CS297 Paper

Expanding JavaScript Metaobject Protocol

Student: Tom Austin (tom@bias2build.com)
Advisor: Prof. Cay Horstmann (horstmann@cs.sjsu.edu)

Abstract

JavaScript has been a much maligned programming language. Browser incompatibilities, poor implementations, and some superficial flaws in its design have led to numerous headaches for developers, and for a long time, it was seen as an evil to be avoided.

All of this belies the fact that JavaScript is a very powerful language. It has support for closures, functional programming, and metaprogramming. In fact, it offers many of the same features that have helped to make Ruby the recent darling of the programming world.

More importantly, JavaScript might be a better scripting language choice for Java programmers. Much of JavaScript's syntax and conventions follows those of Java. Furthermore, it boasts a strong, robust JVM implementation in Netscape/Mozilla's Rhino. Rhino was rated as a close second to Jython in Steve Yegge's celebrated "JVM Language Soko-Shootout". (The original site is down, but http://www.oreillynet.com/ruby/blog/2006/01/a_little_antiantihype.html discusses this).

However, JavaScript has only a somewhat limited Metaobject Protocol (MOP). Expanding this could be a powerful addition to the language. Some possible uses include:

- Allowing web browsers like Firefox or Opera to support Microsoft-specific JavaScript, without cluttering up their API.
- Dynamic multiple-inheritance – Using method intercession, missing methods could be looked for in more than a single prototype chain.

This might also help to make JavaScript a viable server-side language. Ruby on Rails makes extensive use of some of these metaprogramming techniques, particularly in its ActiveRecord object-relational tool.

In this paper, I will explore JavaScript and Ruby's existing metaprogramming features. I will also propose extensions to JavaScript that could give it much of the same power that Ruby has.

Metaprogramming and Metaobject Protocols

These two topics are so closely tied together that I'll slip back and forth between them throughout this paper. However, it is worthwhile to point out the differences between these two concepts.

Metaprogramming, simply put, is the writing of programs that can write and modify other programs. A metaobject protocol is a refinement of metaprogramming focused on objects within these languages. The authors of [4] use this definition:

> Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language.

In other words, a metaobject protocol allows us to modify the way that the building blocks of the language behave. The Common Lisp Object System (CLOS) is the most famous example of a metaobject protocol, and is often cited as the archetype for these systems in general.

Traditionally, metaobject protocol research has been focused on class based object-oriented systems. While class-based design is the more common approach, it is not the only one.

JavaScript instead relies on prototypes. Prototype-based object systems instead define a prototype object. New objects are created by cloning the prototype. This is an inherently more flexible system. It is easy to modify the behavior of a single object or a whole group of objects at runtime. In contrast, this is something that most class-based object-oriented languages cannot do. It is worth noting that Ruby can do this with some metaprogramming magic.

Ruby

Ruby has recently skyrocketed to fame as a well designed, flexible, and powerful scripting language. It is usually described as a combination of Smalltalk and Perl, or Java and Perl for those without Smalltalk experience. The creator of Ruby is Yukihiro Matsumoto, more commonly known within the Ruby community as simply "Matz". In his own description of Ruby he attributes much of the design to Lisp as well [12]:

> Ruby is a language designed in the following steps:
> • take a simple lisp language (like one prior to CL).
> • remove macros, s-expression.
> • add simple object system (much simpler than CLOS).
> • add blocks, inspired by higher order functions.
> • add methods found in Smalltalk.
> • add functionality found in Perl (in OO way).

While Ruby and Lisp have very little superficial resemblance to one another, some of Ruby's features do illustrate the influence. One example is implicit returns; in Ruby, every statement is an expression. The return statement still exists, but with the exception

of early returns, its use is mostly a matter of taste.

Ruby's alleged similarity to Lisp has been a highly contentious issue. Two blog posts in particular managed to stir up a heated debate: Eric Kidd's "Why Ruby is an acceptable LISP" and Steve Yegge's follow up "Lisp is not an acceptable Lisp". The central point of both articles was that Ruby has much of the same flexibility and is much more practical for daily programming tasks. The comments on these articles ranged greatly in their opinions. Steve Yegge himself commented on this [6]:

> [Eric Kidd's article] got approximately 6.02e23 comments, ranging from "I agree!" through "I hate you!" to "I bred them together to create a monster!" Any time the comment thread becomes huge enough to exhibit emergent behavior, up to and including spawning new species of monsters, you know you've touched a nerve.

Regardless of Ruby's background, it has established a reputation as a cleanly designed and user-friendly scripting language. While it is not without its critics, its popularity is clearly on the rise.

In this section I'll highlight some specific features of Ruby's design.

Object Oriented Design

In Ruby, everything is an object. Unlike Java (and JavaScript for that matter), there is no split between primitives and objects. As a result, '1.to_s()' is a valid statement. This leads to a simpler model, since programmers do not have to worry about this dichotomy between primitives and objects.

Ruby, like most object-oriented languages, uses a class-based system. It only supports single inheritance, but has the concept of "mix-ins". Mix-ins are modules that can be included in other classes in order to add functionality. Comparable and Enumerable are two examples of this. These serve in much the same role as interfaces do in Java, with the obvious benefit that they add actual functionality, instead of just obligations. (They do add in obligations as well -- the added methods typically make use of other methods that must be defined in the class. For example, Comparable requires that the <=> operator has been defined).

One notable distinction of Ruby's class system is that all classes are open. While this seems rife with possibilities for abuse by creative programmers, it does give a great degree of flexibility. Here is an example adding the car/cdr functions from Lisp to Ruby Arrays:

```
class Array
  # Returns the head element
  def car
    first
  end
  # Returns the tail
  def cdr
    slice(1,length)
  end
```

```
  def to_s
    s = "[ " + car.to_s
    self.cdr.each do |elem|
      s += ", " + elem.to_s
    end
    s += " ]"
  end
end
list = [1, 2, 3, 4]
puts list.car # prints 1
puts list.cdr.to_s # prints [ 2, 3, 4 ]
```

I'll leave it to the reader to decide whether this is an example of why classes should be open or should not be open.

Both mix-ins and the open nature of Ruby's classes are important for metaprogramming, so we will revisit these again later.

Type System

Ruby is dynamically typed, but not weakly typed. Although programmers do not need to specify the type of a new object, they may be required to convert it before some operations. For instance, here is an attempt to mix a String and an Integer in Ruby:

```
irb(main):002:0> "32" + 1
TypeError: can't convert Fixnum into String
        from (irb):2:in `+'
        from (irb):2
```

Instead, the type conversion must be manually specified. Either way will work:

```
irb(main):003:0> "32".to_i + 1
=> 33
irb(main):004:0> "32" + 1.to_s
=> "321"
```

In contrast, here is Rhino JavaScript:

```
js> 32 + "1"
321
js> "32" + 1
321
```

Ruby on Rails

It has been argued that every new language needs a “killer app” to bring it to the world's attention [7]. For Ruby, this has been the web development framework “Ruby on Rails”. Rails is a nicely designed web framework. It has built-in facilities for testing, a clean division of the model/view/controller pieces, and a friendly object-relational tool named ActiveRecord.

A major axiom of Ruby on Rails is “Don't Repeat Yourself”, often simply referred to as the DRY principle. To achieve this, Rails makes heavy use of default settings. The philosophy of “convention over configuration” means that there is very little

configuration in a typical Rails application. While Rails does provide the ability to override the defaults, this is generally done only for legacy applications.

As a result, Rails seems to work by magic. Or perhaps by voodoo.

It takes some time to get used to the various conventions. However, once you begin to follow the logic, it greatly speeds development. Claims have been made that developers can expect a 10x boost in their productivity.

ActiveRecord is the core to Rails. Without it, Rails would not offer much to entice people to use it. Rails would still be an obscure (though nice) framework in an obscure (though nice) language. It also makes use of "convention over configuration" more than any other single piece of the framework.

Here are two examples of ActiveRecord classes. The names of the database tables, the field to uniquely identify each record, and the foreign key to relate the objects is all determined by default values:

```
# In album.rb
class Album < ActiveRecord::Base
  belongs_to :artist
  has_many :songs
end
# In artist.rb
class Artist < ActiveRecord::Base
  has_many :albums
end
```

Setters and getters are added automatically to the language. As a result, the programmer could then right a script like the following:

```
mark_growden = Artist.new
mark_growden.name = "Mark Growden"
live_at_the_odeon = Album.new
live_at_the_odeon.artist = mark_growden
live_at_the_odeon.title = "Live at the Odeon"
live_at_the_odeon.save()
```

This would save both objects into the database, since ActiveRecord is aware of their relationship.

Metaprogramming

Ruby has many powerful tools for metaprogramming. Many of these also exist in JavaScript; some do not. These will be discussed in more detail later. The important point to note here is that Ruby's metaprogramming features are core to Ruby on Rails and ActiveRecord. Eric Kidd has argued that these offer nearly as much power as Lisp's macros do [5]:

> The real test of any macro-like functionality is how often it gets used to build mini-languages. And Ruby scores well here: In addition to Rails, there's Rake (for writing Makefiles), Needle (for connecting components), OptionParser (for parsing command-line options), DL (for talking to C APIs), and countless others. Ruby programmers write everything in Ruby.

JavaScript

JavaScript is a study in contrasts. It has many ugly, superficial quirks. At the same time, it has a surprisingly elegant core design. On the surface, it has a syntax that seems to be a deliberate clone of Java, but its prototype-based design and its first-class functions are alien concepts to the Java world. It has been regarded as a toy language, and yet it has powered many recent, beloved AJAX applications.

Douglas Crockford offers one of the most concise descriptions [11]:

> JavaScript is a sloppy language, but inside it there is an elegant, better language.

Rhino

With Netscape/Mozilla's Rhino, there is finally a JavaScript that lives up to its name. It is consistently rated as one of the best JVM scripting languages. In addition to adding in tools to script Java, Rhino also includes a number of additional functions that make up for shortcomings in the language's basic design.

As a result, developers have begun to bring JavaScript outside of the browser. Two notable applications that use Rhino are HttpUnit and Phobos. HttpUnit is a tool that can be combined with JUnit to facilitate testing page flow for web applications. Phobos is a Rails-inspired web development framework.

Also, Rhino is now included in Java 6. It seems to have become the flagship of JVM scripting languages.

Prototype-based Object Design

JavaScript is the most widely used prototype-based programming language. While this is an unfamiliar model to most programmers, it is a surprisingly flexible and powerful one. Also, every JavaScript object is a collection of properties. The combination of these two characteristics means that there are very few points that need to be considered when designing a metaobject protocol. In contrast, Ruby has its MOP functionality spread across the Class, Object, and Module classes among others.

First-class Functions

JavaScript functions are first class citizens. They can be passed as arguments, returned from other functions, or stored as properties. Functions are also closures. David Flanagan discusses this in his authoritative reference book on JavaScript [8]:

> The fact that JavaScript allows nested functions, allows functions to be used as data, and uses lexical scoping interact to create surprising and powerful effects.

Throughout his book, Flanagan demonstrates multiple uses for this feature of the language. They can be used to create private namespaces, set breakpoints, and create unique number generators.

Properties

Properties are both the strength and the central failing of JavaScript. Anything can be set as a property to an object, and this is a very powerful feature. As a result of this, JavaScript can easily mimic many of Ruby's metaprogramming features.

However, properties are intrinsically public. This is often undesirable, and it makes it difficult to intercept calls to set or get properties. While nested functions can be used to create getters and setters for private data, this is not the JavaScript way. It breaks with the conventions of the language and loses much of the power and flexibility that JavaScript's properties offer. This will be one major issue that will be addressed with the proposed extensions.

Metaprogramming: Ruby vs. JavaScript

This section will focus on the metaprogramming features within Ruby and the equivalent features within JavaScript. David Black's "Ruby For Rails" covers most of these features in great detail [9]. Outside of digging through the source code for Rails, this was the primary reference for this section.

Singleton Classes

Singleton classes are used to add methods or attributes to individual objects rather than to classes. Ruby's syntax allows the programmer to either define individual methods of the singleton class, or to pry open the singleton class and add methods or variables that way. I'll show an example of the former first, since its syntax is easier to follow:

```
greeting = "Hello"
bob = "Bob"
def greeting.say_twice
  puts self
  puts self
end
greeting.say_twice # This will print "Hello" twice
bob.say_twice # This will throw a NoMethodError
```

Rails uses this technique in its DRb (Distributed Ruby -- one of the several options for storing session information) server setup for ActionController. With this technique, access to the session_hash is synchronized. They use the alternate syntax of 'class <<obj' since they are adding several methods to the class at once. Here is an excerpt:

```
session_hash.instance_eval { @mutex = Mutex.new }
class <<session_hash
  def []=(key, value)
    @mutex.synchronize do
      super(key, value)
    end
    # More methods omitted
  end
end
```

For JavaScript, this is nothing special. JavaScript's prototype-based design inherently provides the same functionality. For instance, the JavaScript equivalent of the say_twice method would be the following:

```
var greeting = new String("Hello");
var bob = new String("Bob");
greeting.sayTwice = function() {
  print(this);
  print(this);
}
greeting.sayTwice(); // This will print "Hello" twice
bob.sayTwice(); // This will throw an Exception
```

The code is longer here, mostly due to the unfortunate split between primitive strings and String objects in JavaScript. The rest of the code is no shorter than the Ruby equivalent, but the syntax seems decidedly cleaner. Ruby's singleton classes seem like a bolted on measure to emulate prototypes.

Eval Methods

This is one of the most powerful metaprogramming features in Ruby. This allows the execution of arbitrary strings as Ruby commands. There are 4 different eval functions:

- eval()
- instance_eval()
- class_eval()
- module_eval()

Eval is the most basic. Also, it is probably the most powerful and the most dangerous. Probably for this reason, it does not seem to be used much in Rails.

The other 3 eval methods are more often used. They differ from the basic eval in that they can also accept blocks of code, meaning that they can be used with much less risk.

The main purpose for instance_eval() is to gain access to the private members of another class. The class_eval()/module_eval() methods are designed to add to the functionality of a class or module and to include variables from the current scope. Together, all 3 of these serve to allow the programmer to inject functionality into another class.

JavaScript has the same basic eval() function. The apply() and call() methods of Function generally fill the same role as the other versions. Because of the elegance of JavaScript's prototype design, fewer MOP tools are needed. This proves to be a recurring theme when comparing metaprogramming in these two languages.

Aliasing a Method

This is heavily used in ActiveRecord, and seems to be one of the core pieces of the design in Rails. The 2 methods used primarily in this are 'alias_method' and (to a lesser extent) 'define_method'. These are used in tandem to create a wrapper around methods.

The method is aliased to a new name, and the original method name is overridden by the wrapper method (typically defined with 'define_method').

In Rails, this is often used to change the functionality of a method. For example, ActionController uses these methods to change what happens when page.render() is

called.

This is nothing exciting for JavaScript. Moving around methods is easy since they are just functions stored as properties.

Callable Objects

Proc, block, and lambda are collectively referred to as 'callable objects'. All three are variations of the same idea -- they are ways to define temporary pieces of executable code. Javascript can already create anonymous functions, so there is little that it is missing.

Ruby has 'method', which returns a reference to the named method. This is mostly needed because of the blurred line between properties and methods in Ruby. JavaScript does not have this issue. 'music.method(:play)' in Ruby would translate to just 'music.play' in JavaScript.

Often used along with 'method' are 'bind' and 'unbind'. Together, these can be used to allow method references to be moved around between objects. The need for this is unclear, and Rails seems to make little use of this feature. In fact, in his discussion on the subject, David Black suggests that if you are using this, you most likely have a problem in your design:

> This is an example of a Ruby technique with a paradoxical status: It's within the real of things you should understand, as someone gaining mastery of Ruby's dynamics; but it's outside the realm of anything you should probably be doing.

JavaScript does all of this already. Its functions seem to be more powerful and flexible. They can have properties of their own (which is not true for Ruby methods), they can be passed as arguments, and they can be bound and unbound at will. Ruby's methods are close, but they are not quite as flexible, which seems to require this extra complexity to achieve the same results.

Mix-ins

As discussed before, mix-ins are used in Ruby in place of multiple-inheritance. They are ways of adding a chunk of functionality to another class. JavaScript has no built in function to do this, though it is easily mimicked. In section 9.6 of his book, Flanagan provides a 6-line method to achieve this. Again, the combination of properties and first class functions provide JavaScript with the power that it needs.

Callbacks and Hooks

Ruby has several different points where a programmer can hook in to the application. They are:

- Module#method_missing
- Module#included
- Class#inherited
- Module#const_missing

Of these, const_missing is used the least. It does not seem to be particularly important. David Black suggests that it could be useful for giving default values to uninitialized constants, but why constants would need default values is a little unclear.

Method_missing comes up frequently. It is often used for creating shortcuts. ActiveRecord uses this to allow calls like 'Employee.find_by_last_name("Austin")'. Behind the scenes, method_missing converts this to 'Employee.find(:first, :last_name => "Austin")'.

Extending the earlier Lisp-like additions to the Array class, we could use method_missing to do the following:

```
class Array
  # This will give more advanced list functions, like cadar or caar.
  # However, unlike in Lisp, there will be no limit to the available
  # methods.  'caadaaddadaadaaddadr' could be called if the
  # programmer so decided (and had a really sick sense of humor).
  def method_missing(method_called, *args, &block)
    meth_name = method_called.to_s
    if meth_name =~ /^c(a|d)+r$/
      list = self
      meth_name.reverse.scan(/./).each do |op|
        if op == 'a'
          list = list.car
        elsif op == 'd'
          list = list.cdr
        end
      end
      return list
    else
      super(method_called, *args, &block)
    end
  end
end
list = [[0, [1, 2], 3], 4]
puts list.caadar  #prints 1
```

While method_missing can do some flashy things, it does not seem to offer any impressive leaps in what you can achieve. However, when combined with JavaScript's prototype-based object design, it does suggest some interesting possibilities. For one, this might be a technique for creating multiple-inheritance. If a method did not exist in one prototype chain, a second prototype chain could be searched.

The included and inherited methods seem to be the core of Ruby metaprogramming, at least for how it is applied in Rails. This is used heavily in ActiveRecord and even more so in ActionController. Here is an example from the base ActionController class:

```
module Layout
  def self.included(base)
    base.extend(ClassMethods)
    base.class_eval do
      alias_method :render_with_no_layout, :render
      alias_method :render, :render_with_a_layout
      class << self
```

```
        alias_method :inherited_without_layout, :inherited
        alias_method :inherited, :inherited_with_layout
      end
    end
  end
  # ... Rest omitted
```

When the Layout module is included, it rewires the render method of the host object so that it will use the layout. It also changes the behavior of the inherited method.

JavaScript does not seem able to compete here. It has no real equivalent to any of these. In particular, it seems like it could benefit from something similar to the included/inherited methods. Fortunately, JavaScript's design makes it easy to cover all of these by intercepting a couple of points.

Setting new properties in JavaScript covers both inclusion of other modules and inheritance (via the prototype chains). By intercepting the getting of properties from an object, method_missing and const_missing could both be mimicked as well. If a mechanism can be created for intercepting the setting and getting of properties, JavaScript's metaprogramming features could become every bit as powerful as those of Ruby.

JavaScript Metaobject Protocol Proposal

JavaScript's power can be greatly increased by adding callbacks and hooks to the language. Fortunately, since JavaScript makes heavy use of properties, we can add most of our hooks at a single point.

JavaScript has only two metaobjects that we need to consider. The first is Object, which is the parent of all other object types. The second is Function, which is a child of Object. Because JavaScript has no classes, this is all that we really need to consider. In contrast, Ruby has Class, Module, and Proc metaobjects to deal with as well.

As it turns out, we can add the additional power we need with Object alone.

Mix-ins

JavaScript can mimic this already, though it is not built in to the language. We can fix this by adding these methods to Object:

- addMixIn(*mixIn*)
- mixedIn(*recipient*) – not automatically added, but reserved by convention.

The addMixIn method is just a modification of David Flanagan's version. It is done in a more object-oriented manner and with a callback mechanism added:

```
Object.prototype.addMixIn = function(mixIn) {
   var from = mixIn;
   var to = this.prototype;

   for (method in from) {
```

```
            if (from.hasOwnProperty(method)) {
               if (typeof from[method] !== "function") continue;
               if (method === "addMixIn" ||method === "mixedIn") continue;
               to[method] = from[method];
            }
         }
         // If the mix-in object has a mixedIn method, it will be called.
         //  This emulates Ruby's Module#Included callback method.
         if (mixIn.mixedIn) {
            mixIn.mixedIn(this);
         }
      }
```

Whenever a mix-in is added to another module, the recipient checks the mix-in for a mixedIn() method. If it finds one, it calls that method and passes itself as the object. This also illustrates how we could track clones of a prototype, although we will need a mechanism to track their creation.

Here is an example mix-in. In this case, we are again adding car/cdr functionality to Arrays, but we are doing it as a mix-in instead:

```
function LispListMixIn() {
   this.mixedIn = function(receiver) {
      var recvMatch = receiver.toString().match(/function (.*?)\(/);
      var recvName = recvMatch ? recvMatch[1] : "primitive";
      print("Adding Lisp functionality to " + recvName);
   }
   this.car = function() {
      return this[0];
   }
   this.cdr = function() {
      return this.slice(1);
   }
}
Array.addMixIn(new LispListMixIn());

var numbers = [1,2,3];

print(numbers.cdr().car()); //This will print 2
```

Property Interceptors

Mix-ins can be done by JavaScript alone. As such, they are really only convenience methods without any real additional power. We will need to make a more fundamental change to the language. The following methods will help us to do that:

- manageProperty(*property*)
- manageWildcardProperty()

These methods will work with a new mechanism. When a program attempts to either get or set any property for an object, it will first check that object for a relevant interceptor. The convention is that *setFoo* and *getFoo* will be the interceptors for property *foo*. (We will call these *named interceptors* to contrast them with the *wildcard interceptors*). If there are no named interceptors, it will check for a setWildcardProperty or getWildcardProperty method.

For ease of use, no error is thrown if any of these methods do not exist. While this could be a source of some debugging challenges, it seems more flexible, and therefore more in keeping with the spirit of JavaScript.

Also, *getFoo* and *setFoo* method calls are not intercepted by the getWildcardProperty method. This reduces the risk of infinite loops and makes the extensions easier to use. Note that setWildcardProperty and any of the named interceptors still apply to the setter and getter methods.

Applications of the New Extensions

The new extensions allow JavaScript to do many things that have not been possible before. In addition, the getWildcardProperty/setWildcardProperty methods allow it to do things that are not easily done in most languages.

For instance, this syntax could easily be used to set an object to a read only state:

```
date.manageWildcardProperty();
date.setWildcardProperty = function(p) {
   throw new Error('Sorry, date.' + p + ' cannot be changed.');
}
```

The getWildcardProperty can be used to emulate Ruby's method_missing idiom. Although it does not execute anything itself, it can create a new function and return that. Here is an example mimicking the Ruby Lisp list example:

```
var list = [[0, [1, 2], 3], 4];
list.manageWildcardProperty();

Array.prototype.car = function() {
   return this[0];
}
Array.prototype.cdr = function() {
   return this.slice(1);
}

list.getWildcardProperty = function (propName) {
   if (propName.match(/^c(a|d)(a|d)+r$/)) {
      var list = this;
      return function() {
         var chars = propName.match(/a|d/g).reverse();
         for (var i=0; i<chars.length; i++) {
            var op = chars[i];
            if (op === 'a') list = list.car();
            else if (op === 'd') list = list.cdr();
         }
         return list;
      }
   }
   else return this[propName];
}
print(list.caadar()); //This will print 1
```

The downside compared to Ruby's method_missing idiom is that it creates a new function object, which is slower. However, with a little adjustment, getWildcardProperty could add this new function to the object, which would greatly speed future calls.

Implementaion

Currently, a prototype has been put together that includes the proposed changes. This has been extended from Rhino JavaScript.

Because of Rhino's design, it is easiest to add new functions to the global level. As a result, some of the added functions look more procedural than object-oriented. To fix this, methods were added to Object. The following pairs of lines are identical in function:

```
manageWildcardProperty(employee);
employee.manageWildcardProperty();

manageProperty(employee,'salary');
employee.manageProperty('salary');
```

This may have some other advantages. Because of Microsoft's influence, ECMA has been very reluctant to add any features that could break existing programs [10]. While the object-oriented style is more aesthetically pleasing, the procedural version might be seen as a safer change.

One limitation of the current implementation is that it only allows the properties to be managed for given objects. Managing properties for a prototype does not make those properties managed for its clones.

Conclusion and Future Work

JavaScript boasts an amazingly powerful design, and it accomplishes this with only a few basic constructs. By making a few simple changes to the language, we can give it all of the same metaprogramming power that Ruby enjoys.

Future work will involve extending this implementation to better deal with prototypes and creating more realistic examples of its usage. For that reason, the next portion of this project will involve building RhinoRecord, an object-relational tool patterned after Ruby on Rails' ActiveRecord.

An initial pass at RhinoRecord has been put together. Currently, this is not using any of the expanded features, but they will be helpful in building out some of the more advanced features of ActiveRecord.

RhinoRecord itself will be used to help build out a web-development framework. Some work for this has been done with JavaServer Faces already. However, Phobos is another logical framework to consider, since it is already built with Rhino. Also, Grails has begun to generate some interest as a potentially language-neutral base for JVM scripting

language web frameworks. At the moment, this is still a Groovy-only framework, but it may be worth more investigation as well.

Because of JavaScript's simple but powerful design, a few simple extensions can greatly expand its capabilities. Future work will focus on illustrating this more concretely.

References

[1] Rivard, Fred. Smalltalk: a reflective language. (Accessed November 2006). http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html.

[2] Paepcke, Andreas. User-level crafting introducing the CLOS metaobject protocol. (Accessed December 2006). http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[3] Tanter, Éric, Noury M. N. Bouraqadi-Saadani, and Jacques Noyé. Reflex – towards an open reflective extension of Java. (Accessed December 2006). http://www.dcc.uchile.cl/~etanter/research/publi/2001/tanterBouraqadiNoye-reflection2001.pdf.

[4] Kiczales, Gregor, Jim des Rivières, and Daniel G. Bobrow. The art of the metaobject protocol. MIT Press, 1991.

[5] Kidd, Eric. Why Ruby is an acceptable Lisp. (Accessed April 2007). http://www.randomhacks.net/articles/2005/12/03/why-ruby-is-an-acceptable-lisp.

[6] Yegge, Steve. Lisp is not an acceptable Lisp. (Accessed April 2007). http://steve-yegge.blogspot.com/2006/04/lisp-is-not-acceptable-lisp.html.

[7] Tate, Bruce. Beyond Java. O'Reilly Media Inc. 2005.

[8] Flanagan, David. JavaScript: the definitive guide, 5th ed. O'Reilly Media Inc. 2006.

[9] Black, David. Ruby for Rails. Manning Publications Co. 2006.

[10] Crockford, Douglas. The JavaScript Programming Language. Yahoo presentation. (Accessed May 2007). http://yuiblog.com/blog/2007/01/24/video-crockford-tjpl/.

[11] Crockford, Douglas. JSLint: The JavaScript Verifier. (Accessed April 2007). http://www.jslint.com/lint.html.

[12] Matsumoto, Yukihiro. Ruby's Lisp features. Ruby-talk mailing list archives. (Accessed May 2007). http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/179642.