

Metaprogramming in Ruby – A Pattern Catalog

Sebastian Günther, School of Computer Science, University of Magdeburg
and Marco Fischer, School of Computer Science, University of Magdeburg

Modern programming languages provide extensive metaprogramming facilities. We understand metaprogramming as the utilization and modification of available language constructs and abstractions while staying inside the language. Metaprogramming requires a rich semantic model of the program that is represented as language constructs. The careful utilization of the constructs allows changing programs and sometimes even the semantics of other language constructs.

There are several use-cases for these kinds of extensions: Runtime adaption, program generation, language customization or the design of Domain-Specific Languages (DSL). However, for using metaprogramming effectively, developers need to understand the mechanisms of metaprogramming and need a language that addresses these mechanisms. These are the prerequisites to communicate about metaprogramming and to actively use it in development.

This paper's goal is to explore and highlight the metaprogramming facilities of the Ruby programming language. Ruby is an interpreted and fully object-oriented language. Any class and method – even those of the built-in entities – can be modified at runtime. We explain the mechanisms in the form of a pattern catalog that is structured along the object's initialization, declaration, composition, and modification. The core contribution is the sensitive grouping of related patterns that exist in literature and to reduce them to the dominant techniques they use. The pattern catalog can be used to communicate about Ruby's metaprogramming capabilities effectively. We explain the patterns with their context, problem, goals, consequences, known uses, and illustrate them in the context of an example application.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns*

General Terms: Languages

Additional Key Words and Phrases: metaprogramming, dynamic programming languages, Ruby

1. INTRODUCTION

Ruby is a dynamic and fully object-oriented programming language. Ruby's roots go back to Japan in 1990, and since then it has evolved from a system scripting language to its today's focus on web applications. Web applications such as Twitter¹ and Amazon² use Ruby. Several interpreters for Ruby exist. The two most mature ones are the original MRI³ written in C and JRuby⁴ written in Java. A complete language specification is available: In the form of executable tests⁵, and as a formal specification draft⁶.

¹<http://twitter.com>

²<http://amazon.com>

³<http://www.ruby-lang.org/en/>

⁴<http://jruby.codehaus.org/>

⁵<http://rubyspec.org/wiki/rubyspec>

⁶http://ruby-std.netlab.jp/draft_spec/draft_ruby_spec-20091201.pdf

Author's address: Sebastian Günther, Universitätsplatz 2, D-39106 Magdeburg, Germany, email: sebastian.guenther@ovgu.de; Marco Fischer, Universitätsplatz 2, D-39106 Magdeburg, Germany, email: marco.fischer@st.ovgu.de

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7 PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7

Ruby supports multiparadigm programming with a mix of imperative, functional, and object-oriented expressions. Object-orientation is the basis, as classes and even methods are objects with (re)definable properties. Ruby has extensive modification capabilities: Module and class redefinition, method extension, saving code in the form of proc objects or strings, evaluating code in any place, and much more. A special property is that even the core classes like `Array` or `String` can be modified – overwriting or extending the basic entities is a great way to customize Ruby. All these modifications are done by metaprogramming.

The Ruby programming language offers metaprogramming facilities to a degree that developers use it unconsciously. However, to use the full potential of the provided facilities, developers need to understand the mechanisms of metaprogramming with a structured pattern catalog that addresses intents and solutions. With this catalog, developers can communicate effectively about metaprogramming and actively use it in development.

This paper's goal is to present a structured catalog of metaprogramming capabilities for the Ruby programming language. We explain the mechanisms in the form of a pattern catalog that is structured along the core object's (modules, classes, methods, procs) initialization, declaration, composition, and modification. This grouping and the patterns explanation focusing on the dominant technique they use is the core contribution of this paper. We explain the patterns with their context, problem, goals, consequences, known uses, and illustrate them in the context of an example application. The target audience of this catalog are programmers that want to understand metaprogramming mechanisms in general, with an obvious focus in Ruby developers. We assume the developer to have a firm grip on common language properties and programming experience.

Section 2 explains more background regarding metaprogramming, Ruby, and the used explanation form for the patterns. Section 3 shortly lists all presented patterns. Sections 4, 5, and 6 explain the patterns. Section 7 discusses the patterns and their usage. Section 8 explains related work, and finally Section 9 summarizes the paper. We apply the following textual formats: *keywords*, *source code*, *PATTERN*, *Subpattern*.

2. BACKGROUND

This section first discusses some definitions of metaprogramming and then uncovers our understanding of metaprogramming in the context of dynamic programming languages. The second part then continues with Ruby and explains the most necessary language concepts that are used as the technical terminology in the pattern descriptions. Finally the last part presents the structure of each pattern and a common example which is used to illustrate the application and implementation of each pattern.

2.1 Metaprogramming

Metaprogramming is a controversially discussed term. The following metaprogramming definitions demonstrate the terminological evolution:

- “code that writes code” [Czarnecki and Eisenecker 2000] or “programs that write programs” [Flanagan and Matsumoto 2008].
- “the creation of new abstractions that are integrated into the host language” [Thomas et al. 2009].

This terminological evolution was motivated by several new dynamic programming languages and their metaprogramming capabilities. As these definitions are not sufficient to describe our understanding of metaprogramming, we will use an adapted form of [Thomas et al. 2009]. In the context of Ruby, we see two difficulties with this definition. First, metaprogramming can be used equally at runtime and at the program's initialization time. Second, the execution of an expression can either be the first-time declaration of an entity or its modification. These difficulties are shown in ►Figure 1 – metaprogramming lies at the border between declaration and modification, and between a program's initialization and runtime. Because of these language-specific conditions, we use the following definition:

Metaprogramming is the application of abstractions integrated with the host-language to (typically) modify the language's metaobjects⁷ or the program's objects (typically) at runtime.

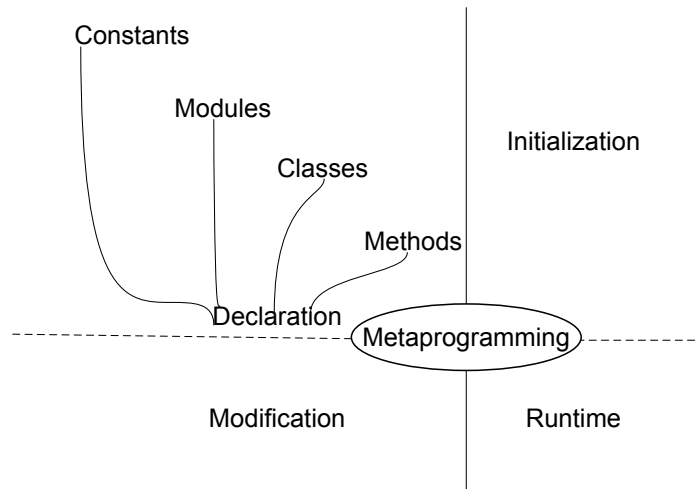


Fig. 1: Applicability of metaprogramming in Ruby.

2.2 The Ruby Programming Language

This section details Ruby's class model, core objects, eigenclass, and method invocation. The information used for the following explanations stems from related work [Flanagan and Matsumoto 2008; Thomas et al. 2009; Perrotta 2010] and our own experiences.

2.2.1 Class Model

Six classes are the foundation of Ruby. The root is `BasicObject`⁸. It defines just a handful of methods needed for creating objects from it and is typically used to create objects with minimal behavior. `Object` is the subclass of `BasicObject` and the superclass from which all other classes and modules inherit. However, most of its functionality (like to copy, freeze, and print objects) is mixed-in from the module `Kernel`. Another important class is `Module` that mainly provides introspection mechanisms, like getting methods and variables, and metaprogramming capabilities, like to change module and class definitions. `Method` is also a first-class entity of Ruby, it is used to represent methods and provides information such as the method's arity. Finally, `Class` defines methods to create instances of classes. ►Figure 2 summarizes the relationships. Two facts are noteworthy. First, `BasicObject`, `Object`, `Module` and `Method` are all `Class` objects. Second, the relationship between `Module`, `Kernel`, and `Object` is circular and thus tells us that all three objects are simultaneously brought into existence by the starting interpreter.

⁷[Kiczales et al. 1995] introduced the term *metaobject protocol* – the summary of all functions and methods that could be used to introspect and modify the semantics of metaobjects (such as classes, methods, and instances)

⁸This is true for the current Ruby 1.9 branch. In Ruby 1.8.x, `Object` remains the top-level entity

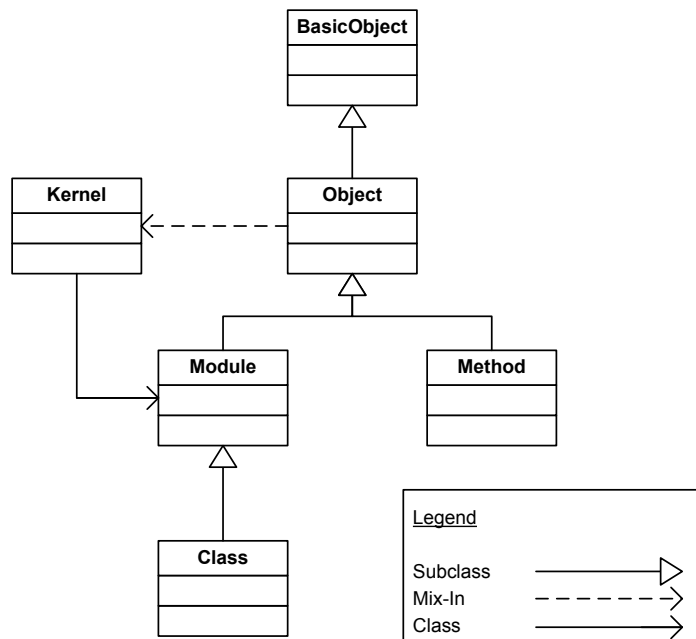


Fig. 2: Ruby's class model.

2.2.2 Core Objects

There are four objects in Ruby that play a fundamental part in the language: `Proc`, `Method`, `Class`, and `Module`. Most of these objects should be familiar to readers experienced with object-oriented programming. However, Ruby's dynamic nature makes the following objects more versatile compared to static languages. Moreover, we see these core objects as metaobjects – modifications of these objects have direct impact on the behavior of the Ruby language.

- *Proc*: A proc is an anonymous block of code. Like other objects, procs can be created either explicitly (referenced by a name) or implicitly (argument of a method). Procs allow two kinds of usage: On the one hand they can reference variables in their creation scope as a closure⁹, and on the other hand they can reference variables that do not yet exist. Procs are executed with the `call` method.
- *Method*: Method declarations consist of the method's name, a set of optional parameters (which can have default values), and a body. Methods belong to the module or class they are declared in. There are two kinds of method objects. The normal `Method` is bound to a certain object upon which it is defined. The `UnboundMethod` has no such object, and, to be executed, it must be bound to an object of the same class in which the method was originally defined.
- *Class*: A class declaration consists of a name and a body. New classes are instances of the class `Class` and they can create instances with the `new` method. Classes can have different relationships. First, they can form a hierarchy of related classes via single subclassing, inheriting all methods of their parents. Second, they can mix-in arbitrary modules.
- *Module*: Like classes, module declarations consist of a name and a body. Modules cannot have subclassing like inheritance relationships – they can only mix-in other modules.

⁹Closures stem from functional programming and capture values and variables in their defined context.

2.2.3 Eigenclass

Some objects in Ruby, especially instances, do not actually implement methods but have pointers to method declarations in their class. An eigenclass is a private class object that is put between the object's original class.

Ruby's class model has a cleanly defined place in which the methods of an object exist. Most of the methods are actually stored in an object's class or its eigenclass.

Consider the example shown in ►Figure 3. Here we define the string "world" (Line 1) and a eigenclass that defines the `hello` method (Line 3–7). As we see in Line 9–10, this method only exists for the specific `world` string, but not for newly created ones. The relationship between the superclass `String` and the eigenclass of `world` is shown in ►Figure 3: the eigenclass becomes the new superclass of the object.

```
1 world = "world"
2 class << world
3   # in the eigenclass
4   def hello
5     "hello " + self
6   end
7 end
8
9 world.hello #=> "hello world"
10 "world".hello #=> NoMethodError
```

Fig. 3: Opening the eigenclass of a `String` instance and define a new method.

Technically, an eigenclass is an additional class in the hierarchy of classes for this object. ►Figure 4 shows that the class of "world" is still `String`. The eigenclass of "world" has `String` as its superclass. If a method on "world" is called, however, the eigenclass is checked prior to the `String` class, the eigenclass is checked first. In the eigenclass, private methods can be defined as well as methods of the superclass overridden.

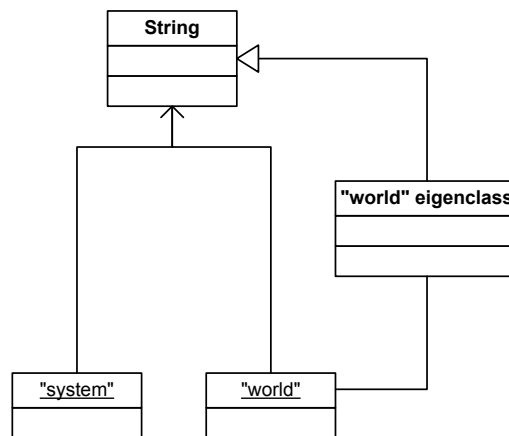


Fig. 4: Different class hierarchies for string objects with and without an eigenclass.

2.2.4 Method Invocation

Method invocation follows a strict protocol. Consider the case of calling `object.method`, the following checks occur.

- (1) – Check the eigenclass of `object` for `method`, and all included modules.
- (2) – If `object` is an instance, check the class of `object` and all included modules.
- (3) – If `object` is a class, check the superclass of `object` and all included modules.
- (4) – Replace `object` with its class/superclass, and repeat step 1 to 4 until the method is found.

If any of these checks finds the method, it is invoked and returned to the caller. If not, the method `method_missing` is called with the purpose to throw an error. However, the declaration of this method is searched in the same way like calling `method` at the first place, but ultimately stops with `Object.method_missing`. We will later see how important this concept is to customize method calls in Ruby.

2.3 Pattern Structure and Common Example

In the pattern catalog, we use a form which is closely related to the original introduction of patterns [Gamma et al. 1997] and more recent approaches [Correia et al. ; Guerra et al. 2009]. The detailed structure thereby is the following:

- *Context*: Explains the coarse situation in which the pattern is applied.
- *Problem*: A question expressing the central concern the pattern addresses.
- *Goals*: Short description of the implementation goals for which this pattern can be used.
- *Solution*: Explanation what language mechanisms are used to provide a solution.
- *Example*: The example section shows how to apply the pattern in the specific context of Ruby and the Hydra example which is explained in the next passage. First a diagram is used to show the application of each pattern and how it affects Hydra. Each diagram shows an abstract representation of the program, the used code to modify the program, and the resulting modification. Used symbols are shown in ►Figure 5. Following the diagram, we detail the implementation of the pattern by showing the code that is utilized for each modification.
- *Consequences*: Additional hints to using the patterns, with a focus on potential pitfalls and tradeoffs.
- *Known Uses*: List of applications that use this particular pattern. We analyzed one or more of these applications:

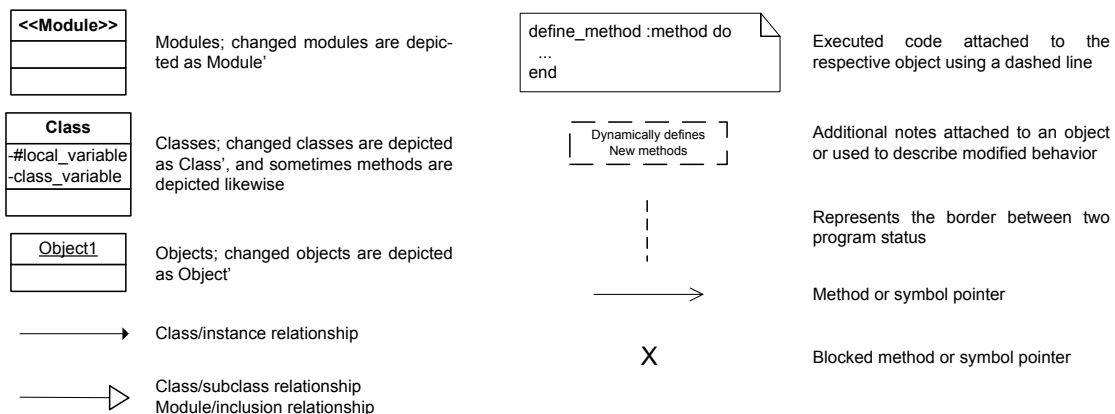


Fig. 5: Symbols used to explain the pattern's structure.

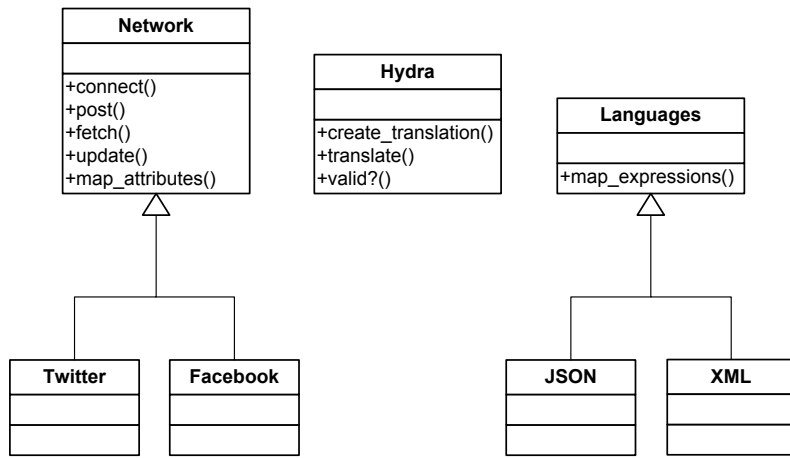


Fig. 6: Hydra's architecture.

- *Rails*: Web-Application framework that implements the Model-View-Controller design pattern [Reenskaug 1979], used for several open-source and commercial projects. We also checked *Builder* that is used to build configuration objects representing XML-like data structures (<http://rubyonrails.com>).
- *ActiveRecord*: Rails standard database abstraction (<http://ar.rubyonrails.com>).
- *RSpec*: Framework and DSL for behavior-driven development (<http://rspec.info>).
- *Sinatra*: Lightweight Web-Framework that uses declarative expressions which look like a DSL for web applications (<http://sinatrarb.com>).
- *Related Pattern*: Finally, a list of closely related patterns.

Except the example section, each pattern is described in a language-independent way and can therefore be applied to other programming languages as well.

The common example we use to illustrate the patterns is called Hydra. Hydra translates and interprets several markup languages that are used to encode data for a specific social network, for example to translate twitter messages to the format used by Facebook. Hydra supports language to language translation and attribute mapping. Language to language translation is a necessity for translating different data formats like XML to JSON. Attribute mapping defines the relationships between attributes of different social network data. Thereby Hydra uses the public API for each network.

Hydra's architecture is shown in ►Figure 6. The main class `Hydra` provides the core method to evaluate a translation configuration, to validate the given configuration, and to start the translation. Furthermore, Hydra defines the two superclasses `Network` and `Language` from which specific classes are derived.

```

1 translation = Hydra.create_translation do
2   from :xml
3   to :xml
4   source_network :twitter
5   target_network :facebook
6   data "post.xml"
7 end
  
```

Fig. 7: Hydra initialization: translating XML markup between Twitter and Facebook.

One concrete example shows how to utilize Hydra for transforming data between social networking services. ►Figure 7 shows the `create_translation` method as it is used to transform XML markup between Twitter and Facebook. We use a `proc` with the `do...end` notation to execute setters for various instance variables. We check all given inputs, and if no errors are reported, the `translate` method is called. This method combines code from the classes `Language` and `Network` for the language translation and attribute mapping. The result is a custom object: it contains the specific attributes, the translation result, and the composed method used for the translations. In the course of the following explanation, we will focus on Hydra for the creation and composition pattern, and then show modification patterns for the Twitter social network.

3. PATTERN OVERVIEW

►Figure 8 lists all pattern that we propose in this paper. They are grouped in accordance with their intent to create or modify objects. Following paragraphs list the problems which each pattern solves.

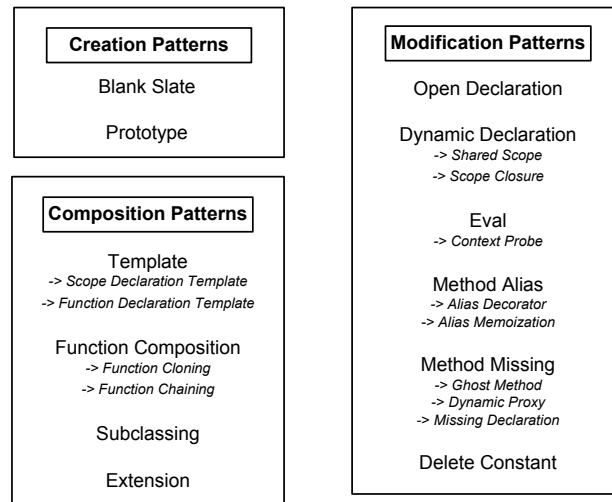


Fig. 8: Pattern overview.

Figure 8 gives a pattern overview and displays their relations. Following sections detail the patterns.

Creation

Creation patterns initially create new objects with properties that are different from common objects.

- BLANK SLATE: How to provide objects with a minimal set of methods so that arbitrarily named methods can be added? (►Section 4.1, page 9)
- PROTOTYPE: How to clone objects instead of creating new instances to save computing time? (►Section 4.2, page 11)

Composition

Composition patterns are concerned with the declaration and instantiation of new objects and thereby reuse functionality that is already available in the program.

- TEMPLATE: How to form an internal and mutable representation of code that facilitates runtime creation, modification, and execution? (►Section 5.1, page 13)

- *Scope Declaration Template*: Declaration of modules and classes.
- *Function Declaration Template*: Declaration of method-like entities, including procs and lambdas.
- FUNCTION COMPOSITION: How to dynamically compose functions out of declared functionality? (►Section 5.2, page 15)
 - *Function Cloning*: Store existing method-like entities in an internal representation.
 - *Function Chaining*: Chain several *Function Cloning* representations as filters in one method call together.
- SUBCLASSING: How to pass method declarations and variables from one class to another, even at runtime? (►Section 5.3, page 17)
- EXTENSION: How to add functionality stemming from different sources into classes, instances, and singletons? (►Section 5.4, page 18)

Modification

Finally the modification patterns take an existing object to modify it.

- OPEN DECLARATION: How to change a class or module after its initial declaration using entity declarations? (►Section 6.1, page 20)
- DYNAMIC DECLARATION: How to change a class or module after its initial declaration using metaobjects? (►Section 6.2, page 21)
 - *Shared Scope*: Share the scope of multiple declarations between each other, providing access to local variables for including them in declarations.
 - *Scope Closure*: Conserve local variables in declarations, completely hiding the variables from any outside access.
- EVAL: How to change a class or module after its initial declaration using code stored in strings or an internal representation? (►Section 6.3, page 23)
 - *Context Probe*: Execute code in the context of another object to check object properties.
- METHOD ALIAS: How to transparently change the behavior of an existing method while preserving the old behavior? (►Section 6.4, page 25)
 - *Alias Decorator*: Add functionality around an existing method.
 - *Alias Memoization*: Replace a method implementation with a fixed return value to save computational time.
- METHOD MISSING: How to enable an object to answer arbitrary method calls and to forward the calls to other methods or define called methods on the fly? (►Section 6.5, page 27)
 - *Ghost Method*: Depending on the method's name, return a value to the caller that simulates a complete method call.
 - *Dynamic Proxy*: Forward the method call to another module or class.
 - *Missing Declaration*: Check the method name and define methods on the fly.
- DELETE CONSTANT: How to completely delete modules and classes together with their methods? (►Section 6.6, page 29)

4. CREATION PATTERNS

The task of creation patterns is to provide objects with a custom, non-standard behavior from which other objects are created. With such objects, developers obtain a large space of customization or tweak the way how instances are created.

4.1 Blank Slate

Context

Ruby objects initially contain more than 200 methods. This reduces the space of method names that can be added to such objects. There are utility functions like `inspect` (prints the object's string representation) or `==` (compares

object identity). Additionally, various introspection methods and the ability to `freeze`, `marshal`, and `taint` the object exist. However, we may not wish to have all those methods defined or want to customize the meaning of those methods.

Problem

How to provide objects with a minimal set of methods so that arbitrarily named methods can be added?

Goals

- *Blank domain objects*: Require objects with method names that represent the application domain only.
- *Save execution time*: Require bare minimum objects to save execution time.
- *Understandability and simplicity*: Objects are much easier to maintain and understand by programmer.

Solution

A BLANK SLATE object is created by taking an existing object and removing all unneeded functions from it. Needed functions are language-specific, but they usually serve to identify a concrete object and to serialize or marshal it. Once the methods from this object are removed, methods with any name can be defined with them.

Example

The central Hydra class serves as the API of our application, it is called for the translation of markup data. We wish to freely extend the Hydra class with custom method names, so we define it as a `BlankSlate`. Our implementation of `BlankSlate` is to use a module that is mixed-into another object and thereby removes unneeded methods.

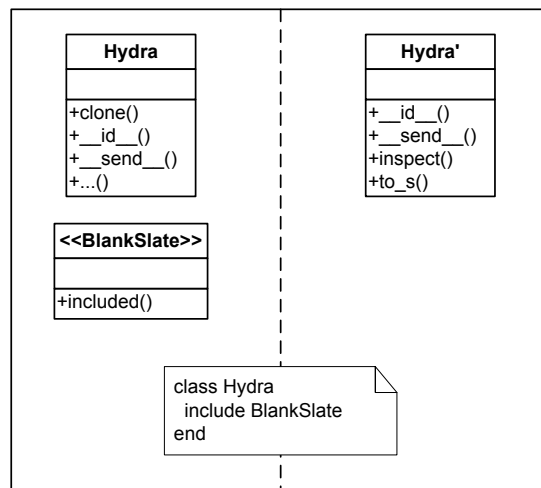


Fig. 9: BLANK SLATE: reducing the methods of `Hydra` from 199 to 4 methods by including the `BlankSlate` module.

The `BlankSlate` module uses `undef_method` to delete the method from its class. All methods except `__send__` and `__id__` are removed in this way.

BLANK SLATE was so popular that Ruby 1.9 introduced the class `BasicObject` at the top of the class hierarchy. It has some methods more than BLANK SLATE proposes, but can be used alternatively.

```

1 module BlankSlate
2   def self.included(base)
3     base.class_eval do
4       methods = instance_methods - ["__send__", "__id__", "inspect", "to_s"]
5       methods.each { |m| undef_method m }
6     end
7   end
8 end

```

Fig. 10: BLANK SLATE: by including the BlankSlate module, all private and public methods of a class except `__send__`, `__id__`, `inspect`, and `to_s` are removed.

Consequences

- To build valid BLANK SLATE objects, carefully select the methods that are to be removed (removing `__send__` disables all method calls on the object!).
- Provide BLANK SLATE objects per application namespace to avoid removing methods which are needed by BLANK SLATE objects in other applications.

Known Uses

- Rails/Builder (defines BLANK SLATE as the base class for building objects that represent XML documents, `builder-2.1.2/lib/blankslate.rb`).
- Ruby 1.9 Basic Object

Related Patterns

- OPEN DECLARATION & DYNAMIC DECLARATION: Alternatively remove selected methods only by executing `undef_method` in class declarations or metaobjects.

4.2 Prototype

Context

Creating new instances is a computational intensive operation. The new object gets an allocated space in the computer, it receives a pointer to a class, and it stores local information. This cost-intensive operation may be inefficient for applications that create a large amount of objects.

Problem

How to clone objects instead of creating new instances to save computing time?

Goals

- *Save computing time*: Prevent instantiating objects to save some computational resources.
- *Objects independent of the class hierarchy*: Create objects outside the borders of the application's class hierarchy.

Solution

At first, a suitable base object must be found. This can be an generic object or an object with specific behavior. The base object should also have no related classes because the PROTOTYPE needs to exist on its own. We modify this object so that no instances can be created from it, but instead clones of it are returned.

Example

Back in ►Section 2.3, we explained that the `create_translation` method is used to specify a particular translation. Normally, a new instance of `Hydra` is returned and the translation occurs within the instance. We tweak this behavior by using `PROTOTYPE`. The translation stores all attributes, the translation result, and even the composed `translation` method (which is specific for the used networks and languages). We use the `Prototype` module for this modification.

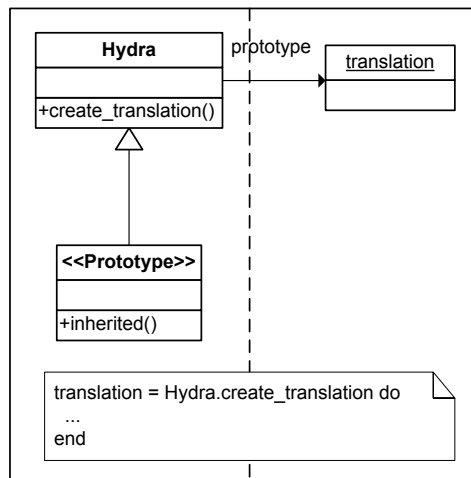


Fig. 11: PROTOTYPE: modifying Hydra to return cloned objects instead of instances.

```
1 module Prototype
2   def self.included(base)
3     base.class_eval do
4       def initialize
5         raise "InstantiationError"
6       end
7     end
8
9     base.instance_eval do
10      def create
11        prototype = self.clone
12        prototype.instance_eval { def initialize; raise "InstantiationError"; end }
13        prototype.instance_eval { def is_prototype?; true ; end }
14        return prototype
15      end
16    end
17  end
18 end
19
20 class Hydra
21   include Prototype
22 end
```

Fig. 12: PROTOTYPE: implementing the Prototype module and using it inside the Hydra class.

The `Prototype` is a module that is mixed-into other classes. The effect is to prevent the creation of new objects by changing the `initialize` method to raise an error, and to add the `create` method that returns a cloned version of the class as the replacement for instantiation.

When the `Hydra` class includes the `BlankSlate` module, the `initialize` method is overwritten to throw an error, and the class receives a `is_prototype?` validator.

Consequences

- As independent objects, prototyped objects can't use `SUBCLASSING` to receive default behavior.
- Independent prototyped objects can't be part of "normal" class hierarchy, but can implement custom relationships and be extended with modules.

Known Uses

Outside `Hydra`, none known.

Related Patterns

- `BLANK SLATE`: Use objects with a minimal amount of methods to further boost the cloning process.

5. COMPOSITION PATTERNS

Composition patterns are concerned with the declaration and instantiation of new objects and thereby reuse functionality that is already available in the program. This helps developers to effectuate program modifications based on run-time code representations.

5.1 Template

Context

Runtime adaptation of code requires a format that can be modified internally. Such modifications are for example methods with custom bodies, or classes and modules with names that are determined at runtime. Anticipating all these changes with a static supply of code is out of question, so another mechanism has to be found.

Problem

How to form an internal and mutable representation of code that facilitates runtime creation, modification, and execution?

Goals

- *Store functions as strings*: Store parts of an application's functionality as string objects.
- *Combine strings representing functions*: Flexible combine string objects to implement modules, classes, methods, and procs.

Solution

Some programming languages can use an `eval` method to execute code. The argument to `eval` can either be an immutable representation like `Proc` objects in Ruby or a mutable form like strings. Strings are supported with several processing methods, often including regular expressions. We suggest to see such strings as templates that embed "anchors" in which they can be modified for a specific task. Templates can be used for the following subpatterns:

- *Scope Declaration Template*: Declaration of modules and classes. The anchors are the name of the class or module, the name of the subclass, and the entity's body.

– *Function Declaration Template*: Declaration of method-like entities, including procs and lambdas. Anchors are the function name, parameters, and the body.

Example

We use the *Function Declaration Template* to provide the basic form of Hydra’s `translate` method. Therefore, we introduce the `Template` module. It contains methods that return a string representing classes of method declarations. Passed parameters are inserted at the template’s anchors.

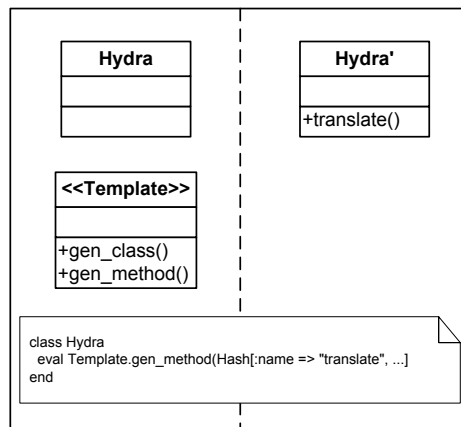


Fig. 13: *Function Declaration Template*: declaring the `translate` method in the `Hydra` class.

The *Function Declaration Template* shown in the following figure uses a Ruby multiline string, its begin and end are marked with the token “RUBY”. The template contains anchors for the method name, parameter, and body. Passed parameters are inserted in the place of the anchors.

```
1 module Template
2   def gen_method(hash)
3     <<-RUBY
4       def #{hash[:name]}(#{hash[:params]})
5         #{hash[:body]}
6       end
7     RUBY
8   end
9 end
```

Fig. 14: *Function Declaration Template*: declaration of a method template with anchors for the name, parameter, and body.

The *Scope Declaration Template* can be implemented likewise, it has inline expressions for the anchors class name and class body.

```

1 module Templates
2   def gen_class(hash)
3     <<-RUBY
4     class #{hash[:name]}
5       #{hash[:body]}
6     end
7     RUBY
8   end
9 end

```

Fig. 15: *Scope Declaration Template*: declaration of a class template with anchors for the name and body.

Consequences

- Code embedded in strings is currently not supported by IDE's for syntax highlighting, error detection and more.
- String code requires rigorous testing to protect against errors.

Known Uses

- Sinatra (create a method with a passed method name, delegate the method call and make the method private, `sinatra-0.9.4/lib/sinatra/base.rb`, Lines 1077 – 1082).
- RSpec (define a method which is automatically registered if it is called and not found, `rspec-1.3.0/lib/spec/mocks/proxy.rb`, Lines 177 – 179).

Related Patterns

- EVAL: Strictly required by TEMPLATE to actually execute the created code.
- FUNCTION COMPOSITION: Compose methods out of existing code objects using procs instead of strings.
- OPEN DECLARATION & DYNAMIC DECLARATION: Using stored templates to modify a program.

5.2 Function Composition

Context

The functionality available inside an application can be used for building new functions at runtime. A running application provides several methods on a global scope or narrow scope (inside modules or classes). These methods are the building block of the application's behavior. The existing functionality can be used to compose new functions at runtime.

Problem

How to dynamically compose functions out of declared functionality?

Goals

- *Store functionality in a non-method object*: Copy existing functionality and store it with an object.
- *Compose stored functionality*: Compose functions out of several existing functions and use them like filters operating on common data.

Solution

Once a program runs, a huge amount of functionality is already available. This functionality can be reused when the language allows to obtain representations of the code, for example as Ruby Proc objects or Python's Code objects. Once detached from its original context, the code can be called independently. In Ruby proc Objects can be called in two ways. First, `Proc.call` executes the proc at the exact position it was defined at, even cloning the

proc to another scope won't change the scope of it's declaration. Second using `class_eval` or `instance_eval` it can be executed at any scope.

With this technique, we see the following use cases:

- *Function Cloning*: Store existing method-like entities in an internal representation.
- *Function Chaining*: Chain several *Function Cloning* representations in one method call together.

Example

Hydra's central `translate` method uses *Function Cloning* to copy existing transformation methods as Proc objects and returns them to the caller to be stored with the transformation object.

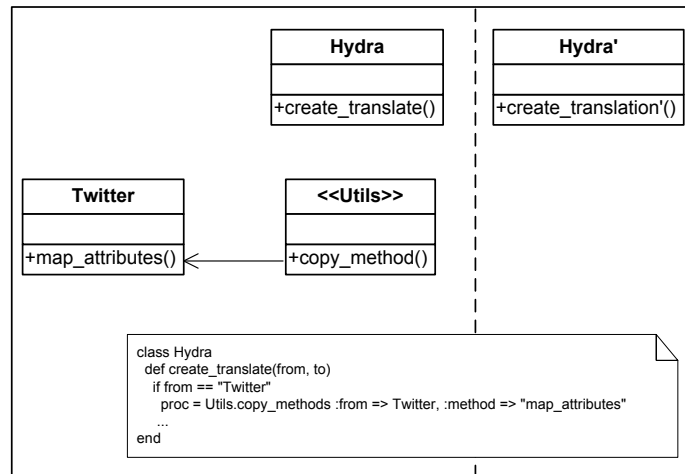


Fig. 16: FUNCTION COMPOSITION: Using *Function Cloning*.

The `translate` method is generated depending on the passed parameters. We check the parameters, and copy the existing methods from the `Network` and `Language` subclasses. The created procs are returned to the caller.

```

1 class Hydra
2   def create_translation(from, to)
3     from_proc = nil
4     if from == "Twitter"
5       proc = Utils.copy :from => Twitter, :method => "map_attributes"
6     elsif from == "Facebook"
7       ...
8     end
9     return [from_proc, to_proc]
10  end
  
```

Fig. 17: FUNCTION COMPOSITION: the `create_translation` method creates two procs using *Function Cloning*, which are returned and used in the created instance for a custom method.

We could use a simple function call too, but there is one decisive reason for this particular approach. Whenever Hydra's `translation` method is called, it will create an instance using the `PROTOTYPE` pattern. The instance then stores a fixed version of the composed translation method.

Consequences

- Calling cloned functions that modify variables of the surrounding scope may provide unintended state changes.
- Since code objects are not readable in the source code, understanding their true semantics may be difficult.

Known Uses

- RSpec (used to define a method chain (methods that always return the same objects), `rspec-1.3.0/lib/spec/matchers/matcher.rb`, Lines 84 – 87).

Related Patterns

- TEMPLATE: Although template functionality is not strictly predefined, we can combine several templates to create new methods.
- SUBCLASSING & EXTENSION: Use subclasses or modules to pass functionality to other entities.
- OPEN DECLARATION & Dynamic Declaration: Composed functions can also be used to modify existing functions at runtime.
- EVAL: Strict requirement to execute the proc at any scope in the program.
- METHOD ALIAS: Alternatively to *Function Cloning*, use alias for a simple 1:1 copy of a method.

5.3 Subclassing

Context

A common goal in object-oriented programming is to reuse existing code with subclassing. From a compositional perspective, this allows to define which entities of the program have a set of default methods and values which determine their behavior. Also the hierarchy of classes represents the structure of the application entities.

Problem

How to pass method declarations and variables from one class to another, even at runtime?

Goals

- *Use existing classes*: Subclasses inherit their parent's functionality and variables.
- *Show class relationships*: Show relationships between different classes, such as the hierarchy of domain objects.

Solution

SUBCLASSING is a common operation in object-oriented languages. For dynamic languages like Ruby and Python, subclassing can be used even at runtime which makes it a metaprogramming operation according to our definition. Depending on the used programming language, different properties are transferred from the parent to the child. The child can gain methods of its parent class, including protected and private methods, as well as fields and attributes.

Example

We explained Hydra's architecture in ►Section 2.3. The classes `Network` and `Language` are common superclasses. `Network` defines a minimal API used to communicate with the respective service. By using SUBCLASSING, the specific network gains these methods too.

SUBCLASSING is invoked by using the interpreter syntax `Subclass < Superclass` or by using the `Class` metaobject in an expression like `Subclass = Class.new(Superclass) do... end`.

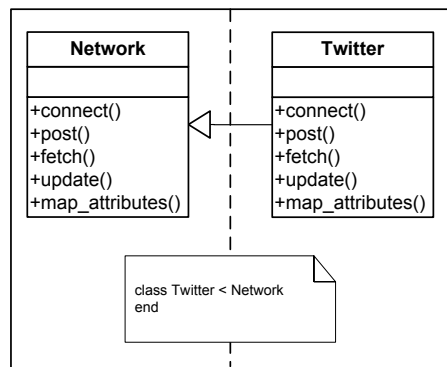


Fig. 18: SUBCLASSING: the `Twitter` class inherits all methods and variables of the `Network` class.

Consequences

- Subclassing is a coarse-grained reuse mechanism, it cannot copy individual methods out of a collection to another class.
- For the child classes, be careful with method naming – method definitions in the child class can override methods inherited by the parent class.
- Although class relationships show the hierarchical structure of the objects, avoid too deep inheritance because code understandability is severely impacted.

Known Uses

- Rake (the `Filetask` class inherits from the `Task` class, rake-0.8.7/lib/rake.rb, Line 766).
- Rails (the `Rails::OrderedOptions` class inherits from the `Array` class, rails-2.3.4/lib/initializer.rb, Line 1097).

Related Patterns

- TEMPLATE & FUNCTION COMPOSITION: Compose methods out of arbitrary other templates or existing functions.
- EXTENSION: Alternatively to using one superclass for method declaration, mix-in collections of methods from multiple modules.

5.4 Extension

Context

Subclassing ties parent and child in a fixed relationship. All methods from the parent are copied to the child. To use subclassing just for providing functionality is not good for those cases where a complex hierarchy of domain objects needs to be represented. Another problem is that only methods from one source can be copied.

Problem

How to add functionality stemming from different sources into classes, instances, and singletons?

Goals

- *Enhance existing objects*: Enhance modules, classes, instances, and eigenclasses with functionality from different sources.

Solution

Modules are the “better” container than classes for shared functionality. Modules can serve functions available in the global scope. And the functions they provide can be copied to other modules, classes, and instances, and even eigenclasses. The relationship of an entity and its class is orthogonal to the subclassing structure which should be used to model the domain.

Dependent on the language, including a module can also work just like a pointer. Changing the module then immediately changes all objects that included the module – this facilitates coarse-grained runtime adaptation. Dependent on the target of including modules, we speak of *Module Extension*, *Class Extension*, and *Eigenclass Extension*.

Example

Hydra defines several helper methods. Some configuration entities (objects that are prototyped from Hydra and define a specific translation) receive the `print_config` method defined in the module `PrintUtils`. We use *Eigenclass Extension* for this purpose.

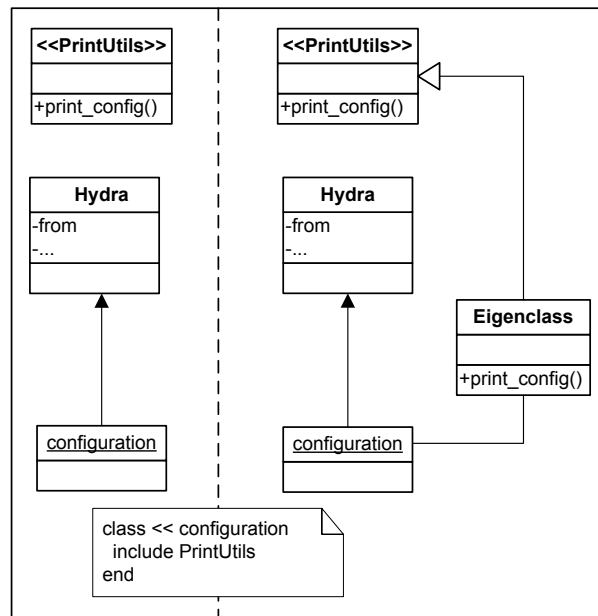


Fig. 19: EIGENCLASS EXTENSION: adding the method `print_config` to the eigenclass of `configuration`.

There are two different methods for extending an entity with a module’s methods. The `extend` and `include` methods can be used to embed a module into a class or another module. By using `include`, the methods of a module are embedded as instance methods. If `extend` is used, the methods of the module are embedded as class methods. The respective statements have just to be executed in the context of the module, class, or eigenclass.

Consequences

- Extending other entities will inevitably override existing methods that have the same name.
- If not composed properly, extending an object with modules may lead to methods being overwritten by modules imported later.

Known Uses

- Sinatra (including `Rack::Utils`, `Helpers` and `Templates` in the Base class, `sinatra-0.9.4/lib/sinatra/base.rb`, Lines 351 – 353).
- RSpec (extending the `Spec::Runner` modules `eigenclass` with configuration options, `RSpec1.30/lib/spec/runner.rb`, Lines 24 – 64).
- RSpec Rails (extending the `Rake` modules `eigenclass` with getters and setters for a Rake application, `rspec-rails-1.2.9/lib/spec/rails.rb`, Lines 267 – 281).

Related Patterns

TEMPLATE & FUNCTION COMPOSITION: Add multiple, individual methods that are composed as procs or strings to modules, classes, or eigenclasses.

SUBCLASSING: If the target is a class and all required methods stem from one source only, than use a subclass relationship.

6. MODIFICATION PATTERNS

The modification patterns are concerned with the modification of objects at runtime. This supports developers in all development challenges where run-time adaptation of systems is important.

6.1 Open Declaration

Also known as: Open Classes, Monkeypatching.

Context

Classes and modules provide the scope for an application's methods. Many are stemming from the standard library or from custom extensions loaded into the application. However, these classes and modules may not exhibit the correct behavior or names for methods, and need to be changed therefore.

Problem

How to change a class or module after its initial declaration using entity declarations?

Goals

- *Modify behavior of methods*: Modify the methods of built-in, external, or application specific classes and modules.
- *Modify the declaration place or visibility of methods*: Adapt existing methods to the environment by copying them to another context or change their visibility.

Solution

To change a class or module at runtime, the important prerequisite is that a suitable language mechanism is found to facilitate this change. For example in Ruby, executing a class declaration again opens the scope of the class and executes any contained code in the context of this class. Ruby even allows modifying built-in classes.

Example

In this example, we add the `translate_text` method to the `Tweet` class. This allows us to call an external service that translates the Twitter text into another natural language. We use the normal class expressions for the OPEN DECLARATION.

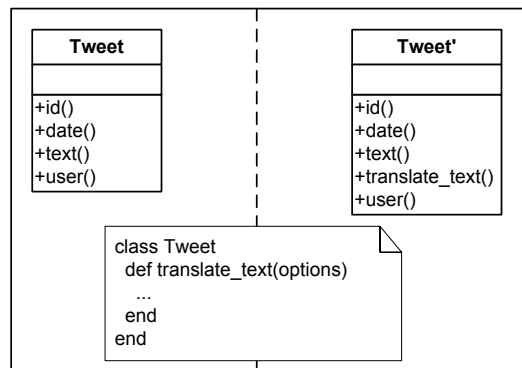


Fig. 20: OPEN DECLARATION: extending the `Tweet` class to include a `translate_text` method.

OPEN DECLARATION works as the pattern name suggests – we use the same class expression again, and the body changes the class definition. For opening the eigenclass of `Tweet`, we execute `class < Tweet`.

```
1 class Tweet
2   def translate_text(options)
3     TranslationWebServer.translate :to => options[:target], :from => self.language
4   end
5 end
```

Fig. 21: OPEN DECLARATION: extending the `Tweet` class to contain the `translate_text` method.

Consequences

- Modifying the behavior of built-in classes may lead to incompatibilities with other libraries, inexecutable programs, or even crashing the running Ruby interpreter.
- Changing the visibility of methods may open them for unintended changes.

Known Uses

- RSpec (extension of `Object` with `Spec::Mocks::Methods`, `rspec-1.3.0/lib/spec/mocks/extensions/object.rb`, Lines 1 – 3).
- Sinatra (modifies the built-in `String` class by using the ALIAS METHOD pattern to copy the `each_line` to the `each` method and the `bytesize` to `length` method, `sinatra-0.9.4/lib/sinatra/base.rb`, Lines 1111 – 1119).

Related Patterns

- DYNAMIC DECLARATION: Modify classes and modules with additional options to reuse or enclose objects of the surrounding scope.
- TEMPLATE & FUNCTION COMPOSITION: Modify classes and modules by executing strings or procs that contain other module, class or method declarations.

6.2 Dynamic Declaration

Also known as: Flat Scope, Nested Lexical Scoping.

Context

In Ruby, the declaration of modules, classes, and methods change the scope of the surrounding code, which makes it difficult to include contextual information. But for example, we want to declare methods with a name that is computed at runtime, or in which the method's body reference to variables of the surrounding scope.

Problem

How to change a class or module after its initial declaration using metaobjects?

Goals

- *Use metaobjects for object creation:* Use the metaobjects `Class`, `Module`, and `Method` for modification instead of language given expressions.
- *Shared Scope:* Share the scope of multiple declarations between each other using metaobjects.
- *Provide invisible local variables:* Provide protected access to local variables to include them in declarations.
- *Scope Closure:* Conserve local variables in a closure.

Solution

Some programming languages provide metaobjects that represent the language's structural entities such as module, classes, and methods. Similar to the relationships between class and instance in object-oriented programming, these objects are the parent for all concrete modules, classes, and methods. Modifying these metaobjects can change the behavior of the language at a very high level. The capability of such modifications are summarized in the *metaobject protocol* of the language [Kiczales et al. 1995].

Ruby provides the metaobjects for modules, classes, and methods. They provide a special constructor that allows to access and incorporate surrounding variables in the naming and behavioral declaration of objects. Ruby metaobjects are invoked like follows: (i) Classes can be defined with `Class.new`, (ii), Modules are defined with `Module.new`, and (iii) methods can be added with `define_method`.

Using metaobjects also modifies the scope in which declarations occur. Normal method declarations using the language given `def` expression changes the execution scope of the expressions, thereby prohibiting access to the surrounding scope. However, by using the metaobjects, code in the surrounding scope, for example local variables are visible in the declaration too. Following usage type can be derived:

- *Shared Scope:* Share the scope of multiple declarations between each other, providing access to local variables for including them in declarations.
- *Scope Closure:* Conserve local variables in declarations, completely hiding the variables from any outside access.

Example

In the following example, we want to extend the `Tweet` class with a `retweet` variable. Retweets are tweets that cite or link other tweets, and thus are an indicator of popularity. This variable shall be invisible to any other method, but still be usable in the method declaration.

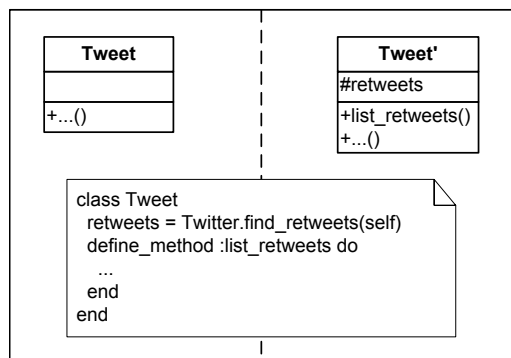


Fig. 22: DYNAMIC DECLARATION: utilizing *Scope Closure* to define an inaccessible `retweets` variable that only exists within an accessor method.

For this purpose, we use *Shared Scope* to include the local variable `retweets` within the `list_retweets` methods, and at the same time use *Scope Closure* to protect this variable from any outside access. We use the method `define_method` for this implementation. The methods to create metaobjects for the other entities are `Module.new` and `Class.new`.

Consequences

- Classes can not be extended with DYNAMIC DECLARATION, because the class object of the declaration is bound to the named variable. If the variable is already defined, its declaration is not extended, but replaced.
 - When sharing the scope, one could use a local variable that unintentionally refers to variables in the surrounding scope
 - The *Scope Closure* is destroyed when the scope containing method is overwritten.
- This has several differences in comparison to OPEN DECLARATION. The class statement must always define a receiving object, a constant if it's going to be globally visible, or else the class declaration will not be bound to any object.

Known Uses

- RSpec-Rails (access local defined `model` variable in a returned class object, `rspec-rails-1.3.2/lib/spec/rails/mocks.rb`, Lines 98 – 111).

Related Patterns

- OPEN DECLARATION: Using common class or module declarations limits the availability of entities from the surrounding scope for modifications.
- TEMPLATE & FUNCTION COMPOSITION: Modify classes and modules by executing strings or procs that contain other module, class or method declarations.

6.3 Eval

Context

Similar to OPEN DECLARATION and DYNAMIC DECLARATION, this pattern modifies an existing module, class, or instance – but reuses existing code. The code stems from existing methods and is put in a form that is modifiable and executable at runtime.

Problem

How to change a class or module after its initial declaration using code stored in strings or an internal representation?

Goals

- *Evaluate code in an object*: Evaluate code in the context of a class, module, or instance.
- *Define objects contained in strings*: Define objects and methods that are expressed in a string template.
- *Context Probe*: Examine an arbitrary object by executing code in it.

Solution

Finding a suitable code representation usually takes the form of either a string or a built-in object. Using strings has more advantages because they can be processed with the mechanisms of the language. This facilitates powerful adaptation and transformation approaches. Independent of the form, the code can express any operation. One interesting subpattern is called *Context Probe* [Perrotta 2010]: Execute code in the context of another object to check object properties. We can for example access local variables or call private methods by just putting some of this code in a string and executing it in the context of the object.

Example

The Tweet class receives two different introspection methods. The `public_introspection` method is defined for the Tweet class and all its instances, it returns a hash of a tweet's id, text, user. The `private_introspection` method is defined for selected Tweet instances only, and provides more detailed information such as the date when the tweet was inserted into the database.

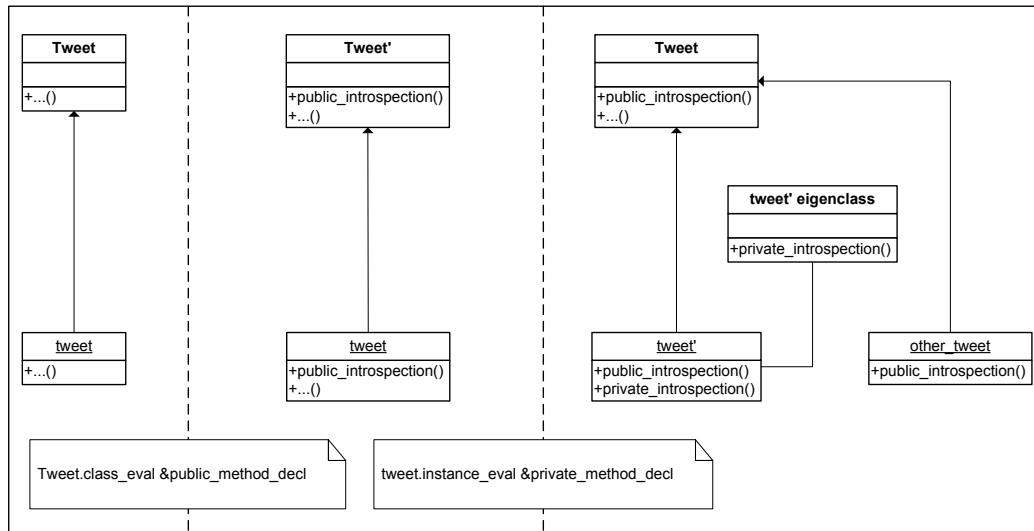


Fig. 23: EVAL: Applying `class_eval` to the Twitter class adds a method to the class that can be used from all other instances, but using the `instance_eval` executes a method declaration in the instance's private eigenclass, thereby only changing one instance.

There are three EVAL methods: `eval`, `class_eval` (and its alias `module_eval`), and `instance_eval`. They differ in the following properties:

- `class_eval` changes the two pointers `self` and `class` to the object's class, and executes the code in this context.
- `instance_eval` changes the two pointers `self` and `class` to the object's eigenclass, and executes the code in this context.
- `eval` only accepts string objects, and can receive an optional binding (represents a certain program state, like a specific execution context that contains scope and local variable information) that is used as the execution context.

We use `Proc` objects together with `class_eval` to modify the `Tweet` class and `instance_eval` to modify the instances.

```

1  pub_method = lambda do
2    def public_introspect
3      Hash[:id => self.id,
4           :text => self.text,
5           ...]
6    end
7  end
8
9  priv_method = lambda do
10   def private_introspect
11     Hash[:creation_date => self.timestamp,
12          ...]
13   end
14 end

```

Fig. 24: EVAL: Using `class_eval` to change the `Tweet` class globally, and use `instance_eval` to change a single `Tweet` instance (but not all instances of the class).

Consequences

- The evaluated code can unintentionally change the objects semantics.
- In Ruby, using string objects for defining new methods is a very slow operation compared to the common declaration using text written in a compiler.
- Using context probe may break the encapsulation principle.

Known Uses

- Sinatra (Uses `eval` with a string to define delegation methods that forward to `Sinatra::Application`, `sinatra-0.9.4/lib/sinatra/base.rb`, Lines 1077 – 1082).
- RSpec (uses `class_eval` with a string to define a method that registers the method call with an internal proxy, `rspec-1.3.0/lib/spec/mocks/proxy.rb`, Lines 176 – 181).

Related Patterns

- TEMPLATE & FUNCTION COMPOSITION: To execute declarations contained in string or `proc` objects at any scope, EVAL is a strict necessity.

6.4 Method Alias

Context

The lure to change the built-in classes traps developers into risking compatibility breaks with other libraries. Ruby's flexibility allows customization of external libraries or the core language itself. However, compatibility needs to be

ensured especially when the application is used as a library in the future. This demands an approach of careful aliasing existing methods.

Problem

How to transparently change the behavior of an existing method while preserving the old behavior?

Goals

- *Transparently wrap methods*: Modify built-in classes while staying compatible with other libraries.
- *Hide existing methods*: Permanently hide a method from being called.
- *Define a method dispatcher*: Use an existing method as a dispatcher and forward the call to another method.
- *Alias Memoization*: Memoize a complex computation.

Solution

Aliasing an existing method is done by copying an existing method under a new name and then redefine the original method. This operation is common in Ruby so that it even got it's own name: `alias`. The *Method Alias* pattern allows to insert custom logic how such method calls should be treated properly and yet retain the original behavior. This can be used for the following two reasons:

- *Alias Decorator*: Add functionality around an existing method. For example, enhance the original methods functionality by calling an additional logger or augmenting the return value.
- *Alias Memoization*: Replace a method implementation with a fixed return value to save computational time. For example, complex computations that are based on static data waste computational resources when they are calculated each time. With alias memorization, the original method is replaced by a fixed return value. The alias checks from time to time whether changes in the data occurred, and updates its return value.

Example

The `translate_text` method of `Tweet` is used more frequently in the course of the applications utilization since we want to use the translated version of a `Tweet` to be synchronized with other social networks. But the modification should not be stored in the database, and we do not want the translation service to be called every time. `METHOD ALIAS` comes to the rescue. We alias the original method as `old_translate`, and redefine the method to return a local variable each time it is called. Only at the very first time the method is called, we call the original method and then store its result in the local variable.

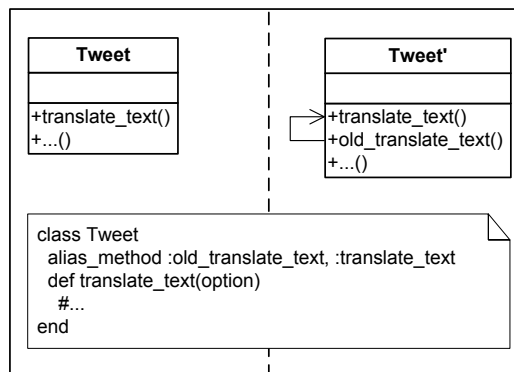


Fig. 25: METHOD ALIAS: Executing a *Alias Memoization* to store the translation result of a tweet's text.

The Ruby language defines two alternatives for this kind of modifications: `alias` and `alias_method`. Their difference is that `alias` is a keyword of the language, while `alias_method` is defined in `Module` and thus modifiable too.

```
1 class Tweet
2   alias_method :old_translate_text, :translate_text
3   @text_translation = ""
4   def translate_text(options)
5     @text_translation ||= old_translate(options)
6   end
7 end
```

Fig. 26: METHOD ALIAS: using *Alias Memoization* to memoize the `translate_text` method for a specific `Tweet` instance.

Consequences

- If methods are hidden from direct call, then the comparability with other classes has to be ensured.
- Memoization requires a careful implementation, because values stalled for too long can result in logical bugs.

Known Uses

- RSpec (uses `alias` to create temporary proxies for methods, and removes them later, `rspec-1.3.0/lib/spec/mocks/proxy.rb`, Lines 172 – 236)

Related Patterns

- FUNCTION COMPOSITION: Store an existing method declaration as a `proc` to retain the original behavior. Then, redefine the method and optionally use the stored `proc` for executing the unaltered behavior.
- METHOD MISSING (*Dynamic Proxy*): Instead of providing a fixed behavior, forward the call to a proxy that decides how to respond.

6.5 Method Missing

Context

Developers that wish to flexibilize their application need to provide a dynamic interface that allows not yet defined methods to be called. This interface does not strictly define the available methods, but is open to respond to any method calls and can then even define methods on the fly.

Problem

How to enable an object to answer arbitrary method calls and to forward the calls to other methods or define called methods on the fly?

Goals

- *Missing Declaration*: Add a called method at runtime.
- *Contextualized method dispatch*: Use regular expressions to check the method's name, dynamically deciding which method to call.
- *Dynamic Proxy*: Forward the method call to another object.
- *Ghost Method*: Return values so that it looks like an existing method was called.

Solution

Method call semantics are a fixed part of the language. Some languages allow intercepting method calls directly by overriding a hook method. This is the preferred way to extend the method to which an object responds without actually implementing the methods. Using such a mechanism allows the following patterns as they are explained in [Perrotta 2010]:

- *Ghost Method*: Depending on the method's name, a value is returned to the caller and a normal method call is simulated.
- *Dynamic Proxy*: Forward the method call to another module or class (or `method_missing` of another object).
- *Missing Declaration*: Check the method name and define methods on the fly. For example, Rails *Dynamic Matcher* checks the method name by applying regular expressions, and if they reference correct fields in the database, define this method with a body that executes a database query.

For checking the method name, we can compare it to a fixed string or use regular expressions to check for certain parts of the method call [Perrotta 2010].

Example

The `Twitter` class should be augmented with the *Missing Declaration* pattern for defining assessors to its values on the fly. Methods that we would like to use start with `find_by`, and then mix attributes and the `and` keyword. The method checks whether the attributes contained in the called method exist in the class. Only if all methods are defined, we define the new method.

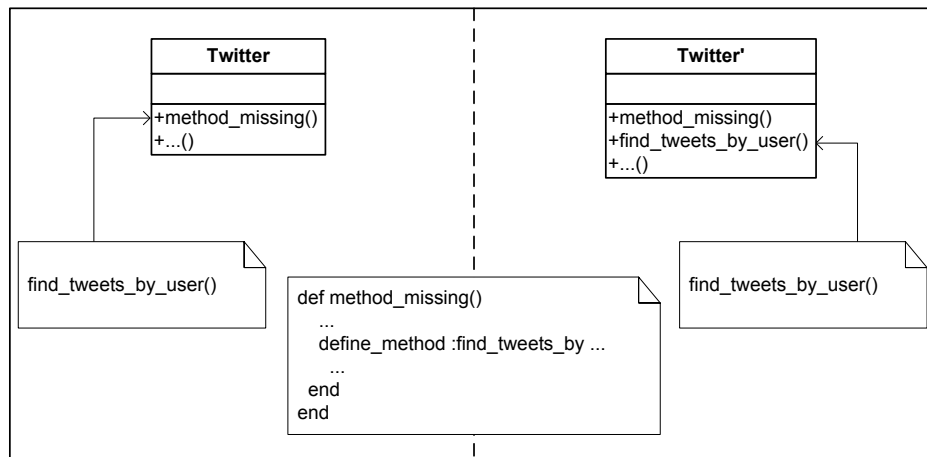


Fig. 27: METHOD MISSING: using *Missing Declaration* to define `find_by_*` methods for `Twitter`.

Here is the prototypical implementation of the *Missing Declaration* pattern. We check whether the called method starts with `find_by`. Then the other tokens of the method call are analyzed and checked if they correspond to attributes of `Tweet`. We use the attributes to construct a custom return value, and use a `DYNAMIC DECLARATION` for the actual method declaration.

```
1 class Twitter
2   def method_missing(name,*args, &block)
3     if name.to_s =~ /^find_by/
4       parts = name.to_s.split("_")
5       parts.each do |part|
6         case part
7           when "find" then
8             #...
9             define_method name do
10              ...
11            end
12          else
13            super(name, *args, &block)
14          end
15        end
16      end
17    end
18  end
19 end
```

Fig. 28: METHOD MISSING: using *Missing Declaration* to create a method searching for tweets by an username.

The *Dynamic Proxy* can be used to add convenience functions to `Hydra`. Method calls that start with `translate` are further analyzed for languages, and then we call the normal `translate` method with the other parts of the method call as parameters.

```
1 class Hydra
2   def method_missing(name, *args, &block)
3     if name.to_s =~ /^translate_/
4       from, to = nil, nil
5       parts = name.to_s.split("_")
6       parts.each do |part|
7         if part == "twitter"
8           from = "Twitter"
9         end
10        ...
11        translate from, to
12      end
13    end
14  end
15 end
```

Fig. 29: METHOD MISSING: declaration of *Dynamic Proxy* in `Hydras Core`, that forwards and orchestrates method calls.

Consequences

- *Missing Declarations* should be checked to not accidentally override an existing method.
- When *Ghost Methods* are explicitly defined as part of an object's interface, it is difficult to document them with tools that parse an application's source code
- *Ghost Methods* are not supported by current IDEs in terms of syntax highlighting, error detection and more.

Known Uses

- `ActiveRecord` (*Missing Declaration* for database queries, `activerecord-2.3.5/lib/active_record/base.rb`, Lines 1839 – 1961).

Related Patterns

- **TEMPLATE & FUNCTION COMPOSITION:** Define the called methods by executing a predefined string or proc object.
- **METHOD ALIAS:** When one method can be accessed by different method names, define the missing ones by aliasing.

6.6 Delete Constant

Context

Systems with a high data throughput will create several objects inside the Ruby VM, and these methods occupy memory space. Although the garbage collector may eventually remove these objects, they may occupy space far longer than necessary. Dependent on the programming language, some may explicitly represent class and method objects that are only deleted when all references to them vanish. Instead of waiting for the garbage collection, we need an explicit operation to delete constants and their defined method objects immediately.

Problem

How to completely delete modules and classes together with their methods?

Goals

- *Cleanly remove objects:* Remove a class or module completely with their methods so that calling its methods fails.

Solution

In order to completely remove a module or class and its defined methods, we need to access the language-dependent mechanisms that are used to store entities in the first place. In Ruby, we can use the built-in `remove_const` method to remove the entity, and extend this method to also delete referenced methods.

Example

In the course of Hydra's implementation, we use several social networks for a one-time synchronization, but soon after, we do not need the instance anymore and want to remove it completely.

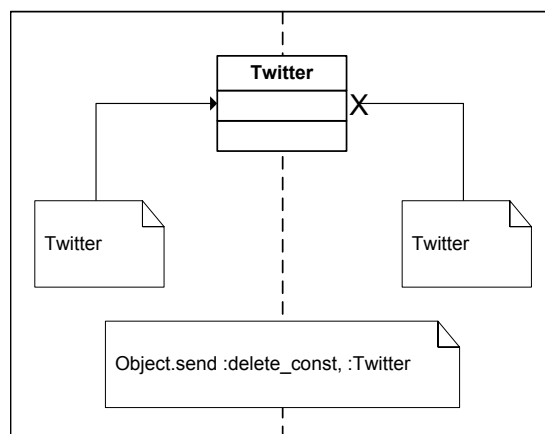


Fig. 30: DELETE CONSTANT: after using a `Twitter` instance to synchronize some messages, we delete this class from our application.

Using the built-in `Object.remove_const :Twitter` method does not completely remove all traces of the instance. The solution is to use a custom method that first deletes all methods of the class, and then the symbol. Here is the implementation:

```
1 class Object
2   def delete_const(con)
3     object = const_get(con)
4     object.instance_methods.each { |m| object.send :undef_method, m }
5     remove_const(con)
6   end
7 end
8
9 Object.delete_const :Twitter
```

Fig. 31: DELETE CONSTANT: removal of the constant and deletion of all its methods.

Consequences

- Use with care since changes to the current program are irreversible.

Known Uses

Outside of Hydra, none known.

Related Patterns

- BLANKSLATE: Alternatively to immediately deleting the methods, defining the objects as a BLANKSLATE prevents existing methods from being called.
- DYNAMIC DECLARATION: Alternatively override an existing class or module and “virtually” delete the methods. New instances, subclasses, or mixins can not respond to the method calls. However, objects that existed before the re-declaration still reference the original method declarations.

7. DISCUSSION

This sections broadly discusses when and how to use the explained patterns and the differences between them. We also explain security considerations about applying some metaprogramming techniques.

7.1 Creation Pattern

There are two creation patterns: BLANK SLATE and PROTOTYPE. PROTOTYPE objects are intended to save some computational time by copying instead of instantiating new objects. Our solution prevents to create any instances from prototype objects. Instead, BLANK SLATE creates normal objects from which instances can be created. Both mechanisms, however, prevent to use SUBCLASSING for defining additional methods in the classes themselves.

We suggest to use these patterns only for their original need: BLANKSLATE when objects need to have a wide space for naming additional methods, and PROTOTYPE when a large amount of objects need to be created and saving computational resources is of importance. Both concepts can be combined to have minimal and resource-effective objects available.

7.2 Composition Patterns

We group the composition patterns along the amount of objects and methods they can define.

TEMPLATE is a very powerful pattern. Strings are common objects in Ruby, and many manipulation mechanisms exist, including regular expressions. This facilitates composing strings from very different sources: local text files,

environment variables, databases, or web services. Strings are appended to each other and they can contain arbitrary object declarations, methods definitions, and modifications of existing objects. `TEMPLATE` should be used wisely by the developers, and it requires careful integration with the other techniques.

`FUNCTION COMPOSITION` is primarily concerned with reusing existing code at runtime. It allows to combine several existing methods into one new method. Because proc objects are immutable, this pattern is not as powerful as `TEMPLATE`. But we can chain the result of several procs together to stack functionality in a functional programming style. Like `TEMPLATE`, `FUNCTION COMPOSITION` is very fine grained and allows to use individual method declarations.

The goal of `SUBCLASSING` and `EXTENSION` is to reuse existing methods. It may be controversial to define them as metaprogramming patterns, but in accordance with our definition of metaprogramming, they can be used at runtime for modifying programs. `SUBCLASSING` is a very restricted mechanism: It can only be used during the initial class declaration, and cannot be changed once set. Indeed, this is one of the rare cases where Ruby does not allow further modification. Since only one superclass can exist, method reusing would be severely limited in Ruby. And this is the particular motivation for using modules. `EXTENSION` allows to mix-in methods of several modules. Also, only `EXTENSION` allows to augment the eigenclass of objects. We argue to use `SUBCLASSING` for representing object-hierarchies similar to the application's domain, and `EXTENSION` for the purpose of reusing existing sets of grouped methods.

The difference between `TEMPLATE` and `FUNCTION COMPOSITION` on the one hand, and `SUBCLASSING` and `EXTENSION` on the other, is the granularity: The first group is concerned with the provision of individual methods, and the second group bundles a set of methods that can be shared. `TEMPLATE` is outstanding in its capability to freely recombine existing declarations, and it is not limited to method declarations only.

7.3 Modification Patterns

For the modification patterns, we see one group that is about class and module modifications (`OPEN DECLARATION`, `DYNAMIC DECLARATION`, and `EVAL`), and one group to modify methods (`METHOD ALIAS`, `METHOD MISSING`).

In the first group, the different techniques synthesize parts of the same goal. We suggest to use `DYNAMIC DECLARATION` and `EVAL` for the initial declaration of entities. If developers want to hide local variables in closures, or need to share variables in different scopes, than `DYNAMIC DECLARATION` (*Scope Closure*) can be used. That is the only option to truly hide a local variable, since the `EVAL` (*Context Probe*) pattern can be used to access class variables or instance variables. Using `EVAL` with procs has a similar effect of sharing local variables. Both patterns can be used in conjunction with `FUNCTION COMPOSITION` to reuse existing functionality. For very dynamic and adaptive code, `EVAL` with the `TEMPLATE` pattern can be used. Once the entities are created by these patterns, developers should stick with `OPEN DECLARATION` and `EVAL` for further customizations, since `DYNAMIC DECLARATION` overrides the definition of existing entities.

The techniques of the second group form alternatives to each other. `METHOD ALIAS` premier role is to customize existing functionality. The modification is transparent and reversible since the original method remains available – to “un-alias” an aliased method is simple. As long as the modification happens around the original method, this technique is sufficient. For inner method modification however, we recommend to combine `METHOD ALIAS` to save the original methods, and then to use `FUNCTION COMPOSITION` or `TEMPLATES` for the method modification. Alternatively, one can use a richer semantic model as provided by `rbFeatures` [Günther and Sunkle 2009b] or even holistic manipulations including an abstract-syntax tree like representation of code [Günther and Sunkle 2009a] to customize method bodies.

When one method should be callable by many different names, `METHOD MISSING` is the alternative to `METHOD ALIAS` because it eliminates the need to define concrete methods. In this case, using `METHOD ALIAS` would bloat the implementation since we need to define one method for each call that should be proxied. Now, `METHOD MISSING` does not only center the declaration of this behavior, but it is also the sole solution for answering any

method call or to define called methods on the fly. It is crucial to define METHOD MISSING precisely for the objects that need this pattern – the best place is the object’s eigenclass.

7.4 Pattern Relationships

In ►Figure 32, we summarize the above discussions and all relationships that we explained in the related pattern sections.

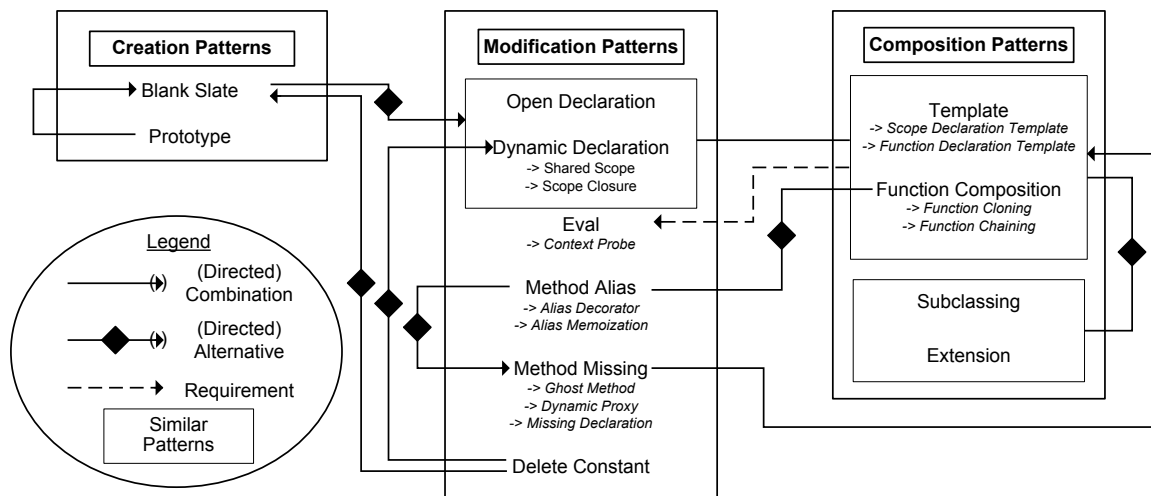


Fig. 32: Pattern Relationships.

7.5 Security Considerations

All presented patterns work on the metaobject layer of a program’s model. They allow coarse and fine grained modifications and extensions of all program entities, including built-in classes. It is important to thoughtfully restrict the changes to only those parts of the program where they are needed. Developers need to structure and architect changes precisely. Another important point is to carefully check what is produced by the TEMPLATE and FUNCTION COMPOSITION patterns. User input should be examined at any case and never executed directly. For example, if system input is directly evaluated, users could insert any code in the string and thus get internal access to the system where they can read data or manipulate the system. Instead, the input should be checked for certain patterns – for example with the help of regular expressions – and only execute code dependent on thoughtfully defined conditions.

Additionally to these design consideration, Ruby provides another mechanism: Safe levels and tainted objects [Thomas et al. 2009]. The safe level is an interpreter process specific variable. There are five different safe levels. Higher levels disallow to read code from global writable locations, prohibit introspection like listing methods of an object, or disallow to evaluate tainted strings. Per default, all objects that stem from an environment variable or all strings read from the surrounding system are considered tainted. If objects are created out of those objects, they are considered tainted too. Using safe levels and tainted objects purposely supports security of applications altogether and can be used effectively with the explained patterns.

8. RELATED WORK

Books about Ruby, like [Thomas et al. 2009; Flanagan and Matsumoto 2008] also explain several Ruby metaprogramming capabilities. The presentation is oriented at the methods and their occurrence within the class model. They are not providing a pattern catalog like we did, so that the understanding of the bigger context and utilization of the techniques is not provided. Furthermore, the explanation is not structured into creation, composition, and modification, making it difficult to see when and where the techniques should be used best.

The original design patterns explained in [Gamma et al. 1997] were ported to Ruby in [Olsen 2007]. The book showed how easy structural patterns are expressed with Ruby since the language is so open to customization. Our perception was that most design patterns do not need a structure of collaborators, but can be expressed with a single entity using several programming techniques. We see these design patterns as providing the coarse structure of a program, and our metaprogramming patterns for more fine-grained design decisions.

Finally, there is one book giving a practical-oriented introduction to metaprogramming in Ruby [Perrotta 2010]. The book uses the metaphor of a spell to explain a certain metaprogramming mechanisms together with an intent. Our work differs in several points. First, we used the classical form of presenting patterns while [Perrotta 2010] sticks to name, intent, examples, and only sometimes provides known uses. Second, we group the patterns into creation, composition, and modification instead of providing a list oriented at the book's narrative content, which is better to pinpoint the patterns utilization. And third, we discuss the explained patterns and their alternatives to each other to detail the application of pattern combinations.

For metaprogramming in other languages, several articles and work exist. For C++, [Czarnecki and Eisenecker 2000; Vandevoorde and Josuttis 2003] explains template metaprogramming. Templates allow defining functionality independent for the used type. In the compilation phase, a preprocessor uses the templates to generate and link code for specific types. The books do not identify patterns for template metaprogramming, but they differentiate into class and function templates. One article explains how the traditional design patterns in [Gamma et al. 1997] can be flexibilized with C++ metaprogramming [Bachmann 2006]. This work uses a pattern explanation similar to ours and groups the patterns according to the structural intents of the original design patterns. A similar approach for Lisp is explained in [von Dincklage 2003]. This work shows that the classical design patterns can be implemented using functions that encapsulate the creation and modification of structured classes and methods.

9. SUMMARY

Metaprogramming is an important property of the Ruby programming language. Many libraries use several metaprogramming techniques: The definition of classes and methods out of string templates, the provision of method invocation proxies, and runtime adaptation including the composition of new method calls out of or around existing functionality. Because many mechanisms are built into the language, they can be used for the declaration and modification of entities both at the program's initialization and runtime. We identified 23 patterns and subpatterns which are grouped along their main usage creation, composition, and modification of entities. The patterns build a rich language that can be used to plan and explain the implementation of a program.

Acknowledgements

We thank Maximilian Haupt, Matthias Splieth, and the shepherd for feedback on earlier drafts of this paper. Sebastian Günther and Marco Fischer work with the Very Large Business Applications Lab, School of Computer Science, at the Otto-von-Guericke University of Magdeburg. The Very Large Business Applications Lab is supported by SAP AG.

REFERENCES

BACHMANN, P. 2006. Static and Metaprogramming Patterns and Static Frameworks. In *Proceedings of the the 13th Conference on Pattern Languages of Programs (PLOP)*. ACM.

- CORREIA, F. F., FERREIRA, H. S., FLORES, N., AND AGUIAR, A. Patterns for Consistent Software Documentation. In *Proceedings of the 16th Conference for Pattern Languages of Programs (PloP)*.
- CZARNECKI, K. AND EISENECKER, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, San Francisco et al.
- FLANAGAN, D. AND MATSUMOTO, Y. 2008. *The Ruby Programming Language*. O-Reilly Media, Sebastopol, USA.
- GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1997. *Design Patterns - Elements of Reusable Object-Oriented Software* 10th Ed. Addison-Wesley, Reading, Harlow et al.
- GUERRA, E., SOUZA, J., AND FERNANDES, C. 2009. A Pattern Language for Metadata-based Frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PloP)*. ACM.
- GÜNTHER, S. AND SUNKLE, S. 2009a. Enabling Feature-Oriented Programming in Ruby. Technical report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, Germany.
- GÜNTHER, S. AND SUNKLE, S. 2009b. Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, New York, 11–18.
- KICZALES, G., RIVIÈRES, J. D., AND BOBROW, D. G. 1995. *The Art of the Metaobject Protocol* 4th Ed. The MIT Press, Cambridge, London.
- OLSEN, R. 2007. *Design Patterns in Ruby*. Addison-Wesley, Upper Saddle River, Boston et al.
- PERROTTA, P. 2010. *Metaprogramming Ruby*. The Pragmatic Bookshelf, Raleigh.
- REENSKAUG, T. 1979. THING-MODEL-VIEW-EDITOR: An Example from a Planning system. Xerox PARC technical note. Available online <http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf>.
- THOMAS, D., FOWLER, C., AND HUNT, A. 2009. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, USA.
- VANDEVOORDE, D. AND JOSUTTIS, N. M. 2003. *C++ Templates: The Complete Guide*. Pearson Education, Boston.
- VON DINCKLAGE, D. 2003. Making Patterns Explicit with Metaprogramming. In *Proceedings of the 2nd international conference on Generative Programming and Component Engineering (GPCE)*. Springer-Verlag, New York, 287–306.