A Layer-based Approach to Hierarchical Dynamically-scoped Open Classes

MATTHIAS SPRINGER¹,a) HIDEHIKO MASUHARA¹,b) ROBERT HIRSCHFELD²,c)

Received: July 8, 2016, Accepted: October 26, 2016

Abstract: Open classes are frequently used in programming languages such as Ruby and Smalltalk to add or change methods of a class that is defined in the same component or in a different one. They are typically used for bug fixing, multi-dimensional separation of concerns, or to modularly add new operations to an existing class. However, they suffer from modularity issues if globally visible: Other components using the same classes are then affected by their modifications. This work presents Extension Classes, a hierarchical approach for dynamically scoping such modifications in Ruby, built on top of ideas from Context-oriented Programming (COP). Our mechanism organizes modifications in classes and allows programmers to define their scope according to a class nesting hierarchy and based on whether programmers regard an affected class as a black box or not. Moreover, Extension Classes support modularizing modifications as mixins, such that they can be reused in other components.

Keywords: Open Classes, Extension Classes, Context-oriented Programming, Mixins, Ruby

1. Introduction

Open classes are used for modifying existing classes: adding or modifying methods in an already existing class (target class) that is typically defined somewhere else in the same program or in an external library. The former case is a method addition and the latter one a method refinement.

There are a variety of use cases for method additions. The most common use case in mainstream programming languages such as Ruby is adding auxiliary methods in an object-oriented way. For example, the Ruby library ActiveSupport provides methods like Fixnum.minutes and Fixnum.hours to make it easy to perform time calculations such as 4.hours + 2.minutes. Another use case is multi-dimensional separation of concerns [26]: A class or group of classes may exhibit a number of different concerns which programmers may want to group together for understandability reasons. While a traditional object-oriented design allows only for a “single, dominant dimension of separation”, open classes can be used to group methods of a class family belonging to the same concern at a dedicated place.

Method refinements are typically used for bug fixing or implementing behavioral variations. Multiple behavioral variations can target the same classes and methods. In that case, there must be a way to specify which variation should be used and possibly combined. Context-oriented programming [15] (COP) is a mechanism for modularizing context-dependent behavioral variations. The mechanism presented in this paper is similar to context-oriented layer activation, focusing on scoping modifications dynamically. Other mechanisms have been proposed for Ruby and other languages, and will be addressed in the next section.

1.1 Background

This paper presents the design and implementation of Extension Classes for layer-based hierarchically-scoped open classes in the Ruby programming language. Ruby is a class-based object-oriented programming language and has support for class nesting and modules (mixins). Our design is amenable to other dynamically-typed languages supporting these features (e.g., Python, Scala, certain Smalltalk implementations [23]).

In Ruby, a class can be either a top-level class or be nested within another class. Nested classes are static members like constants (i.e., they are shared by all instances of a class) and their purpose is typically to serve their enclosing classes [8]. Mixins are units of code reuse, called modules in Ruby, and implemented as classes that are inserted into the (single inheritance) superclass hierarchy. In this implementation, they are the key to sharing modifications among multiple classes, but the mechanism itself is not specific to mixins and could also be implemented with traits [22] or other composition mechanisms.

1.2 Requirements

Fig. 1 illustrates the problem with open class modifications in its most basic form (example taken from Method...
Shells [25]). We would like to design two applications Viewer and Browser, which are both using the library WebPage. That library can render web pages and show popups. In Scenario (a), Viewer opens a web page and a popup must be shown. In Scenario (b), the Browser programmer overwrites the popup method with a no operation. When Browser opens a web page no popup must be shown. The main challenge is to ensure that Viewer and Browser can be used together and still function properly, i.e., Browser’s modification to WebPage do not affect Viewer, and Browser works even if the popup method is called indirectly.

Our goal is to develop an open classes mechanism supporting method additions and method refinements. That mechanism has the following properties, whose benefits will be described in more detail in Section 3 and then formally defined in Section 4. Properties 6–8 account for consistent mechanism has the following properties, whose benefits will be described in more detail in Section 3 and then formally defined in Section 4. Properties 6–8 account for consistent

modifications must typically be activated at some point, which is usually done with imports. Importer granularity and importee granularity denote the unit of scope and the unit of importable changes. A mechanism supporting units of fine granularity gives programmers precise control over what code will be affected by an import or over which changes should be imported, respectively. If a mechanism is dynamically scoped, modifications are applied even beyond (usually static) import statements, i.e., activation continues beyond method invocations (cf. local rebind). In this way, indirectly called methods can be adapted. Some mechanisms have a way to limit the extent of dynamic scoping, such that modifications do not propagate into arbitrary program parts. If changes are composable, there must be a way to access either an original implementation (in case of a method refinement) or another active modification that targets the same variation point. Modifications can be instantiable if they are encapsulated in a class-like structure. The following paragraphs highlight specialties of related techniques and compare them with Extension Classes.

Ruby has open classes, which allows programmers to open existing classes/modules to add new methods or to override them. Extensions for Ruby [10] are based on mixin or traits, class nesting, and inheritance (such as Ruby).

Our goal is to develop an open classes mechanism supporting method additions and method refinements. That mechanism has the following properties, whose benefits will be described in more detail in Section 3 and then formally defined in Section 4. Properties 6–8 account for consistent

modifications must typically be activated at some point, which is usually done with imports. Importer granularity and importee granularity denote the unit of scope and the unit of importable changes. A mechanism supporting units of fine granularity gives programmers precise control over what code will be affected by an import or over which changes should be imported, respectively. If a mechanism is dynamically scoped, modifications are applied even beyond (usually static) import statements, i.e., activation continues beyond method invocations (cf. local rebind). In this way, indirectly called methods can be adapted. Some mechanisms have a way to limit the extent of dynamic scoping, such that modifications do not propagate into arbitrary program parts. If changes are composable, there must be a way to access either an original implementation (in case of a method refinement) or another active modification that targets the same variation point. Modifications can be instantiable if they are encapsulated in a class-like structure. The following paragraphs highlight specialties of related techniques and compare them with Extension Classes.

Ruby has open classes, which allows programmers to open existing classes/modules to add new methods or to override them. Extensions for Ruby [10] are based on mixin or traits, class nesting, and inheritance (such as Ruby).

Our goal is to develop an open classes mechanism supporting method additions and method refinements. That mechanism has the following properties, whose benefits will be described in more detail in Section 3 and then formally defined in Section 4. Properties 6–8 account for consistent

modifications must typically be activated at some point, which is usually done with imports. Importer granularity and importee granularity denote the unit of scope and the unit of importable changes. A mechanism supporting units of fine granularity gives programmers precise control over what code will be affected by an import or over which changes should be imported, respectively. If a mechanism is dynamically scoped, modifications are applied even beyond (usually static) import statements, i.e., activation continues beyond method invocations (cf. local rebind). In this way, indirectly called methods can be adapted. Some mechanisms have a way to limit the extent of dynamic scoping, such that modifications do not propagate into arbitrary program parts. If changes are composable, there must be a way to access either an original implementation (in case of a method refinement) or another active modification that targets the same variation point. Modifications can be instantiable if they are encapsulated in a class-like structure. The following paragraphs highlight specialties of related techniques and compare them with Extension Classes.

Ruby has open classes, which allows programmers to open existing classes/modules to add new methods or to override them. Extensions for Ruby [10] are based on mixin or traits, class nesting, and inheritance (such as Ruby).
existing methods. It is a common pattern to alias a method before overwriting it, such that the original implementation is accessible under a different name [13]. Changes are globally visible, potentially leading to destructive modifications.

Refinements were introduced in Ruby 2.0 and allow programmers to limit the scope of open classes. Refinements can be activated at “top-level” or inside a class, and remain active for the remainder of the current file. They are not dynamically scoped: If a method in another file is called, the refinement is deactivated [28]. There is a discussion in the Ruby community as to whether refinements should be dynamically and hierarchically scoped [11]. Reasons against include implementation issues (performance), complexity of the lookup semantics, and unintuitive behavior of the `using` keyword, which is used to activate refinements [19]. Even though our approach adds some complexity to the lookup semantics as well, it automates parts of the activation process and does not require a `using` equivalent in many cases.

A classbox [6] is a container and namespace (package in the Java implementation [5]) for classes. Classes can be imported from other classboxes and changed locally; a classbox is the unit of scoping, whereas modifications are scoped by classes in our approach. Refining a class conceptually defines a new class, i.e., changes to imported classes are visible only in the extending classbox or in classboxes importing extended classes. Similar to our approach, including a class \( C \) from another classbox into a classbox \( B \) extends the scope of \( B \) onto \( C \) (local rebranding). An application can be represented by a classbox defining its own classes, importing external classes, and maybe extending them locally. In Ruby, an application is typically a single top-level class, whose modifications are visible in all nested classes with our approach. Classboxes are most similar to our approach, but lack hierarchical scoping and inheritance scoping, and they introduce an additional organizational unit for classes and modifications.

Local rebinding can lead to accidental method overwriting if two classboxes provide conflicting modifications. Method Shelters have been proposed as an extension to Ruby, combining features from Ruby refinements and Classboxes [1]. Modifications can be defined in a hidden chamber or in an exposed chamber. Modifications defined in an exposed chamber are subject to local rebinding, whereas modifications in a hidden chamber cannot be overridden locally and behave similar to lexical scoping of Ruby refinements. Our approach is closer to the Classboxes model and can currently not handle conflicting modifications as Method Shelters do. However, our approach lets programmers compose multiple (layered) modifications.

In plain layer-based context-oriented programming (COP) [15], partial methods can be used to encapsulate modifications in COP layers. Most COP implementations have pure dynamic scoping [3], i.e., if a layer is activated for a block (e.g., using the `with` method in ContextS [14]), then that method remains active until the execution of the block finished, unless programmers deactivate the layer explicitly. This can lead to destructive modifications if a method is called in a class that is not compatible with the modifications defined in an active layer (e.g., see full example in Section 3.1), because the extent of dynamic scoping is not limited. ContextJS allows a form of hierarchical scoping, where a layer can be activated for a morph (user interface element) and all of its submorphs [17]. COP frameworks with layer instances (e.g., JCop [4]) are the only mechanisms with instantiability in this comparison, but our approach could be extended accordingly.

<table>
<thead>
<tr>
<th>Base Language</th>
<th>Ruby</th>
<th>St., Java</th>
<th>(many)</th>
<th>Java</th>
<th>Java</th>
<th>Ruby</th>
<th>Java</th>
<th>Py.</th>
<th>Scala</th>
<th>St.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Additions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Method Refinements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Variable Additions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Locality of Changes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(who is affected?)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Importer Granularity</td>
<td>class</td>
<td>n/a</td>
<td>file</td>
<td>classbox</td>
<td>block, object</td>
<td>pkg,</td>
<td>meth.,</td>
<td>shelter</td>
<td>pkg, n/a</td>
<td>file</td>
</tr>
<tr>
<td>(unit of change)</td>
<td>class</td>
<td>n/a</td>
<td>refine-</td>
<td>class, classbox</td>
<td>layer</td>
<td>expander,</td>
<td>meth.,</td>
<td>shelter</td>
<td>meth., pkg, n/a</td>
<td>class</td>
</tr>
<tr>
<td>Dynamic Scoping</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Extent Limit</td>
<td>scope of class</td>
<td>n/a</td>
<td>n/a</td>
<td>scope of classbox</td>
<td>✓</td>
<td>n/a</td>
<td>link</td>
<td>hidden chamber</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 1 Comparison of Mechanisms and Implementations for Open Classes
MultiJava [10] and Expanders [29] support statically-scoped method additions. The scope of class additions is confined to the source code file where they were defined, unless they are imported. MultiJava and Expanders take into account class additions during type checking at compile time, making it possible to detect and prohibit destructive modifications. In contrast to our approach, there is no scoping mechanism extending modifications to collaborating classes, because method refinements are forbidden and method additions can only be referred to in a type-safe way if they were imported explicitly. MultiJava and Expanders support polymorphic overriding of methods (and accessing original methods). Overwriting arbitrary methods, even other method additions, is not allowed.

Method Shells [25] have been proposed as an open classes mechanism for Java. Classes and revisers (containers for modifications) are contained in a method shell. Modifications are visible only within the extending method shell or when the method shell is imported into another one. Classes from other method shells can also be imported with the `link` keyword, which will not include revisers in the current method shell and switch the context (active method shell) to the method shell of the included class when a method from that class is executing. Our approach cannot `link` other classes; however, the `scope` of a class controls deactivation of modifications, which is similar to “linking” and sufficient to implement the examples shown in that paper [25].

Python does not provide language support for open classes, but exposes the method dictionary of classes. A method addition or refinement can be defined by adding a method to that dictionary, but changes to the method dictionary are globally visible [7]. In Squeak/Smalltalk, a package can define methods for classes in a different package [18]. These methods are called extension methods and installed when the packaged is loaded. Existing methods are overwritten, making modifications globally visible. Scala supports implicit classes [20] which are syntactical sugar for implicit type conversions [21]. Such a conversion is attempted automatically if a non-existing method is called. Consequently, implicit classes cannot be used for method refinements.

### Previous Work

This work is based on our previous work [23] on class extensions for the Matriona module system. Matriona is a module system for Squeak/Smalltalk [24] that brings class nesting and class parameterization to Smalltalk, inspired by concepts from Newspeak and BETA. The present work simplifies the previous class extensions mechanism and presents it in a distilled form, leaving away Matriona-specific details.

In Matriona, the (method) lookup has an additional level of complexity, because nested classes are virtual and can be overridden. The semantics of `super` calls (`proceed` in this work) in particular are complex, because superclasses are also virtual and cannot be determined statically, making it hard for programmers to predict/control the `scope` of a class at development time. Furthermore, mixins are effectively represented as methods with a superclass parameter and returning a class object, making reusability of class extensions harder to grasp for programmers that are not familiar with mixin theory. In contrast, Ruby nested classes and superclasses are not virtual and there is a dedicated language construct for mixins (`modules`).

### 3. Extension Classes by Example

In this section, we present three examples to justify the requirements and give an overview of the Extension Classes mechanism that we will formally define afterwards.

#### 3.1 Dynamic Scoping

The first example [25] was already briefly mentioned in Section 1.2 and illustrates the most basic use case. We would like to develop an embedded web browser and an audited viewer for web pages, represented by classes `Browser` and `Viewer`, respectively. Both applications use the same `WebPage` library, containing functionality for rendering websites and for showing popups (Fig. 2).

```ruby
1 class WebPage
2   def open(url)
3     if popup_requested
4       popup(text)
5       end
6     end
7   end
8 def popup(text); ""Show popup dialog""; end
9 end
```

Fig. 2 Definition of Class `WebPage`. Method `popup` may be called internally to display a popup window.

Since the browser application was designed for an embedded system, it should not display popup windows, whereas the audited viewer application should show a popup window whenever a confidential file is accessed (Fig. 5). Showing popup windows is an essential part of the functionality of the audited viewer application.

In this design, the embedded web browser defines a method refinement for `WebPage.popup` to disable popup windows, i.e., replacing it by a `no operation` (Fig. 4). Our mechanism should allow for the following behavior.

First, when another application uses both the embedded web browser and the audited viewer, the viewer should still show popup windows. In other words, the `scope` of the browser’s modifications should be confined to `Browser`.

Second, in another design, where the embedded web browser uses the audited viewer, programmers should be able to choose whether the browser’s modifications should affect the audited viewer or not. In other words, programmers should be able to specify whether the scope of the browser’s modifications includes `Viewer` or not.

### Representation of Modifications and Notation

In our Ruby implementation, class modifications are defined as members of classes[^2]. Following COP terms and

[^2]: They can also be defined as members of modules (mixins). Whenever we say `class`, we also refer to modules.
class Viewer
  def check(file)
    page = WebPage.new
    if file.is_confidential?
      page.popup("<b>confidential</b>")
    end
  end
end

![Fig. 3](image-url) Definition of Class Viewer. This class uses WebPage internally and might trigger a window to pop up.

```ruby
class Viewer  
  def check(file)  
    page = WebPage.new  
    if file.is_confidential?  
      page.popup("<b>confidential</b>")  
    end  
  end  
end
```

Fig. 4 Example: Locality of Changes. Gray boxes indicate classes and their extensions, sets enclosed in curly braces indicate the scope of a class. Classes enclosed in parentheses account for hierarchical scoping.

```ruby
class Editor
  def check(file)
    if file.is_confidential?
      page.popup("<b>confidential</b>")
    end
  end
end
```

Fig. 5 Definition of Class Editor. The class defines a partial class targeting the top-level class WebPage, overwriting method popup.

This statement works similar to `public`, `private`, and `protected`.

```
C's methods is executed. For example, if `Browser.open` is executed (invoked), class `Browser` is activated.

Class deactivation depends on the **scope** of a class and takes place upon method invocation. The scope of a class is a set of classes and determines how long a class should remain activated if activated before, i.e., the scope of a class affects only deactivation but not activation: If a class C is active and the control flow dispatches to a method D.

```ruby
scope(Browser) = \{Browser\} \cup \{WebPage\}
```

Every method invocation can activate and deactivate classes. Our implementation maintains a class activation set containing all activated classes. When a method call returns, the set of activated classes is restored to the original state right before that method call. Partial classes give programmers control over the scope of a class and thus its deactivation. Adding another class D to the scope is simple: Define a partial class targeting D. It does not have to contain any partial methods.

Until now, method dispatch is simple: Before invoking a method C.

```ruby
scope(Browser) = \{Browser\} \cup \{WebPage\}
```

Full Example

In the following examples, the embedded browser and the audited viewer are used together, in order to illustrate deactivation of modifications (classes). Class Application (Fig. 6) is a component that uses both Browser and Viewer. When invoking method run, there are no classes activated other than Application. The browser application will not generate popup windows, whereas the audited viewer application will, because Browser is deactivated when Browser.open returns to Application.main. The reason for that is that the original class activation set is restored when method open returns.

Consider a slightly different case now where Browser uses Viewer internally, while both of them still use WebPage for rendering purposes (Fig. 7). If Browser calls WebPage directly, no popups will be shown. However, once Browser calls a method in Viewer, class Browser is deactivated, be-


3.2 Hierarchical Scoping

In this section, we extend Extension Classes to handle nested classes. Consider a networking library that consists of a module Networking where functionality such as sockets and DNS name resolution is organized in a class nesting structure within Networking (Fig. 9). Network endpoints are represented by instances of class Address, which is nested inside Networking. In this design, the networking library defines a method addition String.to_address to make it easy to convert string representations of DNS names and IP addresses to instances of Networking.Address (Fig. 10).

The open classes mechanism should ensure that the method addition String.to_address is visible in the entire networking library, i.e., also inside the nested module Networking. In other words, the scope of Networking should contain all nested classes of Networking. Moreover, in accordance with the mechanism described in the previous section, its modifications should not be visible in other applications. For example, consider that an address book application contains a class Address representing mail addresses along with a converter method String.to_address. The scope of each class should be confined to itself and its nested classes, but not spill over to other classes in different parts of the class nesting tree.

Scope of Modifications

We extend the notion of the scope of a class such that it also includes the scope of all nested classes of the class. Furthermore, we activate a class not only if one of its methods is executing, but also if a method contained in one of its nested classes is executing. For example, when calling a method in Pinging, the classes Pinging, Networking, and Object are activated. The module Networking remains active even inside Pinging, because Pinging ∈ scope(Networking). Consequently, Pinging uses the to_address method addition defined as part of Networking.

---

1. class Application
2. def main
3. Browser.new.open("http://...")
4. Viewer.new.check("secret.html")
5. end
6. end

Fig. 6 Definition of Class Application. This class contains the entry point in a scenario where both Browser and Viewer are used together.

1. class Application
2. def main
3. Browser.new.open("http://...")
4. end
5. end
6. class Browser
7. def open(url)
8. Viewer.new.open(url)
9. Viewer.new.check("secret.html")
10. end
11. end
12. # Same partial classes as in example above
13. end

Fig. 7 Variation of Application Scenario. Browser overwrites popup and uses Viewer internally.

1. module Networking
2. class Address; end
3. module Pinging
4. def self.ping(address)
5. # address might be a String
6. addr = address.to_address
7. # ...
8. end
9. end
10. end
11. end
12. partial
13. class ::String
14. def to_address
15. # Convert String to Address
16. # ... (partial)
17. end
18. end
19. end

Fig. 9 Definition of Module Networking. Behavioral variations and additions defined for String should be visible in all nested classes/modules of Networking.

1. module Networking
2. class Address; end
3. module Pinging
4. def self.ping(address)
5. # address might be a String
6. addr = address.to_address
7. # ...
8. end
9. end
10. end
11. end
12. partial
13. class ::String
14. def to_address
15. # Convert String to Address
16. # ... (partial)
17. end
18. end
19. end

Fig. 10 Example: Duplicate Auxiliary Methods. Method refinements for String.to_address are only active in their respective owner classes and their nested classes.
Object can define class additions String.to previous section, both classes AddressBook and Networking can define class additions String.to_address and work side by side by side, because modifications defined in AddressBook are not active in Networking and modifications defined in Networking are not active in AddressBook. The reason for that is that Networking $\not\in$ scope(AddressBook) and AddressBook $\not\in$ scope(Networking).

3.3 Importing Extension Classes

In this section, we extend Extension Classes to allow for importing. Consider a library representing abstract syntax trees (AST) that defines a tree-based data structure of nodes, along with a number of operations (e.g., printing and various evaluation strategies). Every operation provides a method per node and optionally additional helper methods. In this design, every operation is represented as a set of method additions for AST node classes. Notice how the concerns evaluating and printing are grouped in their own separate classes (Fig. 11, 12). In terms of multi-dimensional separation of concerns, the AST node type is the dominant decomposition dimension and the other two concerns are encapsulated in an additional dimension for operations.

The open classes mechanism should allow programmers to choose an evaluation strategy for their application, or possibly combine multiple strategies. For example, Evaluating is the default evaluation strategy, but programmers might want to use Mod10Evaluating which is built on top of the default evaluation strategy and takes the result modulo 10. In other words, programmers should be able to import modifications into their applications.

Modularizing Modifications with Mixins

According to the previously mentioned class activation rule, an evaluation strategy like Evaluating is activated only if a method in Evaluating is called. In this example, all operations are defined within Ruby modules which can be included in (multiple) classes. Ruby modules are implemented as mixins [9] and effectively insert a copy of the module in the superclass hierarchy upon inclusion (application). From a technical point of view, it does not matter if a partial method was defined in a superclass or was included from a module, which is why this example is listed under inheritance scoping.

In the example in Fig. 13, Printing and Evaluating trees are encapsulated in modules Printing and Evaluating, which provide method additions for all AST node classes.

---

**Fig. 12** Definition of Module AST. Functionality for printing and evaluating trees is encapsulated in modules Printing and Evaluating, which provide method additions for all AST node classes.

```ruby
module AST
  module Nodes
    class Node; end
    class IntNode < Node; end
    class PlusNode < Node; end
  end
  class Application
    include AST::Evaluating
    def evaluate
      return value
    end
  end
  module Evaluating
    class Nodes::Evaluating
      def evaluate
        return left.evaluate + right.evaluate
      end
    end
  end
  module Printing
    include AST::Printing
    module Application
      def main
        AST::Nodes::IntNode.new(42).evaluate
      end
    end
  end
end
```

**Fig. 13** Definition of Class Application. To use the tree printer or evaluator, the respective module has to be included.

```ruby
class Application
  include AST::Printing
  include AST::Evaluating
  def main
    AST::Nodes::IntNode.new(42).evaluate
  end
end
```

---

Fig. 11 Example: AST Library. Classes enclosed in parentheses in scope sets account for inheritance scoping.
application of mixin `Evaluating`, whose scope includes all AST node classes (for presentation purposes, the following formulas do not account for `Printing`).

\[
\text{scope}(\text{Evaluating}()) = \{\text{Evaluating()}, \text{Node}, \text{IntNode}, \text{PlusNode}\}
\]

\[
\text{scope}(\text{App.}) = \text{scope}(\text{Evaluating}()) \cup \{\text{Application}\}
\]

\[
= \{\text{Application}, \text{Evaluating()}, \text{Node}, \text{IntNode}, \text{PlusNode}\}
\]

The only class being activated is `Application` but not its superclass. However, the method lookup does not only take into account partial classes defined in an activated class `L`, but also partial classes defined in its superclass `L'`, starting with `L` and then `L'` (see Section 4.4). The `proceed` expression\(^*\) can be used in a partial method of `L` to call a partial method defined in `L'` or one of its superclasses (similar to `proceed` in COP and `super`).

Consequently, when executing a method in `Application`, modifications from `Evaluating` are active and remain active as long as methods from `Application` or any AST node class are executed.

**Layered Modifications**

We now want to modify our AST evaluator in such a way that all results and partial results are calculated modulo 10. For that reason, we define a set of class extensions `Mod10Evaluating` that runs on top of `Evaluating`, i.e., whenever the mixin `Mod10Evaluating` is applied, the mixin `Evaluating` is applied first, automatically (Fig. 14, Design INCLUDE). Technically, `Mod10Evaluating` is a module that first includes module `Evaluating` before defining its own partial methods; standard Ruby allows `include` statements inside modules, i.e., this is not a new language feature.

```ruby
module AST
  module Mod10Evaluating
    include Evaluating
    partial
    class Nodes::IntNode
      def evaluate
        return proceed % 10
      end
    end
    class Nodes::PlusNode
      def evaluate
        return proceed % 10
      end
    end
  end
end
```

![Fig. 14 Definition of Mod10Evaluating](image)

From now on, we use a stack (instead of a set) as the data structure for maintaining activated classes. Consequently, we call that data structure the `class activation stack`, in adherence to the layer activation stack in context-oriented programming. A stack can capture the ordering of class activations, such that `proceed` can also be used to dispatch to a method in the next activated class after an exhaustive search in the previous class (the class on top of the `next activated class` in the class activation stack), similarly to method dispatch in context-oriented programming.

To continue with the previous example, note that a mixin application of `Mod10Evaluating` contains partial methods for `IntNode.evaluate` and `PlusNode.evaluate`. Since mixin `Evaluating` is applied first (during mixin application of `Mod10Evaluating`) and mixin `Mod10Evaluating` is applied second, method execution will start with methods from the latter one. The reason is that an application of `Mod10Evaluating` is on the top of the class activation stack. The `proceed` expression in the `evaluate` methods executes the `evaluate` implementations defined in `Evaluating`.

In theory, `Mod10Evaluating` could also be defined as a subclass of `Evaluating` (Design SUBCLASS), resulting in the same behavior in this example (see Section 4.4). This is possible because `proceed` calls are used for both calling a `super` method (i.e., method defined in a superclass) and calling the next partial method encapsulated in a partial class on the class activation stack. However, due to language restrictions in Ruby, modules cannot be subclassed and classes cannot not be included, confining such a design to activation by local rebinding (see Section 4.5); i.e., behavioral variations cannot be shared with mixins.

**4. Formal Concept**

Extension Classes use a variant of context-oriented programming [15] (COP) to support open classes. Every class can not only contain variables, methods, and nested classes, but also `partial classes`, i.e., every class with its partial classes can act as a layer. Conceptually, a partial class is a special form of a nested class which does not define a new class via subclassing but extends a specific existing `target class`. Every partial class can contain a number of partial methods. Such a method can be a method addition or a method refinement.

**4.1 Rationale**

Method additions and method refinements can be beneficial for their defining classes, but they can break other classes or libraries if they are global. Programmers typically treat external libraries as black boxes [25], i.e., it is hard to anticipate the effect of modifications. Therefore, Extension Classes should allow programmers to define the scope of modifications, i.e., where they are active. As a rule of thumb, we propose that a class `L` with its modifications should be deactivated if the control flow is passed to another class or library that the programmer of `L` regards as a black box. Defining a partial class (container for modifi-
cations) for a target class \(C\) within class \(L\) means that \(C\) is no longer regarded as a black box from \(L\)’s point of view.

Class nesting can be used as an orthogonal scoping mechanism to indicate that modifications defined in a class should also be active for all of its nested classes. If a class defines modifications for \(C\), then \(C\) is not being regarded as a black box from the perspective of any nested class (i.e., nested classes are aware of the behavior and modifications of their enclosing classes).

### 4.2 Class Activation

Our Ruby implementation maintains a global stack of activated classes (class activation stack), which is similar to a layer activation stack in context-oriented programming. Modifications defined as part of a class \(L\) are taken into account during method lookup only if \(L\) is on the class activation stack. The exact lookup semantics are described in Section 4.4. Classes are activated and deactivated before performing a method call (but after performing the method lookup) and when a method call returns according to the following activation rule.

**Activation Rule.** Before dispatching to a method \(C\).\text{method}, make a copy \(S’\) of the current class activation stack and perform the following operations.

1. For all active classes \(L\) on the class activation stack, if \(C \notin \text{scope}(L)\), deactivate (remove) \(L\).
2. Push \(C\) and its enclosing classes onto the layer (class) composition stack (start with outermost class).

Once the method call returned, restore the original class activation stack \(S’\).

Note that classes are never activated multiple times. If an already activated class is activated once more, it will be moved to the top of the activation stack.

According to the activation rule, enclosing classes are activated but not superclasses. They are accounted for in the method lookup (Section 4.4). It is debatable whether enclosing classes of superclasses should be activated or not — our approach does not activate them, because it makes the method lookup significantly more complex and difficult to predict. It is also closer to Ruby semantics, which does not take into account names defined in the superclass’s enclosing class during constant lookup.

### 4.3 Scope of a Class

The scope of a class determines if a class remains activated during a method call. It does not affect method activation. A class remains active only as long as the control flow stays within the class’s scope, as specified by the activation rule.

**Definition.** The scope of a class \(L\) is defined as the set containing \(L\), all target classes (and their reachable nested classes\(^6\)) corresponding to partial classes of \(L\), all classes in the scope of all nested classes of \(L\)\(^7\), and all classes in the scope of the superclass of \(L\) (if super\((L) \neq \text{Object}\)).

\[
\text{scope}(L) = \{L\} \quad \text{(reflexivity)}
\]
\[
\cup \{C \mid C \in \text{nested}^*(\text{target}(P)) \land P \in \text{partials}(L)\} \quad \text{(dynamic scoping / local rebinding)}
\]
\[
\cup \{C \mid C \in \text{scope}(N) \land N \in \text{nested}(L)\} \quad \text{(hierarchical scoping)}
\]
\[
\cup \text{scope}(\text{superclass}(L)) \quad \text{(inheritance scoping)}
\]

In this definition, reflexivity accounts for direct method calls. Dynamic scoping/local rebinding accounts for indirect method calls. In the absence of hierarchical scoping, the second line could be simplified as follows.

\[
\cup \{\text{target}(P) \mid P \in \text{partials}(L)\}
\]

**Dynamic scoping is not transitive:** The scope of a class includes all target classes, but not the scope of all target classes. This is an intentional decision as programmers writing modifications for a class \(C\) would otherwise have to be aware of all modifications of \(C\) itself.

Note that modifications remain active even if the control flow changes from one target class \(A\) to another one \(B\). The rationale is that programmers do not regard \(A\) and \(B\) as black boxes, since they are changing both of their behavior. Consequently, programmers should also be aware of the interaction between \(A\) and \(B\).

It is interesting to see that the activation rule cannot be replicated using an inverted scope function without changing its semantics. One reason for that is that the scope of every class is a static property, whereas class activation is dynamic.

### 4.4 Method Lookup

Whenever a method is called on an object, our approach first determines the receiver’s class \(C\). Instead of using \(C\)’s superclass hierarchy for the method lookup, its effective superclass hierarchy is used, which is a combination of \(C\)’s superclass hierarchy and partial classes for \(C\) and its superclasses defined as part of classes on the layer composition stack.

**Effective Superclass Hierarchy**

Extension Classes use the \textit{proceed} expression for both calling overridden methods defined higher in the superclass hierarchy and for calling partial methods defined in a class lower on the class activation stack. From a method lookup point of view, the actual superclass hierarchy and partial classes defined in classes on the class activation stack are combined into an effective superclass hierarchy. The rule for merging both hierarchies is simple: For every class \(C\) in the actual superclass hierarchy, first look up methods in partial classes of \(C\) on the layer composition stack, then

\(^6\) nested\(^*(C) = \{C\} \cup \{D \mid D \in \text{nested}^*(N) \land N \in \text{nested}(C)\}\), i.e., \(C\) and all nested classes of \(C\) and their nested classes etc.

\(^7\) Therefore, \text{nested}^*(L) \subseteq \text{scope}(L)\).
look up methods in C. Partial classes are conceptually sub-classes which are applied dynamically depending on the current class composition.

Definition. The effective superclass hierarchy of a class C is defined as Effective(C), where S is the class composition stack (S[1] is top of stack), \#C is the number of superclasses of a class C, super^i(C) is the i-th superclass of class C, L[C] is the partial class targeting C defined in L (if there is one), \{\} brackets denote a (ordered) list, and summation is used for list concatenation.

\[
\begin{align*}
\text{LayerHier}(L, C) &= \sum_{i=0}^{#L} (\text{super}^i(L)[C]) \\
\text{ClassLayers}(C) &= \left( \sum_{i=1}^{S} \text{LayerHier}(S[i], C) \right) + (C) \\
\text{Effective}(C) &= \sum_{i=0}^{#C} \text{ClassLayers}(\text{super}^i(C)) 
\end{align*}
\]

LayerHier(L, C) is the list of partial classes for class C defined in class L and its superclasses. ClassLayers(C) is the list of partial classes of C (among all activated classes) and C itself.

Fig. 15 illustrates the effective superclass hierarchy in an example. Let us assume that the layer composition stack contains classes A', B' (which is on top), classes A', B, and B' have partial classes for C' and class A has a partial class for its superclass C. Consequently, the effective superclass hierarchy of C is defined as follows.

\[
\begin{align*}
\text{LayerHier}(B', C') &= (B', C', B, C') \\
\text{LayerHier}(A', C') &= (A', C') \\
\text{LayerHier}(C, A) &= (A, C) \\
\text{ClassLayers}(C') &= (B', C', B, C', A', C', C') \\
\text{ClassLayers}(C) &= (A, C, C) \\
\text{Effective}(C) &= (B', C', B, C', A', C', C', A, C, C)
\end{align*}
\]

As another example, Fig. 16 shows the effective superclass hierarchy for class IntNode in the scenario from Fig. 13. If the mixin Mod10Evaluating is applied instead of Evaluating, that hierarchy is prepended with the corresponding partial class for IntNode defined in module Mod10Evaluating (Design INCLUDE). The resulting hierarchy is identical to the hierarchy in an alternative Design SUBCLASS where Mod10Evaluating is a subclass of

\[
\text{Effective(\text{IntNode})} = \text{ClassLayers(\text{IntNode})} + \text{ClassLayers(\text{Node})} + \text{ClassLayers(\text{Object})}
\]

In Design INCLUDE, ClassLayers(IntNode) expands to three summation terms and ultimately results in the superclass hierarchy described above.


In Design SUBCLASS, LayerHier(Mod10Ev, IntNode) expands to both Mod10Ev.IntNode and Eval.IntNode, and ultimately results in the same superclass hierarchy.


4.5 Class Activation

If modifications defined in class L should be active when executing a method defined in class A, one of the following two designs can be applied.

Activation by Local Rebinding

In this design, programmers have to ensure that the control flow reaches A via a sequence of methods defined in classes \(L \rightarrow C_1 \rightarrow \ldots \rightarrow C_n \rightarrow A\) with \(C_i \in \text{scope}(L)\)

\(*\) For presentation reasons, we abbreviate Evaluating with Eval and Printing with Print.
(for all $i = 1 \ldots n$) and $A \in \text{scope}(L)$. The control flow may also originate from a subclass of $L$ or a nested class of $L$. Programmers can enforce that a class $C_i \in \text{scope}(L)$ by adding a partial class targeting $C_i$ to $L$ (partial classes can be empty). For example, in Section 3.1, $n = 1$, $L = \text{Browser}$, $C_1 = \text{WebPage}$, and $A = \text{Name}$. WebPage when calling WebPage, pop up via WebPage.open from Browser.open.

**Mixin-based Activation**

The previous design is hard to accomplish if modifications should be shared among a variety of classes. The following design encapsulates modifications in mixins. A mixin is an abstract subclass that can be applied to a number of superclasses. When partial classes are defined inside a mixin $M$ that is applied to a superclass $C$, all of $M$ ’s modifications are active when the control flow passes through a method in the context of the resulting class $C'$ (i.e., the polymorphic receiver class is $C'$).

Consider Fig. 16 as an example. Evaluating() and Printing() are mixin applications with partial classes for IntNode. Class Application is defined as a subclass of the application of both mixins. When a method is executed in the context of Application, then that class is pushed onto the composition stack. The effective superclass hierarchy of class IntNode contains the partial classes of both mixins.

### 5. Implementation

This section gives an overview of our prototypical Ruby implementation of Extension Classes. Currently, performance is explicitly not a goal. Instead, our prototype is geared towards language design experiments. It is implemented using metaprogramming, reflection, and Ruby libraries providing access to low-level interpreter functionality. Thus, our implementation supports only MRI (Ruby’s reference implementation) at this time.

**Defining Partial Methods**

Partial methods must be defined inside partial classes with a preceding partial statement. partial is an instance method defined on Module, similar to public, private, protected, and tells our implementation that the following method definitions are partial methods. Partial classes reuse the syntax of open classes: In Ruby, any class can be opened even if that code is nested inside another class, as long the fully qualified name of the class is used\(^9\). Newly defined methods are aliased and replaced with a wrapper method that performs our customized method lookup. To determine if a method is a partial method, our implementation uses a Ruby library (implementing the Debug Inspector API as a C extension) to see if the previous stack frame belongs to a method whose class has the partial flag set.

**Layer Activation and Method Dispatch**

Wrapper methods are responsible for activating and deactivating classes, as well as for dispatching to the correct partial method or base method by scanning the class composition stack. Our implementation defines a new instance method proceed on class Object which determines the next partial or base method to be executed. This method uses our customized method lookup.

Method lookup must be performed in wrapper methods and in the implementation of proceed. Our implementation attaches a state object to every wrapper method invocation (stack frame). proceed traverses the stack and searches for the closest state object. That object stores information about the method lookup and contains all information necessary to find the next method to be invoked: the (runtime) class of the receiver, the current subclass of the receiver in the method lookup, the (runtime) class of the current layer, the current subclass of the current layer in the method lookup. Effectively, iterating from one state to the next state traverses the effective superclass hierarchy. For the moment, we assume that the class composition stack remains constant during proceed calls, which is why we can easily determine the next layer from the layer that is currently being processed in the method lookup.

### 6. Discussion

The motivation for supporting open classes in a programming language is twofold: First, method additions can promote modular understandability through multi-dimensional separation of concerns. Second, method additions and refinements provide an easy way to implement behavioral variations and to add new operations to an existing class. At the same time, modifications should be confined to a local scope (e.g., the scope of a certain component) to avoid the problem of destructive modifications.

**Multi-dimensional Separation of Concerns**

Multi-dimensional separation of concerns is a concept for factoring out otherwise scattered concerns in a family of classes, which can result in improved comprehension, reduction of complexity and better code reuse, among other benefits [27]. Two well-known techniques to achieve this are hypermodules [26] in HyperJ and inter-type declarations in AspectJ [16]. The basic idea is to define a set of classes according to one “dominant” dimension and to “encapsulate concerns in dimensions other than the dominant one” [27].

Method additions are another technique to achieve multi-dimensional separation of concerns and discussed in this paper. Consider, for example, an AST library where each node implements methods for the two concerns of evaluating and printing. A class Evaluating can define method additions

\[^9\] For that reason, we have to write ::WebPage or Object::WebPage instead of just WebPage in Section 3.1.
New Operations and Behavioral Variations

Adding new methods to an existing class is difficult without method additions. One approach is to create a subclass and perform changes in the subclass, but this approach fails if the programmer is not in control of instance creation. Another approach for tree-based data structures is to use the Visitor design pattern [12], but this approach results in more overhead and infrastructural complexity due to additional classes and double dispatch. A common approach in Ruby is open classes: New methods can be added to an existing class at any time, but the method addition can be destructive, i.e., it can overwrite existing methods.

It is sometimes necessary to change the behavior of an existing method in order to extend the functionality of the method or to simply replace it with one that behaves differently. This is necessary, if the developer of that class did not anticipate the change and provide a suitable interface. A common use case in Ruby is bugfixing: If a method is buggy, a method refinement can replace that method with a proper implementation. Open classes in Ruby can be used to replace buggy methods (known as monkey patching), but the method refinement will be globally visible and can be destructive. For example, a different component in the system might depend on the buggy behavior and work around it by itself. As illustrated by the examples in Section 3, Extension Classes can handle all of these cases properly.

7. Summary and Future Work

We proposed Extension Classes, a hierarchical and layer-based approach for organizing method additions and method refinements in Ruby, as an alternative to open classes. This approach is similar to context-oriented programming, but class (layer) (de)activation is performed implicitly. A class in our system may contain partial methods that extend (modify) other classes (thus the name Extension Classes) and can be compared to a classbox or a method shell, but it is scoped hierarchically and does not require additional syntactical elements except for the definition of partial classes. Scoping and locality of changes are important to avoid destructive modifications. Our approach lets programmers control the scope of modifications by specifying whether a class should be regarded as a black box or not.

We also showed how our mechanism can be used for multidimensional separation of concerns: Every concern is encapsulated in a mixin and can be activated during class definition. The method lookup is guided by the conceptual model of an effective superclass hierarchy.

Future work might focus on a more formal definition of the semantics of the lookup mechanism and consider performance optimizations. Our current implementation approach is based on a reimplemention of the method lookup using metaprogramming. Two particular problems are implicit class (de)activation, which takes place not only when a method is executed but also when the method returns, and proceed calls: Both are expensive in our implementation, but performance is explicitly not a goal at this time. Future versions might contain optimizations like partial method inlining or layer (class) composition caching to achieve better performance.

This work focuses on dynamically-typed programming languages, but the main ideas could be applied to statically-typed languages if the type system is aware of method additions and method refinements. One particular problem is that method additions can only be referred to in a type-safe way if the type system can prove that at least one class providing a suitable method is always available at runtime [2]. Future work might evaluate such an approach in the context of Extension Classes.

References


© 2017 Information Processing Society of Japan


Matthias Springer is a Ph.D. student at the Department of Mathematical and Computing Science, Tokyo Institute of Technology. He received his B.S. and M.S. degrees from the Hasso Plattner Institute at the University of Potsdam in 2013 and 2015, and was a visiting student at UC San Diego in 2013/14. His research interests are the design and implementation of object-oriented programming languages, GPGPU computing, and modularity.

Hidehiko Masuhara is a Professor at the Department of Mathematical and Computing Science, Tokyo Institute of Technology. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1992, 1994 and 1999, respectively. Before joining Tokyo Institute of Technology, he served as an Assistant Professor, Lecturer, and Associate Professor at Graduate School of Arts and Sciences, the University of Tokyo. His research interests include design and implementation of programming languages and software development environments.

Robert Hirschfeld is a Professor at the Hasso Plattner Institute at the University of Potsdam, Germany. He is interested in improving the comprehension and design of software systems. Robert enjoys explorative programming in interactive environments. He served as a visiting professor at the Tokyo Institute of Technology and The University of Tokyo, Japan. Robert was a senior researcher with DoCoMo Euro-Labs, the European research facility of NTT DoCoMo Japan. Prior to joining DoCoMo Euro-Labs, he was a principal engineer at Windward Solutions in Sunnyvale, California. Robert studied engineering cybernetics and computer science at Ilmenau University of Technology, Germany (see also http://hpi.de/swa/).