Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

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

Matthias Springer[1,a)] Hidehiko Masuhara[1,b)] Robert Hirschfeld[2,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] Department of Mathematical and Computing Science, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] Hasso Plattner Institute, University of Potsdam, Germany
[a)] matthias.springer@acm.org
[b)] masuhara@acm.org
[c)] hirschfeld@hpi.uni-potsdam.de

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

Figure 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*.

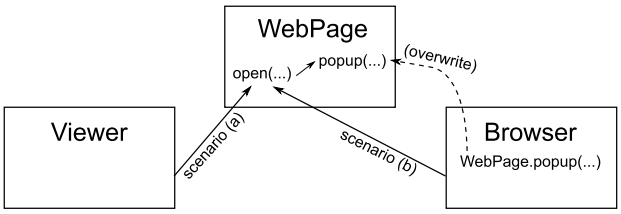

Fig. 1 Example: Simplified problem description.

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 language integration in programming languages with mixins/traits, class nesting, and inheritance (such as Ruby).

(1) *Only Classes:* No additional organizational units (such as classboxes, method shells, etc.) are necessary. Modifications are part of other classes or modules.
(2) *Locality of Changes:* Modifications are visible in a local scope (i.e., not global). Components outside of that scope (e.g., Viewer in Fig. 1) are not affected by them.
(3) *Implicit Activation:* Modifications are automatically activated for source code that is defined in the same class as the modifications (e.g., `popup` for Browser; *reflexivity*).
(4) *Dynamic Scoping:* Once activated, modifications remain active even if a method in a different class is called. This allows for indirect method calls (cf. *local rebinding*). For example, in Fig. 1, no popup must be shown even if `popup` is called from `open` (which is called from Browser).
(5) *No Destructive Changes:* Modifications are deactivated if a method is called in a class that is not known to be compatible with them (limiting the dynamic extent). Multiple activated modifications targeting the same method (i.e., *overwriting* each other) are also potentially destructive.
(6) *Reusability:* Modifications can be shared among multiple classes, in all of which they are activated implicitly.
(7) *Hierarchical Scoping:* Modifications defined in an enclosing class should also implicitly be activated for inner (nested) classes.
(8) *Inheritance Scoping:* Modifications defined in a superclass should also implicitly be activated for subclasses.
(9) *Composability:* Multiple modifications targeting the same method can be combined, which can also be used to resolve destructive modifications.

1.3 Outline and Contributions

The remainder of this paper is structured as follows. Section 2 will compare Extension Classes to related work. Section 3 presents motivating examples and illustrates the requirements. Section 4 describes Extension Classes formally and Section 5 gives a brief overview of our Ruby implementation. Finally, Section 7 gives a summary and ideas for future work. This paper makes the following main contributions.

- An open classes mechanism with local visibility supporting dynamic scoping and scoping according to class nesting hierarchy and inheritance hierarchy.
- A prototypical implementation in Ruby [*1].

2. Related Work

There are a variety of related technqiues and implementations for open classes (**Table 1**). Some programming languages support them out of the box, while others rely on external libraries. Open classes mechanisms with *locality of changes* are mechanisms with a scoping technique, i.e., changes are not necessarily globally visible. In that case, 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 rebinding*). 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 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

[*1] git@github.com:prg-titech/ruby-extension-classes.git

Table 1 Comparison of mechanisms and implementations for open classes.

	Extension Classes (our approach)	Ruby Open Classes	Ruby Refinements	Classboxes	Plain Layer-based COP	Expanders	Method Shells	Method Shelters	MultiJava	Python Class Opening	Scala Implicit Classes	Smalltalk Ext. Methods
Base Language	Ruby			St., Java	(many)	Java	Java	Ruby	Java	Py.	Scala	St.
Method Additions	✓	✓	✓	✓	(✓)	✓	✓	✓	✓	✓	✓	✓
Method Refinements	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓
Variable Additions	(✓)	(✓)	(✓)	✓	(✓)	✓	✗	(✓)	✗	✓	(✗)	✗
Locality of Changes	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗
Importer Granularity *(who is affected?)*	class	n/a	file	classbox	block, object	pkg.	meth. shell	meth. shelter	pkg.	n/a	file	n/a
Importee Granularity *(unit of change)*	class	n/a	refine-ment	class, classbox	layer	ex-pander	meth. shell	meth. shelter	meth., pkg.	n/a	class	n/a
Dynamic Scoping	✓	n/a	✗	✓	✓	✗	✓	✓	✗	n/a	✗	n/a
Dynamic Extent Limit	scope of class	n/a	n/a	scope of classbox	✗	n/a	`link`	hidden chamber	n/a	n/a	n/a	n/a
Composability	✓	(✗)	✓	✓	✓	(✗)	✗	✗	(✗)	✗	n/a	✗
Instantiability	(✗)	✗	✗	✗	(✓)	✗	✗	✗	✗	✗	✗	✗

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 rebinding*). 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 accidential 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 instantiablity in this comparison, but our approach could be extended accordingly.

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**).

Since the browser application was designed for an embedded system, it should not display popup windows (**Fig. 5**), whereas the audited viewer application should show a popup window whenever a confidential file is accessed (**Fig. 3**). 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.

```
class WebPage
  def open(url)
    if popup_requested
      popup(text)
    end
  end

  def popup(text); """Show popup dialog"""; end
end
```

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

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

Fig. 3 Definition of class `Viewer`. This class uses `WebPage` internally and might trigger a window to pop up.

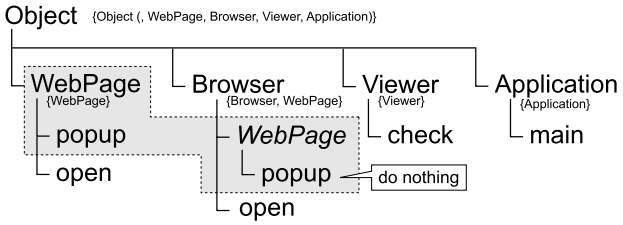

Fig. 4 Example: Locality of changes. Gray boxes indicate classes and their extensions, sets enclosed in curly braces indicate the scope of a class. Classes enclosed in parentheses account for hierarchical scoping.

```
class Browser
  def open(url)
    WebPage.new.open(url)
  end

  partial

  class ::WebPage
    def popup(text); """Do nothing"""; end
  end
end
```

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

Representation of Modifications and Notation

In our Ruby implementation, class modifications are defined as members of classes [*2]. Following COP terms and notation, we call both method additions and method refinements *partial methods*. Partial methods must be wrapped in *partial classes*, which are defined like nested classes, but require a preceeding `partial` statement [*3] and must reference an existing class, which is called *target class*. The target class is the class that is being extended.

For example, in Fig. 5, `Browser` is a class defining a partial

[*2] They can also be defined as members of modules (mixins). Whenever we say *class*, we also refer to modules.

[*3] This statement works similar to `public`, `private`, and `protected`.

class targeting the class `WebPage` and containing a partial method `popup`. Partial methods can be instance methods or class methods; however, we focus on instance methods in the remainder of this work.

Scope of Modifications

To avoid destructive modifications, there are rules to activate and deactivate modifications. (De)activation is supported on a per-class basis, i.e., it is not possible to (de)activate single partial methods or partial classes separately. The term *(de)activation* of class C means that all method additions and method refinements defined *in* a class C (not *targeting* class C!) are (de)activated. The rule for class activation is simple: A class C is activated if one of 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.foo`, where $D \notin scope(C)$, then class C is deactivated. The scope of a class always contains the class itself (*reflexivity*). Therefore, when calling `WebPage.popup` from `Browser`, the method lookup will select the method refinement defined in `Browser`, because `Browser` remains active.

Furthermore, the scope of a class also contains the target classes of all partial classes. For example, `WebPage` $\in$ *scope*(`Browser`). If `WebPage.open` calls `popup`, the method lookup will use the method refinement if `open` was called from `Browser`, because `WebPage` $\in$ *scope*(`Browser`). This is also known as *local rebinding* [6] or dynamic scoping.

$$scope(\texttt{Browser}) = \{\texttt{Browser}\} \cup \{\texttt{WebPage}\}$$

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.foo`, first check if a class in the class activation set has a partial method for `C.foo` and use that method instead. If not, use method `C.foo`. In order to support all requirements stated in the previous section, the following sections will refine the method dispatch mechanism, the data structure used for storing activated classes, the definition of the scope of a class, and the rule for activating classes.

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.

```
class Application
   def main
      Browser.new.open("http://...")
      Viewer.new.check("secret.html")
   end
end
```

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

```
class Application
   def main
      Browser.new.open("http://...")
   end
end

class Browser
   def open(url)
      WebPage.new.open(url)
      Viewer.new.check("secret.html")
   end

   # Same partial classes as in example above
end
```

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

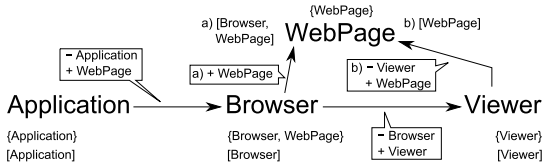

Fig. 8 Example: `Browser` uses `Viewer`. Sets enclosed in curly braces indicate the scope of a class (see Fig. 4). Sets enclosed in square brackets indicate class activation sets after calling a method (specific to this scenario). Arrow annotations indicate add/remove operations to the class activation set.

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, because `Viewer` $\notin$ *scope*(`Browser`). `Viewer` still shows popup windows (**Fig. 8**).

It is unclear what the anticipated behavior of `Viewer` is in this case: Should it show a popup window or not? That decision is left to the programmers. They could apply `Browser`'s modifications to `Viewer` by adding a (potentially empty) partial class targeting `Viewer` to `Browser`.

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

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

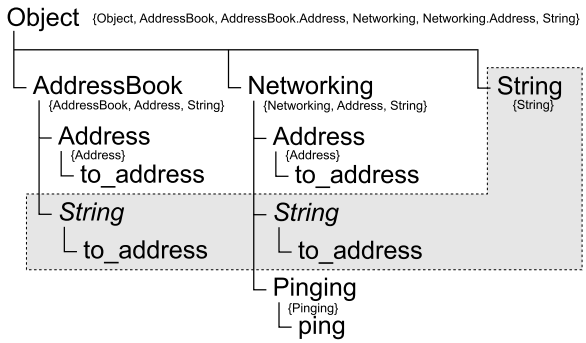

Fig. 10 Example: Duplicate auxiliary methods. Method refinements for `String.to_address` are only active in their respective owner classes and their nested classes.

`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 `Pinging`. 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` $\in$ *scope*(`Networking`). Consequently, `Pinging` uses the `to_address` method addition defined as part of `Networking`.

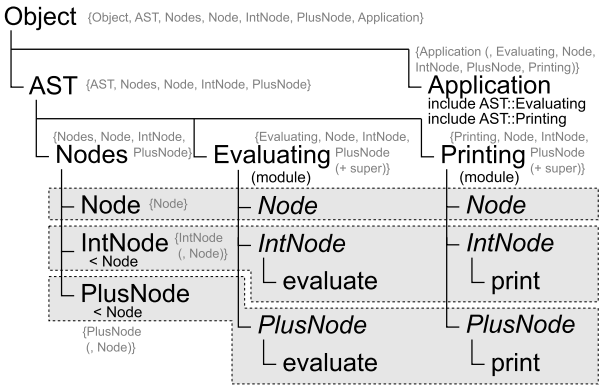

Fig. 11 Example: AST library. Classes enclosed in parentheses in scope sets account for inheritance scoping.

$$
\begin{aligned}
\mathit{scope}(\texttt{Networking}) = {} & \{\texttt{Networking, String}\} \\
& \cup \mathit{scope}(\texttt{Networking.Addr.}) \\
& \cup \mathit{scope}(\texttt{Pinging}) \\
= {} & \{\texttt{Networking}, \texttt{Networking.Addr.}, \\
& \texttt{String}, \texttt{Pinging}\}
\end{aligned}
$$

Moreover, according to the rule described in the previous section, both classes `AddressBook` and `Networking` can define class additions `String.to_address` and work 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` $\notin$ *scope*(`AddressBook`) and `AddressBook` $\notin$ *scope*(`Networking`).

3.3 Importing Extension Classes

In this section, we extend Extension Classes to allow for importing. Consider a libary 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 evalution 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 ef-

```
module AST
   module Nodes
      class Node; end
      class IntNode < Node; end
      class PlusNode < Node; end
   end

   module Evaluating
      partial

      class Nodes::IntNode
         def evaluate
            return value
         end
      end

      class Nodes::PlusNode
         def evaluate
            return left.evaluate + right.evaluate
         end
      end
   end

   module Printing
      # Implementation omitted
   end
end
```

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.

```
class Application
   include AST::Printing
   include AST::Evaluating

   def main
      AST::Nodes::IntNode.new(42).evaluate
   end
end
```

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

fectively 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` are included in class `Application`, which activates both modules (first `Printing`, then `Evaluating`) when running code from `Application` (or a nested class or a subclass). Similarly to standard Ruby, the ordering of `include` statements is important if two modules provide partial methods with the same name for the same target class (see *Layered Modifications*).

Scope of Modifications

We extend the notion of the scope of a class such that it also includes all classes contained in the scope of its superclass [*4]. For example, `Application`'s superclass is an application of mixin `Evaluating`, whose scope includes all AST node classes (for presentation purposes, the following formulas do not account for `Printing`).

[*4] Class `Object` is an exception. Since `Object` is the root of the class nesting hierarchy and the inheritance hierarchy in Ruby, we do not include *scope*(`Object`) in subclasses. In that case all classes would be included.

```
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`. This evaluator builds on top of `Evaluating` by including it internally.

$$scope(\texttt{Evaluating}()) = \{\texttt{Evaluating}(), \texttt{Node}, \texttt{IntNode}, \texttt{PlusNode}\}$$

$$scope(\texttt{App.}) = scope(\texttt{Evaluating}()) \cup \{\texttt{Application}\} = \{\texttt{Application}, \texttt{Evaluating}(), \texttt{Node}, \texttt{IntNode}, \texttt{PlusNode}\}$$

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 [*5] 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.

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.

[*5] `super` is a better name but cannot be overloaded in Ruby.

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 modifications) 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.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 $C \notin scope(L)$, 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 $super(L) \neq$* `Object`*).*

$$
\begin{aligned}
&scope(L) = \{L\} && \textit{(reflexivity)}\\
&\quad \cup \{C \mid C \in nested^*(target(P)) \wedge P \in partials(L)\} \\
&\qquad \textit{(dyn. scoping / local rebinding (+ hierarch. scoping))}\\
&\quad \cup \{C \mid C \in scope(N) \wedge N \in nested(L)\} \\
&\qquad \textit{(hierarchical scoping)}\\
&\quad \cup scope(superclass(L)) && \textit{(inheritance scoping)}
\end{aligned}
$$

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.

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

[*7] Therefore, $nested^*(L) \subseteq scope(L)$.

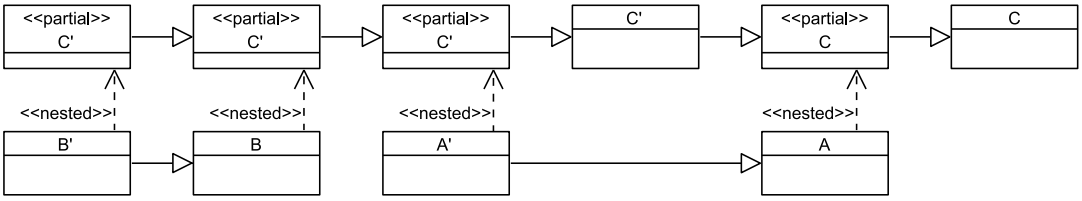

Fig. 15 Example: Effective superclass hierarchy. The class composition stack is ⟨B', A'⟩.

$$\cup \{target(P) \mid P \in partials(L)\}$$
(dynamic scoping / local rebinding)

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 inversed 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 `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 look up methods in C. Partial classes are conceptually subclasses 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.*

$$LayerHier(L, C) = \sum_{i=0}^{\#L} \langle super^i(L)[C] \rangle$$

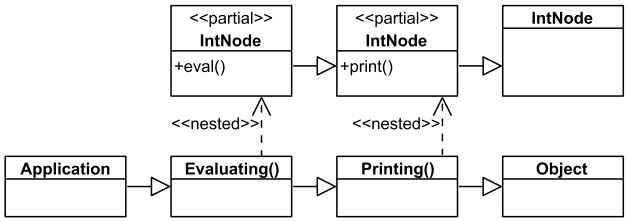

Fig. 16 Example: Effective superclass hierarchy for `IntNode`.

$$ClassLayers(C) = \Big(\sum_{i=1}^{|S|} LayerHier(S[i], C) \Big) + \langle C \rangle$$

$$Effective(C) = \sum_{i=0}^{\#C} ClassLayers(super^i(C))$$

$LayerHier(L, C)$ 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.

Figure 15 illustrates the effective superclass hierarchy in an example. Let us assume that the layer composition stack contains classes `A'` and `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.

$$\begin{aligned}
LayerHier(B', C') &= \langle B'.C', B.C' \rangle \\
LayerHier(A', C') &= \langle A'.C' \rangle \\
LayerHier(A, C) &= \langle A.C \rangle \\
ClassLayers(C') &= \langle B'.C', B.C', A'.C', C' \rangle \\
ClassLayers(C) &= \langle A.C, C \rangle \\
Effective(C) &= \langle B'.C', B.C', A'.C', C', A.C, C \rangle
\end{aligned}$$

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 `Evaluating`: In both designs, the effective superclass hierarchy of `IntNode` is defined as follows.

$$\begin{aligned}
Effective(\texttt{IntNode}) = \; & ClassLayers(\texttt{IntNode}) \\
& + ClassLayers(\texttt{Node}) \\
& + ClassLayers(\texttt{Object})
\end{aligned}$$

In Design Include, $ClassLayers(\texttt{IntNode})$ expands to three summation terms [*8] and ultimately results in the superclass hierarchy

[*8] For presentation reasons, we abbreviate `Evaluating` with `Eval` and `Printing` with `Print`.

described above.

$$\begin{aligned} \mathit{ClassLayers}(\texttt{IntNode}) = {} & \mathit{LayerHier}(\texttt{Mod10Ev}, \texttt{IntNode}) \\ & + \mathit{LayerHier}(\texttt{Eval}, \texttt{IntNode}) \\ & + \mathit{LayerHier}(\texttt{Print}, \texttt{IntNode}) \\ & + \langle \texttt{IntNode} \rangle \\ = {} & \langle \texttt{Mod10Ev.IntNode}, \texttt{Eval.IntNode}, \\ & \texttt{Print.IntNode}, \texttt{IntNode} \rangle \end{aligned}$$

In Design Subclass, $\mathit{LayerHier}(\texttt{Mod10Ev}, \texttt{IntNode})$ expands to both `Mod10Ev.IntNode` and `Eval.IntNode`, and ultimately results in the same superclass hierarchy.

$$\begin{aligned} \mathit{ClassLayers}(\texttt{IntNode}) = {} & \mathit{LayerHier}(\texttt{Mod10Ev}, \texttt{IntNode}) \\ & + \mathit{LayerHier}(\texttt{Print}, \texttt{IntNode}) \\ & + \langle \texttt{IntNode} \rangle \\ = {} & \langle \texttt{Mod10Ev.IntNode}, \texttt{Eval.IntNode}, \\ & \texttt{Print.IntNode}, \texttt{IntNode} \rangle \end{aligned}$$

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 $L \to C_1 \to \ldots \to C_n \to A$ with $C_i \in \mathit{scope}(L)$ (for all $i = 1 \ldots n$) and $A \in \mathit{scope}(L)$. The control flow may also originate from a subclass of L or a nested class of L. Programmers can enforce that a class $C_i \in \mathit{scope}(L)$ by adding a partial class targeting C_i to L (partial classes can be empty). For example, in Section 3.1, $n = 1$, $L =$ `Browser`, $C_1 =$ `WebPage`, and $A =$ `WebPage` when calling `WebPage.popup` 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 preceeding `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.

Performance Considerations

Our implementation effectively rewrites the method lookup in Ruby. This is sufficient for experiments, but unacceptable for production code because of performance issues. A more mature implementation has to either be optimized to work well with a JIT compiler to automatically remove this overhead or be implemented in the interpreter and contain additional optimizations such as class composition caching and (partial) method inlining.

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 op-

[*9] For that reason, we have to write `::WebPage` or `Object::WebPage` instead of just `WebPage` in Section 3.1.

erations 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 complexitity and better code reuse, among other benefits [27]. Two well-known techniques to achieve this are hypermodules [26] in Hyper/J 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 for all nodes, consisting of only the `evaluate` methods and their helper methods. The concern of printing can be implemented in a similar way. In Hyper/J or AspectJ terms, such a class defining method additions is similar to a *hyperslice* or *aspect*, respectively.

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 additonal classes and double dispatch. A common approach in Ruby are 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 [*10].

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 multi-dimensional 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 reimplementation 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.

[*10] Ruby refinements have other limitations and were discussed in Section 2.

References

[1] Akai, S. and Chiba, S.: Method Shelters: Avoiding Conflicts Among Class Extensions Caused by Local Rebinding, *Proc. 11th Annual International Conference on Aspect-oriented Software Development, AOSD '12*, New York, NY, USA, pp.131–142, ACM (2012).

[2] Aotani, T., Kamina, T. and Masuhara, H.: Type-Safe Layer-Introduced Base Functions with Imperative Layer Activation, *Proc. 7th International Workshop on Context-Oriented Programming, COP'15*, New York, NY, USA, pp.8:1–8:7, ACM (2015).

[3] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J. and Perscheid, M.: A Comparison of Context-oriented Programming Languages, *International Workshop on Context-Oriented Programming, COP '09*, New York, NY, USA, pp.6:1–6:6, ACM (2009).

[4] Appeltauer, M., Hirschfeld, R. and Lincke, J.: Declarative Layer Composition with The JCop Programming Language, *Journal of Object Technology*, Vol.12, No.2, pp.4:1–37 (2013).

[5] Bergel, A., Ducasse, S. and Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java, *Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'05*, New York, NY, USA, pp.177–189, ACM (2005).

[6] Bergel, A., Ducasse, S., Nierstrasz, O. and Wuyts, R.: Classboxes: Controlling Visibility of Class Extensions, *Computer Languages, Systems and Structures*, Vol.31, No.3-4, pp.107–126 (2005).

[7] Bicking, I.: Opening Python Classes, available from ⟨http://goo.gl/ln8ozK⟩ (accessed 2016-07-04).

[8] Bloch, J.: *Effective Java* (*The Java Series*), Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition (2008).

[9] Bracha, G. and Cook, W.: Mixin-based Inheritance, *Proc. European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications, OOPSLA/ECOOP '90*, New York, NY, USA, pp.303–311, ACM (1990).

[10] Clifton, C., Leavens, G.T., Chambers, C. and Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, *Proc. 15th ACM SIGPLAN Conference on Object-oriented Pro-*

gramming, Systems, Languages, and Applications, OOPSLA '00, New York, NY, USA, pp.130–145, ACM (2000).
[11] Cooper, P.: *Ruby 2.0 Walkthrough. Ruby 2.0 explained, for Ruby 1.9 developers*, Cooper Press Ltd. (2014).
[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995).
[13] Günther, S. and Fischer, M.: Metaprogramming in Ruby: A Pattern Catalog, *Proc. 17th Conference on Pattern Languages of Programs, PLOP '10*, New York, NY, USA, pp.1:1–1:35, ACM (2010).
[14] Hirschfeld, R., Costanza, P. and Haupt, M.: An Introduction to Context-Oriented Programming with ContextS, *Generative and Transformational Techniques in Software Engineering II*, Lecture Notes in Computer Science, Vol.5235, pp.396–407, Springer Berlin Heidelberg (2008).
[15] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-Oriented Programming, *Journal of Object Technology*, March-April 2008, ETH Zurich, Vol.7, No.3, pp.125–151 (2008).
[16] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proc. 15th European Conference on Object-Oriented Programming*, ECOOP '01, London, UK, UK, pp.327–353, Springer-Verlag (2001).
[17] Lincke, J., Appeltauer, M., Steinert, B. and Hirschfeld, R.: An Open Implementation for Context-oriented Layer Composition in ContextJS, *Science of Computer Programming*, Vol.76, No.12, pp.1194–1209 (2011).
[18] Nierstrasz, O., Ducasse, S. and Pollet, D.: *Squeak by Example*, Square Bracket Associates (2009).
[19] Nutter, C.: Refining Ruby, available from ⟨http://goo.gl/jXLLnO⟩ (accessed 2016-07-04).
[20] Odersky, M.: The Scala Language Specification, Version 2.9 (2014).
[21] Oliveira, B.C., Moors, A. and Odersky, M.: Type Classes As Objects and Implicits, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, New York, NY, USA, pp.341–360, ACM (2010).
[22] Schärli, N., Ducasse, S., Nierstrasz, O. and Black, A.P.: *Traits: Composable Units of Behaviour*, pp.248–274, Springer Berlin Heidelberg (2003).
[23] Springer, M., Masuhara, H. and Hirschfeld, R.: Hierarchical Layer-based Class Extensions in Squeak/Smalltalk, *Companion Proc. 15th International Conference on Modularity, MODULARITY Companion 2016*, New York, NY, USA, pp.107–112, ACM (2016).
[24] Springer, M., Niephaus, F., Hirschfeld, R. and Masuhara, H.: Matriona: Class Nesting with Parameterization in Squeak/Smalltalk, *Proc. 15th International Conference on Modularity, MODULARITY 2016*, New York, NY, USA, pp.118–129, ACM (2016).
[25] Takeshita, W. and Chiba, S.: Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches, *Software Composition*, Lecture Notes in Computer Science, Vol.8088, pp.49–64, Springer Berlin Heidelberg (2013).
[26] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S.M.: N Degrees of Separation: Multi-dimensional Separation of Concerns, *Proc. 21st International Conference on Software Engineering, ICSE '99*, New York, NY, USA, pp.107–119, ACM (1999).
[27] Tarr, P., Ossher, H. and Sutton, Jr., S.M.: Hyper/J™: Multi-dimensional Separation of Concerns for Java™, *Proc. 24th International Conference on Software Engineering, ICSE '02*, New York, NY, USA, pp.689–690, ACM (2002).
[28] The Ruby Contributors: Refinements, available from ⟨http://goo.gl/VXUE5N⟩ (accessed 2016-07-04).
[29] Warth, A., Stanojević, M. and Millstein, T.: Statically Scoped Object Adaptation with Expanders, *Proc. 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06*, New York, NY, USA, pp.37–56, ACM (2006).

Matthias Springer is a Ph.D. student at the Department of Mathematical and Computing Science, Tokyo Institute of Technology. He received his B.Sc. and M.Sc. 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/).