Method Shelters: Avoiding Conflicts among Class Extensions Caused by Local Rebinding

Shumpei Akai Shigeru Chiba
Tokyo Institute of Technology, Japan
akai@csg.is.titech.ac.jp chiba@acm.org

Abstract

A class extension, also known as open classes, allows programmers to modify existing classes and thus it is supported by several programming languages. However, class extensions imply a risk that they supply different definitions for the same method and those definitions conflict with each other. Several module systems have been proposed to address these conflicts. One approach lexically restricts the scope of class extensions but they do not allow us to change the behavior of methods called indirectly. Another approach is to make only class extensions explicitly imported effective while preserving the local rebinding property, which allows us to change the behavior of indirectly called methods. However, this approach causes conflicts if potentially conflicting class extensions are imported together. To address this problem, we propose a new module system named *method shelters*. A method shelter confines a scope of class extensions while preserving the local rebinding property. Hidden class extensions in a method shelter are not visible from the outside. We implemented a prototype of the proposed module system in Ruby. This paper illustrates several examples of the use of method shelters and also shows the results of benchmarks on our prototype.

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

General Terms Languages, Design

Keywords Module, Class extension, Open class.

1. Introduction

Extending existing classes by redefining methods or adding new methods, known as *class extensions*, *open classes* [11] or *revisers* [6], is an important feature for object-oriented programming languages to get better expressiveness. This feature is found in several languages: Smalltalk [8], CLOS [4], Objective-C, Ruby [14] and aspect-oriented languages including AspectJ [10] (as inter-type declarations). In Ruby and especially in Ruby on Rails [13], which is a popular web application framework for Ruby, class extensions are widely used to make the code simpler. Class extensions are used for three major aims: (1) adding convenient methods to core classes, (2) traversing trees without the Visitor pattern, and (3) redefining or adding methods to existing classes (*monkey patching*).

However class extensions also cause a problem; methods added by different libraries may conflict. If library $L1$ and $L2$ required by the same application program define methods with the same name for the same class and they have different behavior, they may crash the program. In Ruby's culture, libraries often modify core-classes and classes in other libraries required by them. Avoiding conflicts among method (re)definitions has been a serious issue.

To address this problem, several mechanisms such as selector namespaces [17], Classboxes [2, 3] and Ruby's Refinements have been proposed. They introduce modules to confine class extensions. Selector namespaces and Refinements allow changing the behavior of methods only in a specific lexical scope, but they cannot change the behavior of methods indirectly called. Classboxes allows us to change the behavior of indirectly called methods (called the *local rebinding* property). Classboxes are useful to create a customized version of an existing library, for example, to create the Swing library from the AWT library of Java. The change is confined within a classbox but not effective out of the classbox. However, it is not possible to address conflicts when a classbox imports other classboxes and they provide conflicting class extensions.

We introduce a new module system named *method shelters* to avoid conflicts among class extensions while addressing the limitations of the previous proposals. A method shelter is a module for restricting a scope of method definitions. By appropriately controlling a scope of every method (re)definitions, programmers can avoid unanticipated conflicts. A method shelter provides:

AOSD'12, March 25-30, 2012, Potsdam, Germany.

1. a scope of method definitions,
2. the ability to import methods defined in other method shelters,
3. the ability to redefine a method defined in an imported method shelter,
4. the ability to protect methods from redefinition and
5. no ambiguity with respect to method lookup.

The main contributions of this paper are: (i) the model of method shelters, (ii) the lookup algorithm written in Scheme and (iii) a proof-of-concept implementation for Ruby with performance benchmarks. According to our benchmarks, overheads due to method shelters are acceptable and one benchmark showed that method shelters may even improve execution performance.

In the rest of this paper, Section 2 presents motivating examples and the problem in the existing systems. In Section 3 we illustrate the model and lookup algorithm of method shelters. Section 4 shows our proof-of-concept implementation on the Ruby virtual machine. In Section 5 we show that method shelters and our implementation can be used to avoid conflicts among class extensions. In Section 6 we discuss the performance of method shelters. Section 7 describes related work. Section 8 concludes this paper.

2. Motivation: class extensions and method conflict

Subclassing and inheritance are popular techniques to extend a program in object-oriented programming languages. Subclassing allows programmers to create a new class with different behavior while partly reusing the implementation of its super class. However it cannot redefine an existing class. Class extensions and open classes are proposals to enable that. They are useful when programmers want to reuse a *whole* program (or framework) and partly customize it to build new software as we mentioned in Section 2.1 of [6].

2.1 Usage of class extensions

Class extensions are frequently used in Ruby[14]. For example, a number of use cases are found in Ruby on Rails[13]. We below show typical usage of class extensions in Ruby.

Convenient methods Class extensions are used to add convenient methods to core classes: Integer, String, Array and so on. For example, in Ruby on Rails, a suite of bytes methods are added to Numeric class, which is a super class of Integer and Float classes. The method call "`n.kilobytes`" (where n is a number) returns $n \times 1024$ and "`n.megabytes`" returns $n \times 1024^2$. These bytes methods are useful when writing a program that handles file sizes. Programmers' intentions will be clear.

Another example is sum method. Ruby on Rails also adds sum method to Enumerable module, which is a mixin [5] module for list-like classes. This method computes the sum of elements in an Enumerable object. This method is simple and useful although the Ruby's standard library does not provide it. By using class extensions, third party libraries such as Ruby on Rails can easily add convenient methods.

Operator redefinition In Ruby, several operators, such as `+`, `-`, `*` and `/`, are normal methods. Thus anyone can redefine them.

Division of integers in Ruby returns an integer by default, for example, `1/2` returns `0`. On the other hand, the Ruby's standard library mathn redefines it. Once you load this library, division of integers returns a rational. `1/2` returns a Rational object that represents $\frac{1}{2}$. This library makes it possible to describe mathematical expressions with normal notations.

Tree traversal Class extensions simplify tree traversal. If you naively write traversal code separately from tree-node classes, the code includes runtime type checking and it must be modified when a new node class is added. Although the code following the Visitor pattern is more extensible, all node classes must have methods for the Visitor pattern in advance. The Visitor pattern is not applicable to a tree if the node classes do not conform the Visitor pattern or they are not modifiable since a third-party library provides them.

If class extensions are available, you can add methods for traversal to node classes on demand. For example, suppose that a tree consists of Integer and Array and you want to sum up every integer elements in a tree. You only have to write the following code

```
1  class Integer
2    def sum_tree
3      self
4    end
5  end
6
7  class Array
8    def sum_tree
9      result=0
10     for child in self
11       result += child.sum_tree
12     end
13     result
14   end
15 end
```

The Visitor pattern is not required to traverse trees if you have class extensions. The code including runtime type checking is not needed.

Serialization libraries for Ruby often use this technique. For example, a JSON [7] library for Ruby adds to_json method to core classes, such as Integer, String, Array and Hash. A JSON serializer uses this method to traverse a tree made by core classes and dump a JSON file.

Monkey patching When a third-party library has a bug, class extensions allow programmers to patch and fix it. A method that includes a bug can be replaced with a correct implementation of that method. Programmers do not have to

directly modify the source code of the library. This technique is known as *monkey patching*.

2.2 A problem: method conflicts

Redefinition of a method by a class extension is visible from all classes. If more than one libraries redefine a method with the same name in the same class, those redefinitions conflict with each other. In Ruby, if there are multiple definitions, only the definition loaded last is made effective. Thus, which method definition is executed when the method is called depends on the order of loading classes and libraries. A library including class extensions implies a risk that it crashes other libraries calling the methods that those class extensions redefines.

This is a real problem in Ruby. If you load mathn library, all integer division results in a rational number. However, almost all programs in Ruby expect it results in an integer. Except writing small scripts, programmers have to treat this library with special care.

Library developers can avoid method conflicts to a certain degree by introducing a naming convention. If all method names include a unique prefix or suffix, a risk of conflicts is decreased against other libraries. This approach avoids conflicts of added methods but it is not suitable for method redefinition. Moreover, this approach may degrade the usability of a library.

2.3 Classboxes and Ruby's Refinements

To address the problem of conflicting method definitions, several module systems have been proposed. *Classboxes*[3] and *Classbox/J*[2] provide a module named a *classbox*. Classboxes allow programmers to write class extensions to modify a library while not affecting other application programs using that library. A classbox can import a class from another classbox and it can redefine a method of the imported class. The redefinition is visible from not only that importing classbox but also the methods of the classes imported by that classbox. Hence a classbox can override the behavior of its imported classbox; this property is called *local rebinding* in the literature.

Although a classbox hides method definitions from the outside[1], it exposes them to classboxes importing it. Thus, if a classbox imports classes from other classboxes and those classes contain conflicting method definitions, the problem occurs. Figure 1 shows an example of the problem. *CB0* is a classbox that provides Integer class and its div method returning an integer. Classbox *CB1* provides List class with avg method. To calculate an average, *CB1* imports Integer class from *CB0* and redefines div method to return a rational number. Since avg method calls div method internally, it returns an average of elements by a rational number. *CB2* imports List from *CB1* to calculate an average, and imports

[1] As far as we know, the original design of classboxes does not provide a mechanism for communicating between a class box and its outside. A whole application program has to run in a classbox.

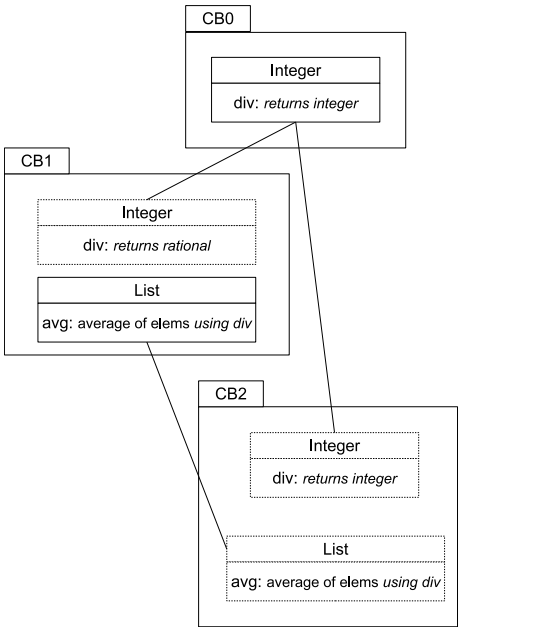

Figure 1. A problem in classboxes

Integer from *CB0* to perform *integer* division. Although the programmer of *CB2* does not know *CB1* internally modifies Integer class, she will expect that avg method returns a rational number. However, Integer from *CB0* overwrites the whole Integer class including the definition of div method due to the local rebinding property. Thus, in *CB2*, avg returns not a rational but an integer number since the two definitions of div conflict in *CB2* and the conflict resolution does not fit the programmer's anticipation.

To address the problem of conflicting method definitions, another approach named *Refinements* were proposed for Ruby in the *ruby-core* mailing list. Refinements make class extensions effective only when the (re)defined methods are directly called within a specific lexical scope. Figure 2 shows a sample code for Refinements. A block starting with refine (line 2) defines class extensions. In the example, the / operator for Fixnum is redefined to return a rational number. Then, the using declaration makes class extensions effective within the current lexical scope. For example, using at line 10 makes the class extensions in MathN effective and hence a call to the / operator at line 12 executes the definition in MathN instead of one in the standard library. However, the class extension in MathN is effective only within the lexical scope from line 9 to 14. If foo method at line 11 calls another method out of this scope and it calls the / operator, then the definition in MathN is not executed. Refinements do not preserve the local rebinding property.

3. Method Shelters

We propose a new module system called *method shelters* to address conflicts among class extensions. Our idea is to

```
1  module MathN
2    refine Fixnum do
3      def /(other)
4        Rational(self,other)
5      end
6    end
7  end
8
9  class Foo
10   using MathN
11   def foo()
12     p(1 / 2)
13   end
14 end
15
16 f = Foo.new
17 f.foo # prints "(1/2)"
18 p(1 / 2) # prints "0"
```

Figure 2. Example of Ruby's *Refinements*

make some class extensions effective only within the module defining them and ones imported by that module. We also protect some class extensions from accidental overriding by outer modules, which directly/indirectly import that module. Thus, if programmers carefully control the scope of class extensions, unexpected conflicts among class extensions are avoidable.

We designed method shelters to provide the local rebinding property but make conflicts avoidable to a certain degree. On the other hand, the refinements of Ruby does not provide the local rebinding property. Classboxes provide it but may cause conflicts among class extensions if multiple versions of class extensions are used in an importing chain.

3.1 Overview

A method shelter, which is a unit of our module system, consists of two *chambers*: an *exposed* chamber and a *hidden* chamber. A chamber contains *import* declarations and method definitions. An import declaration imports another method shelter. A method definition may define a new method added to an existing class and it may redefine an existing method in an existing class.

Figure 3 shows a code sample in Ruby. It is a solution of the problem mentioned in Figure 1. In the code in Figure 3, three method shelters CoreShelter, AverageShelter, and ClientShelter are defined. CoreShelter has an Integer#div method in its exposed chamber. CoreShelter is imported by ClientShelter in its exposed chamber. Importing another method shelter in an exposed chamber is called *exposedly importing*.

If only exposed chambers are used, method shelters are similar to classboxes. The local rebinding property is preserved. The methods in exposed chambers are executed as if they all were in the exposed chamber of the outermost or root method shelter, which exposedly imports their method shelters directly or indirectly. If there are multiple definitions of

```
1  shelter :CoreShelter do
2    class Integer
3      def div(x)
4        # <returns integer result>
5      end
6    end
7  end
8
9  shelter :AverageShelter do
10   class Array
11     def avg
12       s = self.sum
13       return s.div(self.size) # rational version is called
14     end
15   end
16
17   hide
18   import :CoreShelter
19   class Integer
20     def div(x)
21       # <returns rational result>
22     end
23   end
24 end
25
26 shelter :ClientShelter do
27   import :Core
28   import :AverageShelter
29   def calc
30     [1,2,3,4].avg # returns "(5/2)"
31     5.div(2) # returns "2"
32   end
33 end
```

Figure 3. Code sample (a solution of the problem in Figure 1)

the same method m, the method definition of the outermost method shelter S is selected, and other method definitions of m in method shelters imported from S are overridden. Thus, if a method calls another method in the same method shelter, the call selects and executes a different definition of that method in an outer method shelter. Programmers must consider that a method in an exposed chamber may be redefined by another method shelter importing it.

On the other hand, method definitions in a hidden chamber are not visible from the outside. Furthermore, they are never redefined by another method shelter importing them. In Figure 3, AverageShelter has a hidden chamber (below line 17), which contains another Integer#div method. It is visible within AverageShelter but not from ClientShelter, which imports AverageShelter. Thus a call to div at line 31 in ClientShelter never selects the definition in AverageShelter whereas a call to div at line 13 in the exposed chamber of AverageShelter selects the definition at line 20 in the hidden chamber of AverageShelter. The problem in Figure 1 does not happen. However, hidden chambers have trade-off. A method defined in a hidden chamber cannot be redefined even if it has a bug and the user wants to fix it by monkey patching. The concept of exposed and hidden chambers are

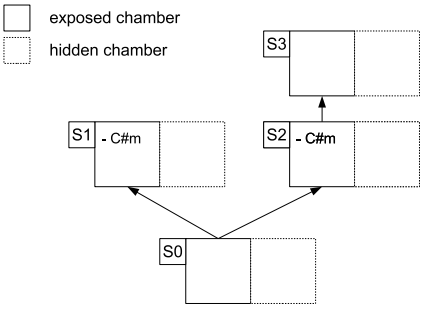

Figure 4. Ambiguous methods in a method shelter

similar to public/private methods in OOP languages. However method shelters are orthogonal to the public/private access control. We decided to use exposed and hidden as keywords to avoid misunderstanding.

Importing another method shelter in a hidden chamber is called *hiddenly* importing. The methods imported in a hidden chamber are visible only within the method shelter importing them, both its exposed and imported chambers. Note that those methods are imported only from an exposed chamber since methods in a hidden chamber are not visible from the outside. The methods imported in the hidden chamber of a method shelter S are not visible from other method shelters importing S.

The local rebinding property is preserved in a method shelter hiddenly imported. A method imported in a hidden chamber may be redefined in that hidden chamber. A hidden chamber is used to import and redefine several classes freely for local use only.

Our method-shelter system does not allow ambiguity with respect to method lookup. For example, in Figure 4, a method shelter S0 imports S1 and S2. Since both S1 and S2 have a method named m in C class, a call to C#m in the method shelter S0 is ambiguous and hence raises an error. It was possible to design the system so that such ambiguity can be implicitly resolved by introducing some precedence rules, for example, the last imported method shelter has the highest precedence. However, we did not adopt such implicit ambiguity resolution since we believe it will confuse programmers.

Global methods. We call methods (re)defined not within a method shelter *global methods*. A method in a method shelter can call a global method. Our module system considers that all global methods are contained in some anonymous method shelter. This method shelter is implicitly exposedly-imported by the method shelter that contains a caller method to a global method. Thus, the global methods can call methods in the exposed chamber where the caller method is defined. A redefinition of a method in that chamber is also effective when a global method calls it. On the other hand, the methods in the hidden chambers of the caller's method shelter are not visible from the global methods. If a global method calls another global method, these two methods can access the same shelters. For example, if a global method m0 is called from a method in a shelter S and m0 calls a global method m1, then m1 can call the same set of methods in the exposed chamber of S that m0 can call.

Entry point. Since a method in a method shelter is not visible from the outside, we need a special mechanism to call it at the beginning. In other words, we have to jump into a method shelter from normal execution contexts. We call that method shelter *an entry point*, which is the outermost method shelter in the import chain. An appropriate strategy depends on the base language:

- define a main function or method in a method shelter if the base language has it. The method shelter containing a main function is an entry point. A main function is a function that is first executed when a program starts.
- define a special code block specifying a method shelter. The code block is executed as if it existed within that method shelter. The entry point is that method shelter. Our ruby prototype adopts this strategy since Ruby does not have a main function like other scripting languages.

Note that in our programming model, every library, framework and application program is in a separate method shelter. The method shelter of an application program imports other method shelters of libraries and frameworks. Hence, having an entry point is natural.

3.2 Lookup semantics

In this section, we present the semantics of method shelters by showing its method lookup algorithm.

3.2.1 Method shelter tree

Method shelters can be imported from other method shelters. Hence the import relation among shelters constructs a directed graph. For the sake of presentation, we first transform this graph into a tree. We will use this tree to describe where we start looking up a method. This transformation is also used in our implementation in Section 4 for performance reason.

Figure 5 shows an example, where method shelter A imports B and C, and B imports C. We do not have to distinguish a type of importing, exposedly or hiddenly, in this transformation. If a method shelter (C in the example) is imported by multiple different method shelters, the node of that imported method shelter C is duplicated and the importing method shelters B and A import a different node of C (Figure 6). The resulting graph after this transformation is a tree, where every (node of a) method shelter is imported by at most one method shelter, that is, every node has at most one parent. We use this property of the tree for describing the algorithm of method lookup. Although this transformation does not work if import relations make a cycle, method shel-

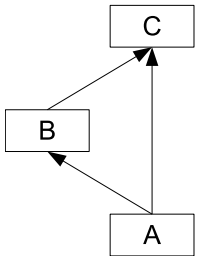

Figure 5. An example of an import graph of method shelters

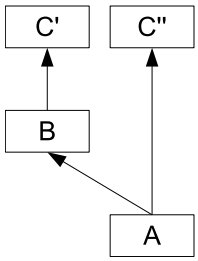

Figure 6. A method shelter tree reconstructed from Figure 5

```
1  (define (lookup context class methodname)
2    (let ((method (lookup−method−of−class context class
           methodname)))
3      (cond
4        (method method)
5        ((superclass class) (lookup context (superclass class)
             methodname))
6        (else (error "no␣method␣error" class name)))))
7
8  (define (lookup−method−of−class context class methodname)
9    (let ((hidden−method (lookup−hidden context class
           methodname)))
10     (if hidden−method
11         hidden−method
12         (let* ((source−chamber (find−source−chamber context))
13                (source−node (node−of−chamber source−))
14                (exposed−method
15                  (if (is−exposed? chamber)
16                      (lookup−exposed source−node class
                            methodname)
17                      (lookup−hidden source−node class
                            methodname))))
18           (if exposed−method
19               exposed−method
20               (lookup−global node class name))))))
```

Figure 7. Method lookup functions of method shelters

ters prohibit cyclic importing. If cyclic importing is detected, this graph-to-tree transformation raises an error.

Our semantics currently supposes that method shelters are immutable. If importing relations of method shelters are changed at run time, a method shelter tree should be reconstructed.

3.2.2 The lookup algorithm

We show the algorithm for looking up a method in a method shelter. Figure 7 lists the algorithm written in Scheme. lookup is the main function. It takes three arguments: *context*, *methodname* and *class*. *methodname* and *class* are the name of a called method and the class of the receiver object. *context* is a node in the tree of method shelters mentioned above. It indicates the method shelter that contains the caller method, which is currently running and attempts to call the method on the receiver object. The result of the method lookup depends on where the caller method is located.

lookup first tries to find a method in a given *class* by calling lookup-method-of-class. If a method is not found there, then lookup tries to find a method in the super class. Note that Ruby adopts single inheritance. lookup and lookup-method-of-class return a pair of the found method and the tree node of the method shelter containing that method, which will be implicitly passed to the found method for further method lookup.

lookup-method-of-class looks up a method in method shelters. First, it looks up the hidden chamber of the given method shelter node. If the method is found in that chamber, the found method is returned. If not found, it tries to look up a method again in the subtree rooted at the *source chamber*.

```
1  (define (hidden−imported? node)
2    <Is the given node is hidden−imported from parent?>)
3  (define (exposed−imported? node)
4    <Is the given node is exposed−imported from parent>)
5
6  (define (find−source−chamber node)
7    (cond
8      ((not (parent−node node)) (list node 'exposed))
9      ((hidden−imported? node) (list (parent−node node) 'hidden))
10     ((exposed−imported? node) (find−source−chamber (
           parent−node node)))
11   ))
```

Figure 8. Definition of source-node and source-chamber

Methods in the given shelter's exposed-side are looked up from the source chamber. The root chamber or hidden chambers which imports the given node can be the source chamber. The one nearest to the given shelter is selected as the source chamber. Figure 8 shows the find-source-chamber function that computes the the source chamber.

Figure 9 shows an example. Suppose that $S0$ exposedly imports $S1$, $S1$ exposedly imports $S2$, and $S1$ also hiddenly imports $S3$. Then the source chamber of $S0$, $S1$ and $S2$ is the exposed chamber of $S0$. Note that $S0$ is the method shelter at the entry point. The source chamber of $S3$ is the hidden chamber of $S1$ since $S3$ is hiddenly imported.

Figure 10 shows the definitions of lookup-exposed, lookup-hidden, and lookup-global functions used in lookup-method-of-class. lookup-exposed first searches the exposed chamber of the given node. If a method is found, the function returns

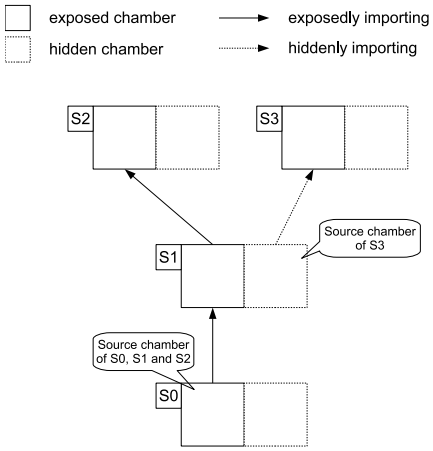

Figure 9. An example of source-node and source-chamber

```
1  (define (lookup−exposed node class name)
2    (if (exposed−method−table−exists? node class name)
3        (list node
4              (exposed−method−table−get node class name))
5        (filter−methods class name
6          (map (lambda (e) (lookup−exposed e class name))
7               (exposedly−importings node)))))
8
9  (define (lookup−hidden node class name)
10   (if (hidden−method−table−exists? node class name)
11       (list node
12             (hidden−method−table−get node class name))
13       (filter−methods class name
14         (map (lambda (e) (lookup−exposed e class name))
15              (hiddenly−importings node)))))
16
17 (define (lookup−global node class name)
18   (if (global−method−table−exists? class name)
19     (list <a node which is exposedly imported by the given node>
20           (global−method−table−get table name))
21     #f))
```

Figure 10. Definition of lookup-exposed and lookup-hidden

a pair of the given node and the found method body. Otherwise, the function recursively calls itself on all the nodes of the subtree rooted at the given node although the nodes hiddenly imported are excluded from the search space. Then the function makes a list of the values returned by the recursive calls. The list is processed by filter-methods, which returns an element if the list contains only one element. filter-methods raises an error if the list contains multiple elements since the method to look up is ambiguous. lookup-hidden is similar. It first searches the hidden chamber of the given node and then the subtree rooted at the given node. It searches only the nodes exposedly imported by the given node directly or indirectly. Finally, lookup-global searches the global method table. If it finds a method, it returns a pair of a method-shelter node and the body of the method found. This method-shelter node represents a method shelter that corresponds to the global method table and it is directly imported by the node given to lookup-global.

4. A proof-of-concept implementation

We made a proof-of-concept implementation [2] of method shelters in Ruby since Ruby already has a class extensions feature and its source code is publicly available. We modified the virtual machine of Ruby 1.9.2.

Since Ruby has already powerful expressiveness, we decided not to extend Ruby's syntax. The syntax of method shelters is based on Ruby's syntax. Figure 11 shows a sample code for illustrating method shelters' syntax. Although shelter looks like a keyword, it is a method name. "shelter" method takes shelter's name and a block. ":S2" represents a symbol S2. "do . . . end" represents a block. The methods defined in the block are contained in the method shelter. By default, those methods belong to an exposed chamber. On the other hand, the methods defined after a call to hide method (at line 15) belong to a hidden chamber. To import another method shelter, call import method. Its argument is a method shelter's name. If import method is called after hide method, the imported shelter is hiddenly imported and hence belongs to a hidden chamber.

The method shelter at the entry point is specified by shelter_eval method. For example, the line 24 and 25 in Figure 11 is executed within the contexts of the method shelter S0.

4.1 Implementation details

When a shelter method is called, we create a shelter object. A shelter object consists of five members: its name, a list of exposedly imported shelters, a list of hiddenly-imported shelters, an exposed method table and a hidden method table. When a method is defined in a method shelter, its method name is converted to a unique name. The mapping between the original method name and the converted one is recorded in the shelter object's method table. At method lookup, the table of converted method names is searched first.

As mentioned in Section 3.2, the method lookup algorithm needs a current node in a method-shelter tree representing import relations. To maintain a current node, we added a new member to the stack frame of the Ruby VM. When shelter_eval method is called, a method shelter tree is constructed from the specified method shelter and the root node of the tree is set to the stack frame of the block.

4.2 Optimization

We implemented a few optimization techniques for method shelters to improve execution performance. First, we added a method cache to every node of a method-shelter tree. It records a mapping from a pair of a class name and a method

[2] The source code is available at
`http://github.com/flexfrank/ruby_with_method_shelters`

```
shelter :S2 do
  class Integer
    def inc(n)
      self + n
    end
  end
end

shelter :S1 do
  class Integer
    def inc10
      self.inc(10)
    end
  end
  hide
  import :S2
end

shelter :S0 do
  import :S1
end

shelter_eval :S0 do
  p(1.inc10) # prints 11
  p(1.inc(1)) # error: method is not found
end
```

Figure 11. The syntax of method shelters

name to a pair of a method entry and the tree node where the method is found. A method entry is a primitive data structure for calling a method in the Ruby VM. A cache entry is updated at method lookup. This cache reduces the overhead of method lookup in particular when an import chain is long.

Since method shelters change the algorithm of method lookup, we also modified the implementation of the inline cache of the Ruby VM. The modified implementation records a current node of a method-shelter tree.

4.3 Compatibility

Our implementation keeps the compatibility with the original Ruby. A normal Ruby program written without method shelters can run on our modified Ruby interpreter. Although we added an additional member to a stack frame of the Ruby VM, this member for maintaining a current node of a method shelter tree is set to NULL at initialization. If the current node is NULL, the method lookup uses the original algorithm for Ruby.

5. Applications

We below illustrate several examples of the use of method shelters.

5.1 Convenient methods in Ruby on Rails

The first example is Ruby on Rails. The ActiveSupport library, which is part of Ruby on Rails, provides a number of convenient methods for Ruby's core classes. Among those methods, we moved time-related methods in the Numeric

```
shelter :ActiveSupportNumericTime do
 class Numeric
   # ** snip **

   def days
     ActiveSupport::Duration.new(self * 24.hours, [[:days, self]])
   end
   alias :day :days

   # ** snip **
 end
end
```

Figure 12. The time-related methods we defined in a method shelter

```
shelter :DateControllerShelter do
  class DateController < ApplicationController
    def days_ago
      @text=params[:id].to_i.days.ago
    end
  end

  hide
  import :ActiveSupportNumericTime
end
```

Figure 13. A client code of Ruby on Rails

class into a method shelter. ActiveSupport adds minutes, hour and days methods to Numeric class. These methods return Duration objects representing time. They simplify writing code for calculating time. For example,

```
10.minutes.ago
```

returns Time object representing the time 10 minutes before the current time.

We can move the definitions of these methods into a method shelter. Figure 12 is a code snippet of the method shelter containing these methods. Figure 13 shows a controller class for Ruby on Rails. Like a servlet in Java, it is executed when a corresponding web page is accessed by a web browser. This controller class is in a method shelter, which hiddenly imports ActiveSupportNumericTime. Thus, days method in Figure 12 is available only in this controller class whereas it is not in the rest of the program. Note that days method is not visible even in method shelters importing the method shelter in Figure 13. To call days, method shelters must import ActiveSupportNumericTime again within the method shelters.

5.2 Operator redefinition

We mentioned a problem of conflicting redefinition of the "/" operator in Section 2.3. The sketch of the solution with method shelters was already presented in Figure 3.

Figure 14 shows a realistic version of the code in Figure 3. In Ruby, numbers are represented by Fixnum objects and "/"

```
1  shelter :MathNShelter do
2    class Fixnum # fixed size integer in Ruby
3      def /(x)
4        Rational(self,x)
5      end
6    end
7  end
8
9  shelter :AverageShelter do
10   class Array
11     def avg
12       sum = self.inject(0){|r,i|r+i}
13       sum / self.size
14     end
15   end
16   hide
17   import :MathNShelter
18 end
19
20 shelter :ClientShelter do
21   import :AverageShelter
22
23   def calc
24     p([1,2,3,4,5,6,7,8,9,10].avg) # prints "(11/2)"
25     p(55/10) # prints 5
26   end
27 end
28
29 shelter_eval :ClientShelter do
30   calc
31 end
```

Figure 14. The code that redefines "/" methods in method shelters

method is defined in this class. Since the original division method "/" of Fixnum is built in, this code does not include CoreShelter shown in Figure 3. The "/" method is redefined in MathNShelter instead of AverageShelter. This simulates Ruby's "mathn" library, which is a separate library providing the redefined "/" method.

Since MathNShelter is hiddenly imported by a method shelter AverageShelter, avg method in Array returns a rational value. The "/" operator at line 13 executes the definition in MathNShelter method shelter. A method shelter ClientShelter can safely import AverageShelter and call avg method without being aware of MathNShelter. Note that since Fixnum is a class in the standard library, calc method can execute the "/" operator at line 25 without explicitly importing Fixnum class. The "/" operator here returns an integer.

Since Ruby is a scripting language, the lines from 28 to 30 compose the code running first when this program is invoked. This "main function" is executed in ClientShelter method shelter.

5.3 Protecting optimized methods

The Ruby VM optimizes several special methods including arithmetic operators. When one of the special methods is called and it is not redefined by the users, the VM directly performs its operation instead of executing that method. The VM manages for every operator a flag indicating whether or not the special methods are redefined. The receiver class is not considered for a reason of performance trade-off. Thus, if "+" operator for Integer is redefined by the users, the VM recognizes all "+" operators including one for Float are also redefined and makes them unoptimized. Redefining a single special method may cause serious performance overhead.

If such a special method is redefined in a method shelter, the VM can directly perform the optimized operation when it is out of that method shelter. Our implementation of method shelters manages the flags per method shelter. Hence, if a method shelter S redefines a special method in a hidden chamber, that redefinition is not visible from other method shelters importing S and the VM performs optimized operations for special methods in these method shelters. Otherwise, if a method shelter S_1 redefines a special method in an exposed chamber and another method shelter S_2 hiddenly imports S_1 for reusing the redefinition, then the redefinition is not visible from method shelters importing S_2, which are ones indirectly importing S_1. The VM performs optimized operations in these method shelters.

5.4 Private instance variables

In Ruby, private instance variables are not available. A method shelter can be used to define private instance variables visible only within the method shelter.

Figure 15 shows the code for defining getter and setter methods for accessing an instance variable with a newly generated unique name. When shelter_accessor method is called, accessor methods with the given name are defined. Note that in Ruby an instance variable is automatically created when it is first used. The code in Figure 15 does not use method shelters but the reflection capability of Ruby. get_var_name_for_current_shelter returns a unique name for the given name and the caller's method shelter. If the name and the shelter are same it returns the same variable name.

Figure 16 shows the client code. Two method shelters S0 and S1 add accessor methods to Object class. Although both the names of the added instance variables are counter, they access different instance variables. The methods defined by a call to shelter_accessor in different method shelters are distinct.

6. Performance

In this section, we discuss the performance of our prototype implementation of method shelters. Our implementation is based on Ruby 1.9.2 [3]. We compare it with the original implementation of Ruby 1.9.2. We ran our benchmark programs on Mac OS X 10.6 with 2.54GHz Intel Core 2 Duo processor and 4GB memory.

[3] The revision number of Ruby's subversion repository is 30579

```
class Module
  def shelter_accessor(name)
    define_method name do
      ivname= get_var_name_for_current_shelter(name)
      self.instance_variable_get(ivname)
    end

    define_method (name.to_s+"=").to_sym do|val|
      ivname= get_var_name_for_current_shelter(name)
      self.instance_variable_set(ivname,val)
    end
  end
end
```

Figure 15. The code for defining getter and setter methods to access a private instance variable

```
shelter :S0 do
  class Object
    shelter_accessor :counter
  end
end
shelter :S1 do
  class Object
    shelter_accessor :counter
  end
end

o=Object.new
shelter_eval :S0 do
  o.counter=0
  p o.counter #prints 0
end
shelter_eval :S1 do
  p o.counter #prints nil
  o.counter=1
  p o.counter #prints 1
end
shelter_eval :S0 do
  p o.counter #prints 0
end
```

Figure 16. The client code using accessor methods to a private instance variable

6.1 Micro benchmark

First, to measure an overhead of method lookup, we ran a program that calls a method with an empty body. The benchmark program calls an empty method 10,000,000 times. We prepared five environments: the original Ruby VM, our modified VM without method shelters, our VM with one method shelter and our VM with five method shelters imported. The benchmark code with five method shelters is shown in Figure 17. We ran the benchmark programs 1,000 times on each environment.

Table 1 shows the results. When method shelters are not used, our VM runs 10% slower than the original VM. This is because our VM must check whether a method shelter is passed or not on method lookup. When one method shelter is used, the overhead is about 3%. Method shelters make method lookup faster, this is due to method caches that we added. When five method shelters are used, it works with comparative speed to one method shelter. This result is also due to the caches.

```
shelter :S0 do
  def a
  end
end
shelter :S1 do import :S0 end
shelter :S2 do import :S1 end
shelter :S3 do import :S2 end
shelter :S4 do import :S3 end

shelter_eval :S4 do
  10000000.times do
    a
  end
end
```

Figure 17. The benchmark program that calls an empty method under five method shelters

	Avg. time (s)	SD[4]
On the original VM	1.430	0.010
On our VM without method shelters	1.575	0.018
With 1 method shelter	1.476	0.013
With 5 method shelters	1.493	0.018

Table 1. Execution time of empty method (1,000 tries)

	Avg. time (s)	SD
On the original VM	1.000	0.005
On our VM without method shelters	1.141	0.004
With 1 method shelter	1.180	0.036
With 5 method shelters	1.192	0.049

Table 2. Execution time of fib(33) (1,000 trials)

We also measured execution time of the Fibonacci function under the same environments as above. Table 2 lists the results. In this case the overhead of our VM is about 14% and with method shelters is 18% to 19%.

6.2 tDiary

To measure the performance of method shelters on a real application, we applied method shelters to *tDiary* [15], a web-based diary system written in Ruby. We used tDiary 3.0.1 for this benchmark. tDiary 3.0.1 redefines three methods in String class: to_a, each and method_missing. We redefined these three methods in a method shelter and ran the main code of tDiary in a method shelter importing it. We ran tDiary on Apache 2.2.17 with CGI and measured response time

[4] standard deviation

	Avg. time (ms)	SD
On the original VM	704	7.1
On our VM without method shelters	704	6.6
With method shelters	627	6.5

Table 3. Response time of tDiary (300 trials)

by ApacheBench. For comparison, we used three versions of tDiaries: tDiary without method shelters on the original Ruby VM, without method shelters on our Ruby VM and with method shelters on our Ruby VM. We accessed the top page of each diary 300 times.

Table 3 lists the results. This results show that our modified VM does not impact performance of existing applications when method shelters are not used. It also indicates method shelters improve the execution speed. This is due to Ruby VM's optimizations that we mentioned in Section 5.3. method_missing, which we confined into a method shelter, is the one of special methods. method_missing is a hook method that is called when an undefined method is called. If method_missing is not redefined, the VM can skip a call to it since the default definition is empty. In this benchmark, we redefined method_missing for String in a method shelter. Hence this redefinition does not affect the performance of the code out of that method shelter. On the other hand, tDiary running on the original Ruby VM gets performance penalties due to the redefinition of method_missing. This is why method shelters improved the execution performance of this benchmark test.

6.3 Ruby on Rails

We applied method shelters to Ruby on Rails in Section 5.1. We measured the performance of a Ruby on Rails application with method shelters. Figure 18 is a benchmark program we used. index method is an action method, which calculates time and accesses a database once. We used SQLite 3.6.12 for a database engine. The version of Ruby on Rails is 3.0.7. We ran this application on WEBrick, a web server written in Ruby. We requested the action 1,000 times through ApacheBench and measured response time.

Table 4 lists the results in *development environment*. In this environment, user-defined application classes are reloaded per request. In this case, method shelters made the execution performance about 50% slower. Table 5 lists the results in *production environment*, in which application classes are not reloaded per request. In this environment, the overhead is less than 4%. This difference between two environments result from the hit ratio of method caches. In the development environment, whenever classes are reloaded, the VM invalidates method caches for method shelters. This implies serious performance penalties.

Table 6 lists the hit ratio of method caches in method shelters after warming-up. In the production environment, over 90% and 100% of lookups hit inline method cache.

```
1 class TestController < ApplicationController
2   def index
3     @text="#{(1.day.ago␣+␣1.day)}"
4     @accesses=Access.order("id␣desc").limit(10).find_all.to_a.
          inspect
5   end
6 end
```

Figure 18. The benchmark program for Ruby on Rail

	Avg. time (ms)	SD
On the original VM	53.131	14.7
On our VM without method shelters	53.341	14.7
With method shelters	78.871	16.2

Table 4. Response time of Rails application (1,000 trials, development env.)

	Avg. time (ms)	SD
On the original VM	10.865	7.7
On our VM without method shelters	11.049	7.8
With method shelters	11.296	7.7

Table 5. Response time of Rails application (1,000 trials, production env.)

	development	production
Inline cache hit (%)	58.35	92.55
Total cache hit (%)	74.0	100.0

Table 6. Cache hit ratios of Rails application (1,000 trials, production env.)

In the development environment, less than 75% hit inline caches. This result indicates that method shelter is not so slow when method caches are appropriately filled.

7. Related work

We have already mentioned several related languages and mechanisms. This section presents other related work.

Java class loader. Since every Java class loader makes a separate name space, a different class loader can load a differently declared class with the same name. However, an instance of that class cannot be passed into the name spaces constructed by other class loaders while method shelters allow exchanging any instance among shelters.

Selector Namespaces. The concept of *selector namespaces* was introduced by Modular Smalltalk [17]. Selector namespaces allow scoped class extensions hence method conflicts can be resolved to a certain degree. However, they do not preserve the local rebinding property. In selector

namespaces, you can add new methods to existing classes but cannot redefine existing methods.

Open Classes. *MultiJava* [11] introduces *open classes* and *multiple dispatch* into Java. Open classes allow you to add new methods to existing classes although redefining a method is not allowed. The added methods are available only within a compilation unit that explicitly imports them. Here, a compilation unit is a module lexically specified.

Context-oriented programming. *Context-oriented programming* languages [9] allows multiple definitions of a method for the same class. Which definitions are executed at a method call depends on the runtime contexts. On the other hand, in method shelters, it depends on not the runtime contexts but the static contexts, which are static relations among modules with respect to importing. Although we could make these relations dynamically changeable according to the runtime contexts, that is not necessarily required.

Open modules. Since aspect-oriented programming (AOP) also allows class extensions, several approaches proposed in the contexts of AOP are included in the related work. For example, *Open modules* [1, 12] are modules for AOP; they expose only selected join points to the outside. Other join points are hidden in the modules. The exposed join points correspond to method definitions, which are newly added to existing classes, in an exposed chamber while the hidden join points correspond to ones in a hidden chamber. However, open modules do not control the visibility of advices, which correspond to method redefinitions, which modify an existing method, in method shelters.

Type classes. Type classes [16] in functional programming languages also provide a way to add methods or generic functions to existing types. Although type classes help extension of software, they do not support local rebinding property. Method shelters target programming languages with a construct supporting it.

8. Conclusion

We propose method shelters to address conflicts among class extensions. Each method shelter consists of exposed and hidden chambers. While methods defined in an exposed chamber are visible from method shelters that import it, methods in a hidden chamber are visible only from the same method shelter. We provide hidden chambers for defining class extensions only for internal use. Hence, if programmers carefully put class extensions in an appropriate chamber, unanticipated conflicts will be avoidable. When updating a method, programmers must preserve the backward compatibility if the method is in an exposed chamber. If programmers want to keep freedom for future updates, methods must be in a hidden chamber and other method shelters internally used must be hiddenly imported in a hidden chamber.

We presented the lookup algorithm for method shelters and showed a prototype implementation of method shelters on the Ruby virtual machine. According to our benchmarks, general overheads due to method shelters are acceptable. Furthermore, one benchmark revealed that method shelters may even help performance optimization and boost the execution speed.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP '05*, pages 144–168, 2005.

[2] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05*, pages 177–189. ACM, 2005.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. In *Computer Languages, Systems and Structures*, 2005.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23:1–142, September 1988.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, pages 303–311. ACM, 1990.

[6] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *OOPSLA '10*, pages 539–554. ACM, 2010.

[7] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`.

[8] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3): 125–151, March-April 2008.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–353. Springer-Verlag, 2001.

[11] T. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *OOPSLA '03*, pages 224–240. ACM, 2003.

[12] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to aspectj. In *AOSD '06*, pages 39–50. ACM, 2006.

[13] Rails core team. Ruby on rails. http://rubyonrails.org/, 2011.

[14] Ruby community. Ruby programming language. http://www.ruby-lang.org/, 2011.

[15] tDiary.org. tDiary. http://sourceforge.net/projects/tdiary/, 2011.

[16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76. ACM, 1989.

[17] A. Wirfs-Brock and B. Wilkerson. A overview of modular smalltalk. In *OOPSLA '88*, pages 123–134. ACM, 1988.