Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches

Wakana Takeshita, Shigeru Chiba

Method Shells: avoiding conflicts on destructive class extensions by implicit context switches

Wakana Takeshita[1] and Shigeru Chiba[1]

The University of Tokyo
www.csg.ci.i.u-tokyo.ac.jp

Abstract. We propose *method shells*, which is a module system for avoiding conflicts on customization by language mechanisms such as aspects in AspectJ and open classes in Ruby. These mechanisms allow programmers to customize a library without rewriting original source code but by only describing differences in a separate file. We call these mechanisms *destructive class extensions*. A problem with destructive class extensions is conflicts on customization. Different customizations may differently modify the same class. To address this problem, we propose a new module system named *method shells*. With this system, programmers can avoid conflicts since the module system automatically switches a set of customizations that has to be applied together according to the contexts declared by programmers. We present the idea of this module system and then its formal semantics. We also present an extension of Java that supports method shells.

1 Introduction

Aspect-oriented programming (AOP) [1] is a programming paradigm where crosscutting concerns can be separated into different modules called *aspects*. Aspects can modify the behavior of the code contained in a different module so that they will implement their concerns. This mechanism, however, can be used for not only implementing a crosscutting concern but also customizing an existing class library or framework to fit an application program. A class library (or framework) can be extended by subclassing but subclassing does not enable all kinds of customization. Some kinds of customization need to directly modify a class contained in the library. To modify such a class, aspects are useful language constructs from the viewpoint of software maintenance.

Aspects are not only the mechanism for customizing a class library without directly changing the library source code. For this purpose, several other mechanisms have been proposed such as open classes in Ruby [2] and refines in AHEAD [3]. In this paper, we call this category of mechanisms *destructive class extensions* [4] since they directly modify the behavior of existing classes.

A problem with destructive class extensions is that extensions often conflict with each other. This problem has been actively studied in the context of AOP but these studies have focused on conflicts observed when the implementations of different crosscutting concerns are woven at the same join point[5, 6]. On the

other hand, in the context of library customization, an important issue is to deal with a library customized by destructive class extensions (or aspects) as a *black box*. This is non-trivial if two libraries require the same sub-library but differently customize it by destructive class extensions. An application program using the two libraries together will cause conflicts on the customization of the sub-library and thus the programmer has to be aware of the customization by the libraries; the libraries cannot be considered as black boxes.

To avoid conflicts on the customization in this scenario, we propose a new module system named *method shells* and present an extension of Java that supports this module system.[1] As a mechanism for destructive class extension, we use a reviser [4]. A method shell is a module that can contain classes and revisers. It can include other method shells to extend and customize their classes. Furthermore, it can *link* to other method shells. The linked method shells are dealt with as black boxes. A method shell can invoke the code in the method shells linked to it but the invoked code is executed in a separate context so that it will not be affected by the customization effective in the method shell invoking the code. No unexpected conflicts happen between method shells linking to each other.

In the rest of this paper, we first show a motivating example and then present method shells. We also present the formal semantics of the method shells and a brief sketch of their implementation. Finally, we mention related work and conclude this paper.

2 Destructive class extensions

When programmers need to customize a class library, the customization would be convenient if they can modify it without directly modifying the original source code by only describing differences in a separate file. Such customization is modular and easy to maintain. Even if the customization includes a bug, they can easily obtain the prior code by deleting the file describing the customization.

Although subclassing is often used to describe such customization, it does not perfectly fit the aim. Suppose that the library contains a class C and she wants to modify a method in that class. Describing a subclass of C that overrides the method is not sufficient to make a customized library. All the classes in the library that create an instance of C must be modified to create an instance of that subclass. Subclassing is, therefore, not the perfect approach for customizing an existing library in a modular fashion.

For modular customization of existing libraries, several language mechanisms have been proposed in languages like Ruby [2], AspectJ [7], AHEAD [3], MultiJava [8], Jiazzi [9], and GluonJ [4]. In this paper, we call these mechanisms *destructive class extensions* since they are mechanisms for directly modifying

[1] The first author submitted a summary of this work to ACM Student Research Competition (SRC) held in March at AOSD 2013. The submission will be reviewed and oral and poster presentations will be scheduled in March. Submitting a full paper to another conference is permitted by the SRC moderators.

```
1  // in the browser library
2  revise WebPage {
3    void popup(HTML text) {
4      warning("disabled");
5    }
6  }
```

Figure 1. A reviser for the WebPage class

```
1  // in the HTML-renderer library
2  class WebPage {
3    void popup(HTML text) {
4      // show a popup window.
5    }
6    void onClick(Mouse m) {
7      URL url = m.getURL();
8      if (isPopup())
9         popup(url);
10     ...
11   }
12     ...
13 }
```

Figure 2. WebPage class

existing implementation. They allow programmers to append new methods to an existing class and substitute a new implementation of an existing method. The new implementation is described in a separate source file and thus the original source files are not modified. Describing customization in a separate module is not sufficient for scalable modular customization. It also has to enable modular reasoning; the customization must change the original implementation only through a public interface or well-designed extension points. Some languages such as AspectJ provide a powerful mechanism like pointcuts and thus their ability for modular reasoning is controversial [10, 11]. Since they enable changes of any parts of module, preserving modularity in large scale software is not straightforward. On the other hand, in other languages like GluonJ, the customization changes the implementation by redefining public methods and hence they enable as modular reasoning as normal object-oriented programming.

However, even in the latter languages, enabling modular customization is not easy. The customization through public methods will not scale as the number of methods increase. Different customizations may conflict on the same method. For better scalability, a scoping mechanism must be introduced so that the customization will be effective only within a limited space. This is the aim of this paper.

Suppose that we have a library l_1 for rendering an HTML text and we write another library l_2 for constructing a web browser that will be embedded in an application software. For code reuse, the library l_2 should be implemented on top of the former library l_1. Since an embedded web browser should not show a popup window, which will surprise application users, we have to customize the library l_1 so that a popup window will be blocked.

A mechanism of destructive class extension allows this customization without modifying the source code of the library l_1. Figure 1 shows the code for that modification. In this paper, we use the syntax of GluonJ [4]. The code modifies the original implementation of the WebPage class shown in Figure 2, which is contained in the library l_1. It directly replaces the original implementation of the popup method with a new one in Figure 1. If the library l_2 contains the code in Figure 1, which is called *a reviser*, the behavior of the library l_1 is revised and no popup window will not be displayed when l_2 uses l_1. The original source code of popup does not have to be modified.

We next write the third library l_3, which provides an audited viewer of local files written in HTML. The viewer shows a popup dialog for an alert when a confidential file is opened. To show a popup dialog, we use the popup method in the WebPage class supplied by the library l_1 for rendering an HTML text. Furthermore, we modify several methods by revisers, for example, the getBorder method in the WebPage method so that the rendered HTML text will be shown in a specially decorated window.

An application using either the library l_2 or l_3 will work well. However, since both l_2 and l_3 commonly use the library l_1 but differently modify the classes in l_1, an application using both of them will not work. The revisers in l_2 and l_3 will conflict. For example, the reviser in l_2 disables to show a popup window by the popup method whereas the library l_3 needs that the popup method shows a popup window as its original implementation does.

This conflict is a well-known problem with destructive class extensions [12, 13, 6] but it is more crucial than usual when destructive class extensions are used for customizing a library. A library is usually dealt with as a black-box; library users should be unaware of which other libraries are internally used by that library and how those other libraries are customized. It should be hidden that both the libraries l_2 and l_3 internally use l_1 and they differently customize l_1. Thus, a conflict on l_1 between l_2 and l_3 will be a surprise to their user programmers. This is a similar problem happening when an application requires two libraries and the two require other libraries that are different versions of the same library. A library providing basic functionality is often included by other third-party libraries but, if it is popular and being actively developed, these third-party libraries often require different versions of it. Such third-party libraries are difficult or impossible to use together. Our scenario of conflicts on customization can be regarded as a conflict between two versions of the library l_1, each of that is implemented by revisers describing differences from the base version.

3 Method Shells

To address the problem presented in the previous section, we propose a new module system named *method shells*. With this module system, a set of revisers that must be applied together is implicitly switched to fit execution contexts during runtime. As a prototype of method shells, we have developed an extension of Java. In this extended Java, a new language construct called *a method shell* is available. It is a construct similar to package and it specifies a module that classes and revisers in the source file belong to. Figure 3 presents a renderer method-shell. The first line is a methodshell declaration, which declares that the following WebPage class is contained in the method shell named renderer. This method shell represents the HTML-renderer library l_1 shown in the previous section.

```
1  methodshell renderer;
2
3  class WebPage {
4    void popup(HTML text) {
5      // show a popup window.
6    }
7    void onClick(Mouse m) {
8      URL url = m.getURL();
9      if (isPopup())
10       popup(url);
11     ...
12   }
13     ...
14 }
```

Figure 3. The renderer method shell

```
1  methodshell browser;
2  include renderer;
3
4  revise WebPage {
5    void popup(HTML text) {
6      warning("disabled");
7    }
8  }
9
10 public static void main(String[] args){
11   WebPage w = new WebPage();
12   w.popup("Available?"); // not shown
13 }
```

Figure 4. The browser method-shell

Include declarations and revisers

In the previous section, the HTML-renderer library l_1 was used by the web-browser library l_2. With the method shells, this relation is represented by an include declaration. Figure 4 presents a reviser in the browser library l_2 reimplemented with method shells. The second line is an include declaration. It represents that the browser method-shell includes the renderer method-shell. All the classes and revisers contained in the renderer method-shell are also contained in the browser method-shell. This relation by include declarations is transitive.

The reviser in Figure 4 belongs to the browser method-shell and it modifies the implementation of the WebPage class. The WebPage class is called *a target class* and it must be in the same method shell that the reviser belongs to. Since include declarations constructs transitive relations, the WebPage class could be in a method shell included by the method shell that the reviser belongs to. A method shell is a scope of the visibility of classes and revisers. Classes and revisers can refer to only the class names contained in the same method shell.

The implementations of the methods declared in a reviser substitute the original ones in the target class or they are appended to the target class if they are new methods. The reviser in Figure 4 substitutes the implementation of the popup method in the WebPage class. Although the source code of the original implementation is not modified, the modification by the reviser is directly applied to the target class. This is a difference from subclassing. The modification by a subclass of WebPage will not affect the instances of WebPage but the modification by a reviser for WebPage affects the instance of the target class WebPage.

A method shell can contain a special function main. It is a main method where the whole program starts. In Figure 4, the main method makes an instance of WebPage and calls the popup method on it. Since this method shell contains a reviser for WebPage, the implementation of popup in the reviser is selected and executed. A popup window is not displayed.

When the program starts from the main method in a method shell S, it runs with the modifications by the revisers contained in S unless the program contains link declarations mentioned later. For clarity, if a program is running with the modifications by a method shell S, we call S *the current context*. Note

```
1  methodshell viewer;
2  include renderer;
3
4  class Viewer {
5    void check(File f) {
6      if (isConfidential(f))
7        new WebPage().popup("<b>Confidential</b>");
8    }
9      ...
10 }
11 revise WebPage {
12   Border getBorder() {
13     // return a decorated window border
14   }
15 }
```

Figure 5. The viewer method shell

that all revisers contained in the same method shell are applied together to classes in that method shell. If multiple revisers share the same target, they are applied in the precedence order given by the programmer. For backward compatibility, our extended Java language allows classes in a source file without a methodshell declaration. Such classes belong to a special method shell that are implicitly included by any method shell. We call this special method shell *the global context*.

Link declarations

In the previous section, the library l_1 was also used by the audited-viewer library l_3. Figure 5 shows one of the source files of the library l_3 after being reimplemented with method shells. It contains a class and a reviser as well as the include declaration for including the renderer method-shell. This reimplementation of l_3 will work correctly if it is used independently. However, if we define a new method shell that naively includes both l_2 and l_3, the customizations of l_1 by the revisers in the two libraries l_2 and l_3 will conflict as we already saw in the previous section.

To address this conflict, the method shells provide a link declaration so that programmers can deal with a method shell as a black box. Here, being a black box means that the mere *users* of a method shell are not aware of its internals: which sub method-shells are included and how they are customized. For the developers who customize a method shell, it is still a gray-box; its internals are partly visible and customizable through a public interface. In large-scale applications, we believe that this sense of being a black box and/or a gray box would be necessary. It would be error-prone to construct such a large application by combining only gray-box libraries while manually avoiding conflicts.

The method shell linked by a link declaration is not included but the classes and the revisers in that method shell become visible. See Figure 6. This source file belongs to the application method-shell and it includes the browser method-shell by the include declaration. Since the third line is a link declaration, the application method-shell does not include the viewer method shell. However, the main method in the application method-shell can refer to the Viewer class, which belongs to the viewer method-shell.

```
1  methodshell application;
2  include browser;
3  link viewer;
4
5  public static void main(String[] args) {
6    new WebPage().popup();
7    new Viewer().check(new File("secret.txt");
8  }
```

Figure 6. The application method-shell and the link declaration

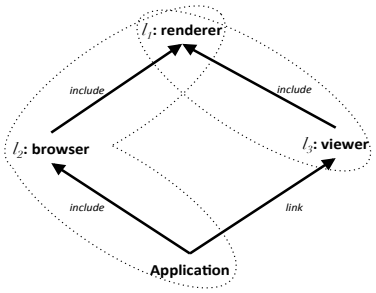

Figure 7. The method shell commonly shared

A unique feature of link declarations is the current context during the execution of the code in the linked method-shell. While a method implementation contained in the linked method-shell is executed, the current context is set to that linked method-shell. For example, when the main method in Figure 6 calls the check method, since the implementation of check is in the viewer method-shell, the current context is changed from the application method-shell to the viewer method-shell. During the execution of the check method, therefore, only the revisers in the viewer method-shell are effective. The revisers in the application method-shell are not effective. The reviser for the WebPage class in the browser method-shell, which is included by application, is not effective and thus the check method can execute the original implementation of the popup method. The current context is switched back to the application method-shell when the execution of the check method finishes.

A link declaration allows a program to execute a method in a space separated from other modules'. In our example scenario, the original implementation of popup method is in the l_1 library, or the renderer method-shell. It is included in the application method-shell through the library l_2 and l_3, or the browser method-shell and the viewer method-shell, respectively. The problem is that popup is modified differently by the paths through l_2 and l_3 and our solution is to make two versions of popup for each path. One is for the path from application to l_2 and l_1 while the other is for the path from l_3 to l_1. The former version is modified by using the application method-shell as the current context while the latter is by using the viewer method-shell. The two versions are switched when a method implementation in the linked method shell is invoked. This problem and the solution are similar to the diamond inheritance problem [14] and its solution.

4 Semantics and Implementation

This section presents the formal semantics of method shells. It also presents the sketch of the implementation technique of method shells. This technique was used to develop a prototype compiler of our extended Java, which supports method shells. This compiler was developed by using the JastAddJ framework [15].

4.1 Syntax

We first present a simple calculus for the formalization. It is an extension of Featherweight Java (FJ) [16] and GluonFJ [4]. The syntax is given as follows:

SL	::= `methodshell` S; $\overline{\text{IL}}$ $\overline{\text{LL}}$ (CL $\|\|$ RV) $*$ MF	method-shell declaration
IL	::= `include` S;	include declaration
LL	::= `link` S;	link declaration
CL	::= `class` C `extends` C{$\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$}	class declaration
RV	::= `revise` C{$\overline{\text{M}}$}	reviser declaration
M	::= C m($\overline{\text{C}}$ $\overline{\text{x}}$){`return` e;}	method declaration
MF	::= `void main`(){`return` e;}	main function declaration
e	::= x | e.f | e.m($\overline{\text{e}}$) | `new` C($\overline{\text{e}}$) | e `in` S; S_C	expressions
v	::= `new` C($\overline{\text{v}}$)	values

The metavariables S and T range over method shell names; B, C, and D range over class names; f range over field names; m ranges over method names; x ranges over parameter names; K ranges over constructor declarations; v and w range over values. The syntax of K and body is not shown here but it conforms to FJ. In this syntax, we use an overline to represent a sequence. For example $\overline{\text{x}}$ equals to "$\text{x}_1, \text{x}_2, ..., \text{x}_\text{n}$" and $\overline{\text{C}}$ $\overline{\text{f}}$ equals to "C_1 f_1, C_2 $\text{f}_2, ..., \text{C}_\text{n}$ f_n".

SL is a method shell. It consists of its name, include declarations, link declarations, class declarations CLs and reviser declarations RVs, and a main-function declaration MF. A class declaration CL consists of its name, its super class, fields, a constructor, and methods. A reviser declaration RV consists of its name and methods. A reviser cannot have a field. An expression e may take a new form e `in` S; S_C, which is used to mark which method shell e originates from and the current contexts in the operational semantics.

We denote the class and reviser table by CRT. It is a mapping from a pair of a method shell S and a class name C to a class declaration CL or a reviser declaration RV. A program is a pair of CRT and a method-shell name. The program execution starts from the main function included in the method shell with that name.

4.2 Lookup semantics

The reduction relation is of the form $\text{S}; \text{S}_\text{C} \vdash \text{e} \rightarrow \text{e}'$, reading "expression e reduces to expression e′ in one step in a method shell S and the current context S_C. If a program starts from the main function in a method shell S, then the

program execution is to reduce its expression e in the method shell S and the current context S. Most reduction rules are given in a straightforward manner from FJ's and GluonFJ's. Interesting rules are the followings:

$$\frac{\mathtt{T;T_C}\vdash \mathtt{e_0}\longrightarrow \mathtt{e_0}'}{\mathtt{S;S_C}\vdash \mathtt{e_0\ in\ T,T_C}\longrightarrow \mathtt{e_0}'\ \mathtt{in\ T;T_C}} \quad \text{(R-In)}$$

$$\frac{\mathtt{mbody(m,C,S,S_C)}=\ \bar{\mathtt{x}}.\mathtt{e_0}\ \mathtt{in\ T;T_C}}{\mathtt{S;S_C}\vdash \mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{w}})\longrightarrow([\bar{\mathtt{w}}/\bar{\mathtt{x}},\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e_0})\ \mathtt{in\ T;T_C}} \quad \text{(R-Invk)}$$

The first rule is straightforward. $\mathsf{e_0}$ is reduced in $\mathsf{T};\mathsf{T_C}$ although $\mathsf{S};\mathsf{S_C}$ are given. The second rule is for method invocation. Unlike FJ's, a function to look up a method body, named *mbody*, takes four parameters. It looks up a method body by referring to the method name m, the class of the target object C, the method shell S that the expression originates from, and the current context $\mathsf{S_C}$. Both S and $\mathsf{S_C}$ are ones at the caller-side. If a method body e with parameters $\bar{\mathsf{x}}$ is found in a method shell T and the new current context is set to $\mathsf{T_C}$, then the method body is executed with the arguments $\bar{\mathsf{w}}$ in T and $\mathsf{T_C}$.

The definition of *mbody* is presented in Figure 8. $mbody(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ returns the body of m called on the C class from the method shell S with the current context $\mathsf{S_C}$, written $\bar{\mathsf{x}}.\mathsf{e}\ \mathsf{in}\ \mathsf{T};\mathsf{T_C}$, where $\bar{\mathsf{x}}$ are parameters, e is the method body, T stands for the method shell where the body is found, and $\mathsf{T_C}$ stands for the current context used to execute the body.

mbody uses a few auxiliary functions. $includings(\mathsf{S})$ returns a set of method shells directly included by S. $linked\text{-}shells(\mathsf{S})$ returns a set of method shells linked by S. $mbodyshell(\mathsf{m},\mathsf{C},\mathsf{S})$ is a function to search the method shell S. It returns the body of method m in class C found in S. It first searches the method bodies directly contained in S and then recursively searches ones in method shells included by S. Finally, $mbodyglobal(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ searches the global context, which contains classes in source files without methodshell declarations. The global context is a special method shell implicitly included by any method shell. In Figure 8, Global stands for the global context. Note that, if the body of m in the class C is not found in the global context, *mbodyglobal* searches the method bodies declared in a super class of C by recursively calling *mbody*.

$mbody(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ searches in the following order. First, it searches the method shells *linked* by S. If a method body m is found, the new current context is set to the linked method shell where the body is found. Otherwise, *mbody* searches the current context $\mathsf{S_C}$. Note that it does not search the method shell S, which the method-call expression originates from. S is used only for obtaining the linked method shells searched at the first step. If a method body is not found in either the linked method shells or the current context, then *mbody* searches the global context. Finally, if a method body directly declared in the class C is not found in any method shells, *mbody* looks up a method body declared in a super class of C. The current context does not change except the first step.

4.3 Implementation

Our prototype compiler transforms a program using method shells into plain Java program, which is then compiled into Java bytecode. During the trans-

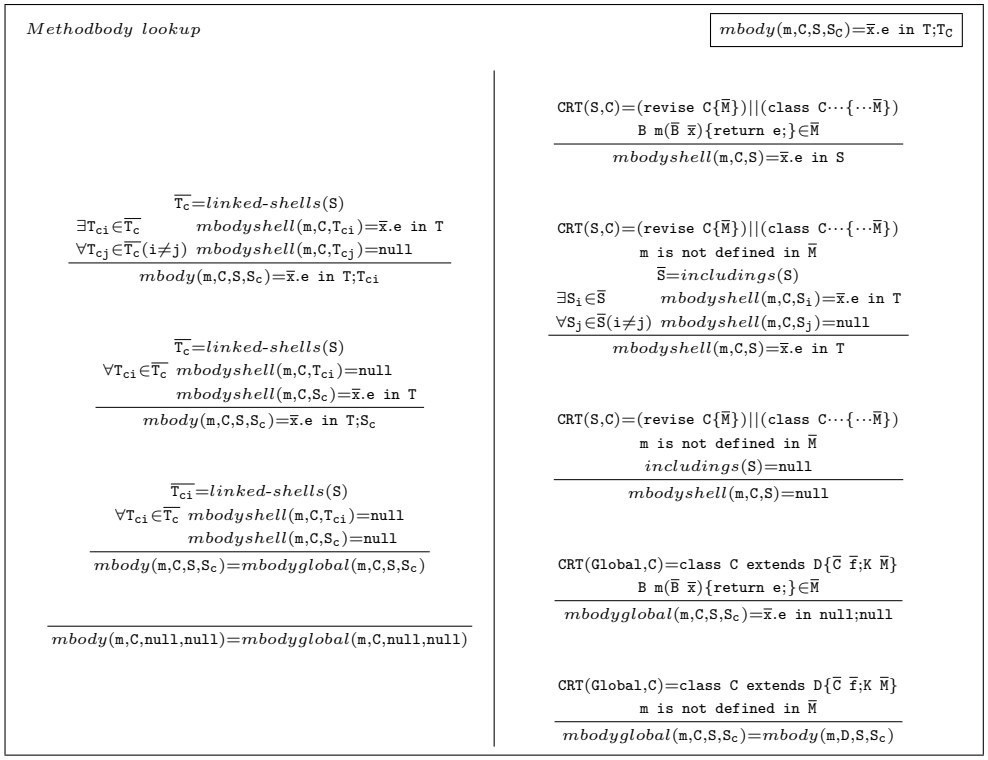

Figure 8. A function to look up a method body

formation, the methods in revisers are copied into the declaration of the target class. If the method already exists in the target class, the method copied from the reviser substitutes the original one. Our prototype compiler/language has not supported a mechanism to invoke the overridden implementation of a method by a call on super.

However, if multiple revisers in different method shells modify the same method, the naive approach above will not work. Our compiler hence copies a method implementation after renaming the method into the name mangled from the original method name and the method-shell name. A method m declared in a reviser for a class C is copied into the declaration of C after the method name is changed into the name $\mathsf{m_S}$ mangled from the method name m and the method shell S, which the reviser is contained in.

When a method m is called on an instance of a class C, the appropriate implementation is selected among the available versions $\mathsf{m_{S1}}, \mathsf{m_{S2}}, \mathsf{m_{S3}}, \ldots$ for each method shell $\mathsf{S_i}$. According to the semantics we showed above, the selection depends on the method shell that the method-call expression originates from and the current context. The former one is statically determined but the latter one is not. Thus, a naive implementation will have to check the latter one at runtime for method dispatch. This will cause a runtime penalty.

To minimize runtime penalties due to method shells, our prototype compiler duplicates a method implementation for different current contexts and transforms the method body to customize. Therefore, a method $\mathsf{m_{Si}}$ is duplicated into $\mathsf{m_{Si\cdot T1}}, \mathsf{m_{Si\cdot T2}}, \mathsf{m_{Si\cdot T3}}, ...$, where $\mathsf{T_i}$ is a current context. The body of the method $\mathsf{m_{Si\cdot Tj}}$ is transformed for $\mathsf{S_i}$ and $\mathsf{T_j}$. The transformation for a method shell S and a current context $\mathsf{S_C}$ under type environment Γ is written $\mathsf{S};\mathsf{S_C};\Gamma \vdash \mathsf{e} \Longrightarrow \mathsf{e'}$. For most expressions, the transformation is trivial. Only method calls must be changed:

$$\frac{\mathtt{S};\mathtt{S_C};\Gamma\vdash \mathtt{e_0}\Longrightarrow \mathtt{e_0}' \quad \mathtt{S};\mathtt{S_C};\Gamma\vdash \bar{\mathtt{e}}\Longrightarrow \bar{\mathtt{e}}' \quad \mathtt{S};\mathtt{S_C};\Gamma\vdash \mathtt{e_0}:\mathtt{C} \quad mname(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m}'}{\mathtt{S};\mathtt{S_C};\Gamma\vdash \mathtt{e_0.m}(\bar{\mathtt{e}})\Longrightarrow \mathtt{e_0}'.\mathtt{m}'(\bar{\mathtt{e}}')}$$

Here, $\mathsf{S};\mathsf{S_C};\Gamma \vdash \mathsf{e_0} : \mathsf{C}$ is read "expression $\mathsf{e_0}$ is given type C under type environment Γ." In other words, the static type of $\mathsf{e_0}$ is C. *mname* is a function to look up a method like *mbody*. It is defined as following:

$$\frac{mbody(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\bar{\mathtt{x}}.\mathtt{e\ in\ null;null}}{mname(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m}}$$

$$\frac{mbody(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\bar{\mathtt{x}}.\mathtt{e\ in\ T;T_C}}{mname(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m_{T\cdot T_C}}}$$

If the method implementation is selected from the global context, the method name is not changed during transformation.

Finally, the body of the main function is transformed so that the appropriate version of methods will be called. If the program starts with the main function in a method shell S, then the body is transformed for S and S.

5 Related Work

In the context of AOP, a number of researchers have been studying conflicts of advices, or *aspect interference*. Aksit et al. proposed a mechanism for detecting aspect interference by using graph transformation [6]. Several linguistic constructs have been proposed to resolve the interference. Douence et al. proposed a new composition operator of aspects [12, 13]. It allows programmers to describe a safely-composable aspect. Airia provided a new kind of around advise called *resolvers* for resolving the interference [17]. A uniqueness of our work is that we have designed a language construct specialized for a specific use-case scenario where aspects are used for building a custom library to be used as a black box.

AOP and destructive class extensions can be regarded as a special case of virtual classes [18, 19], where all base-level classes are implicitly contained as virtual classes in a single enclosing class and all aspects (or corresponding constructs like revisers) are in a subclass of that enclosing class, if the differences in how to specify the target base-level classes are ignored (the targets in AOP are specified by pointcuts while ones in virtual classes are by the super-class names). Our method shells are an approach to introduce a scope mechanism into destructive class extensions. In the analogy above, this approach allows programmers to use more than one enclosing classes in programming with destructive class extensions (*i.e.* AOP) as they can do in programming with virtual

classes. Therefore, the resulting language mechanism is similar to ones for virtual classes but it still has some unique features since it is originated from destructive class extensions. For example, in method shells, programmers are less aware of the existence of enclosing classes. When a program refers to a class contained in a different enclosing class, it does not have to explicitly specify the name of that different enclosing class. That class is implicitly selected by the link declaration so that the obliviousness property [20] is somewhat preserved. Another example is that, in method shells, the method selected for a method call on the same target object changes depending on the caller's contexts. Furthermore, method shells allow multiple shells to be included at the same time like mixing.

In Newspeak [21], all class names are virtual and a subclass can override them. This overriding mechanism corresponds to the include declaration in method shells. On the other hand, Newspeak does not provide a mechanism corresponding to the link declaration, with which the method-lookup context is changed after a method in the linked method-shell is selected. Although Newspeak is dynamically typed, method shells are statically typed and we present a technique for reducing their method-lookup overhead.

The idea of method lookup depending on runtime contexts is found not only in our method shells but also other languages such as JPred [22]. JPred supports predicate dispatch, with which a method is selected by referring to calling contexts such as method arguments and caller objects.

Us [23] allows programmers subjectivity-based programming. In Us, every method call explicitly takes a method-lookup context called a perspective. In method shells, every call does not take such a context, which is declared by a include or link declaration at the beginning of the source file.

Context-oriented programming (COP) [24] is a paradigm where a class definition can be changed depending on the contexts during runtime. A class declaration is divided into multiple parts, which are called layers, and different layers may contain different implementations of the same method. Layers are dynamically switched by with and without clauses. Within the with clause, the specified layer is effective while in the without clause it is ineffective. A layer provides the same ability for destructive class extension as our revisers but a layer must be contained in the declaration of the target class although a reviser is described separately from the target class. Despite this difference, method shells and COP share the idea of changing class definition to fit the current context.

However, the with and without clauses are not adequate for addressing the problem mentioned in this paper. Programmers in COP languages cannot deal with a layer as a black box. They have to understand the dependency among all layers and classes used in their programs. Figure 9 shows a program that is equivalent to the program in Figure 6 but is written in ContextJ, a COP extension of Java [25]. This program starts from the main method in the App class. Since it uses the renderer, the browser, and the viewer, it first activates all the three by with (line 10 to 12). However, while the check method in Viewer is executed, the browser layer must be deactivated since it needs a popup window. In Figure 9, the browser layer is deactivated within the body of the check method

```
1  class WebPage{
2    layer(renderer){
3      void popup(HTML text){
4        // show a popup window
5      }
6      void onClick(Mouse m){
7        URL url = m.getURL();
8        if(isPopup()) popup(url);
9          ...
10       }
11     }
12   }
13   layer(browser){
14     void popup(HTML text){
15       warning("disabled");
16     }
17   }
18 }
19 class Viewer{
20   layer(viewer){
21     void check(File f){
22       if(isCirfidential(f)){
23         without(browser){
24           new WebPage.popup
25             ("<b>Confidencial</b>");
26         }
27       }
28     }
29   }
30 }
31 }
```

```
1  class App{
2    void run(){
3      new WebPage().popup();
4            // a popup is disabled
5      new Viewer().check
6        (new File("secret.txt");
7            // a popup is needed
8    }
9    public void main(String[] args){
10     with(renderer){
11       with(browser){
12         with(viewer){
13           new App.run();
14         }
15       }
16     }
17   }
18 }
```

Figure 9. A program in ContextJ

by without (line 23) but this requires the programmer of Viewer to be aware of the browser layer, which might be independently developed from the Viewer. Another approach is to deactivate the browser layer within the body of the run method in App, for example, just before calling the check method at line 5. However, this requires the programmer of App to be aware that the browser layer must be deactivated while a Viewer is running. The programmer cannot deal with Viewer as a black box. A recent version of ContextJ supports Reflection API [26] and hence the problem above is fairly overcome. In this language, a program can obtain all the layers currently activated and then deactivate them. However, unlike method shells, the programmer still has to be aware of unnecessary layers and explicitly deactivate them.

Classboxes [27, 28] are a module system that also provides a scoping mechanism for destructive class extensions. In Classbox/J, related classes are modularized into a module called a classbox. It can include other classboxes and partly modify them by refine, which corresponds to a reviser in our language. However, Classboxes do not provide a mechanism corresponding to our link declarations and thus they cannot handle the scenario shown in this paper. If the library l_1 is included through multiple paths, all the refines on the paths are applied together.

In Java, every class loader has its own name space. Hence, distinct implementations of the same class can coexist in one program if they are loaded into different class loaders. This is useful to partly address the problem discussed in this paper but moving an object beyond the boundary between class loaders is significantly restricted. In method shells, such restriction known as *the ver-*

sion barrier is not applied. To enable such movement between class loaders, the Java virtual machine has to be modified and support a mechanism like sister namespaces [29].

We have already proposed method shelters, which is a mechanism similar to method shells [30]. Although the two mechanisms share the same approach, method shelters are for a dynamically typed language Ruby. The destructive class extension in Ruby is performed by open classes, which is different from revisers we used in this paper. Furthermore, the design of method shelters is more complicated than that of method shells. A method shelter, which is a unit of module, is divided into an exposed chamber and a hidden chamber. Programmers have to carefully choose which chamber a reviser should be placed to control its visibility. On the other hand, a method shell is simpler but equivalently expressive; it is not divided into smaller containers but a single container. Programmers can intuitively control the visibility of revisers by choosing either an include declaration or a link declaration.

6 Conclusion

We proposed method shells, which are a module system for avoiding conflicts on destructive class extensions. The destructive class extensions are mechanisms for modifying class definitions from a separate module, which include aspects in AspectJ, open classes in Ruby, and revisers in GluonJ. A method shell is a module consisting of classes and revisers. It can include other method shells and the revisers in the included method shells are applied together as well as the revisers in the method shells including them. A unique feature is that a method shell can link to other method shells. The code included in the linked method shells can be invoked but it is executed in a context where only the revisers in the linked method shell are effective. Thus, a linked method shell is dealt with as a black box.

Our contribution is to propose a mechanism for avoiding conflicts on destructive class extensions when we use the extensions for customizing a class library or a framework. The resulting library or framework after customization can be dealt with a black box. The main idea is link declarations. The language automatically switches effective revisers when the thread of control crosses over to a linked method shell. Showing the formal semantics and implementation strategy of method shells is also contribution.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP'97 – Object-Oriented Programming. LNCS 1241, Springer (1997) 220–242
2. : Ruby programming language. http://www.ruby-lang.org/
3. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING **30**(6) (2004) 2004

4. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '10, New York, NY, USA, ACM (2010) 539–554
5. Dinkelaker, T., Mezini, M., Bockisch, C.: The art of the meta-aspect protocol. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. AOSD '09, New York, NY, USA, ACM (2009) 51–62
6. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. AOSD '09, New York, NY, USA, ACM (2009) 39–50
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming. ECOOP '01, London, UK, UK, Springer-Verlag (2001) 327–353
8. Millstein, T., Reay, M., Chambers, C.: Relaxed multijava: balancing extensibility and modular typechecking. In: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications. OOPSLA '03, New York, NY, USA, ACM (2003) 224–240
9. Mcdirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '11, ACM Press (2001) 211–222
10. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proc. of the Int'l Conf. on Software Engineering (ICSE'05), ACM Press (2005) 49–58
11. Steimann, F.: The paradoxical success of aspect-oriented programming. ACM SIGPLAN Notices **41**(10) (2006) 481–497
12. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering. GPCE '02, London, UK, UK, Springer-Verlag (2002) 173–188
13. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Proceedings of the 3rd international conference on Aspect-oriented software development. AOSD '04, New York, NY, USA, ACM (2004) 141–150
14. Malayeri, D., Aldrich, J.: Cz: multiple inheritance without diamonds. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. OOPSLA '09, New York, NY, USA, ACM (2009) 21–40
15. Ekman, T., Hedin, G.: The jastadd extensible java compiler. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. OOPSLA '07, New York, NY, USA, ACM (2007) 1–18
16. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. **23**(3) (May 2001) 396–450
17. Takeyama, F., Chiba, S.: An advice for advice composition in aspectj. In: Proceedings of the 9th international conference on Software Composition. SC'10, Berlin, Heidelberg, Springer-Verlag (2010) 122–137
18. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Abstraction mechanisms in the beta programming language. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '83, New York, NY, USA, ACM (1983) 285–298

19. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: Conference proceedings on Object-oriented programming systems, languages and applications. OOPSLA '89, New York, NY, USA, ACM (1989) 397–406
20. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley (2005) 21–35
21. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In: Proceedings of the 24th European conference on Object-oriented programming. ECOOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 405–428
22. Millstein, T.: Practical predicate dispatch. In: Proc. of ACM OOPSLA, ACM (2004) 345–364
23. Smith, R.B., Ungar, D., Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS **2** (1996) 161–178
24. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008, ETH Zurich **7**(3) (2008) 125–151
25. Appeltauer, M., Hirschfeld, R., Masuhara, H.: Improving the development of context-dependent java applications with contextj. In: International Workshop on Context-Oriented Programming. COP '09, New York, NY, USA, ACM (2009) 5:1–5:5
26. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: Contextj: Context-oriented programming with java. Information and Media Technologies **6**(2) (2011) 399–419
27. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions. In: Computer Languages, Systems and Structures. (2005)
28. Bergel, A.: Classbox/j: Controlling the scope of change in java. In: In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), ACM Press (2005) 177–189
29. Sato, Y., Chiba, S.: Loosely-separated "sister" namespaces in Java. In: Proc. of the 19th European conference on Object-Oriented Programming (ECOOP'05). LNCS 3586, Springer-Verlag (2005) 49–70
30. Akai, S., Chiba, S.: Method shelters: avoiding conflicts among class extensions caused by local rebinding. In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development. AOSD '12, New York, NY, USA, ACM (2012) 131–142