Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct

YungYu Zhuang
The University of Tokyo
zhuang@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
chiba@acm.org

ABSTRACT

To simplify the constructs that programmers have to learn for using paradigms, we extend methods to a new language construct, a *method slot*, to support both the event-handler paradigm and the aspect paradigm. A *method slot* is an object's property that can keep more than one function closure and be called like a method. We also propose a Java-based language, *DominoJ*, which replaces methods in Java with *method slots*, and explain the behavior of *method slots* and the operators. Then we evaluate the coverage of expressive ability of *method slots* by comparing *DominoJ* with other languages in detail. The feasibility of *method slots* is shown as well by implementing a prototype compiler and running a preliminary microbenchmark for it.

Categories and Subject Descriptors

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

Keywords

aspect-oriented programming; event-driven programming

1. INTRODUCTION

The event-handler paradigm has been recognized as a useful mechanism in a number of domains such as user interface, embedded systems, databases [32], and distributed programming. The basic idea of the event-handler paradigm is to register an action that is automatically executed when something happens. At first it was introduced as techniques and libraries [7, 28, 26] rather than supported at language level. Recently supporting it at language level is a trend since a technique such as the Observer pattern [7] cannot satisfy programmers' need. The code for event triggers and observer management scatters everywhere. To address the issues, supporting events by a language construct is proposed in a number of languages [17, 3, 22, 6, 13, 9]. Implicit

invocation languages [8] might be classified into this category.

On the other hand, the aspect paradigm [14] is proposed to resolve crosscutting concerns, which cannot be modularized by existing paradigms such as object orientation. Although the aspect paradigm and the event-handler paradigm are designed for different scenarios, the constructs introduced for them are similar and can work as each other from a certain point of view.

In order to simplify the language constructs programmers have to learn, we borrow the idea of slots from Self [30] to extend the method paradigm in Java. In Self, everything is a slot that may contain either a value or a method. In other words, there is no difference between fields and methods since a method is also an object and thus can be kept in a field. We extend the slot and bring it to Java-like languages by proposing a new language construct named *method slot*. A *method slot* is an object's property which can keep more than one closure. We also present a Java-based language named *DominoJ*, where all methods in plain Java are replaced with *method slots*, to support both the event-handler paradigm and the aspect paradigm.

Our contributions presented in this paper are two-fold. First, we propose a new language construct, a *method slot*, to extend the method paradigm. Second, we introduce *method slots* to a Java-based language named *DominoJ*, and demonstrate how to use for the event-handler paradigm and the aspect paradigm.

2. MOTIVATION

With the evolution of software, more and more programming paradigms are developed for various situations. During programmers' life, they are always learning new paradigms and thinking about which ones are most suitable for the job at hand. For example, the event-handler paradigm is widely adopted by GUI frameworks [31, 18, 24]. When we write GUI programs with modern GUI libraries, we usually have to write a number of handlers for different types of events. The AWT [24] of Java is a typical example. If we want to do something for mouse events occurring on a button, we have to prepare a mouse listener that contains handler methods for those mouse events, and register the listener to the specified button object. A GUI program can be regarded as a composite of visual components, events, and handlers. The visual components and handlers are main logic, and events are used for connecting them. Indeed we have been familiar with using the event-handler paradigm for GUI programs, but it is far from our first "hello world"

program. We are told to carefully consider the total execution order when users' input is read. If the event-handler paradigm is used, we can focus on the reaction to users' input rather than the order of users' input. Whether the mouse is clicked first or not does not matter. Another example is the aspect paradigm. Aspect-oriented programming is developed to modularize crosscutting concerns such as logging, which cannot be modularized by using only object-oriented programming. With the aspect paradigm, crosscutting concerns can be gathered up in an aspect by advices. At the same time, programmers cannot check only one place for understanding the behavior of a method call since advices in other places are possibly woven together. It also takes effort to get familiar with the aspect paradigm since it is quite different from our other programming experience.

To use a paradigm, just learning its concept is not enough. After programmers got the idea of a paradigm, they still have to learn new language constructs for the paradigm. Some paradigms like the aspect paradigm are supported with dedicated language constructs since the beginning because they cannot be represented well by existing syntax. On the other hand, although other paradigms like the event-handler paradigm have been introduced at library level for a long time, there are still good reasons for re-introducing them with direct support at language level [17, 22, 9]. Maybe one reason is that events are complicated in particular when we are not users but designers of a library. Besides GUI libraries, the event-handler paradigm is also implemented in a number of libraries for several domains such as simple API for XML [29] and asynchronous socket programming. Some techniques such as the Observer pattern [7] used in those libraries cannot satisfy the needs of defining events and tend to cause code scattering and tangling. Supporting paradigms by language constructs is a trend since it makes code more clear and reusable. Furthermore, a language supported paradigm may have associated static checks.

However, learning language constructs for a paradigm is never easy, especially for powerful paradigms like the aspect paradigm. Moreover, the syntax is usually hard to share with other paradigms. Even though programmers got familiar with the language constructs for a paradigm, they still have to learn new ones for another paradigm from the beginning. Given that all language constructs we need can be put into a language together, they look too complex and redundant. How to pick up the best language to implement a program with all required paradigms is always a difficult issue. This motivates us to find out an easy, simple, and generic language construct supporting multiple paradigms.

If we look into the language constructs for the event-handler paradigm and the aspect paradigm, there is a notable similarity between them. Both of them introduce a way to define the effect of calling specified methods. The differences are where the reactions are and what the reactions are targeted at. Listing 1 is a piece of code in EScala[1] [9], which is a typical event mechanism, showing how to define a **moved** event for the **setPosition** method in the **Shape** class. Here we specify that **refresh** method on a **Display** object should be executed after **setPosition** method is executed. As shown in Listing 2, the reaction can also be represented in AspectJ [25], the most well-known aspect-oriented language.

By comparing the two pieces of code, we can find that

Listing 1: Defining a reaction in EScala

```
1  class Display() {
2    def refresh() {
3      System.out.println("display is refreshed.")
4    }
5  }
6  class Shape(d: Display) {
7    var left = 0; var top = 0
8    def setPosition(x: Int, y: Int) {
9      left = x; top = y
10   }
11   evt moved[Unit] = afterExec(setPosition)
12   moved += d.refresh
13 }
14 object Test {
15   def main(args: Array[String]) {
16     val d = new Display()
17     val s = new Shape(d)
18     s.setPosition(0, 0)
19   }
20 }
```

Listing 2: Defining a reaction in AspectJ

```
1  public class Display {
2    public static void refresh() {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11 }
12 public aspect UpdateDisplay {
13   after() returning:
14    execution(void Shape.setPosition(int, int)) {
15        Display.refresh();
16   }
17 }
18 public class Test {
19   public static void main(String[] args) {
20     Shape s = new Shape();
21     s.setPosition(0, 0);
22   }
23 }
```

pointcuts are close to events and advices can work as the `+=` operator for handlers. They both refresh the display when the specified method is executed, but there is a significant difference between them. In EScala version, one Display object is mapped to one Shape object and the refresh action is performed within the Shape object. On the other hand, in AspectJ version there is only one Display object in the whole program and the refresh action is in UpdateDisplay, which is completely separated from Display and Shape. From the viewpoint of the event-handler paradigm, such behavior is an interaction between objects, so the reaction is defined inside the class and targeted at object instances; the encapsulation is preserved. From the viewpoint of the aspect paradigm, it is important to extract the reaction for the obliviousness since it is a different concern cutting across several classes. So the reactions are grouped into a separate construct and targeted at the class. Although the two paradigms are developed from different points of view, the language constructs used for them are quite similar. Furthermore, both the two paradigms depend on the most basic paradigm, the method paradigm, since both events and pointcuts cause the exe-
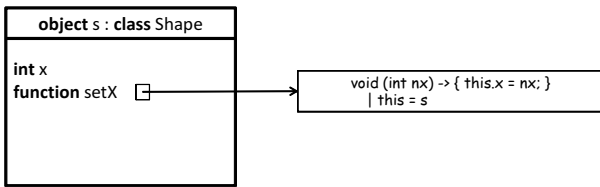
Figure 1: In JavaScript, both an integer and a function are fields on an object
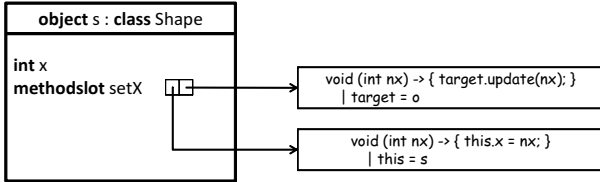

Figure 2: A method slot is an extended field that can keep more than one function closure

cution of a method-like construct. This observation led us to extend the method paradigm to support both the event-handler paradigm and the aspect paradigm. To a programmer, there are too many similar language constructs for different paradigms to learn, so we assume that the integration and simplification are always worth doing.

3. DOMINOJ

We extend methods to a new language construct named a *method slot*, to support methods, events, and advices. We also show our prototype language named *DominoJ*, which is a Java-based language supporting *method slots* and fully compatible with plain Java.

3.1 Method slots

Although methods and fields are different constructs in several languages such as C++ and Java, there is no difference between them in other languages like JavaScript. In JavaScript, a method on an object (strictly speaking, a function closure) is kept and used as other fields. Figure 1 shows a Shape object s, which has two fields: an integer field named x and a function field named setX. We use the following notation to represent a closure:
<*return type*> (<*parameter list*>) -> { <*statements*> }
 | <*variable binding list*>
where the variable binding list binds non-local variables in the closure. The value stored in field setX is a function closure whose return type and parameter type are void and (int), respectively. The variable this used in the closure is bound to s given by the execution context. When we query the field by s.setX, the function closure is returned. When we call the field by s.setX(10), the function closure is executed.

We extend this field in JavaScript to keep an array of function closures rather than just one function closure. As shown in Figure 2, the extended field named a *method slot* can keep more than one function closure. DominoJ replaces a method with a method slot in plain Java. All method-like declarations and calls are referred to method slots. A method slot is a closure array and is an object's property

Listing 3: A sample code in DominoJ

```
1   public class Shape {
2     private int x;
3     public void setX(int nx) {
4       // default closure
5       this.x = nx;
6     }
7   }
8   public class Observer {
9     private int count;
10    public void update(int i) {
11      this.count++;
12    }
13    public static void main(String[] args) {
14      Shape s = new Shape();
15      Observer o = new Observer();
16      s.setX += o.update;
17      s.setX(10);
18    }
19  }
```

like a field. Like functions or other fields, method slots are typed and statically specified when they are declared. The type of method slot includes its return type and parameter types. All closures in it must be declared with the same type.

Listing 3 shows a piece of sample code in DominoJ. It looks like plain Java, but here setX is a method slot rather than a method. The syntax of method slot declaration is shown below:
<*modifier*>* <*return type*> <*identifier*>
  "(" <*parameter list*>? ")" <*throws*>?
  (<*default closure*> | ";")
The default closure is similar to the method body in Java except it is optional. The modifiers can be public, protected, or private for specifying the visibility of the method slot. This ensures that the access to the method slot can be controlled as the methods in plain Java. The modifier static can be specified as well. Such static method slots are kept on the class objects so can be referred using the class name like calling the static method in plain Java. The modifier abstract can also be used to specify that the method slot should be implemented by the subclasses. A method slot can be another kind of "abstract" by being declared without a default closure:

```
public void setX(int nx);
```

Unlike the modifier abstract, this declaration means that the method slot is an empty field and its behavior depends on the closures added to it later. In Listing 3, the method slot setX has a default closure, so the following function closure will be created and inserted into setX automatically when a Shape object, s, is instantiated:

```
void (int nx) -> { this.x = nx; }
  | this = s
```

Now there is only one closure in the method slot setX. If we add another closure to setX, the object may look like the s object in Figure 2. How to add such a closure to a method slot will be demonstrated in the next subsection.

A method slot can also be declared with the modifier final to specify that it cannot be overridden in the subclasses. Although fields are never overridden in either prototype-based

Listing 4: The algorithm of calling a method slot

```scheme
1  ; call a methodslot
2  (define (call-methodslot object slotname args)
3    (let* ((methodslot (get-field object slotname
4                                   (get-type args)))
5           (return_type (get-return-type methodslot)))
6      (let execute-closures
7        ((closures (get-closures methodslot))
8         ($retval (cond ((boolean? return_type) #f)
9                        ((number? return_type) 0)
10                        (else '()))))
11       (if (null? closures)
12           $retval
13           (let
14             (($retval (execute-a-closure (car closures) args)))
15             (execute-closures (cdr closures) $retval)))))))
```

Table 1: The four operators for method slots

Operator	Description
=	*add a new function closure and remove the others from the method slot.*
^=	*insert a new function closure at the beginning of the array.*
+=	*append a new function closure to the end of the array.*
-=	*remove function closures calling the method slot at the right-hand side.*

languages like JavaScript or class-based languages like Java, method slots can be overridden in subclasses. Declaring a method slot with the same signature overrides but not hides the one in the superclass. When a method slot is queried or called on an object, the overriding method slot is selected according to the actual type of the object. It is also possible to access the overridden method slot in the superclass through the keyword super. Note that method slots must be declared within a class and cannot be declared as local variables. Thus the usage of this and super in the default closure are the same as in a Java method, which refer to the owning class and its superclass, respectively. Constructors are method slots as well, and super() is allowed since it calls the overridden constructor.

When a method slot is called by () operator, the closures in it are executed in order. The arguments given to the method slot are also passed to its closures. The return value returned by the last closure is passed to the caller (if it is not the void type). A closure can use a keyword $retval to get the return value returned by the preceding closure in the method slot. If the closure is the first one in the method slot, $retval is given by a default value (0, false, or null). If the method slot is empty, the caller will get the default value and no exception is thrown. It is reasonable since the empty state is not abnormal for an array and just means that nothing should be done for the call at that time. The behavior of a method slot can be dynamically modified at runtime, while still statically typed and checked at compile time. How to call a method slot is described in Scheme as shown in Listing 4.

3.2 Operators for method slots

DominoJ provides four operators for manipulating the closures in a method slot: =, ^=, +=, and -=, as shown in Table 1. These operators are borrowed from C# and EScala, and are the only different syntax from Java. It is possible to add and remove a function closure to/from a method slot at runtime.

Their operands at both sides are method slots sharing the same type. Those operators except -= create a new function closure calling the method slot at the right-hand side, and add it to the method slot at the left-hand side. The method slot called by the function closure will get the same arguments which are given to the method slot owning the function closure. In other words, a reference to the method slot at the right-hand side is created and added to the method

slot at the left-hand side. The syntax of using the operators to bind two method slots is shown below:

$<expr>$".$"<methodslot>$
  $<operator>$ $<expr>$".$"<methodslot>$;

where $<expr>$ can be any Java expression returning an object, or a class name if the following $<methodslot>$ is static. When the binding statement is executed at runtime, the $<expr>$ at both sides will be evaluated according to current execution context and then given to the operator. In other words, the $<expr>$ at the right-hand side is also determined at the time of binding rather than the time of calling. The object returned by the $<expr>$ at the left-hand side helps to find out the method slot at the left-hand side, where we want to add or remove the new function closure. The object got by evaluating the $<expr>$ at the right-hand side is attached to the new function closure as a variable target, which is given to the new function closure along with the execution context at the time of calling. For example, the binding statement in Line 16 of Listing 3 creates a new function closure calling the method slot update on the object o by giving target = o, and appends it to the method slot setX on the object s.

```
void (int nx) -> { target.update(nx); }
  | target = o
```

Then the status of the s object will be the same as the one shown in Figure 2. When the slot setX on the object s is called as Line 17 in Listing 3, the default closure and the slot update on the object o are sequentially called with the same argument: 10. Note that all closures in a method slot get the same execution context, where this refers to the object owning the method slot, and therefore, the callee method slot in target must be accessible from the caller method slot in this. With proper modifiers, a method slot cannot call and be called without any limitation. The behavior avoids breaking the encapsulation in object-oriented programming.

The -= operator removes function closures calling the method slot at the right-hand side from the method slot at the left-hand side. It is also possible to remove the default closure from a slot by specifying the same method slots at both sides:

```
s.setX -= s.setX;
```

Operators manipulate the default closure only when the method slots at both sides are the same one, otherwise operators regard the right-hand side as a closure calling that method slot. Note that the default closure is never destroyed even when it is removed. The algorithms of the four operators are described in Scheme in Listing 5.

Listing 5: The algorithms of the four operators

```scheme
1  ; operator =
2  (define (assign-closure methodslot object slotname)
3    (let ((closure
4            `(call-methodslot ,object ,slotname args)))
5      (set-closures methodslot closure)))
6
7  ; operator ^=
8  (define (insert-closure methodslot object slotname)
9    (let ((closure
10           `(call-methodslot ,object ,slotname args)))
11     (set-closures
12       methodslot
13       (append closure (get-closures methodslot)))))
14
15 ; operator +=
16 (define (append-closure methodslot object slotname)
17   (let ((closure
18           `(call-methodslot ,object ,slotname args)))
19     (set-closures
20       methodslot
21       (append (get-closures methodslot) closure))))
22
23 ; operator -=
24 (define (remove-closure methodslot object slotname)
25   (let ((closure
26           `(call-methodslot ,object ,slotname args)))
27     (set-closures
28       methodslot
29       (remove (lambda (x) (equal? x closure))
30               (get-closures methodslot)))))
```

Listing 6: The algorithm of checking the types

```scheme
1  ; is same type
2  (define (same-type? l_methodslot r_methodslot)
3    (and (equal? (get-return-type l_methodslot)
4                 (get-return-type r_methodslot))
5         (equal? (get-parameter-types l_methodslot)
6                 (get-parameter-types r_methodslot))))
7
8  ; is generic type
9  (define (generic-type? l_methodslot r_methodslot)
10   (and (equal? (get-parameter-types r_methodslot)
11                "Object[]")
12        (if (equal? (get-return-type l_methodslot)
13                    "void")
14          (equal? (get-return-type r_methodslot)
15                  "void")
16          (equal? (get-return-type r_methodslot)
17                  "Object"))))
```

Listing 7: The algorithm of binding method slots

```scheme
1  ; bind methodslots by operators
2  (define (bind-methodslots
3            operator l_object l_slotname r_object r_slotname)
4    (let ((l_methodslots (get-fields l_object l_slotname))
5          (r_methodslots (get-fields r_object r_slotname)))
6      (for-each
7        (lambda (l_methodslot)
8          (for-each
9            (lambda (r_methodslot)
10             (if (or (same-type? l_methodslot
11                                 r_methodslot)
12                     (generic-type? l_methodslot
13                                    r_methodslot))
14               (cond ((equal? operator "=")
15                      (assign-closure l_methodslot
16                                      r_object
17                                      r_slotname))
18                     ((equal? operator "^=")
19                      (insert-closure l_methodslot
20                                      r_object
21                                      r_slotname))
22                     ((equal? operator "+=")
23                      (append-closure l_methodslot
24                                      r_object
25                                      r_slotname))
26                     ((equal? operator "-=")
27                      (remove-closure l_methodslot
28                                      r_object
29                                      r_slotname)))))
30           r_methodslots))
31       l_methodslots)))
```

Although a method slot at the right operand of the operators such as += must have the same type that the left operand has, there is an exception. If a method slot takes only one parameter of the Object[] type and its return type is Object or void, then it can be used as the right operand whatever the type of the method slot at the left operand is. Such a method slot can be used as a generic method slot. The type conversion when arguments are passed is implicitly performed. Listing 6 shows how to check the type of two method slots in Scheme.

DominoJ allows binding method slots to constructors by specifying class name instead of the object reference and giving the keyword constructor as the method slot at the left-hand side. For example,

```
Shape.constructor += Observer.init;
```

means that creating a closure calling the static method slot init on the class object Observer and appending to the con-

structor of Shape. Here the return type of init should be void, and the parameter types must be the same as the constructor. Note that the closures appended to the constructor cannot block the object creation. This design ensures that the clients will not get an unexpected object, but additional objects can be created and bound to the new object. For example, in the default closure of init, an instance of Observer can be created and its update can be bound to the method slot setX of the new Shape object. Using constructor at the right-hand side is not allowed.

Since Java supports method overloading, some readers might think the syntax of method slots have ambiguity but that is not true. For example, the following expression does not specify parameter types:

```
s.setX += o.update;
```

If setX and/or update are overloaded, += operator is applied to all possible combinations of setX and update. Suppose that there are setX(int), setX(String), update(int), and update(String). += operator adds update(int) to setX(int), update(String) to setX(String). If there is update(Object[]), it is added to both setX(int) and setX(String) since it is generic. It is possible to introduce additional syntax for selecting method slots by parameters, but the syntax will be more complicated. Listing 7 is the algorithm in Scheme for picking up and binding two method slots by operators.

Since a method slot may contain closures calling other method slots, a call to the method slot can be regarded as a sequence of method slot calls among objects. When a method slot is explicitly called by a statement in a certain default closure, the method slots bound to it by operators are implicitly called by DominoJ. In order to make method slots more flexible, DominoJ provides keywords to get the preceding objects in the call sequence. In the default clo-
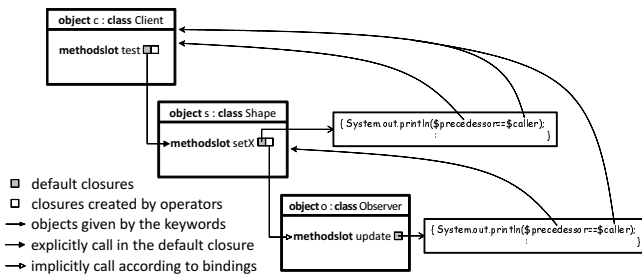
Figure 3: The keywords \$caller and \$predecessor

Figure 4: The average time per method call in Java and DominoJ

sure, i.e. the body of method slot declaration, the caller object can be got by keyword \$caller. It refers to the object where we start the call sequence by a statement. The predecessor object, in other words, the object owning the preceding method slot in the call sequence, can also be got by the keyword \$predecessor. It refers to the object owning the closure calling the current method slot whether explicitly or implicitly. Taking the example of Figure 2, suppose that we have a statement calling s.setX in the default closure of the method slot test in another class Client:

```
public class Client {
  public void test(Shape s) {
    s.setX(10);
  }
}
```

If test on an object instance of this class, for example c, is executed, the relationship between the objects c, s, and o can be described as shown in Figure 3. Note that calling other method slots explicitly by statements in the default closure of test, setX, or update will start separate call sequences. In Figure 3, using \$caller in the default closure of setX and update both returns the object c since there is only one caller in a call sequence. However, the predecessor objects of s and o are different. Using \$predecessor in the default closure of setX returns the object c, but using \$predecessor in the default closure of update returns the object s. Note that both the apparent types of \$caller and \$predecessor are Object because the caller and the predecessor are determined at runtime. If current method slot is called in a static method slot, \$caller or \$predecessor will return the class object properly. The special method call proceed in AspectJ is introduced in DominoJ as well. It calls the default closure of the preceding method slot. In Figure 3, calling proceed in the default closure of update on o will execute the default closure of setX on s since s.setX is the preceding method slot of o.update. If there is no preceding method slot for current one, calling proceed will raise an exception.

4. EVALUATION

To show the feasibility of DominoJ and measure the overheads caused by method slots, we implemented a prototype compiler[2] of DominoJ built on top of JastAddJ [27]. The source code in DominoJ can be compiled into Java bytecode and run by Java virtual machine. In the following microbenchmark, the standard library is directly used without

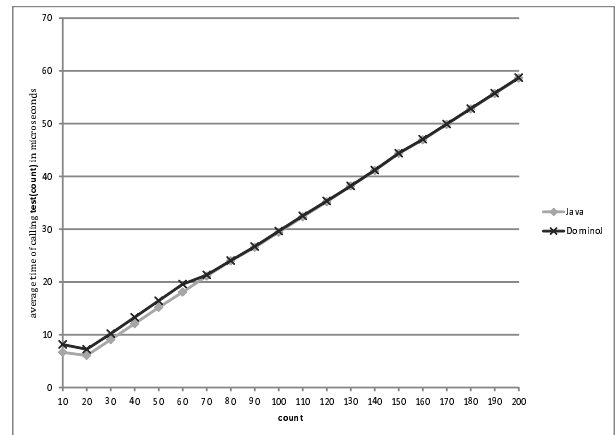---

[2]The prototype compiler of DominoJ is available from `http://www.csg.ci.i.u-tokyo.ac.jp/projects/dominoj/`

recompilation due to the performance concern. All methods in the standard library can be called as method slots which have only the default closure, but cannot be modified by the operators.

4.1 Microbenchmark

In order to measure the overheads of method slots, we executed a simple program and compared the average time per method call in DominoJ and in plain Java. The program calls a method that calculates $sin(\pi/6)$ count times by expanding Taylor series up to 100th order. In the program we call the method with count = 10, 20, 30, ..., 200 sequentially and observe the execution time. In order to ensure the execution time can be measured accurately, for different count we call the method one thousand times and calculate the average. The program was compiled by our prototype compiler and run on the JVM of OpenJDK 1.6.0_24 and Intel Core i7 (2.67GHz, 4 cores) with 8GB memory.

Figure 4 shows the result of the experiment. At first the execution time in DominoJ was 22% slower than in Java ($8.14\mu s$ against $6.67\mu s$), but the gap was getting narrow and almost the same as in Java after count = 70. These numbers on execution overheads are not serious in practice since they are not very large and can be optimized by JVM at runtime.

4.2 Enhance the method paradigm

Method slots extend the method paradigm to support the event-handler paradigm and the aspect paradigm, while still reserving the original behavior in the method paradigm. In DominoJ, if the operators for method slots are not used, the code works as in plain Java. In other words, the method in Java can be regarded as a method slot that has only the default closure.

We also analyze how method slots can be applied to "GoF" design patterns [7], and classify the patterns into four groups as shown in Table 2. The key idea of the patterns in group I can be considered event propagation—from the outer object to the inner object, or among colleague objects. Using method slots can avoid code scattering caused by the pattern code since event implementation is eliminated. Code tangling caused by combining multiple patterns can be eased as well. The patterns in group II use the inheritance to al-

Table 2: Method slots can be applied to design patterns

	Pattern Name	Description
I	Adapter, Chain of Responsibility, Composite, Decorator, Facade, Mediator, Observer, Proxy	*Propagate events among objects*
II	Abstract Factory, Bridge, Builder, Factory Method, State, Strategy, Template Method, Visitor	*Change class behavior at runtime without inheritance*
III	Command, Flyweight, Interpreter, Iterator, Prototype	*Replace inheritance part in the logic*
IV	Memento, Singleton	*Not applicable*

Table 3: The roles and bindings of the event-handler paradigm in EScala and DominoJ

	Type	EScala	DominoJ
role	*Event*	field (evt)	method slot
	Handler	method	
binding	*Event-to-Handler*	+=	+=
		-=	-=
	Event-to-Event	||	+=, ^=
		&&	use Java expression in the default closure of method slots
		\	
		filter	
		map	
		empty	
		any	
	Handler-to-Event	afterExec	+=
		beforeExec	^=
		imperative	explicit trigger is possible

ter the class behavior at runtime. Different implementation can be added to the method slot instead of overriding in subclasses. In that sense, method slots can be used as an alternative to the polymorphism. Although method slots are not perfect replacement for the inheritance, it is convenient in particular when programmers are forced to choose between two superclasses due to single inheritance limitation. The patterns classified under group III also use the inheritance as a part of their pattern code, so programmers may use method slots or not depending on the situation. As to the patterns in group IV, DominoJ is not beneficial to object creation as what AspectJ can do in [10]. The reason is that DominoJ does not support the inter-type declaration and cannot stop the object creation. Details of this analysis is available in [33].

4.3 The event-handler paradigm

To evaluate how DominoJ works for the event-handler paradigm, first we analyze the bindings between the event and the handler in a typical event mechanism like EScala, and compare them with DominoJ. In languages directly supporting the event-handler paradigm, events are usually introduced as fields, which are separate from methods. In order to associate fields with methods, there are three types of binding between events (fields) and handlers (methods). The ways used for each type of binding are usually different in an event mechanism, and also different between event mechanisms. Table 3 shows the ways provided by EScala. The corresponding DominoJ syntax for the three types of binding is also listed, but actually there is only slot-to-slot binding in DominoJ since only method slots are involved in the event-handler paradigm. Every method slot can play an event role and a handler role at the same time. Listing 8 shows how to use DominoJ for the event-handler paradigm for the shape example mentioned in Section 2. Below we will discuss what the three types of binding are, and explain how DominoJ provides the same advantages with the simplified model.

The event-to-handler binding is the most trivial one since it means what action reacts to a noteworthy change. Whether supporting the event-handler paradigm by languages or not, in general the event-to-handler binding is dynamic and provided by a clear manner. For example, in the Observer pattern an observer object can call a method on the subject to register itself; in C# and EScala, += operator and -= operator are used to bind/unbind a method to a special field named event. In addition to the two operators, Domi-

Listing 8: Using DominoJ for the event-handler paradigm

```java
public class Display {
  public void refresh(int x, int y) {
    System.out.println("display is refreshed.");
  }
}
public class Shape {
  private int left = 0; private int top = 0;
  public void setPosition(int x, int y) {
    left = x; top = y;
  }
  public Shape(Display d) {
    this.setPosition += d.refresh;
  }
}
public class Test {
  public static void main(String[] args) {
    Display d = new Display();
    Shape s = new Shape(d);
    s.setPosition(0, 0);
  }
}
```

noJ provides ^= operator and = operator to make it easier to manipulate the array of handlers. In C# and EScala the handlers for an event can be only appended sequentially and removed individually, but in DominoJ programmers can use = operator to empty the array directly without deducing the state at runtime. Using ^= operator along with += operator also makes design intentions more clear since a closure can be inserted at the beginning without popping and pushing back.

The second one is the event-to-event binding that enables event composition and is not always necessary but greatly improves the abstraction. In a modern event mechanism, event composition should be supported. EScala allows programmers to define such higher-level events to make code more readable. An event-to-event binding can be simulated by an event-to-handler binding and a handler-to-event binding, but it is annoying and error-prone. In DominoJ, it is also possible to define a higher-level event by declaring a method slot without a default closure. Then operators += and ^= can be used to attach other events like what the operator || in EScala does. Other operators in EScala such as

&& and `map` are not provided in DominoJ, but the same logic can be represented by statements in another handlers and attached by `+=` operator. For example, in Listing 1 we can declare a new event `adjusted` that checks if `left` and `top` are the same as the arguments given to `setPosition` using the operator && in EScala:

```
evt adjusted[Unit] = afterExec(setPosition)
                     && ((left,top) != _._1)
adjusted += onAdjusted
```

where `_._1` refers to the arguments given to `setPosition` and `onAdjusted` is the reaction. In DominoJ, we can declare a higher-level event `adjusted` and perform the check in another method slot `checkAdjusted`:

```
public void adjusted(int x, int y);
public void checkAdjusted(int x, int y) {
  if(!(x==left && y==top))  adjusted(x, y);
}
```

and then bind them as follows:

```
setPosition += checkAdjusted;
adjusted += onAdjusted;
```

Although the expression in DominoJ is not rich and declarative as in EScala, they can be used to express the same logic. In addition, the event-to-event binding in EScala is static, so that the definition of a higher-level event in EScala cannot be changed at runtime. On the other hand, it is possible in DominoJ since the slot-to-slot binding is totally dynamic.

The last one is handler-to-event binding, which is also called an event trigger or an event definition. It decides whether an event trigger can be implicit or not. In the Observer pattern and C#, an event must be triggered explicitly, so that the trigger code is scattering and tangling. EScala provides two implicit ways and an explicit way: after the execution of a method, before the execution of a method, or triggering an event imperatively. In DominoJ, an event can be triggered either implicitly or explicitly. A method slot can not only follow the call to another method slot but also be imperatively called. More precisely, there is no clear distinction between the two triggering ways. In EScala afterExec and beforeExec are provided for statically binding an event to the execution of a method while DominoJ provides `+=` operator and `-=` operator for dynamically binding a method slot to the execution of another method slot. This sounds like that a method slot has two pre-defined EScala-like events for the default closure, but it is not correct. In DominoJ's model the only one event is the call to a method slot, and the default closure is also a handler like the other closures calling other method slots. This feature makes the code more flexible since the execution order of all handlers can be taken into account together. As to the encapsulation, in EScala the visibility of explicit events follows its modifiers, and the implicit events are only visible within the object unless the methods they depend on are `observable`. On the other hand, the encapsulation in DominoJ relies on the visibility of method slots. The design is simpler but limits the usage because a `public` method slot is always visible as an event to other objects.

There is one more important difference between EScala and DominoJ. In DominoJ, a higher-level event can be declared or not according to programmers' design decision. In
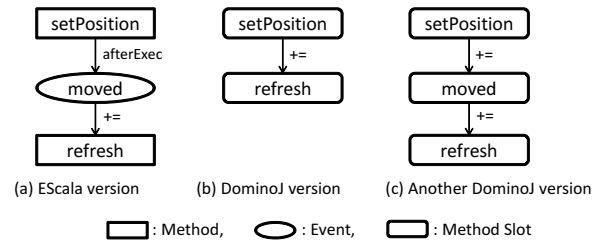

(a) EScala version   (b) DominoJ version   (c) Another DominoJ version

☐ : Method,   ◯ : Event,   ☐ : Method Slot

Figure 5: The execution order of the shape example in EScala and DominoJ

order to explain the difference, we use a tree graph to represent the execution order in the shape example by regarding `setPosition` as the root. As shown in Figure 5, we use rectangles, circles, and rounded rectangles to represent methods, events, and method slots, respectively. When a node is called, the children bound by `beforeExec` or `^=` must be executed first, followed by the node itself and the children bound by `afterExec` or `+=`. Figure 5 (a) is the execution order of Listing 1, and Figure 5 (b) is the one of Listing 8. In the DominoJ version, the event `moved` is eliminated and its child `refresh` is bound to `setPosition` directly since we do not need additional events in such simple case. DominoJ is easier and simpler to apply the event-handler paradigm when events are not complicated but used everywhere. In EScala, events must be created since methods cannot be bound to each other directly. However, such events are still necessary if we want to keep the abstraction. In that case, method slots can be used as the events in EScala by declaring them without a default closure. For example, the event `moved` in Line 11 of Listing 1 can be translated into the following statements:

```
public void moved();
setPosition += moved;
```

Figure 5 (c) is another DominoJ version, which has the higher-level event as the EScala version. In DominoJ, programmers can choose between the simplified one and the original one depending on the situation.

Note that the number of lines of Listing 8 is one line longer than Listing 1 because the syntax of Scala looks more compact than Java. In Java the constructor and the fields used inside a class must be declared explicitly while they are omitted in Scala. In Listing 8 the constructor takes two more lines than Listing 1. If we do not take this into account, the EScala version is one line longer than the DominoJ version due to additional event declaration.

The line of code can also be analyzed according to Table 3. With regard to the roles, additional event declarations are necessary in EScala while they are combined into one declaration in DominoJ as we discussed above. For the event-to-handler binding, both the operators provided by EScala and DominoJ take one line. For the event-to-event binding, the operators provided by EScala can be written in the same line, but in DominoJ `+=` operator and `^=` operator cannot be merged into one line. In that case the code in DominoJ is longer than the EScala one. For example, a higher-level event `changed` can be defined by three events `resized`, `moved`, and `clicked`:

```
evt changed[Unit] = resized || moved || clicked
```

but in DominoJ they must be defined as follows:

```
resized += changed;
moved += changed;
clicked += changed;
```

That is why the expression in EScala is richer but complicated. Introducing appropriate syntax sugar to DominoJ to allow putting operators in one line is also possible, but we think it makes the design complicated. However, in this example we can also find passing the event value in EScala takes effort. In EScala, as far as we understand, only a value is kept in an event field. If we want to gather up the arguments x and y given to setPosition, and then pass to moved and changed, we need to declare additional classes such as Point and declare the events with the new type rather than Unit[3]. The additional classes increase the number of lines as well. For the handler-to-event binding, afterExec and beforeExec in EScala can define an anonymous event and share the same line of an event-to-handler binding. To sum up, in DominoJ the event declarations may be eliminated and thus the number of lines of source code can be reduced. On the other hand, the number of code of DominoJ version is longer when translating a complex EScala expression composed of a number of operators since DominoJ has less primitive syntax. DominoJ makes code clear because each method slot has a name explicitly, and each line for binding only defines the relation between two method slots.

4.4 The aspect paradigm

DominoJ can be used to express the aspect paradigm as well. In order to discuss language constructs concretely, we compare DominoJ with the most representative aspect-oriented language—AspectJ. The call to a method slot is a join point, and other method slots can be bound to it as advices. Note that aspect-oriented programming is broader as discussed in [14] and not restricted to the AspectJ style, which is the point-advice model. In this subsection first we analyze the necessary elements in the point-advice model, and compare the constructs provided by AspectJ and DominoJ. Then we use DominoJ to rewrite the shape example in Listing 2 and discuss the differences.

Since the purpose of the aspect paradigm is to modularize the crosscutting concerns, we need a method-like construct to contain the code piece, a way to attach the method-like construct to a method execution, and a class-like construct to group the method-like construct. In AspectJ, the class-like construct is the aspect construct, the method-like construct is the advice body, and the way of attaching is defined by the pointcut and advice declaration. In DominoJ, the method slot and the class construct in plain Java are used and only operators for method slots are introduced for attaching them. The method slots bound by += operator or ^= operator are similar to after/before advices, respectively. The method slots bound by = operator are similar to around advices and proceed can be used to execute the original method slot. It is expected that DominoJ cannot cover all expression in AspectJ since DominoJ's model is much simpler. For example, in DominoJ the inter-type declaration and the reflection are not provided. According to

[3]In EScala, declaring events with Unit type means that no data are passed [9].

Table 4: The mapping of language constructs for the aspect paradigm in AspectJ and DominoJ

Construct	AspectJ	DominoJ
grouping	aspect	class
code piece	advice body	method slot body (default closure)
pointcut and advice declaration	after returning and execution	+= and $retval
	before and execution	^=
	around	=
	this	$caller
	target	$predecessor
	args	by parameters

the three elements, Table 4 lists the mapping of language constructs in AspectJ and DominoJ.

In AspectJ programmers need to understand the special instance model for the aspect construct, but in DominoJ the class construct is reused. Although the instances of the construct for grouping need to be managed manually, there is no need to learn the new model and keywords like issingleton, pertarget, and percflow. In DominoJ programmers can create an instance of the aspect-like class and attach its method slots to specified objects according to the conditions at runtime. If the behavior of issingleton is preferred, programmers can declare all fields including method slots in the aspect-like class as static since static method slots are supported by DominoJ. The shape example of AspectJ in Section 2 can be rewritten by DominoJ as shown in Listing 9. Here the class UpdateDisplay is the aspect-like class. In Line 14, we attach the advice refresh in a static method slot init, so all Shape objects will share the class object of UpdateDisplay. Furthermore, we let init be executed after the constructor of Shape, so that we can avoid explicitly attaching refresh every time a Shape object is created. Moreover, we do not have to modify the constructor of Shape. If we need to count how many times setPosition is called for each Shape and thus pertarget is preferred, we can rewrite the class UpdateDisplay as shown in Listing 10. Every time a Shape object is created, a UpdateDisplay object is created for it implicitly. Note that the object ud will not be garbage-collected since its method slot count is attached to another method slot.

In DominoJ, there is no difference between methods and advices while in AspectJ they are different constructs. Although an advice in AspectJ can be regarded as a method body, it cannot be directly called. If the code of an advice is reusable, in AspectJ we must move it to another method but in DominoJ it is not necessary.

The pointcut and advice declaration in AspectJ and DominoJ are similar but not the same. First, what they target at is different. AspectJ is class-based while DominoJ is object-based. In other words, what AspectJ targets at are all object instances of a class and its subclasses but what DominoJ targets at are individual object instances. However, it is possible to emulate the class-based behavior in DominoJ by the code attaching to the constructor of a class as shown in Line 16 of Listing 9. Second, unlike AspectJ that has call and execution pointcut, in DominoJ only execution pointcut is supported. This limits the usage but reduces the complexity. In fact, the relation between advices is quite different in AspectJ and DominoJ. In AspectJ an advice is attached

Listing 9: Using DominoJ as the aspect paradigm

```
1   public class Display {
2     public static void refresh(int x, int y) {
3       System.out.println("display is refreshed.");
4     }
5   }
6   public class Shape {
7     private int left = 0; private int top = 0;
8     public void setPosition(int x, int y) {
9       left = x; top = y;
10    }
11  }
12  public class UpdateDisplay {
13    public static void init() {
14      ((Shape)$predecessor).setPosition += Display.refresh;
15    }
16    static { Shape.constructor += UpdateDisplay.init; }
17  }
18  public class Test {
19    public static void main(String[] args) {
20      Shape s = new Shape();
21      s.setPosition(0, 0);
22    }
23  }
```

Listing 10: Rewrite UpdateDisplay for pertarget

```
1   public class UpdateDisplay {
2     private int total = 0;
3     public void count(int x, int y) {
4       total++;
5     }
6     public static void init() {
7       UpdateDisplay ud = new UpdateDisplay();
8       ((Shape)$predecessor).setPosition += ud.count;
9     }
10    static { Shape.constructor += UpdateDisplay.init; }
11  }
```

to methods and cannot be directly attached to a specific advice, but in DominoJ a method slot is not only an advice but also a method. For example, if we need another advice for checking the dirty region in Listing 2, we may prepare an aspect CheckDirty containing this advice as shown in Figure 6 (a). However, the advice can only be attached to setPosition. In DominoJ, the advice can be attached to either setPosition or init as shown in Figure 6 (b).

The behavior of proceed in AspectJ and DominoJ is also a little different. The proceed in DominoJ should be used only along with = operator since it calls the default closure in the preceding method slot rather than the next closure. The root cause of the difference is the join point model: what DominoJ adopts is the point-in-time model while the one AspectJ adopts is the region-in-time model [15]. In other words, in AspectJ the arrays of the three types of advices are separate, but in DominoJ there is only one array. If += operator or ^= operator are used after using = operator to attach a method slot containing proceed, the behavior is not as expected as in AspectJ. Figure 7 shows an example of around advice in AspectJ and DominoJ. In AspectJ, the around advices localCache and memCache are attached to queryData in order. In DominoJ, we can do it similarly:

```
queryData = localCache;
localCache = memCache;
```

then using proceed in memCache and localCache will call the default closure of their preceding method slot, localCache
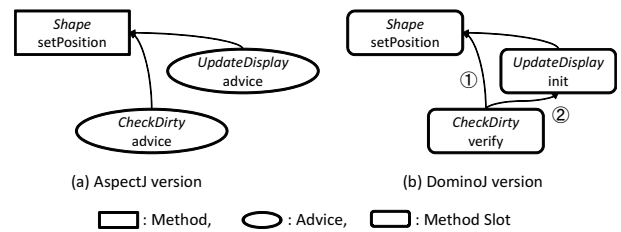

Figure 6: Adding another advice to the shape example in AspectJ and DominoJ
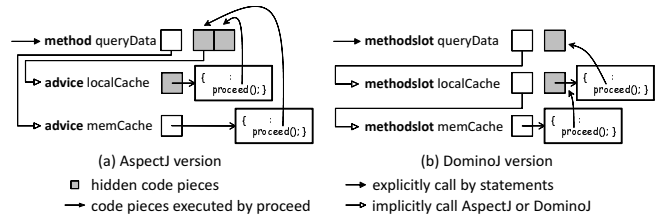

Figure 7: Calling proceed in AspectJ and DominoJ

and queryData, respectively. Another difference is that the args pointcut and the wildcard used in call and execution pointcuts in AspectJ are not supported in DominoJ. Method slots are simply matched by their parameters. If the overloading is not taken into account, the operators in DominoJ only select one method slot in one line statement.

As to the number of lines, the two versions are about the same. Comparing them line by line might not make much sense since there is no simple translation between DominoJ and AspectJ.

4.5 Summary of the coverage

For the event-handler paradigm, there are three significant properties: implicit events, dynamic binding, and event composition. DominoJ supports them all by method slots and only four operators. Rewriting a complex expression of event composition in EScala is also possible though it takes more lines. Introducing additional syntax may resolve the issue but it also complicates the model. As a result of regarding method slot calls as events, giving an event a different visibility from the methods it depends on is not supported by DominoJ.

The aspect paradigm of AspectJ has three important features that cannot be provided by the event-handler paradigm: around advice, obliviousness, and inter-type declaration. In DominoJ the around advice can be emulated by assigning a closure calling another method slot. DominoJ also supports the obliviousness in AspectJ by reusing the class construct as the aspect construct and attaching a method slot to a constructor of the target class. The inter-type declaration is not available in DominoJ. A possible solution is introducing a default method slot for undefined fields in a class like Smalltalk's doesNotUnderstand or what the no-applicable-method does in CLOS. In addition to being used for the event-handler paradigm and the aspect paradigm, DominoJ allows programmers to use both paradigms together.

5. RELATED WORK

The delegation introduced by C# [17] allows programmers to declare an event, define its delegate type, and bind a corresponding action to the event. Event composition is also supported by adding a delegate to two or more events. Although the delegate interface hides the executor from the caller, implicit events are not supported. The event must be triggered manually when the change happens. However, C# is able to emulate DominoJ using an unusual programming style: declaring an additional event for every method and always triggering the event rather than the method. From the point of view, a delegate is very similar to a method slot except the operator += in C# copies the handlers in the event but not creates a reference to the event. However, as in EScala, events and methods are still separate language constructs. Supporting by only one construct means that programmers do not need to decide between using such an unusual style or a normal style at the design stage whether newer modules might regard those methods as events or not. Furthermore, it is annoying that event fields and methods in C# cannot share the same name. Another disadvantage is that we have to ensure that there is at least one delegate for the event before triggering it. Otherwise it will raise an exception. This is not reasonable from the viewpoint of the event mechanism since it just means no one handles the event. In DominoJ no handlers for an event does not raise an exception and the one that triggers an event on a method slot is unaware of handlers.

There are a number of research activities on the integration of object-oriented programming and aspect-oriented programming. Those research use a single dispatch mechanism to unify OOP and AOP and reveal that the integration makes the model clearer, reusable, and compositable. Delegation-based AOP [11, 23] elegantly supports the core mechanisms in OOP and AOP by regarding join points as loci of late binding. The model proposed in [12] provides dedicated abstractions to express various object composition techniques such as inheritance, delegation, and aspects. The difference is that DominoJ integrates the event-handler paradigm and the aspect paradigm based on OOP. Another difference is that we propose a new language construct rather than a machine or language model, which makes it compatible with existing object-oriented languages such as Java. Other work such as FRED [21], composition filters [1], predicate dispatching [5], and GluonJ [2] can also be regarded as such integration work.

The method combination in Flavors and CLOS makes related methods easy to combine but not override. By default the combined method in Flavors first calls the before methods in the order that flavors are combined, following by the first primary method, then the after methods in the reverse order. The return value of the combined method is supplied by the primary method, while the return values of the before and after methods are ignored. Similarly, CLOS provides a standard method combination for generic functions. For a generic function call, all applicable methods are sorted before execution in the order the most specific one is first. Besides the primary, before, and after methods, CLOS provides the around methods and call-next-method for the primary and around methods. From the viewpoint of method combination, the default closure of a method slot looks like a primary method that can be dynamically added to other method slots as a before or after method, and even as an around method by assigning to the target method slot then using proceed as call-next-method. It is also easier to express the method combination as a hierarchy in DominoJ.

With regard to the event mechanism, several research activities are devoted to event declaration. Ptolemy [22] is a language with quantified and typed events, which allows a class to register handlers for events, and also allows a handler to be registered for a set of events declaratively. It has the ability to treat the execution of any expression as an event. The event model in Ptolemy solves the problems in implicit invocation languages and aspect-oriented languages. EventJava [6] extends Java to support event-based distributed programming by introducing the event method, which are a special kind of asynchronous method. Event methods can specify constraints and define the reaction in themselves. They can be invoked by an unicast or broadcast way. Events satisfying the predicate in event method headers are consumed by a reaction. Context-aware applications can be accommodated easily by the mechanism. Both the two research make events clear and expressive, but they do not support implicit events, which is one of the most significant properties as an event mechanism, whereas DominoJ supports it. Moreover, all events in their model are class-based, so that events for a specified object have to be filtered in the handlers. The binding in DominoJ is object-based, so it can describe the interaction between objects more properly.

On the other hand, several research support the event-handler paradigm upon the aspect paradigm. ECaesarJ [19] introduces events into aspect-oriented languages for context-handling. The events can be triggered explicitly by method calls or defined by pointcuts implicitly. EventCJ [13] is a context-oriented programming language that enables controlling layer activation modularly by introducing events. By declaring events, we can specify when and which instance layer is activated. It also provides layer transition rules to activate or deactivate layers according to events. EventCJ makes it possible to declaratively specify layer transitions in a separate manner. Comparing with DominoJ, using events in the two languages may beak modular reasoning since their event models rely on the pointcut-advice model. Furthermore, events are introduced as a separate construct from methods.

Flapjax [16] proposes a reactive model for Web applications by introducing behaviors and the event streams. Flapjax lets clients use the event-handler paradigm by setting data flows. The handlers for an event can be registered in an implicit way. However, unlike other event mechanisms, it requires programmers to use a slightly different event paradigm. The behavior of DominoJ is more similar to the typical event mechanism while it has the basic ability for the aspect paradigm as well.

Fickle [4] enables re-classification for objects at runtime. Programmers can define several state classes for a root class, create an object at a certain state, and change the membership of the object according to its state dynamically. With re-classification, repeatedly creating new objects between similar classes for an existing object can be avoided. Both Fickle and DominoJ allow changing the class membership of an object at runtime, so other objects holding the identity of the object can be unaware of the changes. The difference is that Fickle focuses on the changes between states while DominoJ focuses on the effect of calling specified methods. Fickle provides better structural ability such as declaring

new fields in state classes. However, if the relation between states is not flat and cannot be separated clearly, programmers still have to maintain the same code between state classes. The common code to only part of states can be gathered up into one class in DominoJ. Furthermore, DominoJ is easier to use for the event-handler paradigm.

The lambda expressions [20] will be introduced in Java 8 as a new feature to support programming in a multicore environment. With the new expression, declaring anonymous classes for containing handlers can be eliminated. The lambda expression of Java 8 is a different construct from methods but method slots can be regarded as a superset of methods.

6. CONCLUSIONS

We discussed the similarity between the language constructs for the event-handler paradigm and the aspect paradigm, which motivates us to propose a new language construct, named *method slot*, to support both the two paradigms. We presented how a *method slot* is introduced as a language construct in a Java-based language, *DominoJ*. We then discussed how *method slots* can be used for the two paradigms and the coverage of expressive ability. Although the expression of *method slots* is not as rich as other languages, it is much simpler and able to express most functionality in the two paradigms. We also showed its feasibility by implementing a prototype compiler and running a preliminary microbenchmark.

7. REFERENCES

[1] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[2] 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.

[3] J. Dedecker, T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Ambient-oriented programming in AmbientTalk. In *ECOOP '06*, pages 230–254. Springer, 2006.

[4] S. Drossopoulou, F. Damiani, D. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP '01*, pages 130–149, 2001.

[5] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98*, pages 186–211. Springer-Verlag, 1998.

[6] P. T. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP '09*, pages 570–594, 2009.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[8] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *SIGSOFT '98/FSE-6*, pages 209–221. ACM, 1998.

[9] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. In *AOSD '11*, pages 227–240. ACM, 2011.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02*, pages 161–173. ACM, 2002.

[11] M. Haupt. A machine model for aspect-oriented programming. In *ECOOP '07*, pages 501–524, 2007.

[12] W. Havinga, L. Bergmans, and M. Aksit. A model for composable composition operators: expressing object and aspect compositions with first-class operators. In *AOSD '10*, pages 145–156. ACM, 2010.

[13] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264. ACM, 2011.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*. Springer-Verlag, 1997.

[15] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In *APLAS '06*, pages 131–147, 2006.

[16] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA '09*, pages 1–20. ACM, 2009.

[17] Microsoft Corporation. C# language specification.

[18] Microsoft Corporation. Messages and message queues.

[19] A. Núñez, J. Noyé, and V. Gasiūnas. Declarative definition of contexts with polymorphic events. In *COP '09*, pages 2:1–2:6. ACM, 2009.

[20] Oracle Corporation. OpenJDK: Project Lambda. `http://openjdk.java.net/projects/lambda/`.

[21] D. Orleans. Incremental programming with extensible decisions. In *AOSD '02*, pages 56–64. ACM, 2002.

[22] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, pages 155–179, 2008.

[23] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA '08*, pages 525–542. ACM, 2008.

[24] Sun Microsystems. Abstract window toolkit. `http://java.sun.com/products/jdk/awt/`.

[25] The AspectJ Project. `http://www.eclipse.org/aspectj/`.

[26] The Boost Project. Boost.Signals. `http://www.boost.org/libs/signals/`.

[27] The JastAdd Project. JastAddJ: The JastAdd Extensible Java Compiler. `http://jastadd.org/web/jastaddj/`.

[28] The Qt Project. Signals & Slots. `http://qt-project.org/doc/signalsandslots`.

[29] The SAX project. Simple api for xml. `http://www.saxproject.org/`.

[30] The Self project. `http://selflanguage.org/`.

[31] The X.Org project. Xlib in x window system. `http://www.x.org/`.

[32] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *SIGMOD '90*, pages 259–270. ACM Press, 1990.

[33] Y. Zhuang and S. Chiba. Applying DominoJ to GoF Design Patterns. Technical report, Dept. of Math. and Comp., Tokyo Institute of Technology, 2011.