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

YungYu Zhuang and Shigeru Chiba

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

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 explains 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.

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 [33], 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 [7,29,27] 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 [31] to extend the method paradigm in Java. In Self, an object consists only of slots [24], which may contain either a value or a method. In other words, there is no difference between fields and methods since a

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 70–108, 2014.
© Springer-Verlag Berlin Heidelberg 2014

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 that can keep more than one closure at the same time. 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[1] 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 [32,18,25]. 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 [25] 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

[1] This paper is an extension to the one we presented at Modularity:AOSD2013.

time, there are still good reasons for reintroducing 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 [30] 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 the 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[2] [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 [26], the most well-known aspect-oriented language.

By comparing the two pieces of code, we can find that 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 paradigms depend on the most basic paradigm, the method paradigm,

[2] The syntax follows the example in EScala 0.3 distribution.

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 }
```

since both events and pointcuts cause the execution 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 nonlocal 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

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 }
```

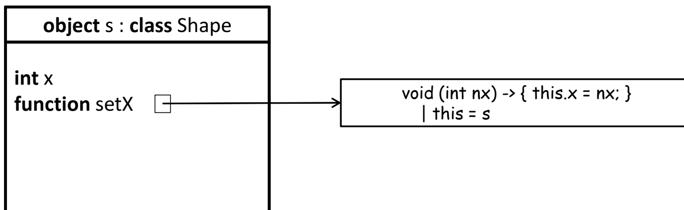

Fig. 1. In JavaScript, both an integer and a function are fields on an object

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 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:

$$\langle modifier\rangle^* \ \langle return\ type\rangle \ \langle identifier\rangle \ \text{“(”} \ \langle parameter\ list\rangle? \ \text{“)”} \ \langle throws\rangle? \ (\langle default\ closure\rangle \mid \text{“;”})$$

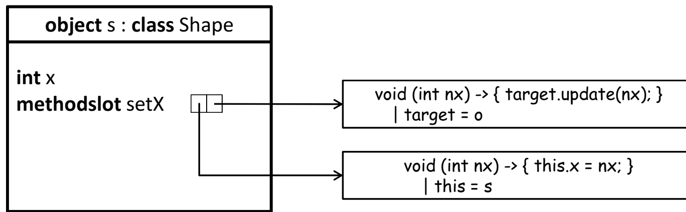

Fig. 2. A method slot is an extended field that can keep more than one function closure

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 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 does not hide 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.

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 }
```

Listing 4. The algorithm of calling a method slot

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

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.

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.*

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 except the side effects caused by the preceding closures in the array of that method slot, where this refers to the object owning the method slot, and therefore, the callee method slot in target must be accessible from the caller

Listing 5. The algorithms of the four operators

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

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.

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 constructor of Shape. Here the return

Listing 6. The algorithm of checking the types

```
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"))))
```

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 language supporting the aspect paradigm must provide a way to retrieve runtime context, for example, AspectJ provides pointcut designators and reflection API for that purpose, DominoJ provides three keywords to retrieve the information about the caller at runtime in the default closure of a method slot. The owner object and the default closure of the method slot at the left-hand side of an operator can be got by using the keywords in the default closure of the method slot at the right-hand side. Unlike AspectJ, which extends the set of pointcut designators available in the language, DominoJ extends the set of special variables such as this and super. In DominoJ a call to the method slot can be regarded as a sequence of method slot calls among objects since a method slot may contain closures calling other method slots. When a method slot is

Listing 7. The algorithm of binding method slots

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

explicitly called by an expression in a certain default closure, the method slots bound to it by operators are implicitly called by DominoJ. Programmers can get the preceding objects in the call sequence. In the default closure, i.e. the body of method slot declaration, the caller object can be got by the keyword $caller. It refers to the object where we start the call sequence by the expression. 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 the 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

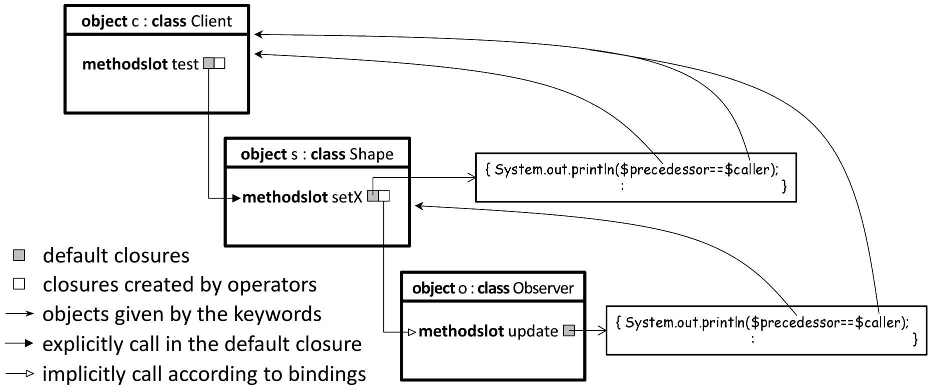

Fig. 3. The keywords $caller and $predecessor

as well. The keyword proceed can be used to call 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 the 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[3] of DominoJ built on top of JastAddJ [28]. 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 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 The Implementation

The DominoJ compiler is a source-to-source compiler which translates DominoJ code to plain Java code and then compiles it into Java bytecode. However, implementing the compiler is not easy since closures are not supported by the current Java version (Java 7). In the DominoJ compiler we use the well-known means of the Java language such as inner classes to represent the closures.

Closure Representations in Java. To emulate closures in Java, a naive implementation is using Java reflection. The compiler could generate the code to

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

record the target objects and the method names, and use the reflection API to invoke the methods at runtime. For example, adding a closure calling o.update to s.setX could be represented as adding a pair (o, "update"), which an object instance of the class Pair<Object, String>, to the array for s.setX. When s.setX is called, all the pair stored in the array will be iterated and the methods such as o.update can be invoked by the reflection API. It is not surprising that the overheads are not small. Another idea is to define an interface like Callable then a closure can be represented by an object instance of a class implementing the interface. This class is generated by the compiler for every closure. Such an object can be stored in the array for a method slot, and the method inherited from the interface, which contains the method call such as o.update, can be called when the object is iterated.

The DominoJ Compiler. The performance of DominoJ code is determined by how the closures are represented and executed at runtime. Using Java reflection is a naive solution, but the overheads are not negligible. Suppose that we have a method slot setX in DominoJ:

```
public class Shape {
    :
  public void setX(int nx) {
      :  // the default closure
  }
}
```

then the compiler will generate the following Java code in Shape: an array field setX$slot and a method setX for iterating the elements in the array setX$slot. In other words, calling a method slot in DominoJ is translated to calling a method in Java to iterate and invoke the elements in an array as follows:

```
// Java code generated by the compiler
public void setX(int nx) {
  Iterator iter = setX$slot.iterator();
  while(iter.hasNext()) {
      :  // invoke a method
  }
}
```

If we use the reflection API to invoke the methods in the iteration, the array setX$slot must store the target objects and the method names for invoking them:

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Pair<Object, String>> setX$slot
    = new ArrayList<Pair<Object, String>>();
    :
}
```

where each element in the array setX$slot holds the target object and the method name. Furthermore, the default closure of setX in DominoJ is translated into a method setX$impl in Java, which contains the statements in the default closure. When an object of Shape, for example s, is instantiated, a pair (this, "setX$impl") is appended to the array setX$slot by default. Suppose that we have another method slot update, the parameter types of which is the same as setX:

```
public class Observer {
    :
  public void update(int i) { ... }
}
```

Then the following binding:

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

where o is an object of Observer, is translated into:

```
// Java code generated by the compiler
s.setX$slot.add(new Pair<Object, String>(o, "update"));
```

When s.setX is called, the pairs (this, "setX$impl") and (o, "update") will be got in order. Here we show the code of setX again for demonstrating how to invoke the methods using the reflection API:

```
// Java code generated by the compiler
public void setX(int nx) {
  Class[] pars = new Class[1];
  pars[0] = Integer.TYPE;
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Pair<Object, String>> iter = setX$slot.iterator();
  while(iter.hasNext()) {
    Pair<Object, String> pair = iter.next();
    Object obj = pair.getFirst();
    String mname = pair.getSecond();
    Class c = obj.getClass();
    Method m = c.getMethod(mname, pars);
    m.invoke(o, args);
  }
}
```

where the Class array pars is used to specify the parameter types for finding the correct method, (int) in this example, since there may be several overloaded methods. The Object array args, which contains the arguments given to setX. In this example the only argument nx, an int, is autoboxed in an Integer instance and put into args. Obviously the cost of finding and invoking a method using the

reflection API is not low. A possible improvement is storing Method instances instead of the method names, so that we can avoid spending time on finding the Method instance when a method slot is called. However, the cost of invoking a Method is still quite high.

The idea used in our prototype compiler is using an interface to simulate the function closure in JavaScript:

```
// Java code used by the compiler
public interface Closure {
  public Object exec(Object[] args);
}
```

Then for each method slot the compiler can declare a field, which is an anonymous class implementing Closure. For example, the field update$closure is declared in Observer for calling update:

```
// Java code generated by the compiler
public class Observer {
    :
  public Closure update$closure = new Closure() {
    public Object exec(Object[] args) {
      this.update((Integer)args[0]);
      return null;
    }
  }
}
```

Note that the individual element in the array args, the arguments to exec, is typecast properly before giving update. If update is a generic method slot, in other words the only parameter of which is Object[], the array args will be directly given to update:

```
this.update(args);
```

then in the default closure of update programmers need to check the type of each element in the array using instanceof and typecast them if it is necessary. Furthermore, in this example we simply return null in exec since the return type of setX is void. The array for the method slot setX, setX$slot, is now an array of Closure rather than an array of the pair (Object, String):

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Closure> setX$slot = new ArrayList<Closure>();
    :
}
```

The binding statement we discussed above is now translated into:

```
s.setX$slot.add(o.update$closure);
```

Similarly, a field setX$impl$closure for calling the method setX$impl, which contains the statements in the default closure of setX, is declated in Shape as well:

```
// Java code generated by the compiler
public class Shape {
    :
  public Closure setX$impl$closure = new Closure() {
    public Object exec(Object[] args) {
      this.setX$impl((Integer)args[0]);
      return null;
    }
  }
}
```

In the constructor of Shape the following line is added for appending setX$impl$closure to setX$slot by default:

```
// Java code generated by the compiler
this.setX$slot.add(this.setX$impl$closure);
```

When the method slot setX is called, all Closure instances in the array are iterated and their exec methods are called with args, the Object array containing the arguments given to the method slot setX, in this example only nx:

```
// Java code generated by the compiler
public void setX(int nx) {
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Closure> iter = setX$slot.iterator();
  while(iter.hasNext()) {
    Closure c = iter.next();
    c.exec(args);
  }
}
```

The iteration is similar to the reflection version, but the code for invoking a method using the reflection API is replaced with a call to the exec method in Closure. In other words, we need more memory to hold the Closure instances, but the overheads of method slots can be reduced to the cost of calling the exec method.

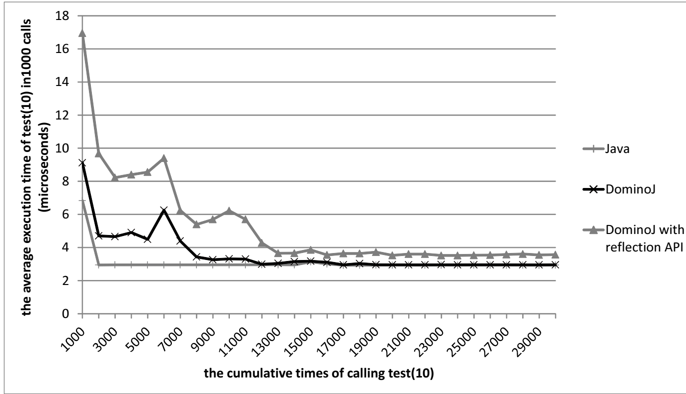

Fig. 4. The average time of continuously calling a method in Java and DominoJ

4.2 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 method we measure is named test, which calculates $sin(\pi/6)$ by expanding Taylor series up to 100th order a number of times according to the argument as shown below:

```
private double x = 3.141592653589793 / 6;
private double result = 0;
public void test(int count) {
  for(int i=0; i<count; i++) {
    double sum = x;
    double n = x;
    double d = 1;
    for(int j=3; j<100; j+=2) {
        n *= - x*x;
        d *= (j-1)*j;
        sum += n/d;
    }
    result = sum;
  }
}
```

Figure 4 shows the results of continuously calling test(10) and calculating the average execution time of calling test(10) every 1000 times of calls until the total amount of calls reaches 30000. For example, the first values we calculate are the average of execution time of 1st 1000th calls in Java and in DominoJ, and the second values are the ones of 1001st 2000th.

The program was compiled by our prototype compiler and run on the JVM of OpenJDK 1.7.0_25 and Intel Core i7 (2.67GHz, 4 cores) with 8GB memory. The result of the naive implementation using the reflection API we mentioned in Section 4.1 is also shown for comparison. After the optimization is sufficiently applied by the JIT compiler, the overhead is negligible (2955ns against 2932ns) although it is initially about 34% (9124ns against 6833ns). On the other hand, the overheads of the reflection version is about 20% (3516ns against 2932ns) after the optimization.

To measure the performance of an operation on method slots such as assigning a closure to a method slot using `=` operator, we repeated the operation and calculated the average time as follows:

```
long start = System.nanoTime();
for(int j=0; j<1000; j++) {
    s.setX = o.update;
}
long estimated = System.nanoTime() - start;
System.out.println(estimated/1000);
```

We also measured other operations by adding one more statement, which uses the other operators such as `+=` operator after the assignment:

```
  :
for(int j=0; j<1000; j++) {
    s.setX = o.update;
    s.setX += o.update;
}
  :
```

Figure 5 shows the result of running such programs 100 times. According to the average time of the operations in the four programs, we can calculate the time of the four operations: the `=` operation takes 427ns, the `^=` operation takes 483ns, the `+=` operation takes 275ns, and the `-=` operation takes 726ns. The `-=` operation might be even slower when the number of closures in the method slot is large since it takes time to check every closure in the array. It is reasonable that the `+=` operation is the fastest one since it simply appends to the array, while the `=` operation have to clear the array and the `^=` operation inserts to the beginning of the array; the performance is relevant to how the method slots are implemented. Finding a more efficient technique to implement method slots is included in our future work. For example, using other structures instead of ArrayList to store the closures or using the new JVM instruction invokedynamic to emulate the closures might be possible solutions to improve the performance of DominoJ code.

4.3 Method Slots and Design Patterns

Method slots extend the method paradigm to support the event-handler paradigm and the aspect paradigm, while still preserving the original behavior in the method

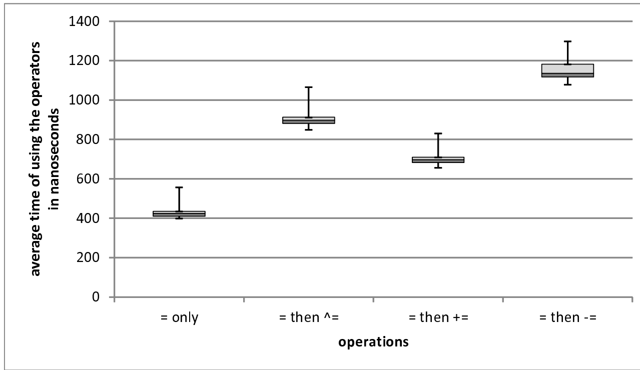

Fig. 5. The average time of using the operators for method slots

paradigm. In DominoJ, if the operators for method slots are not used, the code works as in plain Java. In other words, a Java method can be regarded as a method slot that has only the default closure.

We could regard the inheritance in object-oriented programming as an event mechanism with default bindings. A method declaration in the superclass is an event declaration, and its implementation is the handler bound to the event by default. If the method is overridden in a subclass, the overriding implementation automatically replaces the overridden one and becomes the only handler for the event. In other words, the call to a method on an object is an event, and the method implementation selected by the polymorphism is the handler. The binding from the handler to the event in the inheritance is a one-to-one relation and predefined. Method slots extend the default binding in object-oriented programming to allow the binding of more than one handler to an event.

We have also analyzed how method slots can be applied to "GoF" design patterns [7], and classify the patterns into four groups as shown in Table 2. Furthermore, we implemented the sample code in the GoF book in Java and DominoJ, and compared them with respect to the four modularity criteria borrowed from [10], and a new criterion named noninheritance, which means that method slots can be used as an alternative to the inheritance solution or not. This might remind readers of the mixin. As an alternative to the inheritance both the mixin and method slots allow to execute an implementation in another class or object for a method call at runtime. However, in several mixin mechanisms both fields and methods are included, but a binding between method slots do not involve fields. The comparison is shown in Table 3, where the number of lines of code is listed as well. Note that for the patterns in group III we ignore the comparision on the number of lines of code since in group III method slots do not act a major role as in group I and group II. In this table the locality means the code of defining the relation can be gathered up, the reusability means the pattern code can be abstracted and thus reusable, the composition means the code do not get complicated when applying multiple relationships to the same class, and the unpluggability means it is easy to apply or remove

the pattern. The operators for method slots can be used to cause the execution of a method slot on another object when a specified method slot is called. In general such mixin behavior helps to gather up similar implementations in a class and can be an alternative to the polymorphism. Furthermore, the pattern code for propagating events can be expressed by the bindings, which can be gathered up in one place for the locality. For several patterns such as the Chain of Responsibility pattern, the event implementation is almost eliminated and thus the code tangling caused by the composition can be avoided. If the pattern code can be totally eliminated, reusing it is quite easy since there is no need to implement the pattern every time. It is also possible to make the code easy to plug or unplug for several patterns such as the Proxy pattern since the pattern is applied by a binding rather than passing a different object. However, unlike the polymorphism the switch between different implementations must be manually managed. As to the numbers of lines of code in Java and in DominoJ, basically there are no significant difference since the explicit triggers for events are removed but the bindings for describing the event propagations are added. However, for several patterns such as the Observer pattern the pattern code of which is totally eliminated. On the other hand, using the mixin behavior in several patterns such as the Factory Method pattern takes additional lines of bindings to switch the implementations. Below we discuss the four groups by showing concrete examples.

Table 2. Method slots can be applied to design patterns

	Pattern Name	**Description and Consequences**
I	Adapter, Chain of Responsibility, Composite, Decorator, Facade, Mediator, Observer, Proxy	*Implicitly propagate events among objects by the bindings.* *GOOD: The bindings can be gathered up in one place.* *BAD: The method slots which handle the same event must share the same type.*
II	Abstract Factory, Bridge, Builder, Factory Method, State, Strategy, Template Method, Visitor	*Change class behavior at runtime without inheritance.* *GOOD: A solution to avoid multiple inheritance.* *BAD: Unlike the polymorphism the switch between implementations have to be manually managed.*
III	Command, Flyweight, Interpreter, Iterator, Prototype	*Replace inheritance part in the logic.* *GOOD: Provide an alternative for the inheritance part.* *BAD: Not helpful except the inheritance part.*
IV	Memento, Singleton	*Not applicable*

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 following example is an example of the Chain of Responsibility pattern. In a graphical user interface library a widget such as a button may need a hotkey for showing help. When users are confused with the label of the button, they can press the F1 key to get a pop-up description, which explains the meaning in detail. To implement the help event in Java, the Chain of Responsibility pattern can be used in Widget, the base class for all widgets, as shown in Listing 8. Here we assume that the method handleHelp will be called when users press the F1 key on a widget object. Every subclass of

Table 3. The benefit of applying DominoJ to design patterns

		Modularity Properties					#Lines of Sample Code	
	Pattern Name	Locality	Reusability	Composition	Unpluggability	Non-inheritance	in Java	in DominoJ
I	Adapter	√					51	48
	Chain of Responsibility	√	√	√	√	√⋆	38	28
	Composite		√	√	√	√⋆	41	16
	Decorator	√		√	√		26	20
	Facade	√⋆					34	53
	Mediator	√			√		68	49
	Observer	√	√	√	√	√⋆	71	32
	Proxy				√	√⋆	47	61
II	Abstract Factory	√⋆				√⋆	41	58
	Bridge	√⋆				√⋆	58	64
	Builder	√⋆				√⋆	55	69
	Factory Method	√⋆				√⋆	67	97
	State	√					66	69
	Strategy			√	√	√⋆	36	28
	Template Method	√				√⋆	31	45
	Visitor	√		√	√		63	69
III	Command					√⋆	Ignored	
	Flyweight					√⋆		
	Interpreter					√⋆		
	Iterator					√⋆		
	Prototype					√⋆		
IV	Memento	Same implementation for Java and DominoJ						
	Singleton							

The √ mark means that DominoJ has better modularity than Java when implementing the pattern.
The ⋆ mark means that AspectJ does not provide such modularity when implementing the pattern, while DominoJ does.

the Widget class should override the handleHelp method to implement its own behavior for the help event, and return a boolean value to indicate whether the help event is handled or not. In the Widget class a default implementation is given: propagating the help event to the successor in the chain of responsibility. The successor is kept as a private field and set to its container in the constructor as shown in Line 2-5. If no successor is set, false is returned. When a subclass of Widget such as Button class overrides the handleHelp method, it must explicitly call super.handleHelp for executing the default implementation to propagate the help event to its successor. In DominoJ, the operator += can be used to describe such behavior as shown in Listing 9. Note that in Line 10 the keyword $retval is used to check if the help event is handled by the predecessor, and the explicit call super.handleHelp is removed from all subclasses. It makes the code clear, especially when there are several chain of responsibility for different events in the Widget class. Using DominoJ can avoid the tangling caused by pattern code.

Method slots can also be used to improve the transparency to clients. In a class-based object-oriented language such as Java, it is not allowed to change the class membership of objects as discussed in [4]. Suppose that two classes Student and Employee are given to model the students and the employees in a university. If now a student has graduated and employed by the university, we cannot continue using the original Student object. We have to create a new Employee object according to the original Student object and update all references to the object in clients. A solution is using method slots to implement the Proxy pattern for the Student example. In the Proxy pattern usually the clients are aware of the existence of the proxy object. For example, in order to control the access to Student, giving a proxy class Employee, which owns a reference to its original Student object, then the clients have to use the proxy object instead of the original Student object. In DominoJ the behavior of a Student object such as getInfo can be replaced if it is public:

```
s.getInfo = e.getInfo;
```

Listing 8. The Chain of Responsibility pattern example in Java

```
1  public class Widget {
2    private Widget successor = null;
3    public Widget(Widget container) {
4      successor = container;
5    }
6    public boolean handleHelp() {
7      if(successor == null)    return false;
8      return successor.handleHelp();
9    }
10       :
11 }
12 public class Button extends Widget {
13   public boolean handleHelp() {
14   : //return true if it can offer help,otherwise return super.handleHelp()
15   }
16 }
```

Listing 9. The Chain of Responsibility pattern example in DominoJ

```
1  public class Widget {
2    public Widget(Widget container) {
3      this.handleHelp += container.handleHelp;
4    }
5    public boolean handleHelp();
6        :
7  }
8  public class Button extends Widget {
9    public boolean handleHelp() {
10       if($retval)    return true;
11       :  // return true if it is handled here, otherwise return false
12   }
13 }
```

where s is a Student object and e is its proxy, an Employee object. Then the clients of s may continue using the reference to s. When s.getInfo is called, the method slot getInfo on its proxy object will be executed for access control. In other words, it is possible to make the clients unaware of plugging or unplugging the proxy.

The patterns in group II use the inheritance to alter the class behavior at runtime. Different implementation for a method slot call 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. For example, Listing 10 shows an example of the Template Method pattern in Java. By taking advantage of inheritance, the drawing border step in the class View can be deferred to its subclass FancyView by overriding the method drawBorder. However, unlike mixin or multiple inheritance, in the subclass FancyView we cannot reuse the implementation of other classes due to the single inheritance limitation in Java. For example, the implementation of drawBorder in FancyView may be the same as the one in another class FancyPrint,

Listing 10. The Template Method pattern example in Java

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class FancyView extends View {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
```

which is neither a subclass of View nor a subclass of FancyView. In this case we cannot extract the common part of FancyView and FancyPrint into a new class Fancy. In DominoJ such mixin behavior is possible by using the operator =. As shown in Listing 11 we move the drawBorder implementation to a new class Fancy and let FancyView own a reference to a Fancy object. Then in the constructor of FancyView we can forward the call to its method slot drawBorder to the one in the Fancy object it refers (Line 22). With DominoJ a subclass can still benefit from another class by the binding as using the mixin. It helps to modularize the code when we want to extract parts of the implementation in the subclass. Programmers can decide to use mixin or inheritance for a feature depending on the design.

Another example we want to show here is the State pattern, which allows an object to alter its behavior by switching between the state objects. Using DominoJ the state transitions can be modularized in another class as using AspectJ [10]. Suppose that we have three state classes for the Queue class: QueueEmpty, QueueNormal, and QueueFull. In Java the state transition code scatters across the state classes, for example the transition from QueueEmpty to QueueNormal is checked and performed in the insert method of QueueEmpty class as shown in Listing 12. In DominoJ all transitions can be gathered up in another class UpdateQueueState as shown in Listing 13. The class UpdateQueueState keeps all the state objects (Line 2-4) and manages the transitions such as emptyToNormal (Line 6-11). For example, the transition emptyToNormal is performed after the method slot insert on the object empty is executed as shown in Line 16. Note that the method slots emptyToNormal and insert share the same type.

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. For example, the intent of the Command pattern is wrapping the requests in objects in order to pass around clients, and inheritance is used for overriding the behavior of a request. Suppose that we want to implement a document editor, which allows users to open a document, edit its content, and

Listing 11. The Template Method pattern example in DominoJ

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class Fancy {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
18 public class FancyView extends View {
19   Fancy fancy;
20   public FancyView() {
21     fancy = new Fancy();
22     this.drawBorder = fancy.drawBorder;
23   }
24 }
```

copy a paragraph. First we declare an abstract class Command, which has a method slot execute, to model the commands supported in the editor:

```
public abstract class Command {
    :
  public void execute();
}
```

Then we can implement the individual commands such as OpenCommand and CopyCommand by extending the Command class. In the subclasses we can declare necessary parameters and override execute to define the behavior for individual commands. For example, the implementation of OpenCommand looks like this:

```
public class OpenCommand extends Command {
  private File file = null;
    :
  public void execute() {
    file = getFileFromUser();
  }
}
```

Here, the user has to select a file and then the path in the field file will be stored when its execute is called. By creating the command objects, the requests from users can be wrapped and passed to other UI components. The functionalities such as undo and redo can also be implemented easily. In group III the inheritance is not the core of the pattern code, but helps the implementation.

Listing 12. The State pattern example in Java

```
1  public class Queue {
2    private QueueState state = new QueueEmpty();
3    public void setState(QueueState s) {
4      state = s;
5    };
6    public boolean insert(Object o) {
7      return state.insert(this, o);
8    }
9      :
10 }
11 public class QueueState {
12   public boolean insert(Queue q, Object o) {
13     return false;
14   };
15     :
16 }
17 public class QueueEmpty extends QueueState {
18   public boolean insert(Queue q, Object o) {
19     QueueNormal nextState = new QueueNormal();
20     q.setState(nextState);
21     return nextState.insert(q, o);
22   }
23     :
24 }
```

As the example of the Template Method pattern shown above in group II, the inheritance can be replaced with the mixin by using method slots. Again, using the mixin is not always a good choice and it depends on programmers' design decision.

As to the patterns in group IV, DominoJ is not helpful in dealing with object creation as what AspectJ does in [10]. The reason is that DominoJ does not support intertype declaration and cannot stop the object creation. Further details of this analysis is available in [34].

4.4 The Event-Handler Paradigm

There are three important metrics to evaluate an event mechanism. First, the amount of explicit triggers in a program depends on whether the events can be implicit or not. Second, if dynamic binding is not provided, it is not possible to change the handler at runtime. Third, event composition helps the abstraction though it is not absolutely necessary. In an event mechanism the three properties are determined by how the bindings between the event and the handler are presented.

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

Listing 13. The State pattern example in DominoJ

```
1  public class UpdateQueueState {
2    private QueueEmpty empty = new QueueEmpty();
3    private QueueNormal normal = new QueueNormal();
4    private QueueFull full = new QueueFull();
5    private Queue queue = null;
6    public boolean emptyToNormal(Object o) {
7      normal.insert(o);
8      queue.setState(normal);
9      return $retval;
10   }
11     :
12   public void setup(Queue q) {
13     queue = q;
14     queue.setState(empty);
15     empty.insert += this.emptyToNormal;
16       :
17   }
18 }
```

different between event mechanisms. Table 4 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 14 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 in 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, DominoJ 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 &&

Table 4. 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	||	+=, ^=
		&& \ filter map empty any	use Java expression in the default closure of method slots
	Handler-to-Event	afterExec	+=
		beforeExec	^=
		imperative	explicit trigger is possible

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

Listing 14. Using DominoJ for the event-handler paradigm

```
1  public class Display {
2    public 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   public Shape(Display d) {
12     this.setPosition += d.refresh;
13   }
14 }
15 public class Test {
16   public static void main(String[] args) {
17     Display d = new Display();
18     Shape s = new Shape(d);
19     s.setPosition(0, 0);
20   }
21 }
```

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 predefined EScala-like events for the default closure, but it is not correct. In DominoJ's model the only 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 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 6, 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 6 (a) is the execution order of Listing 1, and Figure 6 (b) is the one of Listing 14. In the

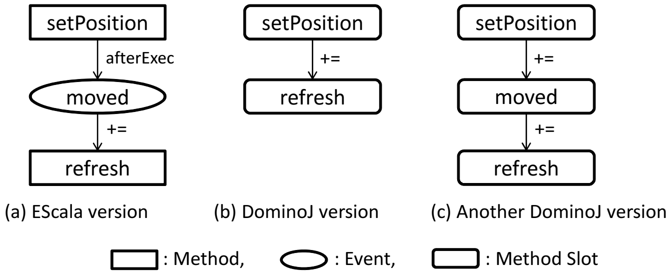

Fig. 6. The execution order of the shape example in EScala and DominoJ

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 6 (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 14 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 14 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 4. 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 to put 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[4]. 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.

4.5 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 AspectJ the important features such as around advices, the obliviousness, and intertype declaration that an event mechanism cannot provide are all supported by constructs. In this subsection first we analyze the necessary elements in the point-advice model in order to 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

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

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 intertype declaration and the reflection are not provided. According to the three elements, Table 5 lists the mapping of language constructs in AspectJ and DominoJ.

Table 5. 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

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 15. 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 16. 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

Listing 15. 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 16. 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 }
```

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 15. 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 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 7 (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 7 (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 8 shows an example of around advices 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 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 for 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.

4.6 Summary of the Coverage

In the previous subsections we have discussed what a language must have for the event-handler paradigm and the aspect paradigm by comparing with EScala and AspectJ, respectively, from the viewpoint of constructs. In this subsection we summarize the significant characteristics of the two paradigms and discuss the support in DominoJ as shown in Table 6. In addition to being used for the event-handler paradigm and the aspect paradigm, DominoJ allows programmers to use both paradigms together.

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 method slots 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 advices, the obliviousness, and inter-type declaration. In DominoJ what the around advices in AspectJ does

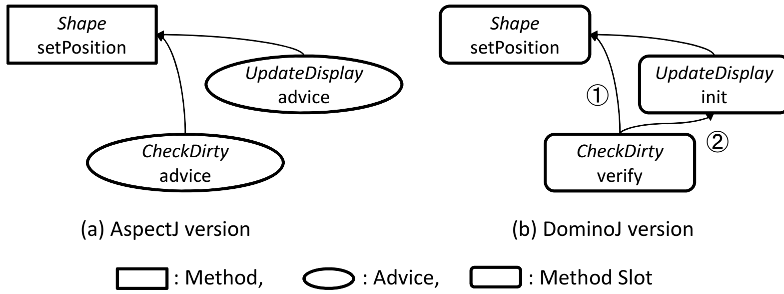

Fig. 7. Adding another advice to the shape example in AspectJ and DominoJ

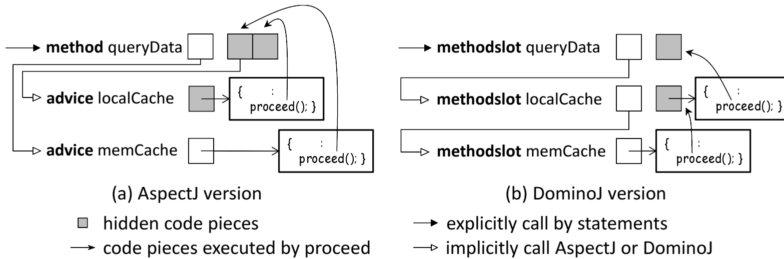

Fig. 8. Calling proceed in AspectJ and DominoJ

Table 6. A summary of the significant characteristics of the two paradigms and the support in DominoJ

the Event-handler paradigm	DominoJ
implicit events	yes
dynamic binding	yes
event composition	yes

the Aspect paradigm	DominoJ
around advices	yes
the obliviousness	yes
inter-type declaration	no

can be archived by assigning a closure calling another method slot using the = operator. DominoJ also supports the obliviousness in AspectJ by using the class construct as the aspect construct and attaching a method slot to a constructor of the target class. In the method slot attached to the constructor, programmers can further attach advices to the method slots at the target class. However, the intertype declaration in AspectJ 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.

4.7 Event-Handler vs. Aspect

Although the event-handler paradigm and the aspect paradigm are developed for resolving different issues, their implementation are almost the same, especially from the viewpoint of virtual machine. They both allow programmers to specify

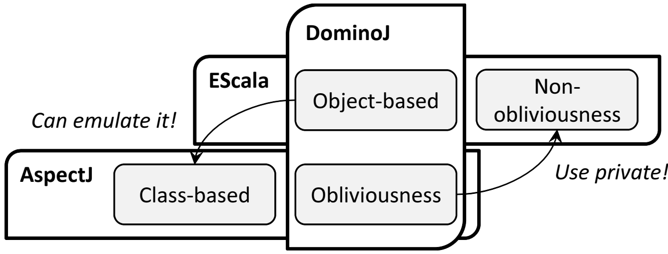

Fig. 9. The design decision of EScala, AspectJ, and DominoJ

which code pieces should be executed after/before the execution of a method. The only difference is the model of specifying and executing the code pieces. First, the behavior of the event-handler paradigm used in EScala is object-based, while the behavior of the aspect paradigm is class-based. Second, in the aspect paradigm used in AspectJ the obliviousness is an important property, but in the event-handler paradigm the non-obliviousness is expected. Obviously, it is impossible to support the contradictory properties at the same time unless we give both constructs into one language. If we just put constructs for the two paradigms into one language, for example providing all syntax of EScala and AspectJ in a new language, it makes the code compilcated and programmers have to learn all of them; it is not what we want to do. Our goal is to make all available by a single construct and let programmers decide how to use it, so we have to choose between object-based behaviors and class-based behaviors, the obliviousness and the non-obliviousness.

The design decisions of EScala, AspectJ, and DominoJ are shown in Figure 9. DominoJ chooses object-based behaviors and the obliviousness since we believe this design is most flexible—the class-based behaviors can be emulated by writing the bindings in the constructors and a method slot can be `private` if the obliviousness is not expected. In this sense, DominoJ can be regarded as either an object-based AOP language or an event mechanism. It does not matter how we call DominoJ since it is just a naming, but here we want to bring up the discussion on the similarities between the event-handler paradigm and the aspect paradigm.

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 does not create 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 a 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 researchers 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 to change 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 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.

References

1. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Commun. ACM 44(10), 51–57 (2001)
2. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: OOPSLA 2010, pp. 539–554. ACM (2010)
3. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in AmbientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
4. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: $\mathcal{F}$*ickle*: Dynamic object re-classification. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 130–149. Springer, Heidelberg (2001)
5. Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 186–211. Springer, Heidelberg (1998)
6. Eugster, P., Jayaram, K.R.: EventJava: An extension of Java for event correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
8. Garlan, D., Jha, S., Notkin, D., Dingel, J.: Reasoning about implicit invocation. In: SIGSOFT 1998/FSE-6, pp. 209–221. ACM (1998)
9. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD 2011, pp. 227–240. ACM (2011)

10. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: OOPSLA 2002, pp. 161–173. ACM (2002)
11. Haupt, M., Schippers, H.: A machine model for aspect-oriented programming. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 501–524. Springer, Heidelberg (2007)
12. Havinga, W., Bergmans, L., Aksit, M.: A model for composable composition operators: expressing object and aspect compositions with first-class operators. In: AOSD 2010, pp. 145–156. ACM (2010)
13. Kamina, T., Aotani, T., Masuhara, H.: EventCJ: a context-oriented programming language with declarative event-based context transition. In: AOSD 2011, pp. 253–264. ACM (2011)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
15. Hansen, K.A., Endoh, Y.: A fine-grained join point model for more reusable aspects. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 131–147. Springer, Heidelberg (2006)
16. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: OOPSLA 2009, pp. 1–20. ACM (2009)
17. Microsoft Corporation. C# language specification
18. Microsoft Corporation. Messages and message queues
19. Núñez, A., Noyé, J., Gasiūnas, V.: Declarative definition of contexts with polymorphic events. In: COP 2009, pp. 2:1–2:6. ACM (2009)
20. Oracle Corporation. OpenJDK: Project Lambda, `http://openjdk.java.net/projects/lambda/`
21. Orleans, D.: Incremental programming with extensible decisions. In: AOSD 2002, pp. 56–64. ACM (2002)
22. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
23. Schippers, H., Janssens, D., Haupt, M., Hirschfeld, R.: Delegation-based semantics for modularizing crosscutting concerns. In: OOPSLA 2008, pp. 525–542. ACM (2008)
24. Smith, R.B., Ungar, D.: Programming as an experience: The inspiration for Self. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 303–330. Springer, Heidelberg (1995)
25. Sun Microsystems. Abstract Window Toolkit, `http://java.sun.com/products/jdk/awt/`
26. The AspectJ Project, `http://www.eclipse.org/aspectj/`
27. The Boost Project. Boost.Signals, `http://www.boost.org/libs/signals/`
28. The JastAdd Project. JastAddJ: The JastAdd Extensible Java Compiler, `http://jastadd.org/web/jastaddj/`
29. The Qt Project. Signals & Slots, `http://qt-project.org/doc/signalsandslots`
30. The SAX project. Simple API for XML, `http://www.saxproject.org/`
31. The Self project, `http://selflanguage.org/`
32. The X.Org project. Xlib in X Window System, `http://www.x.org/`
33. Widom, J., Finkelstein, S.J.: Set-oriented production rules in relational database systems. In: SIGMOD 1990, pp. 259–270. ACM Press (1990)
34. Zhuang, Y., Chiba, S.: Applying DominoJ to GoF Design Patterns. Technical report, Dept. of Math. and Comp., Tokyo Institute of Technology (2011)