Context Holders: Realizing Multiple Layer Activation Mechanisms in a Single Context-Oriented Language

Tomoyuki Aotani  
Tokyo Institute of Technology  
aotani@is.titech.ac.jp

Testuo Kamina  
University of Tokyo  
kamina@acm.org

Hidehiko Masuhara  
Tokyo Institute of Technology  
masuhara@is.titech.ac.jp

Abstract

We propose LamFJ, a calculus for expressing various layer activation mechanisms in context-oriented programming languages. LamFJ extends FeatherweightJava with context holders, which are the abstraction of dynamic layer activation. By encoding programs with different layer activation mechanisms into a program manipulating context holders, LamFJ serves as a foundation to reason about interactions between different mechanisms. This paper presents a sketch of the context holders and encodings of existing layer activation mechanisms.

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

Keywords Context-oriented programming, layer activation mechanisms

1. Introduction

Context-oriented programming (COP) [4] is an approach to modularize behavioral variations of a program from the viewpoint of the context that changes during execution of a program. Changing behavior with respect to changes of the context is achieved by (1) changing the compositions of behavioral variations when the context changes and (2) finding a composition of behavioral variations and selecting a method body with respect to not only the dynamic type of the receiver object but also the composition when a method is called.

COP languages provide partial methods and layers to modularize behavioral variations and layer activation mechanisms to change the behavior of the program. A partial method in a layer describes a behavioral variation of a method. A layer activation mechanism specify how and when compositions of behavioral variations change.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FOAL ’14, April 22, 2014, Lugano, Switzerland.  
Copyright © 2014 ACM 978-1-4503-2798-5/14/04... $15.00.  
http://dx.doi.org/10.1145/2588548.2588552

Figure 1. Example of a class with a layer and a partial method

```java
class C{
    void m() {}
    layer L{ void m(){new C().m();} }
}
```

Figure 2. Example using imperative activation

```java
C c=new C();
deactivate(L);
activate(L);
c.m();
c.m();
```

US [8], one can implement his or her own layer activation mechanism through meta programming in JavaScript.

In this study, we propose LamFJ, a calculus for expressing various layer activation mechanisms. It extends FeatherweightJava [6] with layers, partial methods, context holders, contextual method invocation, and let-binding. Using context holders enables us to not only realize each layer activation mechanism separately but also use several mechanisms in one programming language. In this sense, LamFJ serves as an intermediate language for COP languages and foundation to reason about interactions between different mechanisms.

In this paper, we sketch LamFJ and the context holders, and show how we can realize each layer activation mechanism and use several mechanisms by using them.

2. Activation Mechanisms

This section gives a brief overview of the three layer activation mechanisms in COP languages, namely imperative activation, per-object activation, and syntactic activation.

Figure 1 shows class C that is commonly used to explain each mechanism. Class C has base method m and layer L, which defines a partial method to m in class C. When method m is invoked on an object of C, the program goes to line 3 if layer L is active and to line 2 otherwise. The active layers are usually managed as a sequence L1, L2, ..., Ln. Partial methods in Li precede the ones in Lj if i < j, and Li is activated more recently than Lj in most COP languages.

2.1 Imperative activation

In the imperative activation mechanism, there is exactly one sequence of active layers and its change affects every method dispatch process in the program. Figure 2 shows a simple example that uses imperative activation. Layer L is activated and deactivated in lines 2 and 4, respectively. The method invocation in line 3 and the subsequent method invocations execute partial method m in L.
class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

class C{
    let h=newCH() in
    Obj m(CH h){...} = let c=new C() in
    layer L{
        activate(L,h);
        Obj m(CH h) {
            c.m(h);  // new C().m(h);    // deactivate(L,h);
            c.m(h)
        }
    }
    Obj main() {
        ...
    }
}

Figure 5. Imperative activation using context holders

```plaintext
Figure 3. Example using per-object activation

```c
C c=new C();
C c1=new C();
activate(l,c1);
```c

Figure 4. Example using dynamic scope activation

```plaintext
Figure 3. Example using per-object activation

```c
c=new C();
c1=new C();
activate(l,c1);
```c

Figure 4. Example using dynamic scope activation

```plaintext
Figure 3. Example using per-object activation

```c
C c=new C();
with(l){ c.m(); }
```c

Figure 4. Example using dynamic scope activation

<table>
<thead>
<tr>
<th>Signatures</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH new(CH)</td>
<td>creates an empty context holder</td>
</tr>
<tr>
<td>CH activate(Layer,CH)</td>
<td>activates the layer</td>
</tr>
<tr>
<td>CH deactivate(Layer,CH)</td>
<td>deactivates the layer</td>
</tr>
<tr>
<td>CH clone(CH)</td>
<td>clones the context holder</td>
</tr>
<tr>
<td>CH merge(CH,CH)</td>
<td>merges the two context holders</td>
</tr>
<tr>
<td>Obj dispatch(m,CH,Obj)</td>
<td>executes a partial or base method</td>
</tr>
</tbody>
</table>

Table 1. Operations on CH

(line 3 in Figure 2) because they appear after the layer activation of L. On the other hand, the method invocation in line 5 executes base method m (line 2 in Figure 1) because it appears after the layer deactivation of L.

2.2 Per-object activation

In the per-object activation mechanism, layers are activated on each object. To activate a layer, the programmer specifies the object along with the name of the layer. Figure 3 shows a simple example that uses per-object activation. Layer L is activated on object c1 in line 3. The method invocation in line 4 executes the partial method m in layer L (line 3 in Figure 1) because L is active on object c1. The subsequent method invocation in partial method L.m (line 3 in Figure 1) executes base method m because L is not activated on the receiver object. Similarly, the method invocation in line 5 executes m.

2.3 Dynamic scoping

In the dynamic scoping activation mechanism, layers are activated using a block structure, namely the with-block, and the layers are active during the execution of the body of the block. Figure 4 shows a simple example that uses dynamic scoping activation. Line 2 activates the layer L by using the with-block, and the method invocation in the body executes partial method L.m. Subsequent method invocations also execute the partial method, because they are evaluated during the execution of the body of the with-block.

3. Context Holders

In this section, we informally explain context holders and contextual method invocation, and how we can realize in LamFJ the layer activation mechanisms described in Section 2.

Context holders are instances of the abstract data type CH defined as a collection of pairs of an activation event e and time t and six operations, namely newCH, activate, deactivate, clone, merge, and dispatch, which are summarized in Table 1. An activation event is either \( \alpha \) or \( \beta \), denoting that layer L is activated, or \( \delta \), denoting that L is deactivated. newCH() creates an empty context holder. activate(L,h) and deactivate(L,h) evaluated at t take a layer name L and context holder h and add a pair \( (\alpha_L, t) \) and \( (\delta_L, t) \) to h, respectively. clone(h) takes a context holder h and returns a deep clone of h, merge(h,h') takes two context holders h and h' and returns a new context holder that has every pair \( (\epsilon, t) \) in h or h'. dispatch(m,h,\sigma) takes a method name m, context holder h, and objects \( o_1, \ldots, o_n \), and executes a base or partial method on \( o_1 \) with respect to the sequence of active layers computed from h as follows. Let \( t_L \) and \( t_L' \) be the latest activation event on L and its time in h, respectively. Then in the resulting sequence of active layers, (1) every L such that \( t_L = t_L' \) appears and (2) L appears in front of L' if \( t_L \geq t_L' \).

A contextual method invocation is a method invocation that takes one context holder h along with the receiver and argument objects, i.e., \( e_0, m(e_0), (e_1, e_2, \ldots, e_n) \), where \( e_0 \) is an expression evaluated to a context holder and \( e_1, \ldots, e_n \) are the arguments. The semantics of contextual method invocation \( e_1.m(e_0)(e_2, \ldots, e_n) \) is given by dispatch(m, h, \sigma) where \( e_0 \) is the value obtained by evaluating \( e_i \) for each \( i \in \{1, \ldots, n\} \) and h is the value obtained by evaluating \( e_h \).

The following subsections show how we can realize each layer activation mechanism by using context holders in LamFJ.

3.1 Imperative activation

The imperative activation mechanism can be realized by creating exactly one context holder at the beginning of the execution of the program and using it for every method invocation and layer activation. In Figure 5, we rewrite class C (Figure 1) and Figure 2 to use context holders in this manner. In the declaration of class C (lines 1–7), the method invocation (line 5) in partial method L.m is changed so that it specifies the context holder h. Line 9 creates the context holder that are used by every layer activation and method invocation in the program. Lines 10–14 correspond to the code in Figure 2. To activate and deactivate layer L, the context holder h is explicitly specified (lines 11 and 13). The method invocations in lines 12 and 14 are also extended to specify the context holder h for method dispatching.

3.2 Per-object activation

The per-object activation mechanism can be realized by having a context holder for each object. In other words, we can get a program that uses the context holders by translating each object to a pair comprising the object and a context holder.

In Figure 6, we rewrite class C and Figure 3 to use context holders. Instead of using the pairs, each object of C has its context holder as a member (field h in line 2). Each method invocation to an object of C then uses the context holder by accessing its field h (lines 7, 14 and 15). Activating layer L on object c1 (line 13) is also changed so that it specifies the context holder in c1 instead of the object itself.
let h’=clone(h) in
Obj main(){
merge(gh,h) in
let mh=
layer L{
class C{
C c=new C();
}
}