Efficient Layer Activation for Switching Context-Dependent Behavior

Pascal Costanza[1], Robert Hirschfeld[2], and Wolfgang De Meuter[1]

[1] Vrije Universiteit Brussel, Programming Technology Lab, B-1050 Brussels, Belgium
{pascal.costanza, wdmeuter}@vub.ac.be
[2] Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract. Today's programming platforms do not provide sufficient constructs that allow a program's behavior to depend on the context in which it is executing. This paper presents the design and implementation of programming language extensions that explicitly support our vision of Context-oriented Programming. In this model, programs can be partitioned into layers that can be dynamically activated and deactivated depending on their execution context. Layers are sets of partial program definitions that can be composed in any order. Context-oriented Programming encourages rich, dynamic modifications of program behavior at runtime, requiring an efficient implementation. We present a dynamic representation of layers that yields competitive performance characteristics for both layer activation/deactivation and overall program execution. We illustrate the performance of our implementation by providing an alternative solution for one of the prominent examples of aspect-oriented programming.

1 Introduction

In Context-oriented Programming, programs consist of partial class and method definitions that can be freely selected and combined at runtime to enable programs to change their behavior according to their context of use. In [18], we have introduced this idea and presented the programming language ContextL which is among the first language extensions that explicitly realize this vision.[1] As a motivating example in that paper, an alternative implementation of the model-view-controller framework is illustrated that avoids any secondary non-domain classes and thus increases understandability and flexibility at the same time.

Context-oriented Programming encourages continually changing behavior of programs according to the context of use, and employs repeated changes to class and method definitions at runtime. Therefore, efficient implementation strategies are needed for Context-oriented Programming to become practical.

The contribution of this paper is a novel design and implementation that addresses these needs, yielding the desired efficiency characteristics.

[1] For example, we are also working on similar extensions to Smalltalk and Tweak called *ContextS* and *ContextT* respectively.

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 84–103, 2006.

2 Context-Oriented Programming

2.1 Motivation

In the following, we present examples that motivate the need to be able to write code with a meaning that is not fully self contained, but partially depends on the context in which it is deployed and executed.

- *Mobile applications* running on mobile devices might need to dynamically adjust their behavior according to the geographical context in which they are used [12].
- *Mobile code* typically depends on the context of the runtime environment in which it is executed, such as applets or software agents [21].
- *Exploration environments* create safe contexts in which to execute applications and can considerably help users to learn how to use them, as has been shown by research in the field of Human-Computer Interaction [46].

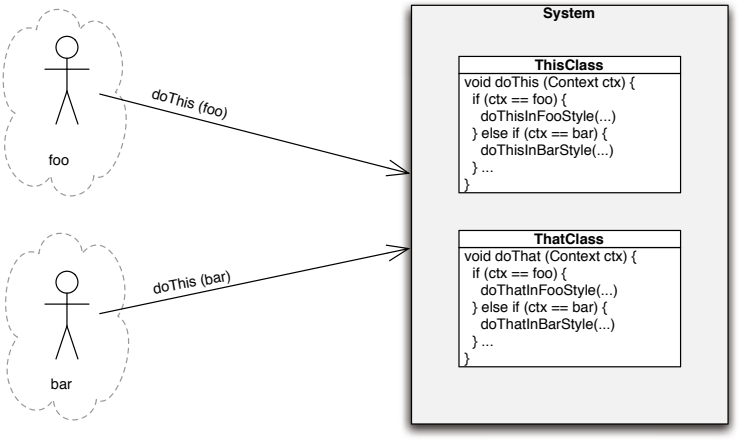

Fig. 1. Context-dependent behavior through `if` statements

With contemporary mainstream programming languages, the only way to introduce context-dependent behavior into a program is either by inserting `if` statements everywhere that check for the context in which a program is running (Fig. 1), violating one of the fundamental principles of object-oriented programming, namely to avoid `if` statements for achieving polymorphic behavior, or else by factoring out the context-dependent behavior into separate objects that can be substituted according to the context in which a program is used. Both approaches lead to unnecessarily complicated code that is hard to comprehend and even harder to maintain. Furthermore, they can only be applied for context

dependencies that are anticipated in the software development process. There are cases in which it is clearly not possible to foresee all context-dependent issues and without explicit support, it is difficult to write maintainable and robust code that handles them well. With Context-oriented Programming on the other hand, we can factor out partial class definitions into separate layers. As illustrated in Fig. 2, we can then, depending on the context of use, select different layers for further program execution. The principal notion of such layers as partial program definitions has been suggested before ([3,40], cf. the section on related work in this paper). In our approach, we extend this idea with the notion of dynamically scoped layer activation (see Sect. 2.4), resulting in a viable approach for expressing context-dependent behavior.

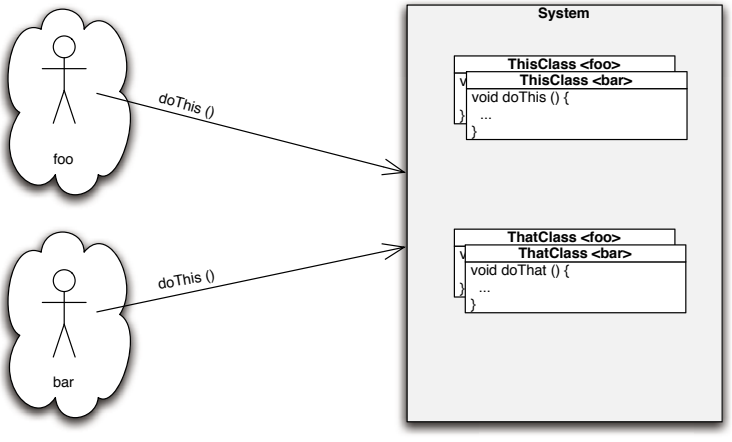

Fig. 2. Context-oriented Programming with layers

2.2 ContextJ/ContextL

ContextL is one of our first programming language extensions that explicitly support a context-oriented programming style [18]. While it is an extension to the Common Lisp Object System (CLOS, [4]), the features we describe are conceptually independent of that particular object model. In order to ease the accessibility of this paper, code examples are given in a Java-style syntax instead of the original Lisp syntax. This is possible because in this paper, we only deal with a subset of CLOS that is compatible with a similar subset of Java. Consequentially, we call this hypothetical Java-style language extension ContextJ, which we refer to in this paper when we discuss Java-specific issues. Since we are concerned with illustrating ContextL features using a Java-style syntax, we do not consider advanced Java language constructs that are not available or necessary in CLOS, like inner classes or generic types, but restrict ourselves to essentially the feature set of JDK 1.0. Adapting Java-specific features, like its static type system, to match the new constructs can be a topic for future work.

2.3 The Figure Editor Example

The figure editor [42] is a popular example widely used to motivate aspect-oriented programming. It is a variation of a similar example used to illustrate the notion of *jumping aspects* [8]. In this example, there is a class hierarchy for graphical objects of which instances are to be presented on a display. Some of these graphical objects are implemented by other, simpler graphical objects. So for example, a line is described by two end points. In order to move such objects to a different location, the contained simpler objects must be moved individually. Whenever the description of a graphical object is changed, its presentation on the screen should be updated accordingly.

The basic class hierarchy can be implemented in a plain object-oriented language as shown in Fig. 3: A figure element is described by an interface which is implemented by concrete classes, such as `Point` and `Line`.[2] Note that the required update of the display is not part of the code yet, but will be added in the following.

2.4 Layers

Layers are the essential extension provided by ContextL on which all subsequent features of ContextL are based. Layers can be defined by the `layer` construct:

```
layer DisplayLayer { /*...*/ }
```

Layers have a name and comprise partial class definitions, as shown in Sect. 2.5. There exists a predefined root layer. All class definitions that are not explicitly placed in a particular layer are by default associated with the root layer.

Layers can be activated and deactivated in the dynamic scope of a program:

```
with (DisplayLayer) { /* ... contained code ... */ }
without (DisplayLayer) { /* ... contained code ... */ }
```

Dynamically scoped layer activation/deactivation has the effect that the layer is active/inactive during execution of the *contained code*, including all the code that the *contained code* executes directly or indirectly. When the control flow returns from the dynamically scoped layer activation/deactivation, a layer's activation state is reverted to the previous state. This time interval between activation/deactivation of a layer and subsequent reversal to the previous activation state is also called the *dynamic extent* of the `with`/`without` block.

Layer activation can be nested, meaning that a layer can be activated/deactivated more than once in an individual flow of control. Furthermore, dynamically scoped layer activation/deactivation only affects the activity state of layers applied in the context of the current thread. The activity state of layers in other threads will remain unaffected.

[2] Since ContextJ would not need to change any of the existing Java language constructs, we can define and use interfaces, classes, fields, and methods as before.

```
interface FigureElement {
  void move (int dx, int dy);
}

class Point implements FigureElement {
  int x, y;

  Point(int newX, int newY) { this.x=newX; this.y=newY; }

  void setX(int newX) { this.x=newX; }
  void setY(int newY) { this.y=newY; }
  int getX() { return this.x; }
  int getY() { return this.y; }

  void move(int dx, int dy) { /*...*/ }
}

class Line implements FigureElement {
  Point p1, p2;

  Line(Point newP1, Point newP2) { this.p1=newP1; this.p2=newP2; }

  void setP1(Point newP1) { this.p1=newP1; }
  void setP2(Point newP2) { this.p2=newP2; }
  Point getP1() { return this.p1; }
  Point getP2() { return this.p2; }

  void move(int dx, int dy) { /*...*/ }
}
```

Fig. 3. A basic implementation of the figure editor example

2.5 Layered Classes

A class definition, or parts of it, can be associated with a specific layer:

```
layer DisplayLayer {
  class Display { /*...*/ }
  // ...
}
```

Here, such an association does not have a useful effect yet: The class can still be instantiated from any other layer. However, placing a class definition in a specific layer gets interesting when we use layers to add to the definition of a class that is already defined in another layer. In Fig. 4, we add the display update mechanism: The layer `DisplayLayer` contains a class `Display` that implements the code for updating the graphical representation of an object on a screen (not shown here). It also contains additional definitions for our classes `Point` and `Line` as well as the interface `FigureElement`, introducing `after` methods for the state changing

```
layer DisplayLayer {
  class Display {
    // ...
    static void update(FigureElement elm) { /*...*/ }
  }

  interface FigureElement {
    after void move (int dx, int dy) {
      Display.update(this);
    }
  }

  class Point {                          class Line {
    after void setX (int newX) {           after void setP1 (Point newP1) {
      Display.update(this);                  Display.update(this);
    }                                      }

    after void setY (int newY) {           after void setP2 (Point newP2) {
      Display.update(this);                  Display.update(this);
    }                                      }
  }                                      }
                                       }
```

Fig. 4. The `DisplayLayer` for the figure editor example

methods `setX`, `setY`, `setP1`, `setP2` and `move`. All these `after` methods contain calls to the `update` method of the `Display` class.

It is important to observe that the original classes `Point` and `Line`, and the interface `FigureElement` are not replaced. They still have their original definitions. The fact that the new extensions are placed in the `DisplayLayer` ensures that the respective `after` methods are executed when and only when the `DisplayLayer` is active. So an update of the display is just visible when the figure elements are changed in the dynamic extent of a `with (DisplayLayer) {...}` activation.

ContextJ would have to add `before`, `after` and `around` method qualifiers along the lines of `before`, `after` and `around` methods in CLOS.[3] They are methods of their own and are combined with other methods of the same signature. This is different from the advice-construct in AspectJ. AspectJ-style advice code adds behavior to pointcuts, that is collections of join-points, which are not necessarily methods, and not necessarily of the same signature. In our example, the `after` methods are all executed after the respective primary methods associated with the "root" layer in Fig. 3, but only if the `DisplayLayer` is active.

Due to the fact that layer activation/deactivation is confined to the current thread, display updates occur only in threads in which `DisplayLayer` is active, but not in other threads unless `DisplayLayer` is utilized within them as well.

[3] Indeed, the corresponding `before`, `after` and `around` methods in ContextL are just taken over from CLOS of which ContextL is an extension.

2.6 Nested Layer Activation/Deactivation

ContextL does not automatically activate layer definitions. Layers must be explicitly activated via the `with` construct to take effect. Indeed, layer activation is provided as a base-level language construct, so layers *can* be activated anywhere in a ContextL program, including layers that are loaded while a program is already running and also in classes that are loaded after a program is already running. One especially interesting case is the nesting of activation and deactivation of the same layer within the same control flow because this allows solving the phenomenon of *jumping aspects* without using AOP-style pointcuts, as shown below.

The figure editor example has been introduced in [8,42] to illustrate the jumping aspects phenomenon: Whenever we change the state of a simple graphical object, we can immediately update its presentation on the screen. However, when we change the state of a complex object that consists of other, simpler objects, the change has to be propagated to those simpler objects, but screen updates should be deferred and combined until all objects are changed that the complex object comprises. This has led to the introduction of `cflow`-style constructs in AspectJ and subsequent AOP approaches [31].

ContextL's `with` and `without` are base-level language constructs that allow us to achieve the effect of deferring the update on the screen by providing `around` methods instead of the above `after` definitions. Figure 5 contains a revised version of the `DisplayLayer` where `after` methods are replaced by `around` methods that deactivate the `DisplayLayer` before they `proceed`[4] to the respective primary method. This has the effect that the method definitions of the `DisplayLayer` are not executed during the extent of these calls to `proceed`, so no display update will take place here. Only after leaving the `without` block, the `update` method is called eventually, and only once.

Now, a crucial question is whether continually activating and deactivating layers is a reasonable approach with regard to efficiency considerations. Sect. 4 discusses this question after the presentation of our implementation approach.

3 Implementation

This section presents an implementation of the language constructs introduced in the previous section. ContextL is an extension to CLOS. In our description we focus on the implementation strategy that is reusable in other languages, without going too much into the CLOS-specific details. The general idea is this:[5]

- Layers are implemented internally as classes.
- Combinations of currently active layers are represented as classes that inherit from the primary layer classes using multiple inheritance.

[4] Similar to `proceed` in AspectJ and `call-next-method` in CLOS.

[5] Some of these building blocks do not exist in languages like Java and C#, especially multiple dispatch and multiple inheritance. However, Section 3.4 refers to existing approaches that can be used for implementing them in those languages.

```
layer DisplayLayer {
  class Display {
    // ...
    static void update(FigureElement elm) { /*...*/ }
  }

  interface FigureElement {
    around void move (int dx, int dy) {
      without (DisplayLayer) { proceed(); }
      Display.update(this);
    }
  }

  class Point {                          class Line {
    around void setX (int newX) {          around void setP1 (Point newP1) {
      without (DisplayLayer)                 without (DisplayLayer)
        { proceed(); }                         { proceed(); }
      Display.update(this);                  Display.update(this);
    }                                      }

    around void setY (int newY) {          around void setP2 (Point newP2) {
      without (DisplayLayer)                 without (DisplayLayer)
        { proceed(); }                         { proceed(); }
      Display.update(this);                  Display.update(this);
    }                                      }
  }                                      }
                                       }
```

Fig. 5. The `DisplayLayer` with `around` methods to defer the `update` of the display

- A dynamically scoped variable contains a prototype instance of such a layer combination class.
- Multiple dispatch is used to dispatch on both the currently active combination of layers and the receiver of a message.
- Efficiency is gained by providing fast caches for layer combinations and reusing efficient implementations for multiple inheritance and multiple dispatch.

3.1 Layers as Classes

Primary layers. Layers which are explicitly introduced by a programmer are called *primary layers*. For example, the following declaration defines a primary layer.

```
layer DisplayLayer { /*...*/ }
```

In the ContextL implementation, such primary layers are internally supplemented by dynamically generated layers which programmers cannot directly refer to (see below).

Layers are implemented as classes. Each layer declaration is represented internally by a corresponding class. So for example, the above `DisplayLayer` is internally represented by the following class.

```
class DisplayLayer { /*...*/ }
```

Active layers are combinations of such primary layers. Active layers are represented by dynamically generated *combination classes* which are ordinary classes that inherit from the classes that represent primary layers. Multiple inheritance is used to connect the various static and dynamic layer representations to create a chain of active layers. In Fig. 6, a combination of `Layer1`, `Layer2`, `Layer3`, and the `RootLayer` is realized as a combination class named `Layer1+2+3*` that inherits both from `Layer1` and a combination class named `Layer2+3*` that represents `Layer2`, `Layer3`, and the `RootLayer`. The latter combination class is in turn formed by inheriting from both `Layer2` and a combination class named `Layer3*` that represents `Layer3` and the `RootLayer`, and so forth.

Multiple inheritance typically results in the possible occurence of conflicting inherited members and thus in the need to determine a linearization of all superclasses [2]. However, in our case the linearization of layers is trivial: Each dynamically generated combination class has exactly two superclasses, one static layer representation (such as `DisplayLayer`, `Layer1`, `Layer2`, etc.) and one previous dynamic layer representation (such as `Layer1+2+3*`, etc.). For each combination class, the static layer representation takes precedence over the previous dynamic layer representation. Therefore after each layer activation, the most recent combination class comes first, followed by the most recently activated layer, followed by the previous combination class, and so on, which naturally leads to the required ordering of layers. For example in the combination illustrated in Fig. 6, the linearization of the class hierarchy starting from `Layer1+2+3*` is `Layer1+2+3*, Layer1, Layer2+3*, Layer2, Layer3*, Layer3, RootLayer` in that order.

Different layer combinations can coexist in the same program. Figure 7 illustrates how both a combination of layers `Layer1` and `Layer3` and a combination of layers `Layer1` and `Layer2` can exist at the same time. Indeed, any combination can be built without interfering with other combinations. Note that this allows the implementation to reflect the order in which layers are activated and deactivated: Whenever a layer is activated, it is ensured that it will be placed in front of all other already active layers. When it is already active itself, it will nevertheless be placed in front of all other already active layers from which it is implicitly removed beforehand as part of the activation process.

For example, assume layers `Layer1` and `Layer3` are already active in that order. An activation of layer `Layer2` will lead to a chain of layers `Layer2`, `Layer1` and `Layer3` in that order. Given that latter order, an activation of the (already active) layer `Layer1` will internally lead to first a deactivation and a subsequent reactivation of layer `Layer1`, and thus to a chain of layers `Layer1`, `Layer2` and `Layer3` in that order.

The various possible combinations do not have to be determined at compile time, but can be created on demand at runtime if the given language allows for creating classes at runtime (as is possible in CLOS, Smalltalk, Java, and so on)

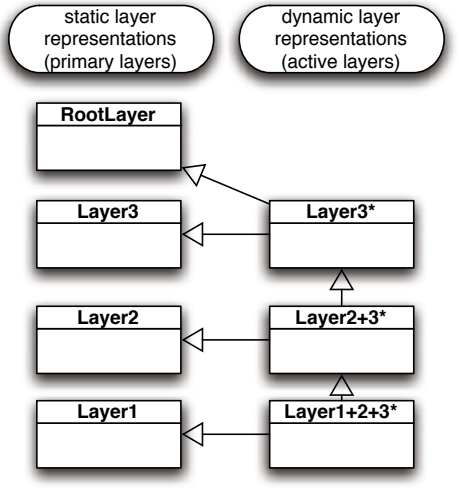

Fig. 6. Static and dynamic layers

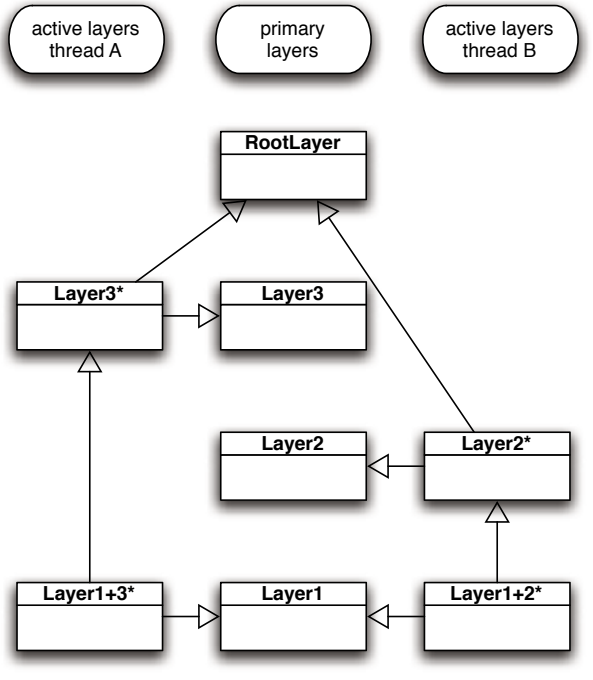

Fig. 7. Different layer combinations in different threads

so that only the actually required combinations are ever created. Layer combinations only need to be created once, because when they have been created they can be cached and reused. Layers are always activated/deactivated relative to the currently active layer combination. Therefore, such caches can be associated with the dynamically generated classes that represent layer combinations and only need to include the new combinations relative to those combinations. This enables the use of very small and fast caches.

3.2 Dynamic Scoping

The `with` and `without` constructs activate/deactivate layers with dynamic extent, that is, they effectively implement dynamic scoping for layers, including the fact that activations/deactivations are confined to the current thread and so do not interfere with activations/deactivations in other threads.

Such a dynamically scoped activation scheme can be easily implemented if the underlying language offers dynamically scoped variables which can already be rebound without affecting other threads [15]. We can then instantiate the class that represents the currently active combination of layers and store it in a dynamically scoped variable. By default, that variable contains an instance of the root layer representation, and can later be rebound to contain instances of the corresponding layer combinations. Such instances are called *prototypical* because they do not contain any state or behavior of their own, but are just used to select the correct behavior for layered classes and methods (see below).

Common Lisp provides dynamically scoped variables directly [41] and Java allows their simulation by storing a stack data structure in a thread-local variable [11]. Thread locality ensures non-interference with other threads, and the stack allows shadowing a previous layer combination with a new one by pushing the new layer combination at the beginning of the execution of a `with`/`without` block, and by popping it at the end.

3.3 Method Invocation

Having modelled layers as classes and layer combinations as dynamically generated classes, we turn to the question of which methods to execute in response to a message. It becomes obvious that this depends on both the class that represents the currently active layer combination and the class of the message receiver. In other words, we need *multiple dispatch*. Common Lisp already has multimethods, and it is possible to add multimethods to Java – see for example MultiJava [13]. To illustrate this further, Figure 8 shows how the method `move` from our figure editor example can be understood to be internally mapped to a multimethod definition using a combination of MultiJava and ContextJ syntax. The `Layer@SomeLayer` notation used in Fig. 8 is taken from MultiJava and specifies that the corresponding parameter is of the (static) type `Layer` but further specialized to be applicable only when the respective parameter is an instance of

```
// in the root layer
class Point implements FigureElement {
  // ...
  void move (Layer@RootLayer layer, int dx, int dy)
  { /*...*/ }
}

layer DisplayLayer {
  // ...
  class Point {
    // ...
    around void move (Layer@DisplayLayer layer, int dx, int dy)
    { /*...*/ }
  }
}
```

Fig. 8. Internal mapping of layered methods to multimethods

`SomeLayer` at runtime. How multimethods in MultiJava are translated into Java bytecode on a per-compilation-unit basis is described in [13].[6]

3.4 Putting It All Together

We have implemented ContextL as an extension to CLOS in a relatively straightforward way. This is because CLOS provides all the necessary ingredients described above, namely dynamic class generation, multiple inheritance, dynamically scoped variables, and multiple dispatch. As indicated, a similar implementation could in principle be achieved in a Java-based implementation as well: Classes can indeed be generated at runtime [20], a variant of dynamically scoped variables is already present in the form of thread-local variables [11], and it has already been described how to add multiple dispatch [13]. Currently, it is not obvious to us how to incorporate the required multiple inheritance mechanism into Java. However, subsets of multiple inheritance and their implementation have already been described, for example, for C# based on traits [37] and for Java based on interfaces with default implementations [35]. It is likely that such subsets are sufficient to support our model, but this needs to be explored further to be answered appropriately.

Note that the implementation we describe is only one possible implementation of layers and dynamically scoped layer activations. For example, we have a prototypical implementation of a minimal version of ContextL that is solely based on dynamically scoped instance variables, a construct of ContextL not described here ("special slots", see [18]). However, the implementation described

[6] Note that MultiJava implements symmetric dispatch while CLOS implements asymmetric dispatch by default. This issue would need to be addressed in an implementation of ContextJ. In the current ContextL implementation, the layer argument in a layered function/method has least priority with regard to argument precedence order.

in this paper yields competitive performance characteristics because multiple active layers are always represented by exactly one generated class. In comparison, straightforward implementation techniques for `cflow`-style constructs in aspect-oriented language implementations introduce `if`-tests for each pointcut that contains a `cflow` expression which is reported to lead to substantial runtime overheads [19]. In our approach, computational overhead occurs exclusively on the first activation/deactivation of a previously unused combination of layers and on the first message send in a previously unused combination of methods [30]. After that, both lookups of layer combinations and method dispatches take advantage of highly efficient caches.

4 Benchmarks

We have used the figure editor example described in Sect. 2.3 as the basis for a benchmark that measures the effect of layer activation and deactivation. In order to measure only the method dispatch and layer activation/deactivation overhead, no actual updates on the screen are implemented, but instead a global counter is incremented on each call of the `Display.update()` method to check the correct number of issued updates at the end of a test run.

We have implemented the benchmark in ContextL and have run the benchmark on six different Common Lisp implementations. We have run two versions of the benchmark, one without and one with layer activations/deactivations. In other words, we have compared the program in Fig. 3 that does not issue any display updates with the program in Fig. 5 that continually switches the `DisplayLayer` on and off: on to enable display updates and off to disable display updates for calls to `proceed` in the `around` methods of the `DisplayLayer`. The main loop of the latter version looks as follows:

```
for (int i=0; i<1000; i++) {
   for (Line line: lines) {
      with (DisplayLayer) {
         line.move(5, -5);
      }
   }
   for (Line line: lines) {
      with (DisplayLayer) {
         line.move(-5, 5);
      }
   }
}
```

The main loop of the version without layer activations/deactivations just omits the `with` blocks around the `line.move()` calls. It is important to note that the version without layer activations/deactivations is essentially just a plain CLOS program.

The results of the various runs on different Common Lisp implementations is presented in Fig. 9. Each run creates a collection of 100 lines, with each line being

Implementation	Platform	Without Layers	With Layers	Overhead
Allegro CL 7.0	Mac OS X	2.292 secs	2.540 secs	10.82% slower
CMUCL 19b	Mac OS X	0.7812 secs	0.7361 secs	7.8% *faster*
LispWorks 4.4	Mac OS X	3.0928 secs	3.1768 secs	2.72% slower
MCL 5.1	Mac OS X	2.3506 secs	2.6412 secs	12.36% slower
OpenMCL 0.14.3	Mac OS X	2.2448 secs	2.5066 secs	11.66% slower
SBCL 0.9.4	Mac OS X	0.8363 secs	0.7795 secs	7.29% *faster*
CMUCL 19a	Linux x86	0.76 secs	0.836 secs	10% slower
SBCL 0.9.4	Linux x86	0.5684 secs	0.638 secs	12.24% slower

Fig. 9. The results of running the figure editor example in various Common Lisp implementations

moved 1000 times. Time required for creating the collection of lines and filling it is not taken into account. The entries in Table 9 are average measurements of five runs. The respective platforms are an Apple PowerBook 1.67 GHz PowerPC G4 running Mac OS X 10.4.2 and a Dell PowerEdge 1600SC dual Xeon 2.8 Ghz running Linux 2.6.12. The overheads in runtime range from very moderate 2.72% in LispWorks for Macintosh to still reasonable 12.36% in Macintosh Common Lisp (MCL), especially when taking into account that we have an additional update of a global counter for each call of `line.move()`. Two implementations show the anomaly that the runs that repeatedly switch layers on and off are actually *faster* than the runs without layers: On CMUCL 19b, the runs without layers are on average 7.8% slower, and on SBCL 0.9.4 they are 7.29% slower. These two environments are based on the same Common Lisp compiler, so this provides an explanation for them showing similar efficiency characteristics. The performance anomaly as such may seem surprising, but in fact such anomalies occur frequently in performance benchmarks [25]. Obviously, factors beyond layer activation/deactivation and method dispatch play a more important role for the overall performance of our test program. Since applications typically spend less than 10% of the overall time in call overhead [28], our numbers suggest an overall estimated cost of 0.3% to 1.3% for inclusion and repeated switching of layers. This is a noteworthy result, despite the fact that, of course, more benchmarks are necessary to measure the effects of, for example, combinations of multiple layers.

This excellent performance is evidently the result of a combination of finding an appropriate runtime representation of layers and reusing existing optimizations for implementing object-oriented language constructs as described in the previous section. It stems from folding all active layers into a single class that represents current combination of active layers, and specializing the involved methods on an implicit argument in addition to the receiver of a message. Ultimately, our implementation relies on efficient multiple dispatch as provided by modern CLOS implementations. See [30] for a discussion of implementation techniques for multiple dispatch in CLOS, and [10] for a general overview.

5 Related Work

5.1 Dynamic Aspect Weaving

The only aspect-oriented technologies we are aware of approaching our notion of dynamically scoped activation of partial program definitions are AspectS [26,27], LasagneJ [43], CaesarJ [34] and the Steamloom virtual machine [6]. They all add constructs for thread-local activation of partial program definitions at the base-program level. However, CaesarJ is limited in that it does not provide a corresponding thread-local deactivation construct, and LasagneJ is even further limited in that it restricts the use of thread-local activation to the `main` method of a Java program [36]. Their lack of thread-local deactivation constructs makes `cflow`-style constructs necessary in those approaches to implement the figure editor example. Our approach allows its modular implementation without using AOP-style pointcuts. Global (non-thread-local) activation/deactivation constructs, like in CaesarJ and ObjectTeams [45] are not sufficient in this regard. Steamloom provides undeployment of thread-local aspects, but cannot thread-locally undeploy a globally active aspect.

With regard to efficiency considerations, it is important to note that the straightforward technique to implement activation/deactivation by using thread-local flags that are subsequently checked for each message impose a substantial runtime overhead, as is reported in [19]. We are aware of two approaches that specifically address efficiency improvements for `cflow`-style constructs and thread-local aspect activation/deactivation respectively.

In [1], optimizations are described that reduce the number of flags to be checked at runtime, with considerable efficiency improvements for `cflow`-style pointcuts. However, the basic implementation as described above remains the same. The main disadvantage of their approach is its reliance on a time-consuming static global program analysis.[7]

In contrast, we gain high runtime performance without limiting applicability of layers to those that have already been available at compile time. In our approach, no dedicated global analysis is required.

The Steamloom virtual machine [6] is another attempt to reduce the overhead of `cflow`-style poincuts. It implements a `deploy` statement that can be used to activate aspects in its dynamic extent by modifying the Jikes virtual machine for Java. It avoids the use of flags for checking applicability of aspects by recompiling the program at each context switch. That paper reports a considerable efficiency improvement for the remaining part of the program execution in the dynamic extent of a `deploy` block when compared to traditional implementation strategies for similar `cflow`-style constructs. However, the `deploy` statement as such is extremely expensive since it recompiles all parts of the program that are affected by such aspect deployment. The benchmark results provided in that paper suggest a performance decrease by a factor of 30, compared to their

[7] For example, a simple AspectJ program that takes less than 5 secs to be compiled with the plain AspectJ compiler can easily take more than 5 mins with their compiler.

original example program without any aspects. [8] The Steamloom manual discusses these "remarkable performance penalties" [5] in conjunction with the display updating aspect in their version of the figure editor example which is triggered by frequently entered and exited control flows.

The latest implementation of Steamloom explicitly addresses the above issues and is described in [7]. That paper reports considerable performance gains of `cflow`-style constructs, and would therefore be a viable candidate for an implementation of ContextJ. As future work, we plan to explore this option and compare the implementation approaches of ContextL and Steamloom.

5.2 Delegation Layers

Delegation layers, as in the prototype-based languages Slate [39] and Us [40] and also combined into a class-based programming language in [38], are very similar to our approach. As in ContextL, delegation layers define layers that group behavior for sets of objects in [39,40] and for sets of classes in [38]. However, the hierarchy of layers is globally fixed in [38]. One can select a layer from which to start a specific message send, but all the other layers below are then predetermined by the original configuration of layers. In [39] and [40], the selection and ordering of layers is not fixed but layers can be arbitrarily recombined in the control flow of a program. However, layer selection and combination has to be done manually, there are no dedicated `with/without` constructs like in ContextL. Providing these constructs as high-level abstractions allows for less straightforward, but more efficient implementation strategies.

5.3 Other Related Work

Related work for special functions, precursors for combinations of methods from different layers, is discussed in [15,16]. Related work for special slots is discussed in [16,17]. Related work for delegation is discussed in [18].

The term Context-oriented Programming has already been used in two other contexts. Gassanenko [22,23] describes an approach to add object-oriented programming concepts to Forth without turning it into an actual object-oriented programming language. Instead, a notion of context is added that essentially comes down to some form of first-class environments [24]. This allows code to behave differently when executed in different environments. The description in Gassanenko's papers focuses on Forth-specific details and it is very hard to tell how much overlap, if any, exists with our approach. For example, it is not clear whether Gassanenko's contexts must be fully defined or can be partial and combinable. The examples provided in [23] only cover fully specified, but no partial contexts. Furthermore, Gassanenko's contexts seem to cover functions only, neither state nor class definitions, the latter due to the explicit goal not to turn Forth into a fully object-oriented programming language. Therefore, it seems

[8] See column "no aspect" compared to column "`cflow`/dynamic" in the Steamloom row of Table 2 in that paper.

that those contexts are most likely similar to dynamically scoped functions [15], one of our own precursors to ContextL.

Keays and Rakotonirainy [29] use the term context-oriented programming for an approach that separates code skeletons from context-filling code stubs that complete the code skeleton to actually perform some behavior. The claimed advantage is that the code stubs can vary depending on the context, for example the device some code runs on. A proof-of-concept implementation using Python and XML is described. Their approach appears to be a reverse macro expansion framework in which code skeletons and code stubs need to be combined at runtime. Furthermore, there is no mention whether different combinations of skeletons and stubs can coexist at the same time.

In contrast, ContextL is essentially an extension to an object-oriented approach that does not rely on runtime source code transformation. ContextL's root layer, whose behavior can be altered in other layers, can already be fully operational, and different combinations of different layers can be simultaneously active in multiple threads.

6 Conclusions

Several examples suggest the need for programming language constructs that allow explicit association of the meaning of code not only with its position in a static hierarchy, but also with the context in which it is running. This is what we call Context-oriented Programming. The essential ideas are exemplified by ContextL's layers which are presented using a Java-style syntax. ContextL allows for partial class definitions that belong to individual layers. Layers can be activated and deactivated with dynamic extent.

We present an implementation of ContextL that relies on CLOS's multiple dispatch, and an analysis on how ContextL constructs can be implemented in more mainstream programming languages such as Java. The experiments with ContextL show that the concepts presented in the paper can be implemented efficiently. A ContextL program with repeated activations and deactivations of layers is about as efficient as one without.

We show that context-dependent layers can be used to implement the popular figure editor example in an elegant and very efficient way, without using aspect-oriented features. Most notably, no `cflow`-style construct is necessary to implement the full example because ContextL includes constructs not only for thread-local activation, but also for thread-local deactivation of layers.

Acknowledgements

We thank Thomas F. Burdick, Brecht Desmet, Johan Fabry, Michael Haupt, Bjoern Lindberg, Oscar Nierstrasz, Andreas Raab, Christophe Rhodes, Dave Thomas, Peter J. Wasilko, and JonL White for fruitful discussions and valuable contributions.

References

1. Pavel Avgustinov, Julian Tibble, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam. Optimizing AspectJ. Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation.
2. Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. OOPSLA '96, Proceedings.
3. Daniel Bobrow and Ira Goldstein. Representing Design Alternatives. Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior. Amsterdam, July 1980.
4. Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon. Common Lisp Object System Specification. Lisp and Symbolic Computation 1, 3-4 (January 1989), 245-394.
5. Christoph Bockisch, Tom Dinkelaker, Michael Haupt, Michael Krebs. The Steamloom Manual, December 2004. Available: http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp
6. Christoph Bockisch, Michael Haupt, Mira Mezini, Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. AOSD 2004, Proceedings, ACM Press.
7. Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, Mira Mezini. Efficient Control Flow Quantification. OOPSLA 2006, Proceedings, ACM Press.
8. Johan Brichau, Wolfgang De Meuter, Kris De Volder. Jumping Aspects. ECOOP 2000 International Workshop on Aspects and Dimensions of Concerns, 2000.
9. Martin Büchi and Wolfgang Weck. Generic Wrappers. ECOOP 2000, Proceedings, Springer LNCS.
10. Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. OOPSLA '99, Proceedings.
11. Patrick Chan. *The Java Developers Almanac 1.4, Volume 1: Examples and Quick Reference*. Addison-Wesley Professional, 2002.
12. Guanlin Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, Hanover, USA, November 2000.
13. Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design Rationale, Compiler Implementation, and Applications. ACM Transactions on Programming Languages and Systems (TOPLAS), 28, 3 (May 2006), 517-575.
14. Pascal Costanza, Günter Kniesel, Armin Cremers. Lava – Spracherweiterungen für Delegation in Java. JIT '99 – Java-Informations-Tage 1999. Springer, Informatik Aktuell, 1999.
15. Pascal Costanza. Dynamically Scoped Functions as the Essence of AOP. ECOOP 2003 Workshop on Object-oriented Language Engineering for the Post-Java Era, Darmstadt, Germany, July 22, 2003. ACM Sigplan Notices 38, 8 (August 2003).
16. Pascal Costanza. A Short Overview of AspectL. European Interactive Workshop on Aspects in Software (EIWAS'04), Berlin, Germany, September 23-24.
17. Pascal Costanza. How to Make Lisp More Special. International Lisp Conference 2005, Stanford. Proceedings.
18. Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming. ACM Dynamic Languages Symposium, San Diego, USA, 2005. Proceedings.

19. Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, Clark Verbrugge. Measuring the dynamic behavior of AspectJ programs. In: *OOPSLA 2004*, Proceedings, ACM Press.
20. Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications Co., 2004.
21. Alfonso Fugetta, Gian Pietro Picco, Giovanni Vigna. Understanding Code Mobility. IEEE Transactions on Software Engineering, Vol. 24, No. 5, May 1998.
22. Michael Gassanenko. Context-oriented Programming: Evolution of Vocabularies. Proceedings of the euroFORTH'93 Conference. Marianske Lazne, Czech Republic.
23. Michael Gassenenko. Context-oriented Programming. euroFORTH'98, Schloss Dagstuhl, Germany.
24. David Gelernter, Suresh Jagannathan, Thomas London. Environments as First Class Objects. POPL '87, Proceedings.
25. Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, Michael Hind. Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. OOPSLA 2004, Proceedings.
26. Robert Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland (eds.), *Objects, Compononts, Architectures, Services, and Applications for a Networked World*, pp. 226-232, LNCS 2591, Springer, 2003.
27. Robert Hirschfeld and Pascal Costanza. Extending Advice Activation in AspectS. European Interactive Workshop on Aspects in Software (EIWAS 2005), Brussels, Belgium, 2005.
28. Urs Hölzle, personal communication, 1999.
29. Roger Keays and Andry Rakotonirainy. Context-oriented Programming. International Workshop on Data Engineering for Wireless and Mobile Access, San Diego, USA, 2003. ACM Press.
30. Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. Proceedings of the 1990 ACM conference on LISP and Functionl Programming.
31. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ. ECOOP 2001, proceedings.
32. Günter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. ECOOP '99, Proceedings, Springer LNCS 1628.
33. Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems. OOPSLA '86, Proceedings.
34. Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In: *2nd International Conference on Aspect-Oriented Software Development* (AOSD 2003). Boston, USA, March 17-21, 2003, 90-100. ACM Press.
35. Markus Mohnen. Interfaces with Default Implementations in Java (extended abstract). Conference on the Principles and Practice of Programming in Java, Dublin, Ireland, June 2002. Proceedings.
36. Adriaan Moors, Jan Smans, Eddy Truyen, Frank Piessens, Wouter Joosen. Safe language support for feature composition through feature-based dispatch. Position paper at 2nd Workshop on Managing Variabilities Consistently in Design and Code (MVCDC2005), OOPSLA 2005, San Diego, California, USA, October 20, 2005.
37. Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, Nathanael Schärli. *Adding Traits to (Statically Typed) Languages*. Technical report no. IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.
38. Klaus Ostermann. Dynamically Composable Collaborations with Delegation Layers. ECOOP 2002, Proceedings, Springer LNCS.
39. Lee Salzman and Jonathan Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. ECOOP 2005, Proceedings, LNCS.

40. Randall Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. Theory and Practice of Object Systems, 2, 3 1996.
41. Guy L. Steele. *Common Lisp the Language, 2nd Edition.* Digital Press, 1990.
42. Perri Tarr, Maja D'Hondt, Lodewijk Bergmans, Cristina Videira Lopes. Worshop an Aspects and Dimensions of Concerns: Requirements on, and Challenge Problems For, Advanced Separation of Concerns. In: *Object-oriented Technology: ECOOP 2000 Workshops, Panels, and Posters*, Sophia Antipolis and Cannes, France, June 2000. Proceedings. Springer LNCS 1964.
43. E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, B.N. Jorgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In: *Proceeedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, May 12-19, (2001) 233-242.
44. David Ungar and Randall Smith. Self: The Power of Simplicity. OOPSLA '87, Proceedings.
45. Matthias Veit and Stephan Herrman. Model-View-Controller and ObjectTeams: A Perfect Match of Paradigms. In: *2nd International Conference on Aspect-Oriented Software Development* (AOSD 2003). Boston, USA, March 17-21, 2003, 90-100. ACM Press.
46. Volker Wulf and Björn Golombek. Exploration Environments: Concept and Empirical Evaluation. In: *Proceedings of GROUP 2001*, ACM 2001 International Conference on Supporting Group Work. Boulder, Colorado, USA, September 30 - October 3, 2001. ACM 2001.