ARTICLE IN PRESS

Science of Computer Programming ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

An open implementation for context-oriented layer composition in ContextJS

Jens Lincke *, Malte Appeltauer, Bastian Steinert, Robert Hirschfeld

Software Architecture Group, Hasso-Plattner-Institute, University of Potsdam, Germany[1]

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
ContextJS
Context-oriented programming
Open implementations
Dynamic adaptation
Scope

ABSTRACT

Context-oriented programming (COP) provides dedicated support for defining and composing variations to a basic program behavior. A variation, which is defined within a layer, can be de-/activated for the dynamic extent of a code block. While this mechanism allows for control flow-specific scoping, expressing behavior adaptations can demand alternative scopes. For instance, adaptations can depend on dynamic object structure rather than control flow. We present scenarios for behavior adaptation and identify the need for new scoping mechanisms. The increasing number of scoping mechanisms calls for new language abstractions representing them. We suggest to open the implementation of scoping mechanisms so that developers can extend the COP language core according to their specific needs. Our open implementation moves layer composition into objects to be affected and with that closer to the method dispatch to be changed. We discuss the implementation of established COP scoping mechanisms using our approach and present new scoping mechanisms developed for our enhancements to Lively Kernel.

1. Introduction

The evolution of a software system is a critical task that often increases the application's complexity. Projects typically start out with a well defined set of requirements and target a specific user group. Over time, systems evolve; their core functionality is extended to support additional features, or is customized for new users. This evolution often requires the specification of variations of existing behavior depending on the context of the new usage. The representation of these context-dependent *behavioral variations* at programming language level requires support for dynamic system adaptation and composition of variations.

Context-oriented programming [18,12] (COP) extends object-oriented programming by providing dedicated language abstractions for defining and composing variations to basic program behavior. Behavioral variations are encapsulated by *layers*, modules that can crosscut classes. Layers can be dynamically de-/activated – and composed with other layers – for the dynamic extent of a code block. This mechanism allows for scoping behavioral variations to specific control flows.

Although COP does not prescribe a certain implementation strategy, most of the COP implementations described in the literature scope layer activations to the dynamic extent of a block of statements [12,17,4,3]. For many control flow driven applications, dynamic extent-based layer composition is an appropriate mechanism. However, behavior adaptations in general can also depend on scopes other than the control flow, such as a dynamic object structure or object state. Having conducted several case studies in which we applied COP to various projects [27,26] related to Lively Kernel [21], we identified needs for scoping strategies different from what has been proposed and implemented so far.

* Corresponding author.
E-mail addresses: jens.lincke@hpi.uni-potsdam.de (J. Lincke), malte.appeltauer@hpi.uni-potsdam.de (M. Appeltauer), bastian.steinert@hpi.uni-potsdam.de (B. Steinert), robert.hirschfeld@hpi.uni-potsdam.de (R. Hirschfeld).
[1] http://www.hpi.uni-potsdam.de/swa.

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.013

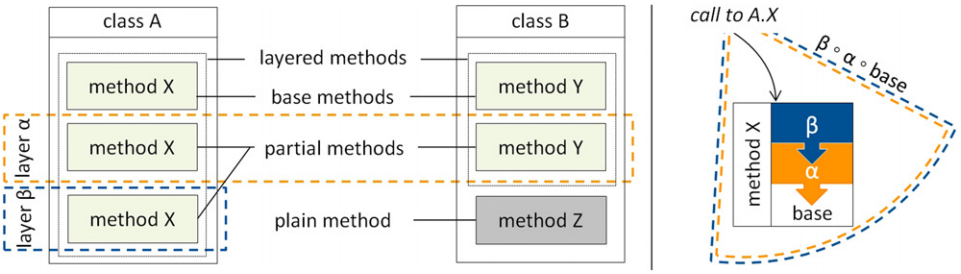


Fig. 1. Denotation of method kinds and sideways composition in COP.

In this paper, we make the following contributions:

- The identification of the need for new dynamic adaptation scopes,
- a generalization of the concept of (control flow specific) layer composition,
- an open implementation for layer activation allowing to customize adaptation rules,
- an implementation of our proposed approach for ContextJS, a COP extension to JavaScript, and
- several examples that serve to illustrate how our approach to domain-related composition rule adaptation eases the expression of adaptations.

The remainder of the paper is structured as follows. Section 2 motivates the extension of state-of-the-art COP layer composition mechanisms by example. Section 3 presents our solution towards a generalization of the composition mechanism and introduces *ContextJS*, a JavaScript-based open implementation for COP. Its usage is illustrated by several examples in Section 4. We discuss related work in Section 5. Finally, Section 6 provides a summary and conclusion.

2. Scoping of layer activation in context-oriented programming

In COP, layers are an encapsulation mechanism orthogonal to object-oriented decomposition. This meets the nature of behavioral variations, whose implementation often affects various parts of a system and cannot be encapsulated by a single object. We present an example for which we employed COP and discuss the benefits of layer-based adaptation. While COP is helpful for control flow driven use cases, existing dynamic scoping mechanisms are not applicable to interactions not centered on control flow. We discuss the inapplicability of existing layer activation mechanisms and the need for alternative specifications in a second example.

2.1. Overview

In the COP execution model, a statement's semantics depends upon the context in which it is evaluated. Behavioral variations, such as variations of method executions, become explicit concepts in COP. Typically, context-specific behavior requires adaptations at several points in a system, constituting its implementation as a crosscutting concern [30]. Behavioral variations are defined within a *layer* allowing for the modularization of functionality that would otherwise be scattered over an object-oriented decomposition.

Depending on the language's features, behavioral variations are implemented by *partial method*, *function*, and/or *class definitions* that encapsulate context-specific functionality; we will focus on partial method definitions. Fig. 1(left) illustrates two classes defining partial methods. To distinguish between plain object-oriented and context-oriented definitions, we introduce the terms *layered method definition* and *plain method definition*. Layered methods consist of a *base method definition* and at least one *partial method definition*. The former is executed when no active layer provides a corresponding partial method. The execution of *plain methods* – methods that have no partial definition – is not affected by layers.

Layers can be activated and composed with others at run time. Therefore, the object-oriented method lookup, which is based on a method's signature, its object, and inheritance rules, is extended with a sideways lookup that considers partial method definitions of active layers. This layer-aware method lookup is also denoted as *sideways* or *layer composition*.

In a layer composition, multiple layers may provide partial definitions of the same method. In that case, a partial method can *proceed* to the next partial definition in the composition, or, if none exists, to the base method definition. When activated, layered method calls are dispatched to the partial method provided by the layer. Partial methods can be executed before, after, or around the base method definition. Fig. 1(right) shows a method dispatch of `A.x` while layers β and α are active. Given that the partial methods proceed with the call, the partial method β is called before α and the base definition.

2.2. Background: programming with dynamically scoped layers

Our first example is an adaptation of a xUnit-like test runner[2] [9]. The test runner as shown in Fig. 2(A) displays only the execution time of whole xUnit test cases, but no fine-grained execution information of individual test methods within

[2] The running example and implementation can be interactively explored at: http://lively-kernel.org/repository/webwerkstatt/demos/contextjs/testrunner.xhtml *(visited 2010-10-28)*. The examples are best viewed with Google Chrome, Apple Safari, or other WebKit-based browsers.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

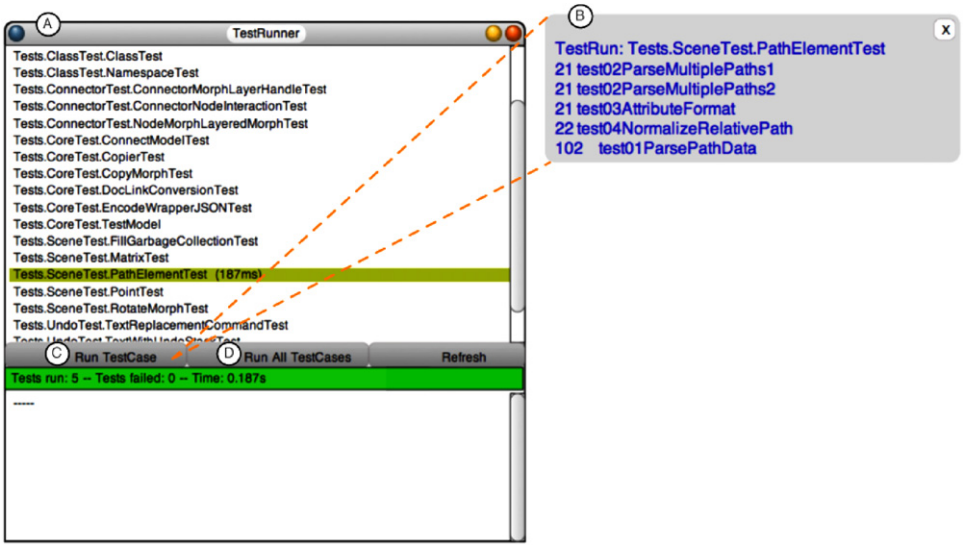


Fig. 2. Test framework adaptation. (A) The test runner shows only the execution time of whole test cases. (B) A user adapted the test runner's behavior by measuring and displaying the execution time of individual test methods when a selected test case is run.

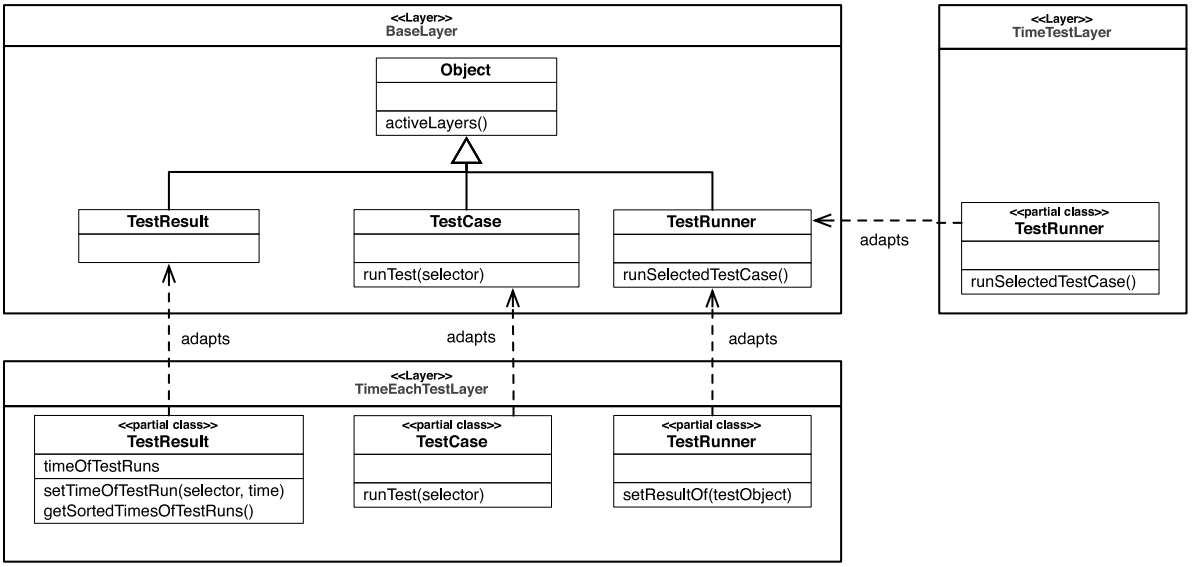


Fig. 3. UML Class diagram showing the adaptation of an xUnit test framework measuring and displaying the execution of each test method.

test cases. A straightforward object-oriented implementation would measure execution time whenever a test method is executed. Therefore the class `TestCase`, which is responsible for the execution of test methods, has to be adapted. Since our system contains a large number of tests, this static solution would decrease execution performance. Instead, measuring and displaying of results should only be active for the execution of an explicitly selected test. Fig. 2 presents a screenshot of a JavaScript-based unit test tool. Whenever the button *Run TestCase* (Fig. 2(C)) of the test runner is pressed, the measurement adaptation should be activated, displaying the result in a separate window (Fig. 2(B)). When the test runner executes all tests at once (Fig. 2(D)) or when test cases are used for other purposes, measuring and displaying should be disabled.

To address this adaptation of the test runner, COP layers can be employed. The core behavior of the test framework is implemented in three classes: `TestRunner`, `TestCase`, and `TestResult`. As shown in Fig. 3, the behavioral adaptation is implemented in two layers: `TimeTestLayer` and `TimeEachTestLayer`. Since time measurement should be implemented as a separate concern, we do not modify the base method definition of `runSelectedTestCase` in the class `TestRunner`, but define a partial method in the layer `TimeTestLayer` as a suitable entry point for the adaptation. To adapt the test runner behavior for any execution of the *Run TestCase* button, we activate `TimeTestLayer` globally.

Measuring execution time of individual methods is the responsibility of the class `TestCase`. Its adaptation is defined in the layer `TimeEachTestLayer`, which is dynamically activated in the `runSelectedTestCase` method. Fig. 4 shows how the `withLayers` statement activates `TimeEachTestLayer` during the execution of an anonymous function, which is used as a

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

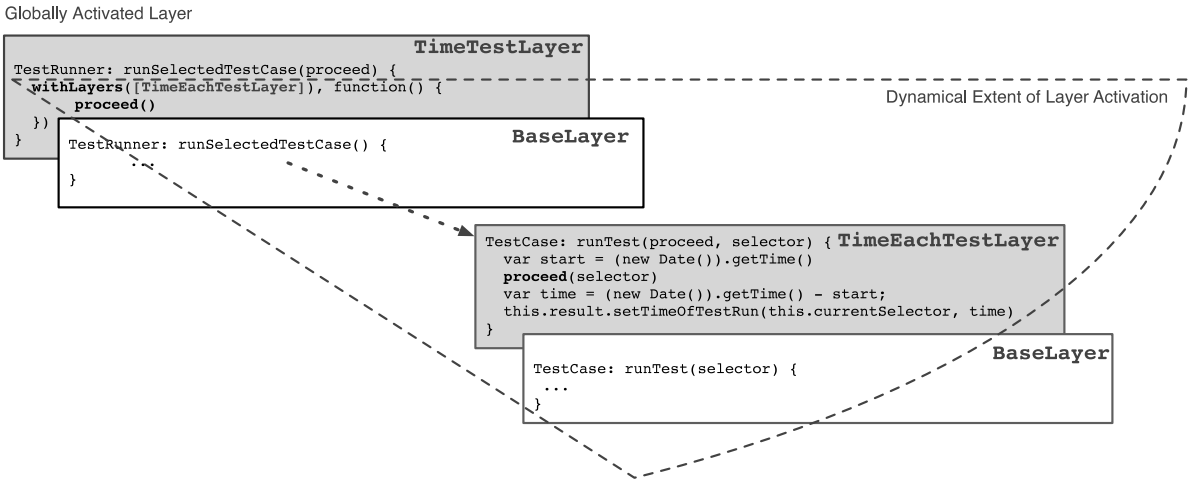


Fig. 4. The refined `runSelectedTestCase` method activates the `TimeEachTestLayer` and thus refines `runTest` of the class `TestCase` for the dynamic extent of this execution.

scoping construct. The `withLayers` statement allows for implicitly passing context information and changing the layered method composition at the time the method `runTest` of the class `TestCase` is executed. The time each test method execution takes is stored in an instance of the class `TestResult`, which was adapted by adding new behavior and new state as shown in Fig. 3.

Our test case adaptation emphasizes an issue of object-oriented adaptation techniques. It requires extensions and refinements of several methods and fields of the abstract class `TestCase`. Using plain object orientation, the adaptations could be either specified within `TestCase` itself or in a new subclass. The former is not desirable with regard to separation of concerns, since context-specific behavior should not be defined within an abstract superclass that handles core concerns. The latter requires changes to the inheritance chains of all concrete test cases, letting them inherit from our new class. However, we cannot assume to have access to the source code of all `TestCase` subclasses; thus, this strategy is fragile. Layers allow for more fine-gained modularization using sideways composition, thereby supporting the definition of method refinements and of new methods and state. Sideways composition extends object-oriented adaptation techniques; it can be used as a delegation technique preserving object identities, and it can be applied where subclassing is not suitable.

2.3. Lack of alternative scoping mechanisms

While layers are useful for defining behavioral variations, the existing dynamically scoped layer activation is insufficient in many situations. To demonstrate this issue, we provide an example of programming graphical objects using a *Morphic* [29,33,28] implementation.

Morphic, known from *Self* [38] and *Squeak/Smalltalk* [20], allows for direct interaction with primitive graphical objects such as rectangles, ellipses, and text fields, that can be composed to more complex objects up to rich user interface applications. All graphical objects are called *morphs*; their classes inherit from the class `Morph`.

We use the example of developing a *connector* to demonstrate the need for new instance-specific and structural scoping of layer activations. One example of a simple graphical object in Morphic is a *line*. Lines can be moved by dragging their handles, which are small rectangles appearing at their ends. These handles – children of their corresponding line in the Morphic scene graph – are instances of the class `HandleMorph`. This parent–child relationship, as depicted in Fig. 5, is an example of how context is constituted by object structure. In the Morphic domain, the scene graph structure is manifested by a bidirectional `submorphs`/`owner` relationship. There are other domains that have similar object structures, e.g., parse trees that may define their object structure in different ways.

Based on Morphic lines, we want to implement connectors. A connector is a line that graphically connects two morphs.[3] Our implementation has to consider two requirements. First, when one of the connected objects moves, the connector should automatically update its position, as shown in Fig. 6. Second, a simple line can be moved by dragging its *handle*, but a connector line should not only be moved but also be reconnected when the handle is dragged onto a new morph, as shown in Fig. 7.

We considered modeling and implementing this scenario by providing special kinds of morphs applying basic object-oriented techniques. We could either have included the new behavior into `Morph`, which would have bloated this already large class. Alternatively, we could have subclassed `Morph` and restricted the potentially connectable graphical objects to instances of this subclass hierarchy.

[3] The running example and implementation can be interactively explored at: http://lively-kernel.org/repository/webwerkstatt/demos/contextjs/connectors.xhtml (visited 2010-10-28).

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

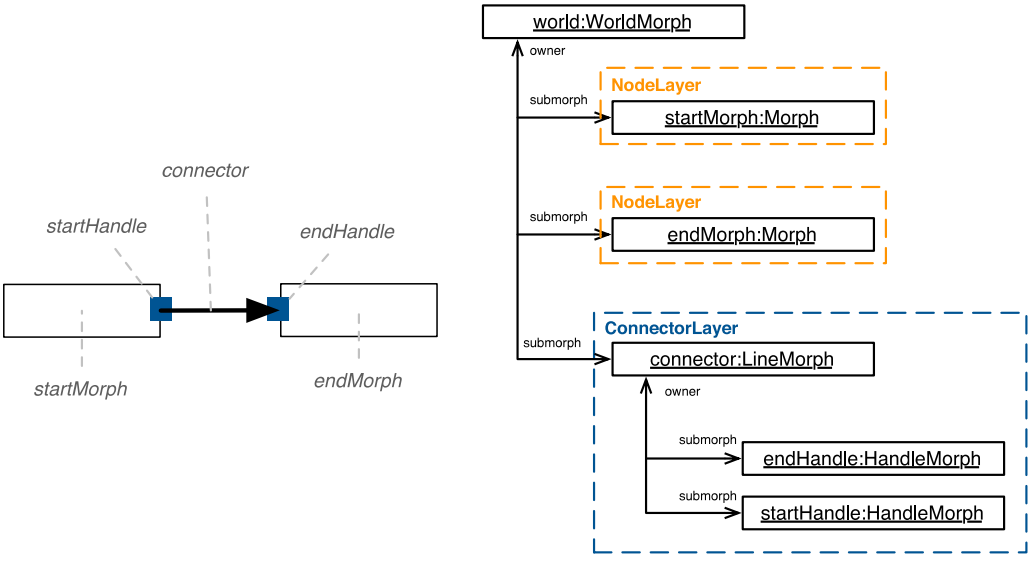


Fig. 5. A Morphic scene graph of the connector example. Both `startMorph` and `endMorph` have instance-specific `NodeLayer` activations. `startHandle` and `endHandle` are in the structural scope of the `ConnectorLayer` which is explicitly activated only in the `connector`.

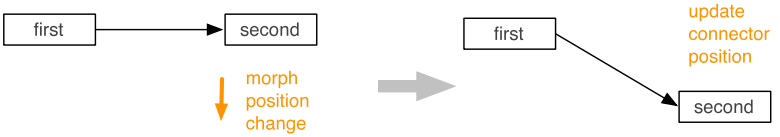


Fig. 6. First requirement in connector example: connectors should update themselves when morphs change their position.

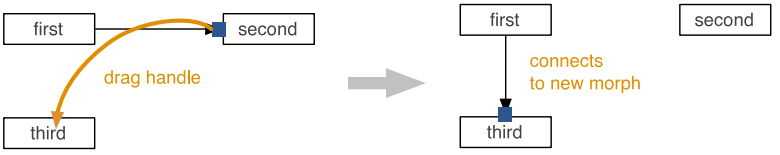


Fig. 7. Second requirement in connector example: dragging a handle (small rectangle) onto a third morph connects the line to that morph.

Instead, we want to use a layer-based sideways composition. Layers allow us to encapsulate our behavioral variation without tangling the `Morph` class declaration and at the same time avoid unwieldy subclassing. We are convinced that layers are a good way to express such class adaptations and to separate them as dedicated concerns.

However, COP's standard scoping mechanisms are not applicable to our scenario. First, morphs need to dynamically adapt node behavior when they are attached to a connector line. Second, handle behavior needs to be adapted when they are part of a connector; i. e., when they are used in the context of a line playing the role of a connector.

With standard COP techniques, the respective layers would be activated upon initiation of the connector's control flow. However, user interface events can also influence the handle's behavior but run in separate control flows that will not touch the connector and its composition statements. This behavioral variation is not control flow centric, but rather depends on specific objects and the structure of the morph object tree (*scene graph*).

We identified the following new layer activation scopes. First, there is a need for layer activations depending on a *specific object*. Second, *structural object hierarchies* should be taken into account. Such structural context information should be used in combination with instance-specific layer activation to extend scoping to object structures. Our desired behavioral adaptation should be defined in `HandleMorph` and activated for handles when they are children of connectors, as shown in Fig. 5.

The new scoping strategies require domain-specific information about structural hierarchies and instance-specific activation algorithms. Hence, we need a flexible implementation of layer activation for application- and domain-specific needs.

3. Open implementation of layer composition

As a result of our analysis, we identify new scoping strategies that must be accessible for application-specific customization. We propose a holistic approach that integrates our new strategies with existing mechanisms. Our solution is based on an open implementation [23,25] in which layer composition strategies are encapsulated into objects. Objects can add other scoping mechanisms or disable layers completely, by overriding the default layer composition behavior.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

This enables developers to implement domain-specific scoping strategies. The definition of a strategy is optional; a default implementation provides the original control flow-specific scoping mechanism.

Moving responsibility for adaptations to objects has some implications. Layer composition can be late-bound, being computed upon layered method execution (contrary to the start of a dynamic extent), which allows for object-specific adaptation within a control flow. However, providing more control can also lead to bad designs. Theoretically, one could specify a different composition strategy for each object in a system, ending up with an unwieldy application design. Nevertheless, we found a set of use cases for which we believe the benefits of such fine-grained and open adaptation strategies to outweigh this drawback. For example, nodes in a tree can provide behavioral variations depending on their position, i.e., structural context, affecting (only) their children; objects can be adapted to play a role independent of their control flow; or specific objects can be closed against any adaptation by providing an empty layer composition. Our experiences with our approach so far suggest that implementations of new scoping strategies require modifications of relatively few classes or objects. Moreover, most applications will not require new adaptation scopes but resort to existing ones. The implementation of new layer activation and composition mechanisms is meta-programming and therefore should be used only where actually needed—specifying scoping strategies should not be part of a standard development process.

3.1. Context-oriented programming with ContextJS

ContextJS, our COP language extension for JavaScript, implements the aforementioned concepts. ContextJS is realized as a library, using only mechanisms provided by JavaScript. It allows for defining behavioral variations of objects as partial methods. In addition, it supports the definition of partial classes for a library-based class system.

ContextJS allows for defining layers that refine methods of objects and classes.[4] ContextJS implements the *class-in-layer* strategy [3], in which partial method definitions are stored inside a layer. Layers are first-class objects and instances of `Layer`. Defining partial methods and classes is realized by calling library functions. The methods `refineObject` and `refineClass` of the class `Layer` take an anonymous object containing these partial methods as an argument.

The following listing presents a simple example for the usage of ContextJS; a more complex one is shown in Section 4.1. The layers `LayerA` and `LayerB` provide partial methods for `m` in class `MyObject`. Layers are created using a library function (Lines 7, 14). Their partial method definitions for `m` (Lines 8–12, 15–19) make use of the `proceed` function to traverse a partial method list at run time. The function `proceed` is prepended to the argument list. When `m` is invoked with the argument `2` without any layer composition, the call is dispatched to the plain method definition that returns `6` (Line 21). The execution of `m` in the dynamic extent of an activation of `LayerA` (Lines 24–25) is first dispatched to `LayerA`'s partial definition of `m`. The `proceed` expression (Line 10) delegates the call to the next partial method – in this case, to `m`'s default definition – and adds `4` to its result. If several layers are activated, for instance `LayerB` within the dynamic extent of `LayerA` (Line 26), the call is first sent to the innermost layer (`LayerB`) and then (using `proceed`) passed to the next one. Besides dynamically scoped layer activation, ContextJS supports global activation using the `enableLayer` function (Line 32). Globally activated layers are active until they are explicitly deactivated using `disableLayer`.

During execution of `refineClass`, the corresponding plain method definition is made layer-aware by replacing it with another function performing layer composition for that method execution and holding a reference to the base method definition. This transformation is done by ContextJS automatically. State (properties) in JavaScript objects can also be layered by defining special JavaScript getter and setter methods in layers.[5]

```
1  Object.subclass("MyObject", {
2    m: function(a) {
3         return a * 3
4    }
5  })
6
7  cop.create('LayerA')
8  LayerA.refineClass(MyObject, {
9    m: function(proceed, a) {
10     return proceed(a) + 4
11   }
12 })
13
14 cop.create('LayerB')
15 LayerB.refineClass(MyObject, {
16   m: function(proceed, a) {
17     return proceed(a * 2)
18   }
19 })
20 var o = new MyObject()
21 o.m(2) // -> 6
22
23 // Dynamically Scoped Layer Activation
24 withLayers([LayerA], function() {
25   o.m(2) // -> 10
26   withLayers([LayerB], function() {
27     o.m(2) // -> 16
28   })
29 })
30
31 // Global Layer Activation
32 enableLayer(LayerB)
33 o.m(2) // -> 12
34
35 withLayers([LayerA], function() {
36     o.m(2) // -> 16
37 })
38
```

[4] Since JavaScript is based on prototypical inheritance between objects, classes are just a convention. The class refinements are therefore only syntactic sugar for refining the class prototype object.

[5] See 11.1.5 Object Initialiser in ECMA-262 [15].

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

3.2. Global and dynamically scoped layer activation

In our library, objects can respond to the message `activeLayers` to compose layers and return a list of active layers. Layer composition is only performed upon method lookup for layered method definitions, so `activeLayers` is only called for a subset of all method invocations. Plain method definitions are not affected. Layering of the `activeLayers` method itself is prohibited, since it would lead to infinite recursion. The sequence diagram in Fig. 8 illustrates a call of `activeLayers` in the invocation process of an adapted `runTest` method.

In the following, we present the implementation of COP's original scoping mechanisms. Our approach covers standard context-oriented scoping mechanisms such as global and dynamically scoped layer activations. The listing below presents an implementation of dynamically scoped layer activation. We use the variable `LayerActivationStack` to keep track of dynamically scoped layer activations.[6] The second variable, `GlobalLayerActivations`, stores the list of globally active layers. The implementation of `activeLayers` uses the `composeLayers` algorithm that recursively composes the layer activation stack and the global activations into an ordered list.

```
1  LayerActivationStack = []
2  GlobalLayerActivations = []
3
4  Object.addMethods({
5    activeLayers: function() {
6      return composeLayers(LayerActivationStack, LayerActivationStack.length - 1)
7    }
8  })
9
10 function composeLayers(activationStack, index) {
11   // 1. Global Layer Activations
12   if (index < 0) {
13     return GlobalLayerActivations
14   }
15   var activation = activationStack[index]
16   var layerComposition = composeLayers(activationStack, index - 1)
17   // 2. Dynamic Layer Activations
18   if (activation.withLayers) {
19     // reject duplicate layer activations
20     layerComposition = layerComposition.reject(function(ea) {
21       return activation.withLayers.include(ea)})
22     return layerComposition.concat(activation.withLayers)
23   }
24   // 3. Dynamic Layer Deactivations
25   if (activation.withoutLayers) {
26     layerComposition = layerComposition.reject(function(ea) {
27       return activation.withoutLayers.include(ea)})
28     return layerComposition
29   }
30 }
```

The layer activation stack is created by the two functions `withLayers` and `withoutLayers` presented in the following listing. The list `layers` is explicitly (de-)activated for the dynamic extent of the closure `func`. Global and dynamically scoped layer activations are available in ContextJS by default for all objects.

```
31 function withLayers(layers, func) {
32   LayerActivationStack.push(
33     {withLayers: layers})
34   try {
35     func()
36   }
37   finally {
38     LayerActivationStack.pop()
39   }
40 }
```

```
41 function withoutLayers(layers, func) {
42   LayerActivationStack.push(
43     {withoutLayers: layers})
44   try {
45     func()
46   }
47   finally {
48     LayerActivationStack.pop()
49   }
50 }
```

The list of globally active layers stored in the global variable `GlobalLayerActivations` can be managed by the two functions `enableLayer` and `disableLayer`. These functions assure that a layer cannot be globally activated twice.

[6] In multi threading environments such as Smalltalk or Java, this variable has to be thread-local, so that different threads will not interfere with each other's layer activations.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

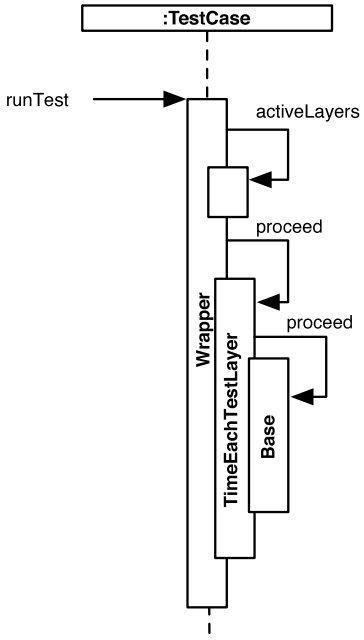


Fig. 8. Sequence diagram for the layered execution of `runTest` in a `TestCase` (from the test runner example).

```
function enableLayer(layer) {
  if (GlobalLayerActivations.include(layer)) {
    return
  }  else {
    GlobalLayerActivations.push(layer)
  }
}
```

```
function disableLayer(layer) {
  if (!GlobalLayerActivations.include(layer)) {
    return;
  }  else {
    GlobalLayerActivations =
      GlobalLayerActivations.reject(
        function(ea) {return ea == layer})
  }
}
```

3.3. Instance-specific and structural layer activation

In Section 2.3, we motivated the need for new scoping mechanisms to leverage COP to new application domains, such as graphical object structures in Morphic. Contrary to the two scoping strategies mentioned above, structural and instance-specific scoping cannot be implemented in a generic way but require domain-specific modifications. To demonstrate the flexibility of our open implementation, we implement these new scoping strategies for our Morphic scenario.

To change layer scoping for all graphical objects, we implement a new composition mechanism in class `Morph`. The implementation of instance-specific layer activation is straightforward: morphs provide a fixed layer composition that is returned when `activeLayers` is called. The list of layers is managed using accessors `(get|set|add|remove)WithLayers`. The following listing shows the implementation of `activeLayers` and two accessor methods for the `Morph` class.

```
Morph.addMethods({
  activeLayers: function () {
    return this.getWithLayers()
  },

  getWithLayers: function() {
    if (! this.withLayers)
      return []
    return this.withLayers
  },
```

```
setWithLayers: function(layers) {
    this.withLayers = layers
  },
  // other accessor methods
  // are omitted here
  ...
})
```

Instance-specific layer activation can be extended to consider structural information of object graphs; in our case, the *submorphs-owner* relationship within a scene graph. The method `structuralLayers` of the next listing recursively walks up the owner hierarchy (as shown in Fig. 5) and collects all instance-specific layer activations. To prevent multiple execution of a partial method, the algorithm also assures that – like in the dynamically scoped layer activation – a layer is included only once in the result (Line 7). Since this implementation of structural layer activation subsumes instance-specific layer activations, we implement `activeLayers` for all graphical objects that allow for using both strategies.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

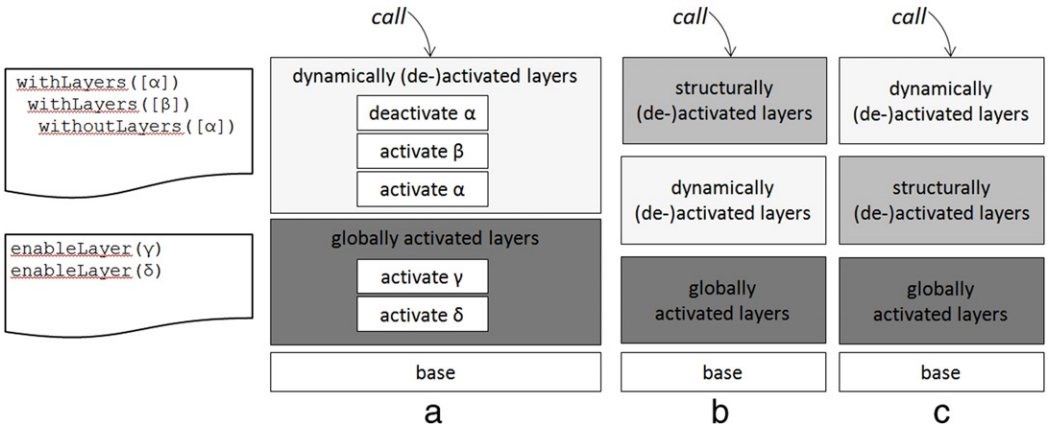


Fig. 9. Order strategies for different layer composition kinds.

```
Morph.addMethods({
  structuralLayers: function () {
    var layers = this.getWithLayers()
    if (this.owner) {
      var ownerLayers = this.owner.structuralLayers()
      // reject duplicate layer activations
      return layers.concat(ownerLayers.reject(function(ea){return layers.include(ea)}))
    }
    return layers
  },

  activeLayers: function () {
    return this.structuralLayers()
  }
})
```

The scoping strategies presented above allow us to implement the connector example as described in Section 4.2. To fully support the default and new scoping mechanisms we have to combine them during layer composition.

3.4. Composition of layer activation strategies

Each layer activation strategy can produce a layer specific composition. When they are used in combination, we need precedence rules. These rules can also be implemented in the `activeLayers` methods, which we will exemplify in the following. The next listing shows a combination of (the default) dynamic extent-based scoping with structural (and therefore also instance-specific) scoping, that is, it includes dynamic and global layer activation mechanisms into one composition.

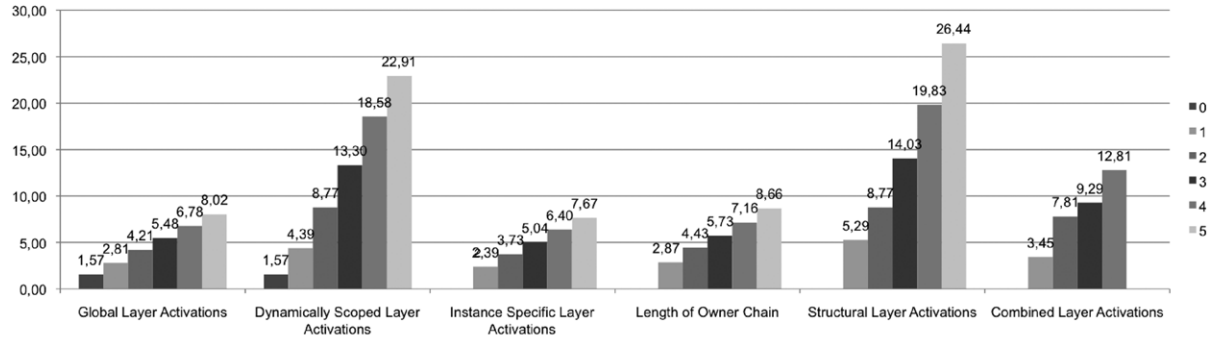
```
Morph.addMethods({
  activeLayers: function () {
    var defaultLayerActivations =
      composeLayers(LayerActivationStack, LayerActivationStack.length - 1)
    var structuralWithoutDefaultLayers = this.structuralLayers().reject(
      function(ea){return defaultLayerActivations.include(ea)})
    return defaultLayerActivations.concat(structuralWithoutDefaultLayers)
  }
})
```

If differently scoped layers do not refine the same method, the execution order does not need any further modifications. Semantic conflicts occur if a method is layered by dynamically and globally activated layers at the same time. In this case, we need precedence rules avoiding ambiguities. The precedence rule applied in our example proceeds from structurally activated layers to dynamically and globally activated layers, as depicted in Fig. 9(B).

The implemented layer composition should provide developers with a coherent metaphor to better understand the layering behavior. Dynamically scoped layer activation follows a stack-like discipline [12]; the stack metaphor as shown in Fig. 9(A) can be stretched to include global layer activations as well:

1. The base layer (every class and method not in a layer) lies at the bottom of the stack,
2. global layer activations come next,
3. dynamically scoped layer (de-)activations are pushed onto the top.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013



	ops	time	ops / ms	overhead
plain method	104,857,600	1,522	68,895	0.03
wrapped method	819,200	1,567	523	3.94
activeLayers	3,276,800	1,589	2,062	1.00

Fig. 10. Micro-benchmark results of various ContextJS `activeLayer` implementations. The chart displays the execution time of the various layer scoping mechanisms relative to the execution time of an empty `activeLayer` method; high numbers are worse.

In this layer composition strategy, dynamically scoped layer deactivations can override global layer activations; i.e., globally activated layers can be deactivated by the dynamically scoped `withoutLayers`. However, depending on the application, other ordering rules may be required as well. Depending on the domain, it may make sense to change the layer composition so that dynamic layer (de-)activations come after the instance-specific and structural layer (de-)activations as shown in Fig. 9(C). Thus, we found no general domain-independent solution for composition ordering. Instead, we allow developers to change the default ordering for their application-specific needs.

By default, classes in ContextJS are open to layered adaptations. ContextL [12], the first COP language, prohibits default adaptation but requires classes to be declared adaptable. ContextJS provides the inverse feature and uses the open implementation to explicitly declare a class as non-adaptable. This encapsulation can be achieved by implementing an empty layer composition. Therefore, a class or object can override the `activeLayers` method to return an empty list, with the consequence that its methods are executed without any adaptations. The decision to move the layer composition into objects hands back control over their method dispatch to them: a context can still change its behavior, but only if the class or object controls its own adaptation.

3.5. Performance observations

Language support for dynamic adaptation requires additional lookups at run time, which affect performance. To measure the actual overhead of layered method definitions, we adopted a set of COP micro-benchmarks [3]. We ran the benchmarks on a MacBook Pro equipped with MacOS X 10.6.4, 4 GB RAM, and 3.06 GHz Intel Core 2 Duo. We used Google Chrome (Dev 5.0.375.99) since it had the fastest JavaScript engine at the time of measurement.[7]

In the benchmark, we activate zero to five layers with the different layer activation mechanisms and measure their execution time. All layers provide a partial definition for a method `m` and proceed to the next layer. The execution of `m` is applied in a loop for two seconds to avoid startup noise within the measurement. Based on this data, we compute the ratio of executions per millisecond. The results are shown in Fig. 10 as a chart of performance overheads of various layer activation mechanisms relative to an empty `activeLayer` method. The absolute results of each benchmark are the layered method executions per time (ops/ms), which are made relative to the method execution of an empty `activeLayers` method which produced approximately 2000 ops/ms. As shown in the table in Fig. 10, a method with empty method body executes approximately 70,000 ops/ms. This relatively high number of operations is a result of aggressive optimization techniques applied to the JavaScript compiler.

In addition to the naïve plain method based implementation, we implemented the benchmark using *method wrappers* provided by the *prototype.js*[8] library. It supports the definition of methods that can be wrapped around any other method, much like partial methods. Since method wrappers make use of meta-programming, which compiler optimizations do not address, a comparison to ContextJS is based on equal premises. The wrapper implementation produces approximately 500 ops/ms and with that is four times slower than an empty `activeLayers` but still faster than most other activation mechanisms.

We can see that there is a considerable performance overhead in methods adapted by ContextJS. The performance of these methods decreased when no layer is activated and further declines with each additional layer activation. The performance of structural layer activation depends not only on the number of active layers as shown in the 5th group of the chart in Fig. 10.

7 Since the micro-benchmark results depend on the performance of JavaScript virtual machines, our results might vary on different Web browsers.

8 http://www.prototypejs.org/ (version 1.6).

It is also affected by the depth of the owner hierarchy (see the 4th group) that has to be traversed even if no layer activations are eventually found. Our benchmarking results show the need for performance improvements in future work. They can be realized by refactoring the layer activation methods to use less expensive expressions. Replacing recursion with iterations, collection functions with `for` loops and introducing caching techniques would probably speed up our implementation.

However, these performance overheads of using ContextJS only affect the performance of refined methods. This differs from other approaches to adaptation such as library-based AOP implementations for JavaScript [37] that wrap every method. These approaches slow down the whole system by a factor of five even when AOP features are not used. In our approach, plain method definitions are executed at full speed, so it depends on the usage of ContextJS how the performance of real programs is affected.

4. Scoping behavior adaptations in the lively kernel

This section shows how our motivating examples – the test runner adaptation and connectors – can be implemented using the various scoping mechanisms provided by our approach. The examples are implemented in the Web-based development environment Lively Kernel[9] [27,26] that provides development tools such as a test runner and class browser.

4.1. Test runner example

In the following, we present a ContextJS-based adaptation of Lively Kernel's test runner. Its base implementation does not measure the execution time of test cases and individual test runs. This execution time should not be logged every time a test case is executed, but only when it is part of a single test run. The execution of each test is the responsibility of the class `TestCase`; without COP, context information would have to be passed (as parameters or in instance variables) to many test cases and their test methods. With ContextJS, the desired behavior can be modeled as dynamically scoped layer activation.

For the implementation, we first separate the time measurement and logging concern from the remaining test runner implementation into a layer. Second, we scope the new layer that should only be active when a user explicitly selects and executes a single test.

The actual behavioral adaptation is defined in layer `TimeEachTestLayer`. As shown in the (extended) class diagram in Fig. 3, the layer refines the classes `TestCase`, `TestResult`, and `TestRunner`,

- adapting existing behavior such as the `runTest` method in class `TestCase`,
- adding new methods such as `getSortedTimesOfTestRuns` in the class `TestResult`, and
- adding new state such as the property `timeOfTestRuns` also in the class `TestResult`.

```
cop.create("TimeEachTestLayer");

// we do not adapt existing behavior in TestResult
// but store our interim results there
TimeEachTestLayer.refineClass(TestResult, {

  setTimeOfTestRun: function(proceed, selector, time) {
    if (!this.timeOfTestRuns)
      this.timeOfTestRuns = {};
    this.timeOfTestRuns[selector] = time;
  },

  getSortedTimesOfTestRuns: function(proceed) {
    var times = this.timeOfTestRuns
    if(!times) return;
    var sortedTimes = Object.keys(times).collect(function(eaSelector) {
      return [times[eaSelector], eaSelector]
    }).sort(function(a, b) {return a[0] - b[0]});
    return sortedTimes.collect(function(ea) {return ea.join("\t")}).join("\n")
  }
});
TimeEachTestLayer.refineClass(TestCase, {
  runTest: function(proceed, selector) {
    var start = (new Date()).getTime();
    proceed(selector);
    var time = (new Date()).getTime() - start;
    this.result.setTimeOfTestRun(this.currentSelector, time)
  },
});
```

[9] http://lively-kernel.org/.

```
33 // after executing all test methods the test runner sets its final result
34 // we use this hook to display our result
35 TimeEachTestLayer.refineClass(TestRunner, {
36   setResultOf: function(proceed, testObject) {
37     proceed(testObject);
38     var msg = "TestRun: " + testObject.constructor.type + "\n" +
39       testObject.result.getSortedTimesOfTestRuns();
40     WorldMorph.current().setStatusMessage(msg, Color.blue, 10);
41   },
42 })
```

Since `TimeTestLayer` refines only the method `runSelectedTestCase` and that adaptation should be active for every execution of `runSelectedTestCase`, we can safely activate this layer globally. The `runSelectedTestCase` adaptation's purpose is to activate the layer `TimeEachTestLayer` in the dynamic extent of the `proceed` statement (Line 8 in the following listing).

```
1 cop.create("TimeTestLayer")
2 enableLayer(TimeTestLayer)
3
4 TimeTestLayer.refineClass(TestRunner, {
5
6   runSelectedTestCase: function(proceed) {
7     cop.withLayers( [TimeEachTestLayer], function() {
8       proceed()
9     })
10  }
11 })
```

The layered method execution of `runTest` is shown in the sequence diagram of Fig. 8:

1. Only those methods that have behavioral adaptations are instrumented, so the execution of most methods in a system is not affected.
2. Before executing the actual method, the object computes the active layers for that message send (`TimeEachTestLayer`, `TimeTestLayer`, `BaseLayer`).
3. Partial methods can be traversed with `proceed` statements.
4. The globally activated `TimeTestLayer` does not define a partial method for `runTest` (see Fig. 3), so it is ignored and the partial method proceeds directly to the base implementation of `runTest`.

The test runner example demonstrates the usage of two default layer activation strategies: global activation and dynamically scoped activation. The following example shows how the new scoping mechanisms can be used.

4.2. Connector example

In Section 2.3, we have motivated the need for a new instance-specific layer scoping mechanism by developing a connector line for two graphical objects (morphs) that is updated when one of the morphs moves (as shown in Fig. 6). We have shown an implementation of such scoping mechanisms in Section 3.3. To demonstrate the usage of instance-specific scoping mechanisms for modeling node and connector roles with layers, we implement the example in Lively Kernel.

Instance-specific layer activation. First, we define `NodeLayer`, which adapts the method `change` of class `Morph` (see Fig. 11). Each node knows its `connectors` and updates them when moved. The connector role is also modeled as a layer that adds a new `updateConnection` method used by the nodes.

```
1 NodeLayer.refineClass(Morph, {
2   changed: function(proceed) {
3     proceed()
4     this.updateConnectors()
5   },
6   updateConnectors: function() {
7     this.connectors.each(function(ea) {
8       ea.updateConnection()
9     })
10  },
11  ...
12 })
```

```
13 ConnectorMorphLayer.refineClass(LineMorph, {
14   updateConnection: function () {
15     // ... algorithm that computes
16     // new start and end position
17   },
18   ...
19 })
20
21
22
23
24
```

The actual instance-specific layer activation, which lets a `LineMorph` dynamically play the role of a connector, is activated by using the `setWithLayers` method, which associates a list of layers with an instance. The same construct is used to let other existing instances of `Morph`, such as rectangles, ellipses, or text fields, play the orthogonal role of a node. This instance-specific layer activation is used in the method `connectToMorph` (Line 33) when the connector is linked to a new morph that takes the role of a node.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

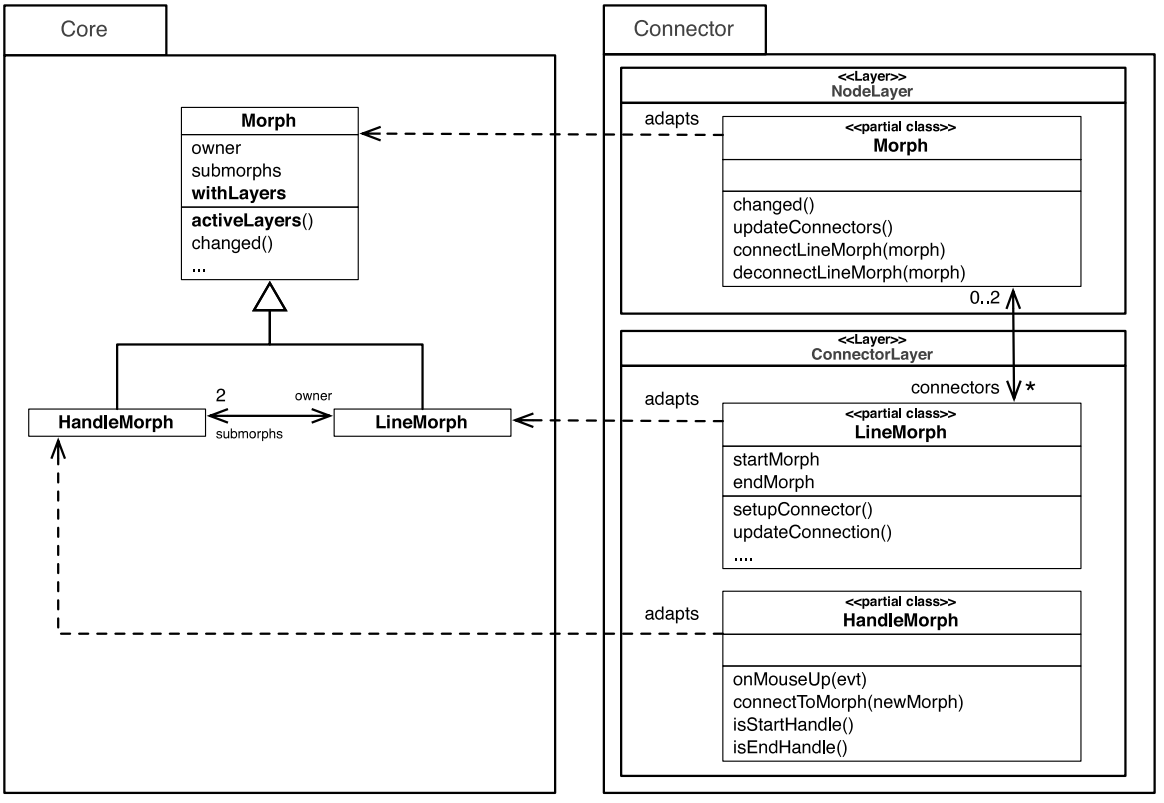


Fig. 11. Adaptation of classes of the Lively Kernel core system in the connector module.

```
25 var connector = Morph.makeLine([pt(0,0), pt(100,0)], 1, Color.black);
26 connector.setWithLayers([ConnectorLayer]);
27 connector.setupConnector();
```

Structural layer activation. The second problem we discussed in Section 2.3 is the behavioral variation that should be active when handles are part of a connector: when dragged onto a new morph, handles should reconnect the connector to that morph (as shown in Fig. 7). The adaptation of the class `HandleMorph` consists of an adaptation of the `onMouseUp` event handler and the addition of helper methods:

```
28 ConnectorLayer.refineClass(HandleMorph, {
29   onMouseUp: function(proceed, evt) {
30     this.connectToMorph(this.findMorphUnderMe())
31     return proceed(evt)
32   },
33   connectToMorph: function(proceed, newMorph) {
34     var connector = this.owner
35     if (newMorph) {
36       newMorph.setWithLayers([NodeMorphLayer])
37       newMorph.connectLineMorph(this.owner)
38     }
39     // ... update startMorph and endMorph
40   },
41   ...
42 })
```

Since handles are a part of the graphical structure of the `LineMorph` connector, as shown in Fig. 5, we can make use of the new structural scoping mechanisms defined in Section 3.3. The behavioral variation of the class `HandleMorph` can be defined in the layer `ConnectorLayer`. The handle's `owner` is a `LineMorph` object; thus, the handle is in the connector's structural scope.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

5. Related work

5.1. Context-oriented programming languages

Original scoping mechanism. Most COP language implementations provide control flow-specific scoping as introduced in Section 2.2. *ContextL* [12,13], based on Lisp and the Common Lisp Object System (CLOS), was the first COP extension to a programming language. Layers can be defined for classes, functions and methods. Classes must specify if they are adaptable by layers (by inheriting from a specific COP class). Thus, classes have to be actively opened for layering, whereas ContextJS classes are open by default but can protect themselves. ContextJS allows objects to protect themselves against any layer refinement by overriding `activeLayers` returning an empty layer composition.

Subsequently, several meta-level libraries for dynamic programming languages were developed, namely *ContextS* [17] for Smalltalk, *ContextR* [32] for Ruby, *ContextPy* [19] for Python, and *ContextG* for Groovy. For the statically typed language Java, some COP prototypes [18,6,2] and the compiler based extension *ContextJ* [4] have been developed. A minimal subset of ContextJ, *cj*, is implemented for the *delMDSOC* kernel [31].

Implicit layer activation. The Python language extension *PyContext* [39], supports a variant of *implicit layer activation* where layers can determine if they are active. Layers can provide a method evaluating an activation condition before layered method invocations. This approach allows for implementing activation mechanisms for specific layers, but it cannot change the whole layer composition. With our approach such implicit layer activations could be achieved by implementing an object-specific layer composition that delegates the layer activation to registered layers.

Event-specific scoping. In some application domains, such as adaptive user interfaces, behavioral variations are related to events rather than to control flows. Events can occur at various points in an execution that may not be obvious in the source code. The Java-based languages *JCop* [5] and *EventCJ* [22] address this problem by providing declarative composition statements adopted from aspect-oriented programming [24]. These declarations describe join points at which certain layers should be composed. In our open implementation, we do not introduce quantification, thus we cannot express declarative layer activation.

5.2. Other scoping and lookup strategies

The *scoping strategies* approach [36] identifies static and dynamic scopes as two scoping dimensions. With scoping strategies, variables are not either statically or dynamically scoped; instead, developers can parameterize variable bindings. A scoping strategy is specified with functions implementing the propagation and activation of a binding. While this approach opens the scoping implementation of variable bindings, ContextJS opens the scoping implementation of layer activation.

The *Ambient Object System* [16] (AmOS) supports context-orientation. AmOS is a prototype-based object system built on top of Common Lisp that supports behavioral adaptations with partial method definitions and context objects, which correspond to COP layers. At any method call in AmOS, receiver methods are first looked up in the current activation and then in further enclosing lexical scopes. If no appropriate method is found in the lexical scope, the lookup continues in a graph of context objects delegating to each other. The delegation chain between these context objects can be modified dynamically, achieving context-specific behavior.

Feature-oriented programming (FOP) [8] addresses the process of step-wise refinement for product-line development. The Java-based AHEAD Tool Suite [7] is an implementation of FOP. As programming language, it supports *Jakarta*, which extends Java with constructs such as class refinements for static feature-oriented composition. Layers in Jakarta are distinct files describing static class refinements. The core ideas of FOP and COP are similar: Both introduce new or alternative program behavior through features or layers, respectively. However, FOP applies compile-time composition of feature variations in contrast to run-time composition as provided by COP.

Aspect-oriented programming (AOP) [24] aims to tame crosscutting concerns by introducing pointcut-based quantification. The main distinction between AOP and COP is that the former allows for a joint specification of *when* in the execution flow *what* kind of functionality should be used, while COP separates *when* (using `with` statements) from *what* (using layers and partial methods). For a comparison of AOP and COP as appropriate representation of behavioral variations, we distinguish between *homogeneous* and *heterogeneous* crosscutting concerns [1]. Homogeneous crosscuts execute the same functionality at multiple locations in a control flow, for which AOP provides well suited abstractions. However, we experienced that most behavioral variations are heterogeneous crosscuts, which introduce different functionality at join points. AOP implementations of heterogeneous crosscuts tend to be less understandable to developers than layer-based implementations [1], since they have to mimic COP behavior using pointcuts with dynamic conditions and advice that are complex and fragile for changes.

Perspectives in the *subjective programming* language *Us* [35] are a way to describe the scope of behavioral variations. Us changes message passing of *Self* [38] to incorporate perspectives. Perspectives define a fixed layer composition by statically connecting a layer with its parent layer. This implies that layers in subjective programming – unlike COP – cannot be part of different compositions at the same time.

Classboxes [10] support static scoping of behavioral adaptations. A classbox is an explicitly named scope in which classes and their members can be defined. Besides common subclassing, Classboxes supports local refinement of imported classes by adding or modifying their features without affecting the originating classbox, much like layers and partial methods. The

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

approach is implemented by analyzing the control flow and modifying the message dispatch accordingly. Since scoping is only controlled by the import and use of objects of a classbox module, structural and instance-specific scopes cannot be expressed by Classboxes.

Modularization approaches such as *traits* [34,14] and *mixins* [11] allow for an additional inheritance relationship next to the class hierarchy, but do not offer dynamic adaptation like layers.

6. Conclusion and future work

Behavioral variations often are concerns that cannot be adequately represented using plain object-oriented means. Their crosscutting and dynamic nature demand alternative encapsulation and scoping mechanisms. Context-oriented programming meets these requirements by providing layers as an encapsulation mechanism orthogonal to objects, and a control flow-specific scoping strategy.

In this paper, we motivated the need for additional scoping strategies – instance-specific and structural scoping – and propose an open implementation for COP layer composition. We presented ContextJS, our COP extension to JavaScript that offers an open implementation allowing developers to define domain-specific scoping strategies. We applied our approach to several examples in Lively Kernel, a collaborative Web-based programming environment.

With our open implementation, we allow for the manipulation of message dispatch via layer composition. As message dispatch is a core feature of object-oriented programming, it should only be cautiously manipulated. The implementation of new layer activation and composition mechanisms is a meta-programming task done by framework developers rather than by application developers. Thanks to its dynamic execution environment and its flexible implementation, ContextJS is a good test environment for rapid COP language prototyping.

In future work, we will analyze the applicability of other scoping mechanisms described in the literature, such as implicit layer activation [39] or Classboxes [10]. In addition, we will apply the results of our experiments with ContextJS to other COP languages, such as the Java-based JCop language [5]. JCop supports declarative event-based layer composition that evaluates an event condition upon layered method execution. Instead of evaluating the event condition, we can open the layer lookup algorithm and provide alternative composition strategies. With that, we will investigate whether the flexibility of our ContextJS approach can be transferred to a statically typed language.

Acknowledgements

We thank Felix Geller and Michael Haupt for their comments on drafts of this paper.

References

[1] Sven Apel, Thomas Leich, Gunter Saake, Aspectual feature modules, IEEE Transactions on Software Engineering 34 (2) (2008) 162–180.
[2] Malte Appeltauer, Robert Hirschfeld, Explicit language and infrastructure support for context-aware services, in: Beiträge der 38. Jahrestagung der Gesellschaft für Informatik, Lecture Notes in Informatics, vol. INFORMATIK 2008 — Beherrschbare Systeme dank Informatik, München, Germany, September 2008. Gesellschaft für Informatik, pp. 164–170.
[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, Michael Perscheid, A comparison of context-oriented programming languages, in: Proceedings of the Workshop on Context-oriented Programming, COP, Co-located with ECOOP 2009, Genoa, Italy, ACM DL, 2009.
[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Hidehiko Masuhara, ContextJ—Context-oriented programming for Java, Computer Software of The Japan Society for Software Science and Technology (2010).
[5] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, Kazunori Kawauchi, Event-based software composition in context-oriented programming, in: Proceedings of International Conference on Software Composition 2010, in: Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, Germany, 2010.
[6] Malte Appeltauer, Robert Hirschfeld, Tobias Rho, Dedicated programming support for context-aware ubiquitous applications, in: UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, IEEE Computer Society Press, Washington, DC, USA, 2008, pp. 38–43.
[7] Don Batory, Feature-oriented programming and the AHEAD tool suite, in: ICSE'04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2004, pp. 702–703.
[8] Don Batory, Jacob Neal Sarvela, Axel Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (6) (2003) 355–371.
[9] Kent Beck, Test-driven Development: By Example, Addison-Wesley Professional, 2003.
[10] Alexandre Bergel, Stephane Ducasse, Roel Wuyts, Classboxes: a minimal module model supporting local rebinding, in: Lecture Notes in Computer Science, 2003, pp. 122–131.
[11] Gilad Bracha, William Cook, Mixin-based inheritance, in: OOPSLA'90: Proceedings of the European Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 1990, pp. 303–311.
[12] Pascal Costanza, Robert Hirschfeld, Language constructs for context-oriented programming: an overview of ContextL, in: DLS'05: Proceedings of the 2005 Symposium on Dynamic Languages, ACM, New York, NY, USA, 2005, pp. 1–10.
[13] Pascal Costanza, Robert Hirschfeld, Wolfgang De Meuter, Efficient layer activation for switching context-dependent behavior, in: David E. Lightfoot, Clemens A. Szyperski (Eds.), Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, in: Lecture Notes in Computer Science, vol. 4228, Springer-Verlag, Berlin, Heidelberg, Germany, 2006, pp. 84–103.
[14] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew P. Black, Traits: a mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems 28 (2) (2006) 331–388.
[15] ECMA, Standard ECMA-262 ECMAScript Language Specification, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, 2009. 5th edition, December 2009.
[16] Sebastián González, Kim Mens, Alfredo Cádiz, Context-oriented programming with the ambient object system, Journal of Universal Computer Science 14 (20) (2008) 3307–3332.
[17] Robert Hirschfeld, Pascal Costanza, Michael Haupt, An introduction to context-oriented programming with ContextS, in: Generative and Transformational Techniques in Software Engineering (GTTSE) II, in: LNCS, vol. 5235, Springer, 2008, pp. 396–407.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013

[18] Robert Hirschfeld, Pascal Costanza, Oscar Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151.
[19] Robert Hirschfeld, Michael Perscheid, Christian Schubert, Malte Appeltauer, Dynamic contract layers, in: 25th Symposium on Applied Computing, Lausanne, Switzerland, ACM DL, New York, NY, USA, 2010.
[20] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay, Back to the future: the story of squeak, a practical smalltalk written in itself, ACM SIGPLAN Notices 32 (10) (1997) 318–326.
[21] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, Tommi Mikkonen, The lively kernel a self-supporting system on a web page, in: Robert Hirschfeld, Kim Rose (Eds.), S3 2008, in: LNCS, vol. 5146, Springer-Verlag, Berlin, Heidelberg, 2008.
[22] Tetuso Kamina, Tomoyuki Aotani, Hidehiko Masuhara, Designing event-based context transition in context-oriented programming, in: COP'10: Second International Workshop on Context-Oriented Programming, ACM Press, New York, NY, USA, 2010, pp. 1–6.
[23] Gregor Kiczales, Beyond the black box: open implementation, IEEE Software 13 (1) (1996) 8–11.
[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin, Aspect-oriented programming, in: Ecoop 1997, Proceedings 11th European Conference on Object-Oriented Programming, vol. 1241, Springer-Verlag, 1997, pp. 220–242.
[25] Gregor Kiczales, Andreas Paepcke, Open implementations and metaobject protocols, Technical Report, Xerox PARC, 1995.
[26] Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, Krzysztof Palacz, Lively wiki a development environment for creating and sharing active web content, in: WikiSym'09, ACM, 2009.
[27] Jens Lincke, Robert Krahn, Dan Ingalls, Robert Hirschfeld, Lively fabrik—a web-based end-user programming environment, in: Proceedings of the Conference on Creating, Connecting and Collaborating through Computing, C5, 2009, IEEE, Tokyo, Japan, 2009.
[28] John Maloney, An introduction to morphic: the squeak user interface framework, in: Squeak: Open Personal Computing and Multimedia, 2001.
[29] John H. Maloney, Randall B. Smith, Directness and liveness in the morphic user interface construction environment, in: UIST'95: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, ACM, New York, NY, USA, 1995, pp. 21–28.
[30] Harold Ossher, Peri Tarr, Multi-dimensional separation of concerns and the hyperspace approach, in: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000, pp. 293–323.
[31] Hans Schippers, Michael Haupt, Robert Hirschfeld, Dirk Janssens, An implementation substrate for languages composing modularized crosscutting concerns, in: Proc. SAC PSC, ACM Press, 2009.
[32] Gregor Schmidt, ContextR & ContextWiki, Master's Thesis, Hasso-Plattner-Institut, Potsdam, 2008.
[33] Randall B. Smith, John Maloney, David Ungar, The Self-4.0 User Interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility, in: OOPSLA'95: Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, 1995, pp. 47–60.
[34] Randall B. Smith, David Ungar, Programming as an experience: the inspiration for self, in: COOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995, Proceedings, in: Lecture Notes in Computer Science, vol. 952, Springer, 1995, pp. 303–330.
[35] Randall .B. Smith, David Ungar, A simple and unifying approach to subjective objects, Theory and Practice of Object Systems 2 (3) (1996) 161–178.
[36] Éric Tanter, Beyond static and dynamic scope, in: DLS'09: Proceedings of the 5th Symposium on Dynamic Languages, ACM, New York, NY, USA, 2009, pp. 3–14.
[37] Rodolfo Toledo, Paul Leger, Éric Tanter, AspectScript: expressive aspects for the web, Technical Report TR/DCC-2009-10, University of Chile, October 2009.
[38] David Ungar, Randall B. Smith, Self: the power of simplicity, Lisp and Symbolic Computation 4 (3) (1991) 187–205.
[39] Martin von Löwis, Marcus Denker, Oscar Nierstrasz, Context-oriented programming: beyond layers, in: ICDL'07: Proceedings of the 2007 International Conference on Dynamic Languages, ACM, New York, NY, USA, 2007, pp. 143–156.

Please cite this article in press as: J. Lincke, et al., An open implementation for context-oriented layer composition in ContextJS, Science of Computer Programming (2011), doi:10.1016/j.scico.2010.11.013