Implicit Layer Composition on UI-Components in a spatial Context

Sven Mischkewitz

Hasso-Plattner-Institute, University of Potsdam, Germany
sven.mischkewitz@student.hpi.de

Abstract Cross-cutting concerns are to be considered in constantly evolving software projects. The programming paradigm context-oriented programming presents language constructs like layers and partial methods to manage cross-cuts. The management of layer composition can be a non-trivial task too. This paper introduces an approach of explicit layer definitions with implicit layer activation. Therefore the position of stateful web UI-components in the DOM-tree is encoded as the component's context. Layers are directly attached to components. Layer scope is propagated through the DOM-hierarchy so that nested components can apply behavioral variations accordingly. The solution[1] is based on ContextJS and React.

Keywords: Cross-cutting concerns, context-oriented programming, implicit layer composition, ContextJS, React Javascript library, Javascript

1 Introduction

Cross-cutting concerns are concerns which affect several modules across the software system. Recent years of research presented programming paradigms which try to tackle cross-cutting concerns, e.g. aspect-oriented programming [6] or context-oriented programming [3]. Context-oriented programming (COP) enables dynamic program behavior variations during runtime. With focus on context during execution behavioral variations are applied using Layers. Layers can be activated or deactivated within the scope of certain code blocks. When a layer is active, behavioral variations get applied to previously *layered methods*. Appeltauer et al. states that "COP assumes context to be everything that is computationally accessible, such as object state, network bandwidth, or user interaction" [2]. To break down this vast definition, this paper focuses on context that is described by visual relations of visual components in web applications. Therefore we combine the research results about COP in Javascript with a modern approach of creating UI-elements in Javascript. Namely we focus on integrating ContextJS [7] into the *React Javascript library*[2] by Facebook[3]. First

[1] https://github.com/mischkew/CopReact
[2] https://facebook.github.io/react/index.html
[3] https://facebook.com

we explain the expected outcome of this integration in chapter 1.1, followed by an introduction to both implementations in chapter 2.1 and chapter 2.2. A current solution is presented in section 2.3. Problems within this solution are discussed in chapter 3.

1.1 Clock Layers provide spatial Context

As context requires an application to behave differently during runtime, we want to adapt to the change of context without scattered or tangled code statements. Context-oriented programming provides several ways to handle cross-cutting concerns. These cross-cutting concerns appear in different manners. We want to address concerns resulting from the spatial position of a context-dependent object in its visual representation. To make this more clear, it is useful to have a look at the Clock Layers[4] built with ContextJS by Lincke et al.

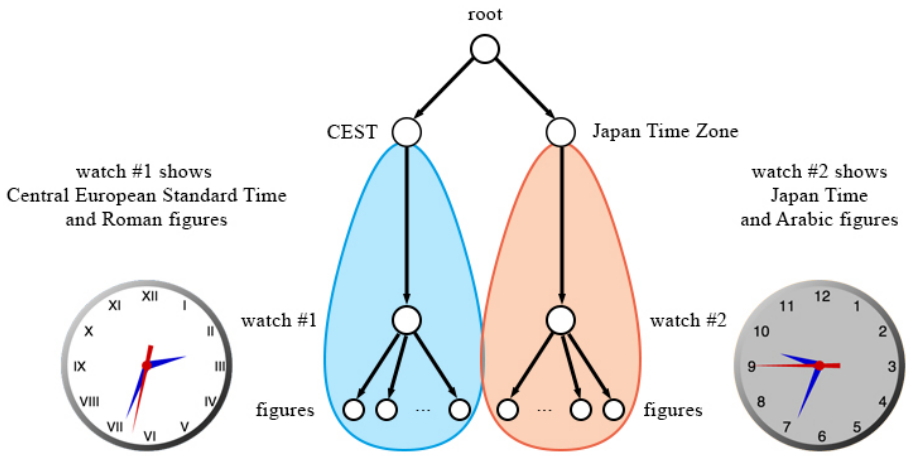

Figure 1. General Scoping explained with Clock Layers. The graph shows the DOM structure of two watches in time zone regions.

In Clock Layers different regions on the screen represent different time zones, which affect the clock when it is places inside a particular region. When a watch is placed in the Japanese Time Zone, the shown time is adapted to the time zone. Additionally the figures are outlined in Arabic style when in scope of the Japanese Time Zone context. Interestingly both watches are identical, but behave different when placed underneath a different time zone node. Speaking in the manner of web development the spatial position of an interactive component is its location in the Document Object Model (DOM). Figure 1 shows how the scope of context evolves through the tree. The CEST node and the Japan Time

[4] http://lively-kernel.org/repository/webwerkstatt2011/demos/contextjs/ClockLayers.xhtml

Zone node of the tree determine the context of all nodes below. The contexts of both time zones are disjoint because the nodes are siblings. The context of a node may only be affected by its ancestors. This means a user can interactively drag and drop a watch from one zone to the other and the outline of the watch will adapt to the new context dynamically. The idea is to introduce a generalized method where layered behavior is directly annotated to a visual component in the DOM-tree. Implicit layer activation will take place whenever a component changes its position in the DOM-tree. Layer activation will automatically scope onto all child elements of the respective node. We believe the restriction of layer scope to subtrees of the DOM adds to a clear understanding of context.

2 Applying context-oriented Programming to the React Javascript Library

2.1 Introduction to ContextJS

The Clock Layers example is hosted inside Lively Kernel [5], a browser-based runtime and development environment. Lively Kernel runs ContextJS, a COP implementation in Javascript. ContextJS provides known concepts of COP like layered methods, global layers and scoped layer activation. ContextJS adds layered methods directly into the object's prototype using the class-in-layer strategy [1]. Thus layers are treated as first-class objects. A basic usage example is seen in listing 1.1.

Listing 1.1. Layering a plain object in ContextJS

```
1  // define a plain object
2  let myObject = {
3    getValue() {
4      return 3
5    }
6  }
7
8  // create a layer
9  let layer = new Layer('MyLayer')
10
11 // add layered behavior to a plain object
12 layer.refineObject(myObject, {
13   getValue() {
14     // similar proceed method known from AspectJ
15     return cop.proceed() + 2
16   }
17 })
18
19 myObject.method() // returns 3
20
21 layer.beGlobal()
```

[5] https://github.com/LivelyKernel/LivelyKernel

```
22 myObject.method() // returns 5
23 layer.beNotGlobal()
```

2.2 Concepts of React Javascript Library

React provides a powerful tool to build user interfaces for web and mobile applications[4]. As React is currently under active development the growing user base emphasizes its practical relevance[6].

React provides a virtual DOM consisting of lightweight representations of DOM nodes, called *Components* [4]. A Component is stateful and provides behavior, e.g. manages click interaction by a user. With focus on flexibility React allows composition of Components. Multiple Components may be nested into a parent Component. The parent Component passes state onto its children. Children can notify their parents by emitting events or executing callbacks. In short, in React data is passed from parent to children, while the event flow is directed from children to parent [5]. To enhance code readibility, React introduces the Javascript Syntax Extensions (JSX)[7]. This code transpilation allows the usage of HTML-tag syntax to instantiate compositions of Components.

The real DOM is re-rendered as soon as React detects changes in the virtual representation of the DOM. This is done with reactive updates. This means the *render* method of a Component is invoked again, when the Component's data changes. This causes React to "throw away the entire UI and re-render it from scratch"[8] [4]. We think that the model of managing context by applying layered behavior to an object fits the way how state is communicated between React Components. A layer scopes a certain control flow. React Components pass data - the "context" - in a unidirectional flow. Also React presents *Mixins* to handle cross-cutting concerns[9]. Mixins provide reusable behavior for a Component but only can be assigned statically. In detail it is a set of functions and properties which is injected into the Component's prototype. Listing 1.2 shows the general usage of Components and Mixins in React.

Listing 1.2. Creating a basic Component with Mixin functionality in React

```
1 // a mixin is a plain object
2 let MyMixin = {
3   generateText() {
4     return 'I am a Component'
5   }
6 }
7
8 const MyComponent = React.createClass({
```

[6] http://facebook.github.io/react/blog/2015/03/30/community-roundup-26.html
[7] https://facebook.github.io/jsx/
[8] http://www.quora.com/How-is-Facebooks-React-JavaScript-library/answer/Lee-Byron?srid=3DcX
[9] http://facebook.github.io/react/docs/reusable-components.html#mixins

```
  // include reusable functionality
  mixins: [ MyMixin ],

  // this method produces DOM fragments
  render() {
    // access the text generator method from the mixin
    return <div>{this.generateText()}</div>
  }
})

// render an instance of the component into the DOM
React.render(<MyComponent />, document.body)
```

2.3 Integrating ContextJS into React

We want to combine COP Layers with React to handle dynamic, implicit activation of context specific behavior. To fully integrate COP the React-way, it is necessary to enable layered behavior on composable Components. Because Components are factories for concrete React Elements, we cannot apply the object refinement of ContextJS directly. Therefore the usage of Mixins proved sufficient. To fuse React and ContextJS we apply layered behavior onto Mixins and then add the Mixin to a Component, listing 1.3 shows the approach.

Listing 1.3. Layering a React Component

```
// the same mixin and component as before
let MyMixin = {
  generateText() {
    return 'I am a Component'
  }
}

const MyComponent = React.createClass({
  mixins: [ MyMixin ],

  render() {
    return <div>{this.generateText()}</div>
  }
})

// add layered behavior to the mixin
let myLayer = new Layer('myLayer')
myLayer.refineObject(MyMixin, {
  generateText() {
    return 'I am a layered Component'
  }
})

// activate the layer
```

```
25 myLayer.beGlobal()
26 React.render(<MyComponent />, document.body)
```

To provide sufficient usability of layers in React, we introduce a structure of Mixins (seen in figure 2) which takes care of layer creation and composition. A *ContextControl* Mixin is responsible for attaching a named layer to a Component. Any child Component that shall be aware of layered content must include a *Context* Mixin. The Context Mixin accesses the marked-active layers and actually applies them with scope respectively to its Component. To enhance a Component with layered behavior the Context Mixin must be extended. Then the Custom Mixin is injected into the Component. A ContextControl and a Context Mixin can be added to a Component at the same time, so new layers can be introduced in a subtree of the virtual DOM. A *Layers* class is used to provide uniform access to layers of ContextJS.

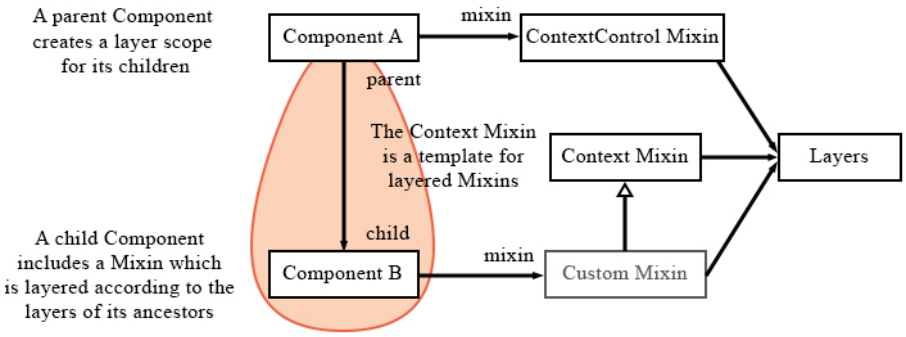

Figure 2. Layer scope passed to child Component in React

Adding Layer Information with ContextControl As mentioned before, in a nutshell the ContextControl Mixin attaches a layer to a React Component. The Mixin stores the name of the layer and its active status in the Component's state. When a layer is marked as *active*, it is pushed into an *activeLayers* array. Initially the activeLayers array is empty or contains layers which were attached to ancestor Components. By using React's parent-based context chains[10] layers can be passed to child Components, when the parent Component provides a context object that contains the activeLayers array. Due to context types [11] only those children which include a Context Mixin receive the layer information. Listing 1.4 shows a use case of ContextControl.

[10] https://blog.jscrambler.com/react-js-communication-between-components-with-contexts

[11] https://blog.jscrambler.com/react-js-communication-between-components-with-contexts

Listing 1.4. Code Example: A parent Component using ContextControl

```
import React from 'react'
import ContextControl from 'lib/ContextControl'

const ParentComponent = React.createClass({
  mixins: [ ContextControl ],

  // a React Lifecycle method
  // when the Component is added to the DOM
  // attach the layer
  componentDidMount() {
    this.setupLayerOnMount()
  },

  render() {
    return (
      <div>
        I am a Parent Component.
        {React.cloneElement(this.props.children)}
      </div>
    )
  }
})

// pass a layer and inital layer state on creation
React.render(
  <ParentComponent layer="MyLayer">
      <ChildComponent />
  </ParentComponent>, document.body)
```

As an experienced ContextJS user may know, there is the concept of *structural layers* [7]. Structural layers apply a layer to all referenced objects within the current layer scope. Because React chooses a top-down unidirectional data flow, structural layers cannot be applied to Components. In chapter 3.1 we give further explanations on the problem with structural layers. So we have to simulate structural scope with the parent-based context chains. Thus each child Component with a Context Mixin applies the layer to itself. The outcome is similar to structural scoping but Components have to explicitly announce that layering is required (see figure 3).

Activating a Layer in Child Scope with the Context Mixin A Component which includes a Context Mixin is aware of layered functionality. The Context Mixin simulates the structural scoping of the ancestors by activating layers locally. This is possible because the Context Mixin extends the *LayerableObjectTrait* object provided by ContextJS. A LayerableObjectTrait enables an object to directly apply layer scopes onto itself [7]. By overwriting the method *activeLayers* with the array of active layers from the parent-based context chain in the

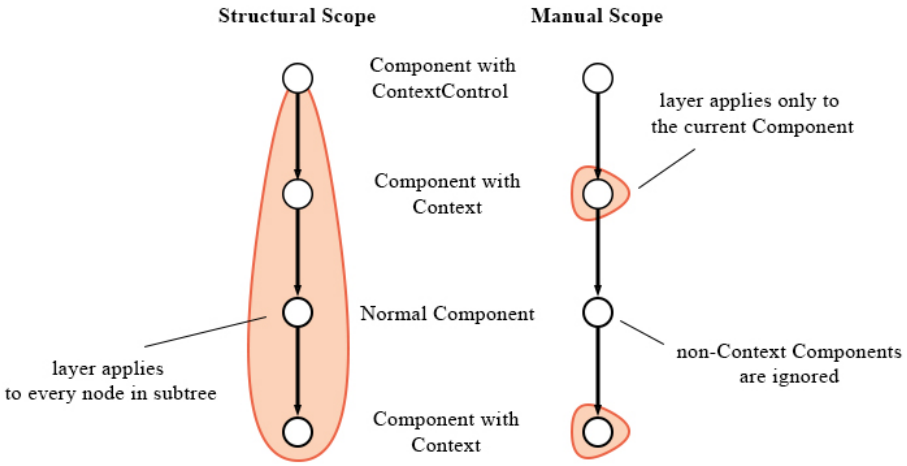

Figure 3. Structural scope in comparison to manual scope

Context Mixin, all accumulated layers from the ancestors are applied on the local Component (see listing 1.5).

Listing 1.5. Applying layers to a Component

```
1  // inside the Context Mixin
2  activeLayers() {
3    return this.hasLayers() ? this.context.layers : []
4  }
```

The Context Mixin is designed to manage the layer activation in the local scope. Though the Context Mixin does not provide behavior. Custom behavior must be fused into the Mixin before it is added to the Component as seen in figure 2 and listing 1.6.

Listing 1.6. Example Code: Including a custom Mixin into a child Component

```
1  const LayeredContext = {}
2  Object.extend(LayeredContext, Context)
3  Object.extend(LayeredContext, {
4    myLayeredMethod() {
5      return 'I am unlayered text.'
6    }
7  })
8
9  Layers
10   // same layer name as attached in the parent component
11   .setupLayer('MyLayer')
12   .refineObject(LayeredContext, {
13     myLayeredMethod() {
14       return 'I am layered text.'
```

```
15      }
16    })
17
18  const ChildComponent = React.createClass({
19    mixins: [ LayeredContext ],
20
21    render() {
22      return <div>{this.myLayeredMethod()}</div>
23    }
24  })
```

A Context Mixin receives the layer information from its assigned Component. When the Component is nested elsewhere in the DOM tree, its context will change. React will refresh the parent-based context chain, thus the *activeLayers* method of the LayerableObjectTrait will return a different set of layers. This procedure makes it possible to implicitely handle layer activation for every Component. Listing 1.7 shows an example where identical Components behave differently in different context scopes.

Listing 1.7. Implicit layer activation is managed by declarative nesting of Components

```
1  // rendering identical ChildComponents in a different
       context
2  React.render(
3    <div>
4      <ParentComponent layerName="MyLayer">
5        <ChildComponent /> // shows 'I am layered text.'
6      </ParentComponent>
7      <ParentComponent>
8        <ChildComponent /> // shows 'I am unlayered text.'
9      </ParentComponent>
10   </div>, document.body
11 )
```

3 Discussion

3.1 Design Constraints

Previously we presented a working design of how to meld the concepts of React and ContextJS by using layers, object refinement and Mixins. The restriction we imposed on ourselves was to avoid altering the implementation of either library. Thus we created a flexible library ourselves which can easily adapt API changes in future development of React or ContextJS. So both libraries can stay up to date without breaking the presented implementation.

We mentioned that Components are the main entity of React-based applications, though we introduced layers in combination with Mixins, a rather weak subset of React's functionality. This is because the *React.createClass* statement is a factory for factories of Components. When we want to layer a Component

directly, we have to create an instance with a concrete factory first. But compared to HTML-style instance creation with JSX transforming this would be a huge overhead of code. Mixins are a rather simple design. Plain objects can be layered using *Layer.refineObject* just fine. Then React takes care of integrating the layered behavior into the Component's prototype.

Another flaw in design can be seen when it comes to structural scoping of layers. We had to simulate the scoping by applying layers manually to each Context-aware Component. We cannot profit of the structural layer implementation of ContextJS because the virtual DOM of React is not managed within the Component. Thus a child cannot access its parent Component directly. The structural layer lookup in ContextJS is bottom-up while React only supports top-down relations.

3.2 Benchmarks

Thinking of practical relevance of this discussed integration performance is an issue. Now we want to measure how much time overhead the integration of COP produces. The following benchmarks were run on *2.3GHz Intel Core i7*, *8GB* memory and running *Max OS X 10.10.4*. In 100 test cycles we rendered 1000 parent Components containing exactly one child Component. The parent Component provides context information and the child component renders text accordingly. We do this in two comparable approaches. The first approach used the described ContextControl and Context mechanism to let the child know which context is set. The second approach uses the parent-based context chains to pass information to the child. The child Component uses string matching to decide if the context is correct. Figure 4 shows the results.

The non-layered benchmark needs *1.13s* in average. The layered benchmark takes an average total time of *3.96s*. We can observe that the COP implementation suffers a performance drop by a factor of *3.5*. This is because each function call requires at least one additional lookup.

So using ContextJS clearly forces a trade-off between clean management of cross-cutting concerns and performance. Web applications, like games or real-time data visualization, focus on fast-running implementations, though for most cases it will be about rendering a few context-aware Components. This results only in an overhead of several milliseconds. We think this is sufficient performance for a natural user experience.

4 Related Work

Comparison of COP languages: Lincke et al. compared several implementations of COP in different programming languages, e.g. Javascript, Smalltalk and Java [1]. At a higher level of detail they showed performance benchmarks and explained and compared concepts of context-oriented programming, like the class-in-layer and layer-in-class strategy we discussed above.

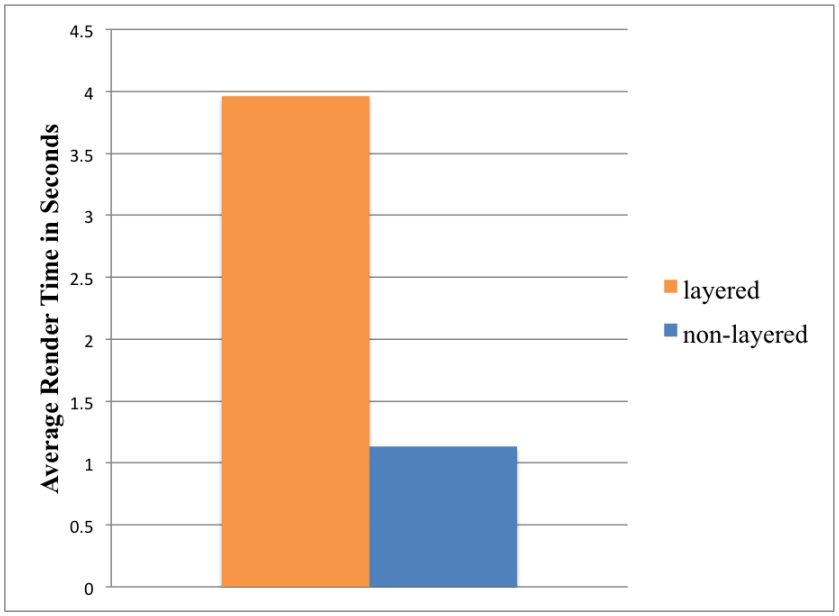

Figure 4. Benchmarks comparing layered and non-layered test case

Structural layers in ContextJS: Lincke et al. presented a solution of layer activation for object hierarchies. The so called structural layer activation requires domain knowledge about the object composition to propagate the scope of the layers correctly along the hierarchy. Using an open implementation of LayerableObjectTraits allows injecting layer information into the object instance. When objects need to know about structural layers the owner chains is walked up recursively and instance-specific layer information is collected. We got inspiration from the way objects extend the structural scope of layers in ContextJS. Though we presented a top-down approach while Lincke et al. proposes a bottom-up aggregation of layer information.

Event-Specific Software Composition in Context-Oriented Programming: Appeltauer et al. studied approaches of event-based layer composition [2]. They introduced JCop, a COP implementation and extension to Java. JCop is about declarative and intuitive definitions of behavioral variations. Progressing work on this paper made clear, that React Components and their spatial context require declarative specifications as well. Furthermore the reactive programming approach of React allows intuitive implicit layer activation. A similar ambition can be seen in the work of Appeltauer et al.

5 Future Work

The presented solution set focus on a combination of object refinement and Mixins. The possiblity of using the layer-in-class strategy instead of the current class-in-layer strategy could enhance the current programming model. Thus we could integrate the layered methods directly into the definition of a Component. As stated before, *React.createClass* is an abstract factory. Impelementing the layer-in-class strategy could require to alter the code base of React itself.

Apart from solutions with context-oriented programming, we should think of an implementation where only conditionals are used to provide the context switch according to the position of a Compontent in the DOM structure. Using explicit Component names, parent-based context chains and a simple switch statement over these names child Components could determine under which Components they are nested. A combination of Mixins as a replacement for layered behavior could produce a similar result as the current implementation. Plus we probably would not suffer from the performance loss COP brings in.

This design alternative could compare the paradigm of COP with the known imperative principle of approaching context with conditionals. This paper merely presented a first solution of handling cross-cutting concerns in React with COP. Its practical relevance has yet to be studied.

6 Conclusion

In this paper we presented an integration of the COP implementation ContextJS and the React Javascript library by Facebook. Due to the virtual DOM concept of React we were able to encode an UI-Component's context in its spatial position. This allowed applying behavioral variations in the manner of COP. Additionally the conditions for layer composition can be determined from the context of a Component. Thus we provided explicit definitions of layers and behavioral adaptions whilst managing layer activation implicitly.

Bibliography

[1] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A Comparison of Context-oriented Programming Languages. In: COP '09: International Workshop on Context-Oriented Programming. pp. 1–6. ACM Press, New York, NY, USA (2009)

[2] Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M., Kawauchi, K.: Event-specific software composition in context-oriented programming. In: Baudry, B., Wohlstadter, E. (eds.) Software Composition, Lecture Notes in Computer Science, vol. 6144, pp. 50–65. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14046-4_4

[3] Hirschfeld, R., Costanza, P., Haupt, M.: An introduction to context-oriented programming with contexts. In: Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science, vol. 5235, pp. 396–407. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-88643-3_9

[4] Hunt, P.: Why did we build react (2013), http://facebook.github.io/react/blog/2013/06/05/why-react.html

[5] Hunt, P.: React documentation - multiple components (2015), http://facebook.github.io/react/docs/multiple-components.html

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP'97 - Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer Berlin Heidelberg (1997), http://dx.doi.org/10.1007/BFb0053381

[7] Lincke, J., Appeltauer, M., Steinert, B., Hirschfeld, R.: An open implementation for context-oriented layer composition in contextjs. Science of Computer Programming X, 19 (2011)

All links were last followed on July 31, 2015.