Improving the Development of Context-dependent Java Applications with ContextJ

Malte Appeltauer Robert Hirschfeld
Hasso-Plattner-Institute
University of Potsdam, Germany
{first.last}@hpi.uni-potsdam.de

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo, Japan
masuhara@acm.org

ABSTRACT

Context-oriented programming languages ease the design and implementation of context-dependent applications. ContextJ is a context-oriented extension to the Java programming language. In this paper, we assess the applicability of ContextJ language abstractions for the development of a graphical user interface-based application. We present a text editor that has been implemented with ContextJ based on the Qt Jambi framework and discuss possible extensions to ContextJ to improve its applicability.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Context-oriented programming, behavioral variations, layer activation, dynamic scoping

1. INTRODUCTION

Context-oriented programming [8] (COP) addresses the development of systems, whose behavior varies depending on their context of use. In most cases, a behavioral variation is not implemented by a single object; instead, it is distributed over a team of collaborating objects. Such distributed functionality is denoted as *crosscutting concern* [10]. The modularization and composition of crosscutting concerns requires additional language abstractions beyond object-oriented programming. COP introduces layers, an encapsulation mechanism for crosscutting behavioral variations. A layer can be dynamically activated and composed with other layers, allowing fine-grained control of an application's runtime behavior. COP has been implemented as extension to several programming languages [5, 7, 13–15] to support various problem domains.

ContextJ [1] is our compiler-based COP extension to the Java programming language. In this paper, we present a ContextJ case study to assess its applicability for the development of graphical user interface (GUI)-based applications. Such applications typically consist of multiple threads handling user interaction. We present a prototypical implementation of a development environment for ContextJ that supports rich text comments in Java/ContextJ compilation units. Depending on the current activity, i.e., programming or documentation, the GUI provides different UI elements that are modularized with layers and that can be activated at run-time.

The rest of this paper is structured as follows. Section 2 introduces ContextJ. Section 3 presents our ContextJ-based GUI application. Section 4 discusses the benefits and drawbacks of the implementation. Previous and related work is considered in Section 5. Section 6 summarizes the paper and presents future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
COP '09 July 7, 2009, Genova, Italy
Copyright 2009 ACM 978-1-60558-538-3/09/07 ...$10.00.

2. CONTEXT-ORIENTED PROGRAMMING WITH CONTEXTJ

2.1 Overview

COP allows for the convenient expression of behavioral variations that cut across a system's dominant decomposition. Context-dependent functionality is explicitly represented and can be dynamically composed at run-time. Context that requires a composition of behavioral variations can be *everything that is computationally accessible* [8], such as state, control flow, or properties of the system's environment.

Layers. Behavioral variations are implemented by *layered methods*. A layered method consists of a *base method definition* and at least one *partial method definition*, which is defined in a layer. A base method denotes the Java method definition that is executed when no active layer provides a corresponding partial method. A partial method definition implements the functionality of a behavioral variation that extends or overrides a base method definition for the time the layer is active.

Dynamic composition. Layers can be composed at runtime. Invocations of layered methods are first send to their active partial method definitions. During its execution, a behavioral variation can proceed to a corresponding partial method in another active layer or, if such method does not exist, to the base method definition. If more than one active layer provides a partial definition for a layered method, the order of layer activation defines the proceed chain, in which

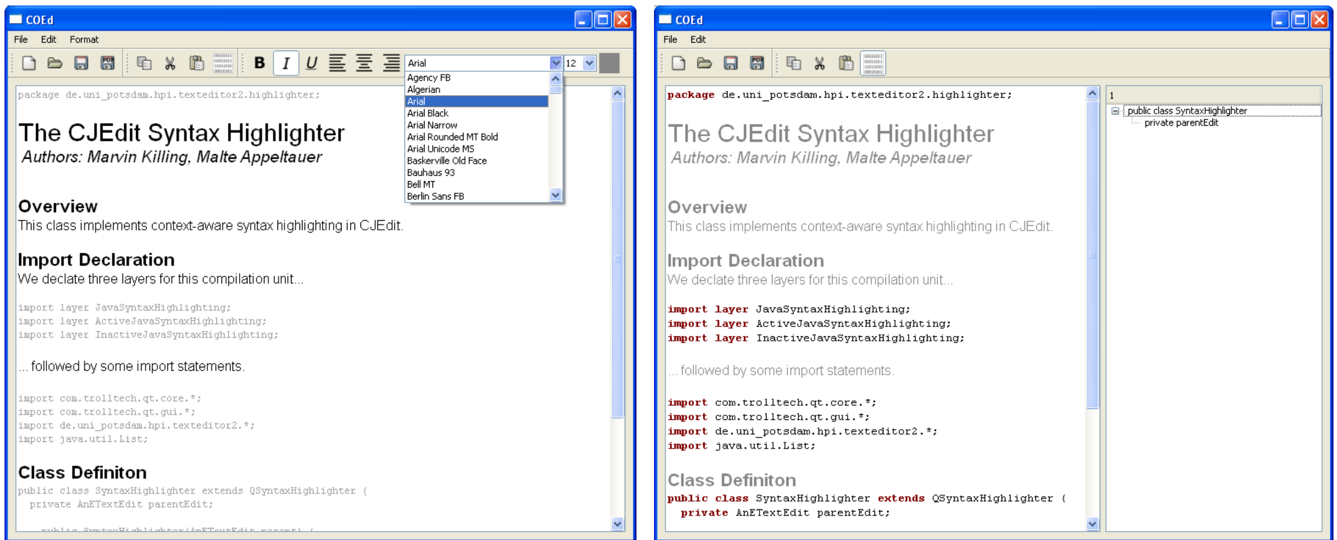

Figure 1: Screenshots of CJEdit. *left*: Richt-text editing is supported by format toolbars and menus. *right*: Program development is supported by an outline and focus on source code blocks.

the layer activated last is accessed first. Per default, layer activation is scoped per thread and to the dynamic extent of a block of statements.

2.2 ContextJ

The ContextJ[1] programming language features the *layers-in-class* style [8]: each class affected by a layer L contains a declaration of L that contains partial methods for its class. Thus, classes carry their own context-specific variations.

A layer definition consists of an identifier and a list of method definitions. These definitions specify either new methods that are visible in the scope of their layer, or partial method definitions, whose signature must correspond to a base method in the hierarchy of the enclosing class. During layer activation, invocations of this method are dispatched to the definition provided by this layer.

The built-in pseudo method `proceed` can be used to explicitly invoke the next partial method definition (or the base method). Both the return type and the expected arguments of `proceed` conform to the method's signature.

In addition to method definitions, layers can also contain fields that are visible within the layer's scope. The state of these fields persists after layer deactivation until the next activation.

To control scoped layer activation, ContextJ provides the `with` block statement that can be used in method bodies. It consists of list of expressions of type `Layer`, `Iterable<Layer>`, or `Layer[]` that specify the layer identifiers to be activated for the dynamic extent of its block. If the `with` arguments are evaluated to an empty list (or `null`), no additional layer will be activated. The `without` block construct, as counterpart to `with`, is used for explicitly disabling layer execution for a certain control flow.

[1]ContextJ is available for download at
`http://www.hpi.uni-potsdam.de/swa/cop`

3. DEVELOPING GUI APPLICATIONS WITH CONTEXTJ

For our ContextJ case study, we developed a programming environment that supports rich text comments within ContextJ programs.

Context-dependent GUI functionality. Modern rich text editing environments, such as *Microsoft Word*, provide extensive functionality to edit, format, and manage text and other elements, for example, pictures and data tables. Such large feature sets cause an additional complexity that also end-users have to cope with. To ease the use of complex work-flows, Word-like applications support context-specific menus and toolbars that emphasize important and hide irrelevant functionality. For instance, *Microsoft Word* provides an environment that partly integrates *Microsoft Excel* features to deal with tables; *Excel*-specific functionality (such as toolbars and menu entries) and behavior (such as cell calculation and formatting) is provided whenever a table is selected by the user. The ContextJ-based application presented in Section 3.1, provides similar behavior for the tasks *programming* and *code commenting*.

Rich text-based source-code comments. The documentation of applications is an important task in the software development process. In addition to architectural and infrastructural descriptions, the documentation of source-code artifacts is vital for understanding and evolving software systems. Therefore, most programming languages support source-code comments. However, most languages and programming environments restrict comments to plain text; text formating is not possible. To overcome this issue, tools like JavaDoc allow for the generation of HTML documents from source-code. This leads to a separation of source code from its formatted documentation, which is unintended in most cases since it requires additional effort to synchronize source code and comments.

3.1 CJEdit

In the following, we give an overview of *CJEdit*, a simple *programming environment* for ContextJ. The editor is equipped with syntax highlighting, an outline view, and a compilation/execution toolbar. Additionally, CJEdit allows to format ContextJ compilation units with rich text comments. For this task, the editor provides *rich text formatting features*, such as font family, size, style, and color modifications. Through the combination of rich text and source code, CJEdit documents are single-source, executable representations of code and documentation.

Both activities require different functionality, therefore our application supports focusing on the actual task at hand by offering only relevant tools, menus, and widgets. A context switch between text editing and programming features is either directly triggered by the user, or on text cursor change: While writing new text, the user can enter the programming mode by pushing a toolbar button. Whenever the text cursor is moved through the document from text to code and vice versa, the GUI elements are changed accordingly. Figure 1 shows two screenshots of CJEdit. The left screenshot presents the application's rich text formatting mode. The toolbar offers various text formatting actions. The right image depicts the programming mode, where the editor comes with an outline and a different toolbar. To support focusing on the source code, any rich text within the document is faded gray.

3.2 Implementation

CJEdit is implemented using ContextJ and the *Qt Jambi GUI Framework* [12]. The editor consists of approximately 1400 lines of code, where most parts are written with plain Java constructs and the help of the Qt GUI Designer. The overlay of task-specific user interfaces and behavior is implemented in ContextJ. The system contains layers that encapsulate rich text and programming widgets such as toolbars and their corresponding behavior.

Encapsulation of GUI elements into layers. Figure 2 shows the implementation of the layers that encapsulate the creation of the task-specific user interfaces. Both layers provide partial methods of `showToolBars` and `showMenus` resp. `showWidgets` (Lines 10-21, 31-42) that are executed after the execution of their base methods. On a task switch, the system hides specific GUI elements by calling `hideWidgets` (Lines 22-25, 46-50) and invokes the `showWidgets`, `showMenus`, and `showToolBars` methods. Thus, the layers extend `hideWidgets` with a partial method that removes the layer-specific widgets.

Text blocks carry their composition. The editor's underlying document tree represents each text line as a text block node. Each block holds a list of layers that should be activated when it is focused. By default, blocks refer to the layers responsible for rich text behavior. If the user switches to the programming activity (by pressing the code button in the toolbar), the following text blocks are linked with programming environment-specific layers.

Task-depended behavioral variations. Layers are recomposed whenever the type of the focused block changes from rich text to code block, and vice versa. This change is explicitly activated by entering or leaving the programming activity (by pressing the code button) or on moving the text cursor between blocks of different types.

For the dynamic extent of the recomposition, the layer list of the current block is activated. The composition is triggerd by the `onCursorPositionChanged` event, as depicted in Figure 3. First, `hideWidgets` is invoked in the context of layers of the previous block to remove their specific widgets. The GUI elements are then recomposed with the layer composition of the current block.

```
1  import layer RTFWidgets;
2  import layer CodeWidgets;
3
4  public class CJEditWindow extends QMainWindow {
5    ...
6    layer RTFWidgets {
7      private QMenu formatMenu;
8      private FormatToolBar formatToolBar;
9
10     after private void showMenus(...)  {
11       if (formatMenu == null)
12         formatMenu = new FormatMenu(...);
13       menuBar().addMenu(formatMenu);
14     }
15     after private void showToolBars(...) {
16       if (formatToolBar == null) {
17         formatToolBar = new FormatToolBar(...);
18         addToolBar(formatToolBar);
19       }
20       formatToolBar.show();
21     }
22     after private void hideWidgets() {
23       formatMenu.hide();
24       formatToolBar.hide();
25     }
26   }
27   layer CodeWidgets {
28     private QWidget outline;
29     private CodeToolBar codeToolBar;
30
31     after private void showWidgets() {
32       if (outline == null)
33         outline = createOutlineWidget();
34       outline.show();
35     }
36     after private void showToolBars(...) {
37       if (codeToolBar == null) {
38         codeToolBar = new CodeToolBar(...);
39         addToolBar(codeToolBar);
40       }
41       codeToolBar.show();
42     }
43     private QWidget createOutlineWidget() {
44       ...
45     }
46     after private void hideWidgets() {
47       outline.hide();
48       codeToolBar.hide();
49     }
50   }
51 }
```

Figure 2: Layered specification of task-dependent GUI Widgets.

4. LESSONS LEARNED

The two main behavioral variations implemented in our example, namely *rich text editing* and *program development* have been implemented using layers. The layers contain partial method definitions that implement the variations of the default behavior of certain methods. The user-based be-

```
1 import layer RTFWidgets;
2 import layer CodeWidgets;
3
4 public class CJEditWindow extends QMainWindow {
5   ...
6
7   void onCursorPositionChanged() {
8     with (getLayersOfPreviousBlock()) {
9       hideWidgets();
10    }
11    with (getLayersOfCurrentBlock()) {
12      showWidgets();
13      showMenus();
14      showToolBars();
15    }
16  }
17 }
```

Figure 3: Layer recomposition on cursor position change.

havioral switch can be mapped directly to dynamic layer composition.

Besides these benefits, we had to consider some characteristics of GUI-based programming that led to additional challenges for the ContextJ-based implementation. The two most important findings are explained in the following.

First, user interaction with GUI behavior is less control flow-centric but rather event-driven. This complicates dynamic extent-based layer composition as proposed originally by COP. Figure 4 depicts a control flow in CJEdit. On user interaction, the `onCodeModeSelected` event is triggered that computes the layers to be activated for the programming environment mode. Subsequent user interactions – such as printing the document, writing new text, or moving the text cursor through the document – activate this composition in their respective control flows. In the source code, this issue is manifested as repeated **with** statements in the event callback methods, which itself is a crosscutting concern.

Another issue is the intrinsic difference between declarative GUI specifications and dynamic behavioral variations. Figure 2 shows the CJEdit implementation of task-dependent GUI elements. The exclusive specification of layer-specific widgets is not sufficient, we also need auxiliary methods such as `showWidgets` and `hideWidgets`, and explicitly trigger their execution after layer activation. In an better solution, we would only need to declare the GUI variations and add them to the internal structure. With the activation of a layer, the layered state of this structure would be activated, too.

5. PREVIOUS AND RELATED WORK

Several COP extensions have been developed for dynamic programming languages such as Lisp [5], Smalltalk [7], Python [14,15], and Ruby [13]. They have been implemented using the respective language's meta-level facilities; none of them utilizes bytecode transformation as done in ContextJ.

First ideas about a COP language extension to Java have been presented in [6], but neither provide a language specification, nor an implementation. The first Java-based prototype is *ContextJ** [8], a Java library that implements the core concepts of COP. The convenient use of ContextJ* is hindered by the lack of a declarative syntax, leading to complex implementations in ContextJ* programs.

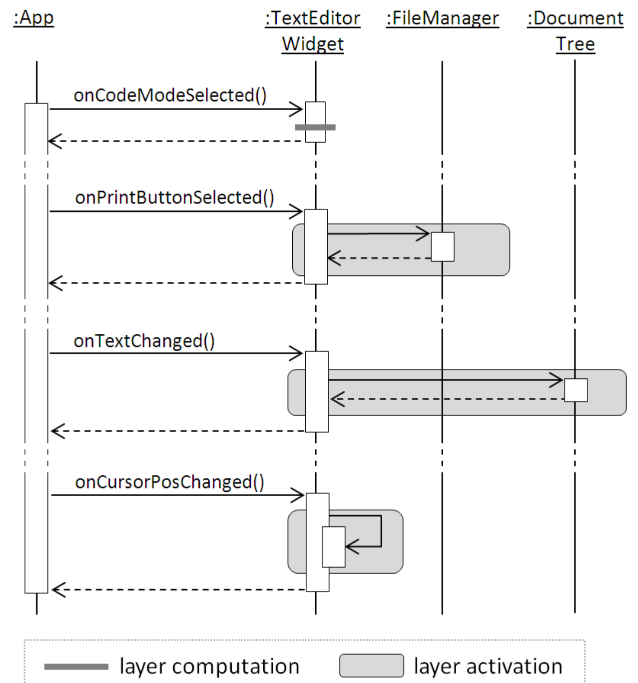

Figure 4: Layer composition computation and activation times. Layer activations are scattered over the system.

Java-based *aspect-oriented programming* languages, such as *AspectJ* [9] or *CaesarJ* [2], also provide abstractions to modularize crosscutting concerns but focus on functionality that is repeatedly used at different points in a program. This view differs from ContextJ, where crosscutting behavioral variations are considered as parts of the classes themselves.

The CaesarJ [2] language unifies classes, aspects, and packages. CaesarJ aspects can be deployed at run-time using different kinds of dynamic scope, much like ContextJ layers. CaesarJ supports virtual classes [11], a concept that enables dynamic class extension, depending on the caller's scope. The ability of virtual classes to extend modules is similar to layers. However, class extension with layers is not bound on the caller's module but differs depending on the current layer composition.

The maintenance and development of software product-lines is addressed by *feature-oriented programming* [4]. The Java-based *AHEAD Tool Suite* [3] provides a Java language extension supporting constructs such as class refinements for static feature-oriented composition. Layers are distinct rules describing static class refinements. Some foundations of AHEAD and ContextJ are the same: Both introduce new or alternative program behavior through layers. However, AHEAD applies compile-time composition of feature variations, whereas ContextJ allows for run-time composition.

6. SUMMARY AND FUTURE WORK

In this paper, we introduced to the ContextJ programming language and considered it as basis for the development of GUI-based applications. We presented a case study, in which we implemented CJEdit, a ContextJ development

environment with the ability to comment source-code with rich text. In our example, ContextJ language features support the separation of activity-specific concerns. In addition, we identified issues for future improvement to provide better support for the development of event-based systems.

One result of our investigation is that the language abstractions provided by ContextJ, though they already ease the development of context-aware systems, need further improvement. We are currently working on alternative layer composition mechanisms to overcome this issue. We will adapt some abstractions of aspect-oriented programming for layer activation specifications to provide a more declarative way of dealing with the composition of behavioral variations, especially in event-based systems.

Context-dependent applications may require to change a system's state for a certain context. A simple approach has already been implemented in ContextJ, that is, layers can introduce new state into classes. However, superimposition of state for the time of a layer activation is not supported, yet. The development of appropriate semantics for stateful layers will be considered in future work.

To evaluate COP for distributed applications such as service-based systems, we will set up such a system based on ContextJ and our new layer composition techniques. We will investigate how COP provides suitable abstractions for application requirements in this domain.

Acknowledgements

We thank Pascal Costanza for helpful discussions and suggestions during the development of ContextJ and Marvin Killing for his initial work on CJEdit. We also thank Michael Perscheid, Jens Lincke, Robert Krahn, Michael Haupt, and the anonymous reviewers for valuable feedback on drafts of this paper.

7. REFERENCES

[1] M. Appeltauer. ContextJ – Context-oriented Programming for Java. In *Proceedings of the 3rd Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*, number 27. Hasso-Plattner-Institut, Potsdam, Germany, 2009.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, 3880:135–173, 2006.

[3] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2003.

[5] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[6] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In D. E. Lightfoot and C. A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103, Berlin, Heidelberg, Germany, September 19 2006. Springer-Verlag.

[7] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-Oriented Programming with ContextS. In J. S. Ralf Lämmel, Joost Visser, editor, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[8] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354, Berlin, Heidelberg, Germany, January 2001. Spinger-Verlag.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[11] O. L. Madsen, B. Mø-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[12] Nokia Corporation. Whitepaper: A Technical Introduction to Qt, 2000. http://www.qtsoftware.com.

[13] G. Schmidt. ContextR & ContextWiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[14] C. Schubert. ContextPy & PyDCL - Dynamic Contract Layers for Python. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[15] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented Programming: Beyond Layers. In S. Demeyer and J.-F. Perrot, editors, *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, volume 286 of *ACM International Conference Proceeding Series*, pages 143–156, New York, NY, USA, 2007. ACM Press.