Context-oriented Programming in ContextL

State of the Art

Pascal Costanza
Programming Technology Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
pascal.costanza@vub.ac.be

ABSTRACT

There is a wide range of scenarios where software systems have to be able to behave differently according to their context of use. In Context-oriented Programming (COP), programs can be partitioned into behavioral variations that can be freely activated and combined at runtime with well-defined scopes, such that the program behavior is affected depending on context. About four years ago, we have introduced our vision of Context-oriented Programming and have presented the programming language ContextL as an extension to the Common Lisp Object System (CLOS), as our first language extension that explicitly realizes this vision. Since then, ContextL has been picked up by various developers world-wide, is now in use in several software systems, and has been continuously improved to meet the demands of its users. For these reasons, ContextL can currently be regarded as the most mature realization of COP concepts. In this paper, we give an overview of the major ingredients of ContextL, describe the developments in ContextL of the last four years, and sketch some future work.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

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

1. CONTEXT-ORIENTED PROGRAMMING

There is a wide range of scenarios where software systems have to be able to behave differently according to their context of use [2, 3, 4, 6, 16, 23, 25, 28]. In Context-oriented Programming (COP), programs can be partitioned into behavioral variations that can be freely activated and combined at runtime with well-defined scopes. Such behavioral variations consist of partial definitions for typical program entities, like classes, methods, functions, procedures, and so on. The essential ingredients of Context-oriented Programming are as follows:

Context is any information that can be computationally accessed in a software system.

Behavioral variations describe the context-dependent behavior of a software system as increments on the underlying context-independent program definitions.

Layers group such behavioral variations as first-class entities that can be referenced in a program at runtime.

Layer activation is achieved by language constructs that ensure that such layers are added at runtime, such that the respective partial program definitions have an influence on the actual behavior of a program.

Scoping of layer activation and deactivation ensures that the behavioral variations are only effective for well-defined parts of a program, and for well-defined durations.

Context-oriented Programming focuses on programming constructs to enable grouping, referencing, and activation and deactivation of layers of behavioral variations. It should thus be seen as a complement to the (equally important) research on context acquisition and reasoning, where the focus is on sensing (low-level) context data and inferring (high-level) context information (for example [25]). Our definition of context is open and pragmatic, in that it treats *any* computationally accessible information as potential parameters for influencing the behavior of a program. In contrast, the widely cited definition for context of Dey at al. [17] states that "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." That definition distinguishes between relevant and irrelevant information, which is important when modelling systems for context acquisition and context reasoning. It differs in this regard from our definition that is targeted at structured ways of how to affect program behavior depending on context, which should be addressed by general-purpose constructs and thus independent of the kinds of context information they may or may not depend on.

About four years ago, we have introduced our vision of Context-oriented Programming and have presented the programming language ContextL as an extension to the Common Lisp Object System (CLOS), as our first language extension that explicitly realizes this vision [8]. Since then we have implemented the underlying ideas as extensions for various other programming languages, like Smalltalk [19], Java [9], Ruby and Python, together with various use cases. During the course of these experiments, the essential ingredients of *context*, *behavioral variations* and *scoped layer activation* have proved to be stable corner stones.[1]

On top of that, ContextL has been picked up by various developers world-wide since its public release as an open-source language extension, is now in use in several software systems in the "real world," and has been continuously improved to meet the demands of its users. For these reasons, ContextL can currently be regarded as the most mature realization of COP concepts.[2]

2. CONTEXTL: STATE OF THE ART

2.1 Essential Features

ContextL is an extension to the Common Lisp Object System (CLOS), and thus includes object-oriented features like classes with multiple inheritance, slots (fields) and methods with multiple dispatch. Like CLOS, ContextL is not based on message sending, but on generic functions [5].[3] For each of the core defining constructs in CLOS, there is a corresponding defining construct in ContextL: For defining classes, ContextL provides `define-layered-class` as an analogue to CLOS's `defclass`. Likewise, ContextL provides `define-layered-function` and `define-layered-method` as analogues to CLOS's `defgeneric` and `defmethod` for defining generic functions and methods.

Layers can be introduced in ContextL with `deflayer` and an associated name, as follows.

```
(deflayer layer-name)
```

Such layers can then be further specified with partial class and method definitions, which can be explicitly associated with such layers, as follows.

```
(define-layered-class class-name
  [:in-layer layer-name]
  ({superclass}*)
  ({slot-specification}*)
  {class-option}*)

(defmethod function-name [:in-layer layer-name]
  {method-qualifier}* parameters method-body)
```

There is always a *root* layer present that defines the context-independent behavior of a program, and class and method definitions can be associated with the *root* layer by either omitting the `:in-layer` specification, or by using `t` as the layer name.[4]

By default, only the root layer is active at runtime, which means that only definitions associated with the root layer effect the behavior of a program. Other layers can be activated at runtime by way of `with-active-layers`, as follows.

```
(with-active-layers ({layer-name}*)
  body)
```

Such a layer activation ensures that all the named layers affect the program behavior for the dynamic extent of the enclosed program code (*body*). Layer activation is dynamically scoped: Both direct and indirect invocations of generic functions (methods) and slot accesses in the control flow of the layer activation are affected. Furthermore, layer activation is restricted to the current thread of execution in multithreaded Common Lisp implementations, to avoid race conditions and interferences between different contexts.

Furthermore, layers can be deactivated at runtime by way of `with-inactive-layers`, as follows.

```
(with-inactive-layers ({layer-name}*)
  body)
```

Such a layer deactivation ensures that none of the named layers affect the program behavior for the dynamic extent of the enclosed program code anymore. As with `with-active-layers`, layer deactivation is dynamically scoped and restricted to the current thread of execution.

Layer activations and deactivations can be nested arbitrarily in the control flow of the program. Both multiple activations and deactivations of the same layer are ignored, each layer is only active or inactive at most once. Furthermore, the order of layer activation determines method specificity when generic functions are invoked: Methods from more recently activated layers in the control flow of a program are executed before methods from less recently activated layers and the root layer.

The above constructs allow referencing layers at runtime, but only as second-class citizens. However, for each of the above constructs, there exist corresponding first-class constructs: The functions `ensure-layered-function`, `ensure-layered-method` and `ensure-layer` enable defining layered functions, layered methods and layers at runtime. Furthermore, `funcall-with-layer-context` and `apply-with-layer-context` enable activating and deactivating computed combinations of layers, and `adjoin-layer` and `remove-layer` enable computing such combinations of layers from first-class layer representations.

Finally, there are cases where it is useful that layers are activated or deactivated globally for all threads without dynamic scope, for example for interactive testing and development, or for program deployment (at system startup time). The functions `ensure-active-layer` and `ensure-inactive-layer` enable such global activation and deactivation of layers.

2.2 Efficient Layer Activation

Context-oriented Programming encourages continually changing program behavior. For example, we have illustrated a use case in [9] where side effects on graphical objects trigger updates of their representations on the screen. Context-oriented Programming can successfully be used in such a scenario to avoid invalid screen updates in situations where graphical objects are in intermediate states during

[1] All this work has been performed in close collaboration with Robert Hirschfeld.

[2] ContextL can be downloaded from `http://common-lisp.net/project/closer/contextl.html`

[3] Note, however, that our experiments with similar extensions for other object-oriented programming languages show that COP is compatible with message sending as well.

[4] The symbol `t` is traditionally used in Lisp dialects to denote "general" concepts, like the boolan truth value, the supertype of all types, the standard output stream, and so on.

composite changes. This use case is characterized by repeated activation and deactivation of a layer that is responsible for the necessary screen updates.

Such examples show that it must be possible to implement layer activation and deactivation efficiently in order to make Context-oriented Programming a sane option in a programmer's toolbox. It is indeed not obvious that layer activation and deactivation can be efficient. However, we have found an implementation for ContextL on top of CLOS with competitive performance. The key ingredients are as follows.

- Layer activation and deactivation leads to a runtime composition of multiple, arbitrary layers. Such layer composition is realized internally in ContextL by reusing CLOS's support for multiple inheritance. This can be achieved by representing composed layers as dynamically created classes that multiply inherit from other layers, on top of which they are activated and which are themselves represented as classes. Dynamic class creation is supported by way of the CLOS Metaobject Protocol [20], and although dynamic class creation is costly in terms of performance, such classes can be cached in a way such that each combination of a given sequence of layers needs to be created only once, and can be looked up very efficiently from then on.

- In ContextL, method selection and combination must depend on the current sequence of active layers. Since layer combinations are already represented as classes, we can internally represent the current sequence of active layers as a (prototypical) instance of such a combined class, and drive method selection and combination by implicitly passing such an instance as an argument to an appropriately specialized parameter that is added to each layered method internally.[5] In this case, efficiency also comes from caches that allow for fast lookup of applicable methods, which is already provided by typical CLOS implementations [21].

- Layer activation and deactivation must modify the current sequence of active layers in such a way that the change is only visible in the current thread of execution, to ensure that they do not interfere with other threads in multithreaded Common Lisp implementations. To achieve this, the representation of the current sequence of active layers is bound to a dynamically scoped, thread-local variable, which are directly supported by Common Lisp and for which implementation alternatives with well-understood performance characteristics exist [1].

In [9], we discuss our overall implementation strategy in more detail, together with some benchmarks that indeed show a low overhead for layer activation and deactivation in terms of performance.

2.3 Reflective Layer Activation

Organizing a program into layers may lead to dependencies between layers: Layers may require each other's presence, or may be mutually exclusive. For example, a typical use case for Context-oriented Programming is to separate the generation of different output formats (html, pdf, json, etc.) from the same document tree into several layers. Activating a layer for one output format should thus deactivate the layers for other output formats. In [10], we illustrated this idea with another example, where different tariffs for cell phone usage are separated into different layers, which must be mutually exclusive and, at the same time, all require the presence of another base tariff layer.

In principle, it is possible to make such dependencies explicit by simply activating and deactivating all the involved layers in all the places where layer activation or deactivation occurs. However, this turns out cumbersome. In [10], we introduce a reflective API for ContextL that can be used to make layer activations and deactivations automatically trigger other layer activations and deactivations behind the scenes. The essential idea is that each layer activation computes the resulting sequence of active layers by invoking the function `adjoin-layer-using-class`, and each layer deactivation computes the resulting sequence of active layers by invoking `remove-layer-using-class`. These two functions are exposed by the ContextL API, are themselves layered functions, and can thus themselves be specialized by user-provided layered methods. Such layered methods can inspect the layers to be activated or deactivated, inspect the current sequence of active layers, and ensure that other required layers are implicitly activated as well, other excluded layers are implicitly deactivated, or invalid layer activations or deactivations are rejected.

In [10], we discuss the details of ContextL's reflective architecture, illustrate its use by giving examples, and show that it can be implemented without affecting the performance achieved by the implementation strategy discussed in [9] (see above).

2.4 Description of Layer Dependencies

In spite of the usefulness of reflective layer activation, implementing extensions on top of ContextL's reflective API proves to be a complex task. In [12], we show a first step of how feature diagrams, or better the feature description language (FDL, [15]) as their textual counterpart, can be used to provide a high-level description of layer dependencies, such as requirement and exclusion dependencies. FDL can be added as an extension on top of ContextL's reflective API without changing any of ContextL's internal implementation details, and without affecting its essential efficiency characteristics.

We have also gained some first experience with using feature diagrams for the requirements elicitation and design phases of context-oriented systems. *Context-Oriented Domain Analysis* (CODA) [14] is a notation and methodology for describing both the context-independent behavior and the context-dependent behaviorial variations of a system. This notation is based on feature diagrams, but extends it to include – potentially dynamically changing – conditions under which layers are required or excluded. We are currently working on mapping CODA diagrams to context-oriented programming languages such as ContextL.

2.5 Impact

Since its original release in 2005, ContextL has been picked up by several developers world-wide, and is now used in a number of software systems in the "real world." ContextL is typically employed in web applications, where layers are

[5] Due to the fact that CLOS supports multiple dispatch, adding such a specialized parameter to a method does not prevent users from dispatching on other arguments.

used to separate generation of different output formats from the same document tree, and to provide different views and modes to different kinds of users depending on their respective tasks. For a description of one exemplary case see `http://p-cos.blogspot.com/2007_11_01_archive.html`.

Furthermore, Context-oriented Programming has started to impact other researchers who have picked up the term and the concepts as starting points for their own work [7, 13, 18, 26]. A detailed discussion of such related work is outside the scope of the overview given here.

3. FUTURE WORK AND CONCLUSIONS

There are several ways in which ContextL can be extended to broaden its scope of applicability:

- ContextL's reflective API can be extended to support first-class dynamic enviroments. The idea is to support capturing the current sequence of active layers as a first-class entity such that it can be reeinstated later for a different context. For example, this would allow passing context from one thread to another and storing the current context of a suspended computation to be later resumed, for example in continuation-based web applications [24]. Since frameworks for such continuation-based web applications are typically based on *partial continuations*, we actually need support for delimited dynamic bindings, as in [22]. Since there are different continuation frameworks available for Common Lisp, we have to design the reflective API for first-class dynamic environments in such a way that they are compatible with different kinds of partial continuations.

- Currently, layers in ContextL contain partial class and method definitions, where the method definitions are typically specialized on classes. Although CLOS supports method specialization on single objects as well, somewhat similar to what is provided in prototype-based object systems, the actual use of layer activation or deactivation for single objects is not straightforward, but would sometimes be useful. After some experimentation with different ideas, we have come up with a generalization of generic function dispatch called *filtered dispatch* which allows for filtering arguments before method selection and combination, while passing the original unfiltered arguments to the thus selected methods for execution [11]. This results in a very powerful and expressive dispatch mechanism, which has a potential to serve as a basis for context-dependent behavior of single objects.

- Context-dependent behavior is also important in distributed settings, especially in the case of ad-hoc mobile networks between mobile devices. Some first experiments in using COP concepts in such a setting are very promising. Interesting research topics are how to distribute layer activation across several nodes of such a network, and how to deal with potentially different context parameters from different interacting nodes. We have described first steps in that regard in the Context-Dependent Role Model [27], and are currently working on integrating a full-fledged distributed architecture with ContextL.

Other important areas of research need to address questions such as:

- What are good software architectures that can take advantage of Context-oriented Programming? There is now a considerable amount of pratical experience with languages like ContextL, and some design principles have started to emerge, but they have not been codified yet, for example in the form of design patterns and architectural patterns, or in the form of libraries and frameworks that take advantage of Context-oriented Programming.

- What are good methodologies for eliciting requirements for, and designing context-oriented software systems? We have made some first steps with CODA [14] (see above), and current research in the field of Software Product Lines and especially Dynamic Software Product Lines are very promising starting points.

- How can context-aware systems be tested and verified? This is an inherently hard issue because of the combinatorial explosion of possibilities induced by the presence of multiple context parameters, which is reflected in Context-oriented Programming by the presence of multiple layers that can be arbitrarily combined with each other in the general case.

At this stage in time, we can rightfully claim that Context-oriented Programming has reached a certain level of maturity, has fostered some interesting research results, and has started to gain traction in industry. ContextL, as based on the Common Lisp Object System, has proved to be a viable tool both for research and for industrial use, and will therefore remain one of our main vehicles for exploring the field in more depth.

Acknowledgments

The central concepts of Context-oriented Programming have been discovered and defined in close collaboration with Robert Hirschfeld over the years.

The following people have generously offered their views and insights to improve our understanding of Context-oriented Programming in general, and the concrete realization in the form of ContextL, at various stages: Eli Barzilay, Elisa Gonzalez Boix, Nick Bourner, Tim Bradshaw, Johan Brichau, Thomas Burdick, Jeff Caldwell, Thomas Cleenewerck, Drew Crampsie, Tom van Cutsem, Theo D'Hondt, Brecht Desmet, Peter Ebraert, Johan Fabry, Paul Foley, Richard Gabriel, Ron Garret, Steven Haflich, Michael Haupt, Charlotte Herzeel, Kaz Kylheku, Ralf Lämmel, Attila Lendvai, Bjoern Lindberg, Barry Margolin, Joe Marshall, Wolfgang De Meuter, Mario Mommer, Stijn Mostinckx, Oscar Nierstrasz, Pekka Pirinen, Kent Pitman, Igor Plekhov, Andreas Raab, Christophe Rhodes, Alexander Schmolck, Nikodemus Siivola, Eric Tanter, Dave Thomas, Jorge Vallejos, Kristof Vanhaesebrouck, Peter Wasilko, and JonL White.

4. REFERENCES

[1] Henry Baker. Shallow Binding Makes Functional Arrays Fast. ACM Sigplan Notices 26, 8 (Aug. 1991).

[2] Matthias Baldauf, Schahram Dustdar, Florian Rosenberg. A Survey of Context-Aware Systems. *International Journal of Ad-Hoc and Ubiquitous Computing*, Interscience Publishers, January 2006.

[3] Jakob E. Bardram. Applications of Context-Aware Computing in Hospital Work: Example and Design Principles. *Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM Press.

[4] Chatschik Bisdikian, Isaac Boarnah, Paul Castro, Archan Misra, Jim Rubas, Nicolas Villoutreix, Danny Yeh, Vladimir Rasin, Henry Huang, and Craig Simonds. Intelligent Pervasive Middleware for Context-Based and Localized Telematics Services. *Proceedings of the 2nd International Workshop on Mobile Commerce*. ACM Press, 2002.

[5] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation* 1, 3-4 (January 1989), 245-394.

[6] Guanling Chen, David Kotz. *A Survey of Context-Aware Mobile Computing Research*. Technical Report, Department of Computer Science, Dartmouth College, November 2000.

[7] Andreas Classen, Arnaud Hubaux, Frans Sanen, Eddy Truyen, Jorge Vallejos, Pascal Costanza, Wolfgang de Meuter, Patrick Heymans, and Wouter Joosen. Modelling Variability in Self-Adaptive Systems: Towards a Research Agenda. Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE'08). Workshop of the Seventh International Conference on Generative Programming and Component Engineering (GPCE'08).

[8] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming. *ACM Dynamic Languages Symposium 2005*. ACM Press.

[9] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. *Joint Modular Languages Conference 2006*, Proceedings, Springer LNCS.

[10] Pascal Costanza and Robert Hirschfeld. Reflective Layer Activation in ContextL. *ACM Symposium on Applied Computing 2007 (SAC 2007)*, Technical Track on Programming for Separation of Concerns (PSC 2007), Proceedings, ACM Press.

[11] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered Dispatch. *Dynamic Languages Symposium 2008) (DLS08)*, co-located with ECOOP 2008. Proceedings, ACM Digital Library.

[12] Pascal Costanza and Theo D'Hondt. Feature Descriptions for Context-oriented Programming, 2nd International Workshop on Dynamic Software Product Lines (DSPL'08), co-located with *Software Product Line Conference 2008 (SPLC2008)*. Proceedings.

[13] Marcus Denker, Mathieu Suen, and Stephane Ducasse. The Meta in Meta-object Architectures. *TOOLS Europe 2008*, Proceedings. Springer, 2008.

[14] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt. Context-Oriented Domain Analysis. *Modeling and Using Context*, Sixth International and Interdisciplinary Conference on Modeling and Using Context. Proceedings. Springer LNCS, 2007.

[15] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1-17, 2002.

[16] Anind Dey, Gregory Abowd. CybreMinder: A Context-Aware System for Supporting Reminders. *Second International Symposium on Handheld and Ubiquitous Computing 2000*, Proceedings, Springer LNCS.

[17] Anind Dey, Gregory Abowd. Towards a better understanding of context and context awareness. Proceedings of Workshop on the What, Who, Where, When and How of Context Awareness, April 2000.

[18] Sebastian Gonzalez, Kim Mens, and Alfredo Cadiz. Context-Oriented Programming with the Ambient Object System. *1st European Lisp Symposium (ELS'08)*. Proceedings, Journal of Universal Computer Science, 2009 (to appear).

[19] Robert Hirschfeld, Pascal Costanza, Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, ETH Zurich, 3/4 2008.

[20] Gregor Kiczales, Jim Des Rivières, Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[21] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. *Proceedings of the 1990 ACM conference on LISP and Functional Programming*. ACM Press.

[22] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited Dynamic Binding. *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM Press.

[23] Zakaria Maamar, Djamal Benslimane, Nanjangud C. Narendra. What Can Context Do For Web Services? *Communications of the ACM*, 49, 12, December 2006.

[24] Christian Queinnec. The Influence of Broswers on Evaluators or, Continuations to Program Web Servers. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*. ACM Press.

[25] Daniel Salber, Anind Dey, Gregory Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. *Proceedings of CHI'99*, ACM Press.

[26] Eric Tanter. Contextual Values. *Dynamic Languages Symposium 2008) (DLS08)*, co-located with ECOOP 2008. Proceedings, ACM Digital Library.

[27] Jorge Vallejos, Peter Ebraert, Brecht Desmet, Tom van Cutsem, Stijn Mostinckx, Pascal Costanza. The Context-Dependent Role Model. *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007)*. Springer LNCS.

[28] Volker Wulf, Björn Golombek. Exploration Environments: Concept and Empirical Evaluation. *Proceedings of GROUP 2001*, ACM 2001 International Conference on Supporting Group Work. Boulder, Colorado, USA, 2001. ACM Press.