Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations

Nicolás Cardozo*†, Sebastian Günther†, Theo D'Hondt† and Kim Mens*
*ICTEAM Institute
Université Catholique de Louvain
Louvain-la-Neuve, Belgium
Email: {nicolas.cardozo,kim.mens}@uclouvain.be
†Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Email: {ncardozo,sguenther,tjdhondt}@vub.ac.be

***Abstract*—Software variability can be supported by providing adaptations on top of a program's core behavior. For defining and composing adaptations in a program, different paradigms have been proposed. Two of them are feature-oriented programming and context-oriented programming. This paper compares an exemplar implementation of each paradigm. For the comparison, a common case study is used in which we detail how adaptations are defined, expressed, and composed in each paradigm. Based on the case study, we uncover similarities and differences of each implementation, and derive a set of characteristics that identify each of them. The experiment shows several overlapping similarities between the two implementations, which is an indicator that there is a similar core set of characteristics for each paradigm. This finding brings the two seemingly disjoint research directions together, and can stimulate future research both in the direction of merging features and context as well as to improve the characteristic strengths of each paradigm.**

***Keywords*-feature-oriented programming; context-oriented programming; language paradigms**

I. Introduction

Software variability is an important factor in design and implementation of programs. Software programs are often developed for high customizability, for example to provide individual variants for particular clients. The implementation of such programs consists of core behavior and of different adaptions that add or modify the functionality. Program variability can be realized by using language level abstractions as introduced by different paradigms tailored to express program adaptations. Two such paradigms are feature-oriented programming (FOP) [1] and context-oriented programming (COP) [2].

The FOP paradigm is concerned with identifying functionality in the form of features. A feature is a stakeholder-relevant functionality [3] that can be implemented coarsely as a module or fine-granular as different lines of code scattered over the source code [4]. In FOP, adaptations are provided by features that can be expressed in several ways, for example by annotating the core program, or by defining adaptations as refinements. To yield different program variants, features are composed with the core program. Normally, feature composition is done statically at compile time, but recent approaches also offer runtime composition [5].

The COP paradigm is concerned with runtime behavior modifications in order to provide functionality that is adapted with respect to the execution environment of a program. In most COP implementations, adaptations are defined as first-class entities, to which context-dependent behavior is associated in a modular fashion. Adaptations are dynamically activated and deactivated at runtime to provide and undo context-dependent behavior [2].

Our objective is to identify the similarities and differences for realizing variability in these two paradigms. To this end, we use the expression product line (EPL) case study, providing an example implementation in each paradigm. For FOP we use rbFeatures, a versatile extension of the Ruby programming language that introduces features as first class entities [6][5][7]. For COP we use Subjective-C [8], a COP implementation for mobile devices that is based on the Objective-C programming language. From a comparison in the expression and implementation of the variability concerns of the EPL case study, we derive a set of characteristics that describe how each paradigm introduces variability.

A clear identification of the core characteristics between the two paradigms is a first result to help in forming a joined research for implementing variability. As we will see, the overlapping set of characteristics is an indicator that the FOP and COP paradigms could be brought together as a hybrid language for software variability.

The paper is organized as follows. We provide background to FOP and COP in Section II. Then in Section III, we provide a side-by-side comparison between the FOP (using rbFeatures) and COP (using Subjective-C) implementations of the case study. We compare both implementations with the help of the Expression Product Line (EPL) case study. Based on the case study, we discuss the similarities and differences of both implementations in Section IV. Sections V and VI respectively present the related work, and the conclusion and future work.

II. Background

This section introduces the feature-oriented programming and context-oriented programming paradigms.

A. Feature-Oriented Programming

The concept of features initially emerged with the goal to express distinct functionality that is targeted towards a specific stakeholder [3]. This notion of a feature is called conceptual [6], because it only regards the end-user visible behavior, but not its implementation. How to implement such conceptual features is considered in the feature-oriented programming paradigm. Basically, a program consists of different artifacts that provide

 ISBN: 978-1-61208-165-6

the program's functionality. Features encompass different parts of these artifacts, and are therefore distinguished into coarse-grained and fine-grained features [4]. Coarse-grained features can be represented with conventional mechanisms provided by a programming language, such as modules and packages. These can then be composed conveniently with the program's core behavior. Fine-grained features are more difficult to represent and compose, because they can consist of individual classes, methods, or even parts of method bodies. Related work shows a diversity of FOP implementation approaches [9]. Each approach differentiates how features are represented, expressed, and composed. We distinguish these approaches as follows:

- *Annotations* – These approaches use the existing program source code and mark the occurrences of feature-related source code. One type of annotations are *source code annotations* such as "#ifdef" statements in C++, which are native preprocessor directives. Before the program gets compiled, all parts of the source code that do not belong to the current feature configuration are pruned. Then, a program variant is created by compiling the remaining source code [10]. Another option is to use *virtual annotations*. In this case, the source code itself is not annotated, but a suitable intermediate program representation, such as the abstract syntax tree [10]. This approach requires tool support for representing the annotations and for generating a program variant.
- *Modules* – These approaches use the programming language modularization concepts to represent features. Among these approaches are traits in Scala [11], atoms and units in Jiazzi [11], Classboxes [12], CaesarJ [13], and Object Teams/Java [14]. The capabilities of modules constraint the level to which especially fine-grained features can be represented and composed.
- *Refinements* – These approaches separate a program into a fixed base program and extensions that are called refinements. Refinements are added to a program, where they change the behavior and the structure. Typically, these approaches add specific language constructs to express these refinements. Some approaches as the AHEAD tool suite [15], for example, use the keyword `refine` as a language construct, other approaches introduce concepts similar to refinements, such as aspects from aspect-oriented programming [16].

We use rbFeatures [6][5][7] as the FOP example language. rbFeatures is a versatile, pure language of Ruby, that allows features to be defined as first-class entities, giving a close integration of features and other application code. In order to express which part of the source code belongs to a feature, semantic annotations, called feature containments in rbFeatures, are used. Containments consist of a condition and a body. A containment condition is a logical expression determining which features need to be active or inactive in order for the body to be included in the program. The containment body is any piece of code: modules, classes, methods, and even individual lines and characters. rbFeatures allows to express the hierarchy and constraints of features with an expressive rule language. A program that is feature-refactored with rbFeatures allows both runtime and compile-time composition. At runtime, features can be activated and deactivated to immediately affect the program behavior, even allowing different variants of a program to exist at runtime [5]. At compile-time, the semantic annotations can be preprocessed to derive a static variant. This is done by pruning source code not define within the configured containments.

B. Context-Oriented Programming

Context-oriented programming paradigm [2] allows software systems to be modularized into behavioral adaptations that can be activated, deactivated and composed at runtime. Adaptations are triggered by changing properties of the execution environment, such as device presence, battery level, or user settings. COP languages typically provide dedicated constructs for the definition of behavior adaptations in a modularized fashion, as well for the composition and execution of such adaptations [2][17]. Program entities in which adaptations are defined are called *layers* [2] or *contexts* [17], which are normally defined as first-class entities of the program. We will refer to them as contexts.

Contexts may specify either behavioral or structural adaptations. The former case focuses in modifying functionality of the program with more suited behavior to particular situations of the execution environment. The later case concerns with providing new entities or adapting existing entities in the program for a particular situation.

Behavioral adaptations are the key concepts of COP. Adaptations rely in the dynamic activation and deactivation of context entities. When a context is activated, its associated behavioral adaptations become available in the current scope of the application. Similarly, whenever contexts are deactivated the behavior adaptations become unavailable to the execution environment, and the observed behavior of the program is restored to its former state. Behavioral adaptations can be associated with more than one context, in such a case, a new context entity is created implicitly, representing the combination of the contexts, to which the adaptation is associated [18]. Combined contexts are made available if and only if all of its components are active.

If not dealt with careful, dynamic activation and deactivation of contexts may lead to unexpected or inconsistent behavior. To manage such situations it is possible to define different dependency relations among contexts [8][19]. Constraints imposed by the dependency relations are verified at runtime when a particular context is to be activated or deactivated.

Contexts are stateful objects, state of variables and objects defined within a context are always preserved between context activations [2][8][20]. Moreover, within a context it is possible to extend the definition of objects already existing in a program by dynamically adding state properties to them. As with behavioral adaptations, these structural adaptations also become available and unavailable as the context in which they are defined is respectively activated or deactivated.

In the remainder of this paper we use Subjective-C [8] a full context-oriented language extension of Objective-C whose design is influenced by the Ambience language [18]. Contexts are defined as first-class entities. Context-dependent behavior adaptations need to be defined as methods in a class. These methods are annotated with the name of the context they belong to. Context-dependent behavior is not accessible by the core program until the context to which they are associated is activated. When a context is activated, a method replacement mechanism (from Objective-C's meta-object protocol) replaces original methods with their context version at runtime. Subjective-C uses a *context manager* to maintain a record

 ISBN: 978-1-61208-165-6

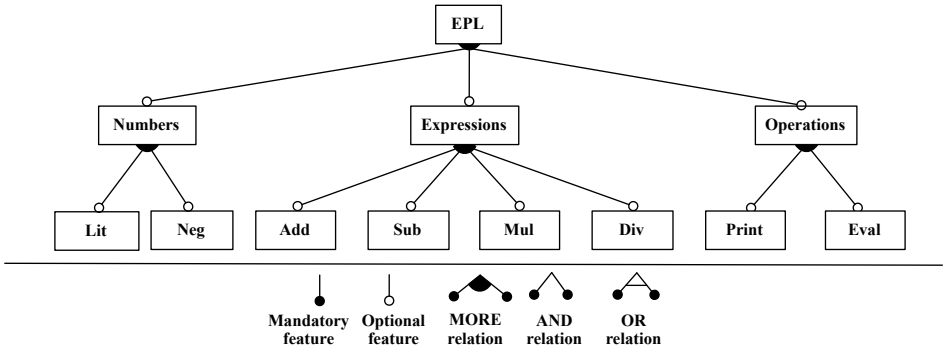

Figure 1: Feature diagram of the Expression Product Line.

of all context objects at runtime, whether they are active or not, and the dependency relations between contexts.

III. Case Study: Expression Product Line

The expression product line (EPL) [11] is a well known case study concerned with finding suitable modularization concepts for representing different types of integers, expressions, and operations over them. All possible program variations of the EPL are shown in Figure 1 – this notation is called a feature diagram, as it depicts the constraints between different features of a program [21]. For the EPL all its features are optional, meaning that they can be build in any combination. We conduct a side-by-side implementation of the case study providing first the rbFeatures example, and subsequently the Subjective-C one. The two implementations are compared based on the principal techniques each uses to realize software adaptations, specifically we are concerned with: (a) The way in which adaptations are declared, (b) The way in which adaptation's behavior is declared, and (c) The way in which modification to the core program is done.

A. Adaptation Declaration

The implementation of the product line starts with its top down definition. As originally expressed in the case study, expressions like Add and Neg, as well as the operations for Print and Eval are defined as features, shown in the following snippet for rbFeatures.

```
class Add
  is Feature
end

class Print
  is Feature
end
```

In Subjective-C, Lit, Add, Neg and the other expression elements are defined as regular (behavior-less) objects, taking advantage of the polymorphic abilities of the language. The Print and Eval operations are defined as contexts providing behavior for expression objects. Operations are declared as named context objects and added to the *context manager*.

```
@interface Add : Exp {
    Exp *left, *right;
}
@end

SCContext* Print = [[SCGlobalContext alloc]
     initWithName:@"print"];
[[SCContextManager sharedContextManager] addContext:
     Print];
```

B. Behavioral Declaration

Once all adaptations have been defined, the next step is to define the specific behavior added by the features to other objects of the program. We consider enhancing `Add` expressions with the printing behavior provided by the Print adaptation.

In rbFeatures, adaptations are introduced by forming feature containments around a pice of feature-specific code, which can be for example a method declaration. In the following example, the containment condition is the Print adaptation, and the containment body is the method declaration. When the Print adaptation is activated, a call to the `print` method will behave as shown in the snippet, otherwise the method will return an error message.

```
class Add
  Print.code do
    def print
      Kernel.print(@left.print + " + " + @right.print)
    end
  end
end
```

In Subjective-C, behavioral adaptations are also introduced by adding context-dependent methods within the body of the object that defines it. This is shown in the following snippet.

```
@implementation Add {
@contexts Print
 - (NSString) print {
     return [NSString stringWithFormat:@"%@ + %@", [left
          print], [right print]];
  }
 @end
}
```

Unlike rbFeatures, in Subjective-C it is not possible to define specific lines within a method as context-dependent. However, this is possible in other COP languages [2][18].

C. Behavioral Modification

Behavior defined for the different EPL expressions and operations is available to the program through the explicit activation of the related feature. For example, in order to have the PRINT

Entity Representation	Annotations	Modules	Refinements	First-Class Entities
Adaptation Constraints	Hierarchy		Rules	
Adaptation Trigger	Internal		External	
Adaptation Activation	Compile-Time		Runtime	
Composition Process	Order-Independent	Order-Dependent	Blocking	Non-Blocking
Adaptation Properties	Stateful	Extensible	Cascading	

Figure 2: Morphologic scheme of all implementation characteristics.

adaptation, in rbFeatures a call to the `Print.activate` method must be made to activate the adaptation. In Subjective-C the `@activate(Print)` keyword is used to process the context activation.

However, there is a difference in the processing of the two activation messages. In rbFeatures the source code enclosed by the feature definition is re-executed, that is, with every activation the code gets redefined and because of changed containment conditions, new behavior is eventually added to the program. Subjective-C, on the other hand, does not re-execute any code. Instead, activation of a context allows its associated methods, variables, objects, and so on, to be visible by the method dispatcher. This is the main reason context-dependent variables are stateful. Whichever the state of a variable is, it remains untouched as long as the context in which the variable is defined is inactive, since it cannot be found by the program.

IV. Comparison of Feature-Oriented Programming and Context-Oriented Programming

In this section, the similarities and differences encountered between our FOP and COP implementations are made explicit. Then we define them as specific characteristics of the implemented paradigm.

We summarize the observed similarities as follows:

- Features and contexts are declared as *first-class entities* of the program.
- Features and contexts add adaptions on top of the core behavior by *annotating source code* at the place where adaptations would normally be defined in.
- Features and contexts can both be *activated and deactivated at runtime*, immediately changing the program behavior.
- Both implementations offer a runtime representation of the *dependencies between adaptions*.

The differences are the following:

- There is *no automatic adaptation* of the dependent features in rbFeatures, while Subjective-C uses the dependency relations defined between contexts to automatically activate or deactivate related contexts.
- Feature activation is *externally triggered* by the user in rbFeatures, while Subjective-C uses *internal triggers* based on the program state to activate contexts.
- There is *no stateful* composition of adaptations in rbFeatures: while instances of objects with feature-dependent behavior retain their state, class variables will be overridden during the program adaptation. Subjective-C uses a *stateful* representation of contexts. Variables declared in a context cannot be accessed or modified unless the context that defines them is available. Maintaining there state between activations.
- Features can be composed at *compile-time* and *runtime* in rbFeatures, while Subjective-C only offers runtime composition.

We use this comparison and related work as the frame of reference for FOP and COP to define a set of characteristics that identify our implementations of each paradigm. These characteristics, also illustrated in Figure 2, are the following ones:

- Entity Representation [Annotations, Modules, Refinements, First-Class Entities] – Specifies how adaptations are represented. On the one hand are pure annotations that are external to the program and receive their meaning as contexts or features from the processing tool. On the other hand we see first-class entities that are high-level abstractions and can be fully integrated with the program.
- Adaptation Constraints [Hierarchy, Rules] – The availability of composition constraints, for example in the form of a hierarchy (a hierarchically higher adaptation is only available if its children are) or rules (arbitrary expressions that state which adaptions need to be active or inactive for a particular adaption to be composed with the program).
- Adaptation Trigger [Internal, External] – The adaptation process is triggered by an internal signal, like a certain program state upon which it reacts, or by an external signal, for example a change in environment that is detected by a sensor or through a command by the user.
- Adaptation Activation [Compile-time, Runtime] – The adaptation can occur statically at runtime, usually loosing the information about the adaptation and producing a program with fixed behavior, or fully dynamically at runtime.
- Composition Process [Order-Dependent, Order-Independent, Non-Blocking, Blocking] – An important difference in the adaptation process is whether the activation order influences the adaptation result, for example when adaptations provide different composition of source code pieces. Furthermore, the adaptation process can block activation of other adaptations during the composition.
- Adaptation Properties [Stateful, Extensible, Cascading] – In a stateful adaptation, defined objects and variables retain their states between deactivations and activations.

 ISBN: 978-1-61208-165-6

Extensible means to modify existing adaptations or to add new ones at the program runtime. Finally, cascading denotes the capability that if an adaptation needs to be added or removed from the program, all dependent adaptations are automatically removed or added.

In terms of these characteristics, we can identify our implementations as shown in Figure 3. As we see, there are 6 common characteristics shared between the implementations, and 6 unique ones. Judging from this representation, the main difference between features and contexts is the availability of compile-time composition of program and the availability of stateful, cascading adaptations.

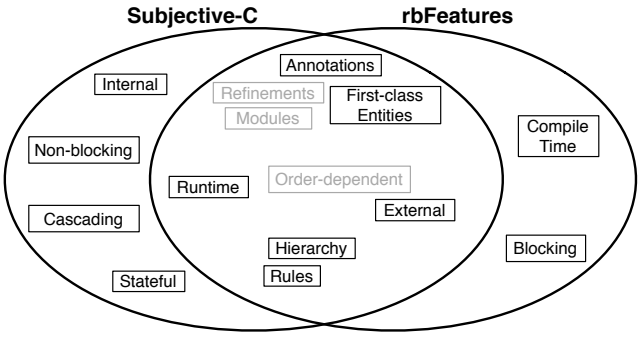

Figure 3: rbFeatures and Subjective-C characteristics.

V. Related Work

To the best of our knowledge, a structured comparison of COP and FOP paradigms as proposed in this paper has not been done. However, several COP ideas are used to build FOP programs and vice versa. We discuss such proposals here.

A first close relation can be seen from the concept of superimposition, which is the process of merging software artifacts by merging their substructures [22]. This mechanism lies at the heart of introducing adaptations of programs, and it is used by several implementations for feature-oriented programming and context-oriented programming.

Context-oriented languages borrow several concepts of feature-oriented programming, at both the implementation and design level. The ContextL [2] COP language uses the concept of mixin-layers [23] normally used as an implementation technique for FOP. Specifically, ContextL uses layers as the main abstraction to define adaptations [24]. Based on the need to express dependencies between layers, and to better control their interaction, a Feature Description Language (FDL) was introduced in ContextL [25] to automatically enforce dependencies between layers.

Additionally, an extension of feature-oriented domain analysis has been used for the design of context-oriented systems, namely Context-Oriented Domain Analysis (CODA) [19]. In this approach, feature diagrams are extended to express resolution strategies whenever there are multiple adaptations available that provide behavior for the same functionality. The CODA approach also introduces *inclusion* and *exclusion* relations between adaptations. The former relation expresses that if an adaptation can be activated all included adaptations are also activated. The later one, expresses that if an adaptation can be activated, all its excluded adaptations are deactivated.

VI. Conclusion and Future Work

This paper shows how feature-oriented programming and context-oriented programming paradigms provide closely related strategies for realizing software variability. To understand the differences and similarities between the two paradigms, we implemented a common case study with an FOP language (rbFeatures) and a COP language (Subjective-C). Based on analyzing how behavioral adaptations are expressed and implemented, we derived a set of characteristic properties constituting each paradigm. We found that six characteristics are common in both paradigms, and six a are different. In essence, the difference lies in the availability of compile time and/or runtime adaptations and in the stateful transition of the program's behavior.

This contribution helps to clarify the commonalities of the two seemingly disjoint research directions, and can help to stimulate research both towards the merging of features and contexts, as well as to improve the characteristic strength of each paradigm.

In future work, the next step is to extend this study with an in-depth analysis of other FOP and COP languages. We wish to further refine the characteristics, and based on it, it would be possible to think about how FOP and COP can be merged in hybrid languages for variability, for example, by adding stateful representation of features or to add compile-time composition to COP implementations that restrict the amount of runtime contexts deployed in devices.

Acknowledgements

This work has been supported by the ICT Impulse Programme of the Brussels Institute for Research and Innovation, and by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy. We thank the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds., vol. 1241. Berlin, Heidelberg, Germany: Springer-Verlag, 1997, pp. 419–443.

[2] P. Costanza and R. Hirschfeld, "Language Constructs for Context-Oriented Programming: An Overview of ContextL," in *Proceedings of the 1st Symposium on Dynamic Languages*. New York, USA: ACM, 2005, pp. 1–10.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, USA, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[4] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. New York: ACM, 2008, pp. 311–320.

 ISBN: 978-1-61208-165-6

[5] S. Günther and S. Sunkle, "Dynamically Adaptable Software Product Lines using Ruby Metaprogramming," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*. New York: ACM, 2010, pp. 80–87.

[6] S. Günther and S. Sunkle, "Feature-Oriented Programming with Ruby," in *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*. New York: ACM, 2009, pp. 11–18.

[7] S. Günther and S. Sunkle, "rbFeatures: Feature-Oriented Programming with Ruby," in *Science of Computer Programming*. Elsevier, 2011, accepted 01.01.2011, in press.

[8] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux, "Subjective-c: Bringing context to mobile platform programming," in *Proceedings of the International Conference on Proceedings of the International Conference on Software Language Engineering*, ser. series-lncs, B. Malloy, S. Staab, and M. van den Brand, Eds., vol. 6563. Eindhoven: Springer, 2011, pp. 246 – 265.

[9] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, 2009.

[10] C. Kästner and S. Apel, "Virtual Separation of Concerns – A Second Chance for Preprocessors," *Journal of Object Technology (JOT)*, vol. 8, no. 6, pp. 59–78, Sep. 2009.

[11] R. E. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Techniques," in *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Berlin, Heidelberg, Germany: Springer-Verlag, 2005, pp. 169–194.

[12] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts, "Classboxes: Controlling Visibility of Class Extensions," *Computer Languages, Systems & Structures*, vol. 31, no. 3, pp. 107–126, 2005.

[13] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An Overview of CaesarJ," in *Transactions on Aspect-Oriented Software Development I*, ser. Lecture Notes in Computer Science, A. Rashid and M. Aksit, Eds. Berlin, Heidelberg, Germany: Springer-Verlag, 2006, vol. 3880, pp. 135–173.

[14] S. Herrmann, "A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java," *Applied Ontology*, vol. 2, no. 2, pp. 181–207, 2007.

[15] D. Batory, "Feature-Oriented Programming and the AHEAD Tool Suite," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. Washington: IEEE Computer Society, 2004, pp. 702–703.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Berlin, Heidelberg, Germany: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[17] S. González Montesinos, "Programming in ambience: Gearing up for dynamic adaption to context," Ph.D. dissertation, Université Catholique de Louvain, October 2008.

[18] S. González, K. Mens, and A. Cádiz, "Context-oriented programming with the ambient object system," *Journal of Universal Computer Science*, vol. 14, no. 20, pp. 3307–3332, 2008.

[19] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt, "Context-Oriented Domain Analysis," in *Modeling and Using Context, Sixth International and Interdisciplinary Conference on Modeling and Using Context*, August 2007, pp. 178–191.

[20] S. González, K. Mens, and P. Heymans, "Highly Dynamic Behaviour Adaptability through Prototypes with Subjective Multimethods," in *Proceedings of the 2007 symposium on Dynamic Languages (DLS)*, ser. DLS '07. New York, NY, USA: ACM, 2007, pp. 77–88.

[21] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston, San Franciso et al.: Addison-Wesley, 2000.

[22] S. Apel and C. Lengauer, "Superimposition: A Language-Independent Approach to Software Composition," in *Software Composition*, ser. Lecture Notes in Computer Science, C. Pautasso and E. Tanter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, vol. 4954, pp. 20–35.

[23] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 215–255, 2002.

[24] B. Desmet, J. Vallejos, and P. Costanza, "Introducing Mixin Layers to Support the Development of Context-Aware Systems," in *3rd European Workshop on Aspects in Software (EWAS)*, ser. Technical Report IAI-TR-2006-6, G. Kneisel, Ed. Universität Bonn, 2006, pp. 23–30.

[25] P. Costanza and T. D'Hondt, "Feature Descriptions for Context-Oriented Programming," in *Proceedings 12th International Conference for Software Product Lines (SPLCL), 2nd International Workshop on Dynamic Software Product Lines (DSPL)*, S. Thiel and K. Pohl, Eds., vol. 2 (Workshops). Ireland: University of Limerick, 2008, pp. 9–14.

 ISBN: 978-1-61208-165-6