Dynamic Visualisation of Features and Contexts for Context-Oriented Programmers

Benoît Duhoux
Université catholique de Louvain
Louvain-la-Neuve, Belgium
benoit.duhoux@uclouvain.be

Bruno Dumas
Université de Namur
Namur, Belgium
bruno.dumas@unamur.be

Hoo Sing Leung
Université catholique de Louvain
Louvain-la-Neuve, Belgium
hoo.leung@student.uclouvain.be

Kim Mens
Université catholique de Louvain
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

ABSTRACT

Context-oriented programming languages allow programmers to develop context-aware systems that can adapt their behaviour dynamically upon changing contexts. Due to the highly dynamic nature of such systems and the many possible combinations of contexts to which such systems may adapt, developing such systems is hard. Feature-based context-oriented programming helps tackle part of this complexity by modelling the possible contexts, and the different behavioural adaptations they can trigger, as separate feature models. Tools can also help developers address the underlying complexity of this approach. This paper presents a visualisation tool that is intricately related to the underlying architecture of a feature-based context-oriented programming language, and the context and feature models it uses. The visualisation confronts two hierarchical models (a context model and a feature model) and highlights the dependencies between them. An initial user study of the visualisation tool is performed to assess its usefulness and usability.

CCS Concepts

•Human-centered computing → Empirical studies in visualization; •Software and its engineering → Integrated and visual development environments; *Abstraction, modeling and modularity; Object oriented languages;*

Author Keywords

Software visualisation; context-oriented programming; feature-oriented software development; dynamic software adaptation

EICS '19 June 18–21, 2019, Valencia, Spain

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6745-5/19/06. . . $15.00

DOI: https://doi.org/10.1145/3319499.3328240

INTRODUCTION

Having information about the surrounding environment and conditions in which a software system operates, enables the creation of systems that can adapt their behaviour dynamically to changing contexts. This information can take the form of user preferences (a user's age, habits, (dis)abilities), information from external services (weather conditions), or internal data about the device on which the system runs (remaining battery level or other sensor information). We refer to any such information as 'contexts' and to systems that adapt their behaviour dynamically to such contexts as 'context-aware systems' [1].

Developing such systems is not straightforward, due to the exponential combination of contexts, their possible behavioural variations, and the high dynamicity of such systems. To build such systems, 'context-oriented programming' [12, 24] proposes dedicated programming languages and abstractions, to implement context-specific behavioural adaptations that can temporarily adapt existing system functionality upon the (de)activation of certain contexts. The notion of context has also been explored in the field of software modelling, and feature modelling and software product lines in particular [7, 11, 14, 4, 3, 21].

Recent work on context-oriented programming proposes a runtime software architecture [22] where contexts and features are handled by separate architectural layers, with explicit dependencies from one layer to the next. The selection and activation of contexts in earlier layers can trigger the selection and activation of their corresponding features.

Nevertheless, keeping track of all possible contexts, features and their intra- and inter-dependencies remains a daunting task for developers of context-oriented systems. It is not easy for a developer to know what contexts or features are available, are currently active, what the impact of activating or deactivating them is, or whether the system exhibits the intended behaviour in a particular situation.

We therefore developed a visualisation tool that can help developers keep an overview of all existing contexts and features, by displaying the context and feature models and their dependencies. For that, we propose a visual strategy to examine how a hierarchical model (the context model) can statically or dynamically interact with another hierarchical model (the feature model). The tool offers more than a mere static visualisation of the context and feature models. It also depicts dynamically at runtime what context and features are active or get activated, and what program code this affects.

We conducted an initial user study with master-level students in software engineering to assess whether the current version of the visualisation tool helps understanding the programming paradigm and programs written in it. We can conclude that our tool helps the comprehension of the underlying approach despite its complexity for programmers developping context-oriented applications.

The remainder of this paper is structured as follows. Section 2 introduces the case study of a context-oriented system that will be used as running example throughout this paper. Section 3 then introduces the feature-based context-oriented programming approach upon which we build. Our visualisation tool is presented in Section 4. Section 5 discusses the initial user study we conducted and discusses its results. Related work is exposed in Section 6. Section 7 concludes the paper and presents avenues of future work.

CASE STUDY

Before introducing our visualisation tool and the underlying context-oriented software architecture upon which it relies, in this section we briefly describe the case study that will serve as running example throughout the paper. Rather than reinventing our own, we decided to revisit a case study introduced by Duhoux et al. [9], who implemented a context-oriented version of a 'risk information system'.

The system provides instructions to citizens on what to do in case of certain emergency situations or risks, like earthquakes or floods. The actual instructions given to a citizen may depend on a variety of contexts, such as the user's age, location or vicinity, weather conditions, or the status of an emergency (the emergency has been announced but has not yet affected the user, the emergency is actually occurring, or being in the aftermath of an emergency situation).

The instructions issued by the context-aware risk information system can be either static or dynamic. Static instructions are just instructions that a user can consult about what to do in case of certain risks [1]. The system can also display information and characteristics about actual emergencies as they occur. For example, when an earthquake is detected, its severity would be computed on the Richter scale and its location shown to citizens as a circular impact zone determined by its epicentre and its radius.

When an actual emergency is observed, the authorities will actively issue instructions specific to the emergency at hand, and specific to the current situation and user profile (e.g. the user's age, the current weather, the status of the emergency). For instance, if an earthquake warning has been issued, and a citizen is stuck at home, an adult may get a specific instruction to "Hide under a table, desk, bed or any other sturdy piece of furniture", while a child may just see a pictogram representing this specific instruction instead.

[1]https://www.risico-info.be/en/hazards/naturals-hazards/floods, for example.

FEATURE-BASED CONTEXT-ORIENTED APPROACH

Our feature-based context-oriented approach is built on top of the context-oriented software architecture of Mens et al. [22] and the multiple-product-line-feature modelling approach of Hartmann and Trew [11]. Fig. 1 depicts an overview of our approach. Context-oriented programming helps programmers build context-oriented systems. In Mens et al.'s framework [22], based on contextual information sensed from the surrounding environment, contexts are selected and activated (CONTEXT HANDLING). Appropriate features corresponding to these activated contexts are then selected and activated (FEATURE HANDLING) and deployed in the system (CODE ADAPTATION), to make the system's behaviour more specific to the context of use. Combining this framework with the work of Hartmann and Trew [11], contexts and features explicitly represent in terms of runtime features models. This overall approach has been shown to be powerful, however due to its intrinsic dynamicity, tools are needed to let developers understand the inter-dependencies and influences between contexts and features.

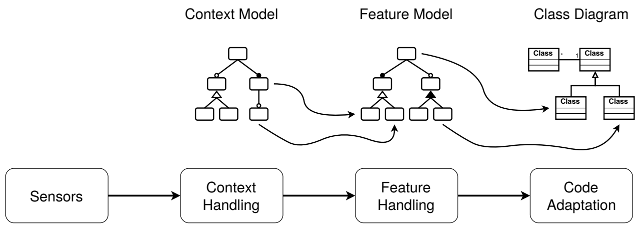
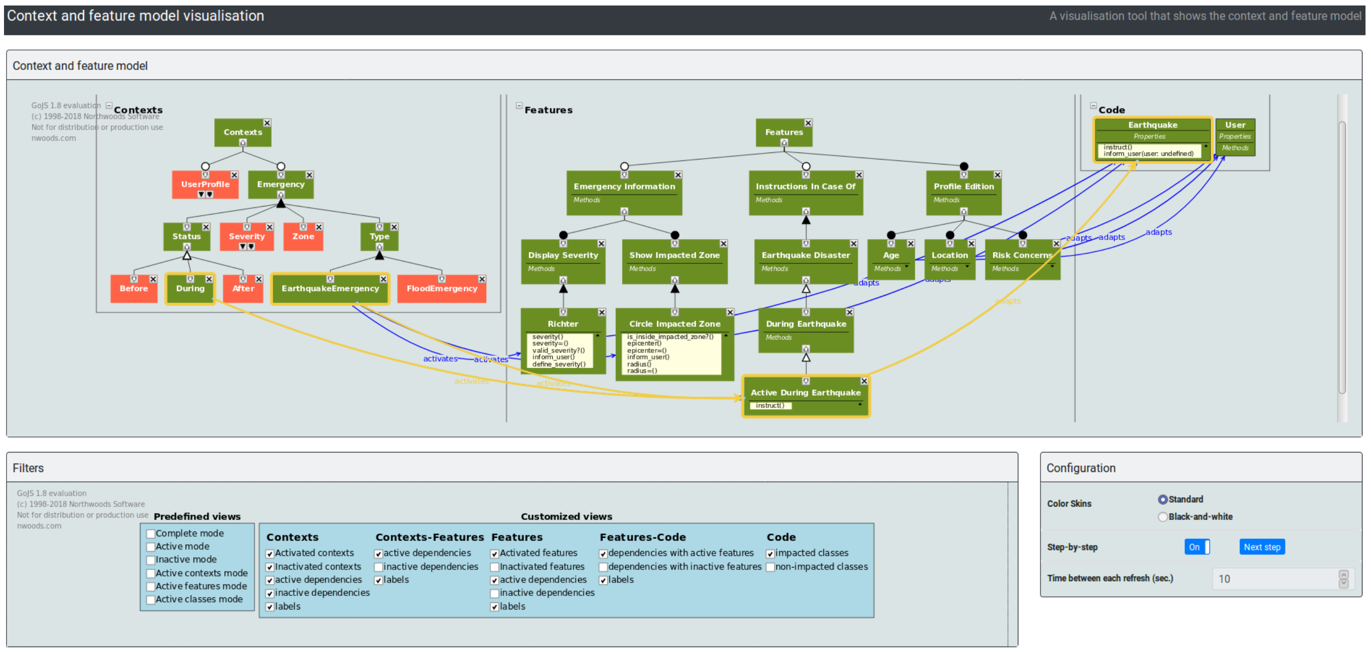


Figure 1: Overview of our feature-oriented context-aware programming approach.

VISUALISATION TOOL

Our visualisation tool, depicted in Fig. 2, lets users inspect the context and feature models with their intra- and inter-dependencies, as well as the classes of the system that get affected by the selected features. The visualisation is partitioned in different parts, corresponding to the different layers of the underlying implementation architecture (Fig. 1). In the tool snapshot shown in Fig. 2, an earthquake emergency is currently occurring, so the system must inform citizens about the characteristics of the ongoing earthquake and the actions they need to take to protect themselves.

In this section, we focus on the different design requirements we followed when creating this tool, according to different usage scenarios focused on a programmer building a context-oriented system using this approach. All these functionalities are also illustrated in a demonstration video provided as supplementary source material.



Predefined views
- [ ] Complete mode
- [ ] Active mode
- [ ] Inactive mode
- [ ] Active contexts mode
- [ ] Active features mode
- [ ] Active classes mode

Customized views

Contexts	Contexts-Features	Features	Features-Code	Code
✓ Activated contexts	✓ active dependencies	✓ Activated features	✓ dependencies with active features	✓ impacted classes
✓ Inactivated contexts	☐ inactive dependencies	☐ Inactivated features	☐ dependencies with inactive features	☐ non-impacted classes
✓ active dependencies	✓ labels	✓ active dependencies	✓ labels	
✓ inactive dependencies		☐ inactive dependencies		
✓ labels		✓ labels		

Color Skins: [✓ Standard] Black-and-white

Step-by-step: On Next step

Time between each refresh (sec.): 10

Figure 2: Snapshot of our visualisation tool applied to the risk information system. Three widgets compose this tool: *Context and feature model*, *Filters* and *Configuration*. Red (resp. green) rectangles represent inactive (resp. active) contexts, features or classes in the *Context and feature model* widget. To keep the picture readable, we deliberately hid some information like for example all child contexts of the `UserProfile` context, all inactive features and all non-impacted classes.

Static visualisation of the context and feature models

A first important usage scenario for a programmer is to get a global overview of the system, in terms of the different contexts, features and classes of which it is composed. Fig. 2 shows what such a visualisation looks like in the tool's *Context and feature model* widget. This static snapshot can show all contexts, features and classes of interest, regardless of whether they are currently active or not: the context model shown in Fig. 2 contains both active contexts (colored in green) and inactive ones (colored in red). The idea of including classes in the visualisation as well is inspired by the *Feature Visualiser* of Duhoux et al. [9]. Furthermore, our tool allows a programmer to inspect in more detail the actual behaviour of features and classes, as illustrated by yellow boxes inside some features (e.g. `Richter`) and classes (e.g. `Earthquake`) in Fig. 2.

Exploring the dynamics of a context-oriented system

In addition to providing a static overview of a context-oriented system, the tool should support programmers in understanding and exploring the dynamic aspects of such a system. The tool should help them inspect what contexts and features are currently active and how that affects the behaviour, in terms of what classes are currently being adapted. For example, suppose that during testing and simulation of the system a programmer discovers that, when an earthquake warning is issued, the system starts displaying instructions related to a flood instead of an earthquake. To understand such undesired behaviour he needs to explore what contexts are currently active, what features were triggered in response to that, and how the classes were then adapted by those features. A possible cause of this bug may be for example a wrong dependency between the earthquake context and its corresponding features.

Filtering and predefined views

To help programmers manage the complexity of understanding big systems consisting of many different context and features, the tool should also come with a set of filters and predefined views that a programmer can select to focus on particular concerns, as depicted in the bottom left of Fig. 2. These filters (called 'Customized views' in the widget) allow a programmer to indicate whether he is more interested in the contexts, the features, the code, or the dependencies between them, and whether he is currently more interested in exploring the active or inactive entities or dependencies.

Highlighting specific elements of interest

As the number of possible contexts, features, classes and dependencies can become quite large, in addition to filtering the diagrams to only show certain elements of interest, *highlighting* is another interesting way to help programmers navigate through the diagrams, letting a developer trace the behaviour of a particular feature.

Hiding and collapsing information

Finally, a programmer should be able to customise his visualisation at an even more fine-grained level through the actions of *hiding* and *collapsing* particular elements. For example in Fig. 2, if the programmer wants to focus his interest on the `Emergency` context, he can hide the subtrees of the `UserProfile` context.

VALIDATION

In this section, we describe and analyse the user study we conducted to get an initial assessment of the usability and usefulness of our visualisation tool and underlying approach. The subjects of our study were 34 master-level students in computer science or engineering following a software engineering course. They were aged 20 to 27 years old and 4 of them were female. To evaluate the tool, we asked them to play the role of programmers working on a context-oriented system. To initiate them to the different technologies used in the project, they participated in two preparatory sessions before the actual user study, so as to train them in using the Ruby language and the approach. Since the study was carried out during a course, we made it clear to the students that this user study would not be related to the course evaluation and would be entirely dedicated to our research and performed anonymously so as to limit any potential bias. In the remainder of this section, we first describe the user study itself, and then present and discuss the results we gathered from the study.

User study

We performed the user study during a two-hour session where our students were asked to assess the usability and usefulness of our visualisation tool. They had to perform two tasks. These tasks aimed to extend the earthquake-specific variant of the risk information system with a new kind of risk and emergency: that of floods. *Task 1* concerned the characteristics of a flood. In this task, they had to implement the fine-grained features to manage and display the 'standard severity' feature and the 'polygon impacted zone' feature needed to represent a flood emergency. *Task 2* was about implementing the flood-specific instructions (either static or dynamic) that citizens must follow before, during or after a flood. We conducted the user study as follows. Students were first asked to report some information about themselves: their age, general background in programming, knowledge of object-oriented programming, context-oriented programming and so on. Then we split the programmers in two separate groups (A and B) to perform their assigned task during a 25-minute time slot. Group A had to start implementing *Task 1* whereas group B had to develop *Task 2*. During this first task, they were not allowed to use the visualisation tool. Afterwards they were asked to answer three questions about the task they performed, to verify if they understood well what was asked of them. Next, they received a quick introduction to the visualisation tool as a preparation for their second task. For this second task, we switched the tasks. Group A now had to develop *Task 2* while group B had to implement *Task 1*. Again, both groups received at most 25 minutes to finish their assigned task and then had to answer two new questions to verify their understanding of the task. Finally, all subjects were asked to answer some questions regarding how they perceived the usability and usefulness of our visualisation tool.

Results and discussion

In this section we analyse the results of this user study. Despite the complexity of our feature-oriented context-aware programming approach, the participants in our study seemed to agree that our visualisation tool is useful for developers when learning our approach or during debugging.

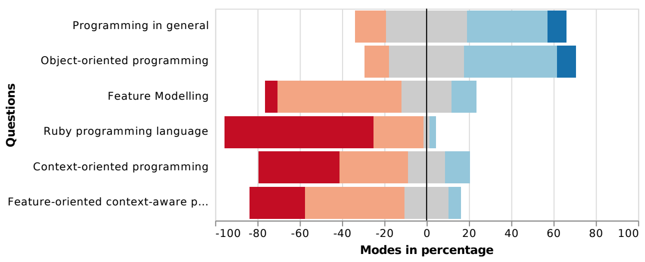


Figure 3: Divergent stacked bar diagram summarising the results of our closed questions about the background knowledge of our participants.

Fig. 3 illustrates the background knowledge of our participants at the beginning of our user study. We can observe that our students have quite a good knowledge of programming and object-oriented programming in particular. But they did not feel as comfortable with more dynamic programming technologies such as the context-oriented programming paradigm or our feature-oriented context-aware approach. Their weak knowledge of the Ruby programming language can be justified by the fact that only a few of the students had prior experience (beyond what they saw in a two-hour preparatory session) in Ruby. From these results we may infer that the preparatory phase of this user study was not sufficient for the complexity of this new context-oriented programming technology. To improve this preparation in the future, we should probably add additional courses and practical experience to help participants understand how to program systems using the feature-based context-oriented programming approach.

Nevertheless, despite the difficulty of our approach, our participants did seem to be interested by the visualisation tool when they must develop a context-oriented system. Fig. 4 depicts their opinions about the tool. The first two questions concern whether they believe the static or dynamic representation of the models and their dependencies are easy to understand. The five values ranged from *hard to understand* to *easy to understand*. For each representation, more or less 58% (taking into account only the positive values) of our participants considered the representation as understandable. For the question about which aspect (static or dynamic) is most interesting in this tool, 50% (considering only the positive values) of our participants consider the dynamic view as more interesting than the static view, as opposed to only 20% (computing only the negative values) who believed the static view to be more interesting. 30% liked the dynamic aspect as much as the static one. In addition, almost 56% of our participants agreed that our visualisation tool is helpful to learn the approach. The ease to use our tool is more mitigated however. Whereas almost 40% of the participants believed our tool to be easy to use, more or less 26% of them did not. Finally, in the open question about which functionality is most useful, several participants answered that the ability to replay changes dynamically using the *Next step* button is really useful.

We can conclude that our visualisation tool seems useful for developers of feature-based context-oriented systems. Users think that the representations of the models are understandable. In addition they consider that this tool helps in learning and understanding the underlying approach. But they do not have a clear opinion concerning the ease of use. However this can be explained by the high complexity of the underlying approach. Indeed, assessing the usability of a visualisation tool such as the one described in this paper is intrinsically linked to the understandability of the underlying programming approach. In the feedback received from our participants, two main requests may be noted: the addition of a *previous step* button to step back in the dynamic visualisation and better support for visualising larger context and feature models.

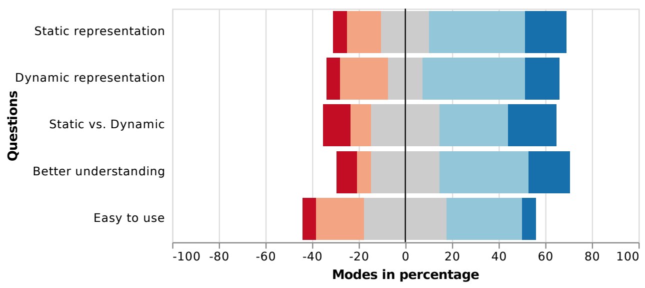


Figure 4: Divergent stacked bar diagram presenting the results of our closed questions about the usability and the usefulness of our visualisation tool.

RELATED WORK

In this section we briefly explore related work on the strong relation between contexts and features. Then we discuss other visualisation tools similar or related to ours.

Context versus feature

While contexts are characteristics of the surrounding environment in which a system runs [1], features can be defined as *"any prominent or distinctive user-visible aspect, quality, or characteristic of a software system"* [15]. Contexts and features are complementary notions when building context-oriented systems that can adapt their behaviour (described in terms of features) dynamically whenever changes (reified as contexts) are detected in the surrounding environment. Notwithstanding their complementarity and differences, their similarities have been observed in the modelling domain [7, 11, 14, 4, 3, 21] and the programming field [5, 23, 6, 22, 17].

Visualisation tools

Several visualisation tools have been created for visualising different aspects of context- and feature-oriented modelling or programming approaches. Many of these works state that such visual support is essential especially when trying to understand and manage large and complex feature or context models.

To visualise feature models, programmers can use a tool like *FeatureIDE* [16], which is an open-source visualisation framework integrated in the Eclipse development environment. For dealing with larger feature models, programmers may prefer to use *S.P.L.O.T.* [20], a web-based system that represents feature models in a much more compact tree-like structure. Illescas et al. [13] propose four different visualisations that focus on features and their interactions at source code level, and evaluate them with four case studies. Urli et al. [25] present a visual and interactive blueprint that enables software engineers to decompose a large feature model in many smaller ones while visualising the dependencies among them. Apel and Beyer [2] present a visual clustering tool that clusters program elements (like methods, fields and classes) based on the features they belong to, as a way to assess the cohesiveness of the features. Features whose elements form clusters are more cohesive than features whose elements are scattered across the layout.

Nieke et al. created a tool suite for integrating modelling in context-aware software product lines [19]. This tool helps developers to model the three dimensions (spatial, contextual and temporal) of the variabilities of such approaches.

Dedicated to the development of adaptative user interfaces, *Quill* is a web-based development approach in which several stakeholders can work together to create a cross-platform model-based design of a user interface [10].

Duhoux et al. built a *Feature Visualiser* tool on top of Mens et al.'s context-oriented software architecture [22] to visualise the interaction between the active contexts and features of the system [9]. In addition to that tool, Duhoux also developed a *COP simulator* [8] to simulate context-oriented systems implemented with this architecture.

CONCLUSION

Managing different models as well as their dependencies in a context-oriented approach is a daunting task, due to the potentially high number of contexts and features, as well as the high dynamicity of such systems. To address this issue, we suggest the use of two linked hierarchical models (illustrating the context model and feature model) with highlights of the dependencies between them. We created a dedicated visualisation tool based on this concept which also shows at runtime the dependencies from the feature model to the code (i.e., the classes of the system).

To assess the usefulness and usability of our approach, we conducted a user study on the visualisation tool with 34 master-level students in the context of a software engineering course. The participants of this study considered that our tool was easy to understand in terms of the different representations it provides. They felt in particular that the dynamic representation of the models helped them understand how the system adapted over time. However, the participants were less convinced when it came to usability of the tool. However, this can be explained by the fact that the tool is strongly linked to the (complex) underlying approach.

As future work we will first integrate the useful comments and feedback received from the participants in our study. To deal with the scalability problem of large models, we intend on relying on other visualisations such as for example a more compact tree-like view à la *S.P.L.O.T.* [20] or alternatively hyperbolic trees [18] or 3D representations [26].

ACKNOWLEDGMENTS

We are grateful to Jean Vanderdonckt for the many fruitful discussions on this topic.

REFERENCES

1. Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. 1999. Towards a Better Understanding of Context and Context-Awareness. In *Handheld and Ubiquitous Computing*. Springer, 304–307.
2. Sven Apel and Dirk Beyer. 2011. Feature Cohesion in Software Product Lines: An Exploratory Study. In *Proc. of ICSE 11*. ACM, 421–430.
3. Rafael Capilla, Mike Hinchey, and Francisco J. Díaz. 2015. Collaborative Context Features for Critical Systems. In *Proc. of VaMoS 15*. ACM, Article 43, 8 pages.
4. Rafael Capilla, Óscar Ortiz, and Mike Hinchey. 2014. Context Variability for Context-Aware Systems. *Computer* 47, 2 (Feb. 2014), 85–87.
5. Nicolás Cardozo, Sebastian Günther, Theo D'Hondt, and Kim Mens. 2011. Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations. In *Proc. of ICSEA 11*. IARIA, 130–135.
6. Nicolás Cardozo, Kim Mens, Pierre-Yves Orban, Sebastián González, and Wolfgang De Meuter. 2014. Features on Demand. In *Proc. of VaMoS 14*. ACM, Article 18, 8 pages.
7. Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. 2007. Context-Oriented Domain Analysis. In *Modeling and Using Context*. Springer, 178–191.
8. Benoît Duhoux. 2016. *L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte*. Master's thesis. UCLouvain, Belgium.
9. Benoît Duhoux, Kim Mens, and Bruno Dumas. 2018. Feature Visualiser: An Inspection Tool for Context-Oriented Programmers. In *Proc. of COP 18*. ACM, 15–22.
10. Vivian Genaro Motti, Dave Raggett, Sascha Van Cauwelaert, and Jean Vanderdonckt. 2013. Simplifying the Development of Cross-platform Web User Interfaces by Collaborative Model-based Design. In *Proc. of SIGDOC 13*. ACM, 55–64.
11. Herman Hartmann and Tim Trew. 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Proc. of SPLC 08*. IEEE, 12–21.
12. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *JOT* 7, 3 (2008), 125–151.
13. Sheny Illescas, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Towards Visualization of Feature Interactions in Software Product Lines. In *Proc. of VISSOFT 16*. IEEE, 46–50.
14. Zakwan Jaroucheh, Xiaodong Liu, and Sally Smith. 2010. Mapping Features to Context Information: Supporting Context Variability for Context-Aware Pervasive Applications. In *Proc. of WIIAT 10*, Vol. 1. 611–614.
15. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. CMU.
16. Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-oriented Software Development. In *Proc. of ICSE 09*. IEEE, 611–614.
17. Alexandre Kühn. 2017. *Reconciling Context-Oriented Programming and Feature Modeling*. Master's thesis. UCLouvain, Belgium.
18. John Lamping, Ramana Rao, and Peter Pirolli. 1995. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proc. of CHI 95*. ACM, 401–408.
19. Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *Proc. of VaMoS 16*. ACM, 41–48.
20. Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. of OOPSLA 09*. ACM, 761–762.
21. Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. 2017. Modeling and Managing Context-Aware Systems' Variability. *IEEE Software* 34, 6 (Nov. 2017), 58–63.
22. Kim Mens, Nicolás Cardozo, and Benoît Duhoux. 2016. A Context-Oriented Software Architecture. In *Proc. of COP 16*. ACM, 7–12.
23. Thibault Poncelet and Loïc Vigneron. 2012. *The Phenomenal Gem: Putting Features as a Service on Rails*. Master's thesis. UCLouvain, Belgium.
24. Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *JSS* 85, 8 (Aug. 2012), 1801 – 1817.
25. Simon Urli, Alexandre Bergel, Mireille Blay-Fornarino, Philippe Collet, and Sébastien Mosser. 2015. A visual support for decomposing complex feature models. In *Proc. of VISSOFT 15*. IEEE, 76–85.
26. Jens von Pilgrim and Kristian Duske. 2008. Gef3D: A Framework for Two-, Two-and-a-half-, and Three-dimensional Graphical Editors. In *Proc. of SoftVis 08*. ACM, 95–104.