A Context and Feature Visualisation Tool for a Feature-Based Context-Oriented Programming Language

Benoît Duhoux, UCLouvain, Belgium
benoit.duhoux@uclouvain.be
Bruno Dumas, UNamur, Belgium
bruno.dumas@unamur.be
Kim Mens, UCLouvain, Belgium
kim.mens@uclouvain.be
Hoo Sing Leung, UCLouvain
Belgium

Abstract

In this paper we present a visualisation tool that is intricately related to a feature-based context-oriented programming language. Context-oriented programming languages allow programmers to develop software systems of which the behaviour evolves dynamically upon changing contexts. In our language, the software behaviour as well as the contexts to which the behaviour adapts, are encoded in terms of separate feature models. Due to the highly dynamic nature of such software systems and the many possible combinations of contexts to which they may adapt, developing such systems is hard. To help programmers manage the complexity of developing such software systems, we created a tool to help them visualise the contexts and features, even at runtime. The visualisation tool confronts two hierarchical models: the context model and the feature model, and highlights the dependencies between them. We conduct an initial user study of the visualisation tool to assess its usefulness and usability.

Keywords: Software visualisation tool, context-oriented programming language, dynamic adaptation, feature and context models, user study.

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at `http://ceur-ws.org`

1 Introduction

Context-aware systems [1] use information about the surrounding environment and conditions in which a software system operates, to *adapt dynamically* their behaviour to such 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). Due to the exponential combination of contexts, their possible behavioural variations, and the high dynamicity of such systems, developing such systems is hard. *Context-oriented programming languages* [19, 35] propose dedicated programming abstractions to implement context-aware 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 feature modelling [9, 17, 21, 5, 4, 29]. Hartmann et al. [17] model multiple software product lines, where the absence or presence of some features depend on the chosen contexts, in terms of a separate *feature model* and *context model*, connected through explicitly declared dependencies between the contexts and features. For example, when modelling a car navigation system, the selection of a context "Europe" would imply the selection of a feature for European maps.

Inspired by this idea, we proposed a context-oriented software architecture [30] 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.

Building upon this context-oriented software architecture, and taking the analogy with Hartmann et al.'s modelling approach a step further, we extended the

proposed architecture by explicitly representing contexts and features in terms of two separate feature models that are used at runtime to guide the selection and activation of both contexts and 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. The tool offers more than a mere static visualisation of the context and feature models. It can also depict dynamically what context and features are active or get activated, and what program code this affects.

To assess whether the tool's visual metaphor helps understanding the programming paradigm and programs written in it, we conducted an initial *user study* with master students in software engineering. We conclude that our tool helps understanding 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 used as running example throughout the paper. Section 3 then introduces our context-oriented software architecture and programming language, as well as corresponding background material. Our visualisation tool is presented in Section 4. Section 5 discusses the initial user study and discusses its results. Related work is exposed in Section 6. Section 7 concludes the paper and presents some future work.

2 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: a 'risk information system' [11].

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.

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 actually emergency is observed, the authorities will actively issue instructions specific to the emergency at hand, and specific to the current situation and user profile. 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.

3 A feature-based context-oriented approach

Context-oriented programming languages and frameworks help programmers build context-oriented systems. We proposed one such framework [30] that, based on contextual information sensed from the surrounding environment, selects and activates so-called contexts in the system. Appropriate features corresponding to these activated contexts are then selected, activated and deployed in the system, to adapt the system's behaviour to the actual context of use. We build upon this work by explicitly representing contexts and features as run-time feature models. At the end of this section we explain this feature-oriented context-aware programming approach in more detail, after having reviewed preliminary work on feature modelling, context modelling and context-oriented programming that lead to this. In the next section we then present a visualisation tool we built to help programmers visualise the underlying context and feature models and how these affect the system at runtime.

3.1 Feature modelling

Fig. 1 shows a simplified feature model depicting a subset of the functionalities of the case study.[1] A user of the risk information system can edit his or her profile (age and location) and the system can display the characteristics of an emergency such as its severity or impact zone.

A feature model highlights the commonalities and variabilities of a system [22]. Such diagrams are often used in software product lines to define a family of similar systems with some variations. In our current

[1] A more complete version is shown later in Fig. 6.

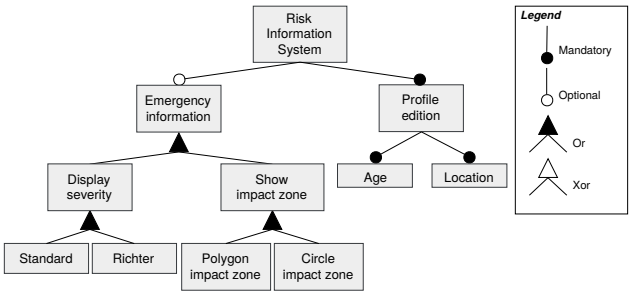

Figure 1: An excerpt of a feature model for a risk information system.

example, the commonalities are the features `Profile Edition`, `Age` and `Location`, which will be active in any instantiation of the risk information system. The other features are variabilities, i.e. features that will be deployed only in some versions of the system, or rather, at particular moments in time (e.g., when a particular emergency is active).

A feature model is represented as a tree, where the nodes represent features and the hierarchical edges represent constraints between these features. Whenever a child feature is selected, all of its ancestor features must be selected. A *mandatory* constraint, depicted with a black circle, means that if the parent feature is selected, the subfeature must be present in the system as well. An *optional* constraint, depicted with a white circle, states that the subfeature may or may not be present. As such, the `Profile edition` feature must always be present in a risk information system, whereas the emergency information is optional (it is only needed when there is an actual emergency).

An *or* (resp. *xor*) constraint, depicted by a black (resp. white) triangle, means that at least one (resp. exactly one) of the child nodes participating in this constraint should be selected in the system. In our example, the features `Display severity` and `Show impact zone` can coexist (and often do) in a same version of the system, as well as their subfeatures (this can happen if two emergencies, for example a flood and an earthquake, happen at the same time).

A particular instantiation of the system is correct only if the selection of features that adapt the system, respects the constraints imposed by the feature model. It is not allowed to activate or deactivate features if that would violate the constraints of the feature model. In Fig. 1, a valid configuration could be `Risk Information System`, `Emergency information`, `Display severity`, `Richter`, `Profile edition`, `Age`, `Location`, and would correspond to a configuration where the system can display the severity of an earthquake emergency (using the Richter scale) and where the user can edit his age and decide if the system can use his current location or not.

3.2 Contexts versus Features

Not only features, but also contexts, are key notions in feature-based context-oriented systems. 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"* [22]. Contexts and features are complementary notions that go hand in hand 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. In other words, the activation or deactivation of certain contexts triggers the activation or deactivation of certain features to adapt the runtime system behaviour.

Notwithstanding their complementarity and differences, it has been observed that the feature modelling notation can also be used to model contexts [9, 17, 21, 5, 4, 29]. For example, Desmet et al. [9] use a notation very similar to Kang et al.'s feature modelling notation [22] to design context models of context-oriented applications. Hartmann and Trew [17] present a Multiple-Product-Line-Feature Model to model several variants of a same product depending on some contexts. In their approach, schematically depicted in Fig. 2, they split the overall model in two separate submodels: a context variability model (representing the contexts and their intra-dependencies) and a traditional feature model. This allows them to model not only what the common and variable features are, but also how contexts affect what features should (not) become part of a product, by declaring explicit dependencies from the context model to the feature model. Murguzur et al. [31] state this strategy increases the number of dependencies between contexts and features but provides a better reusability of context properties.

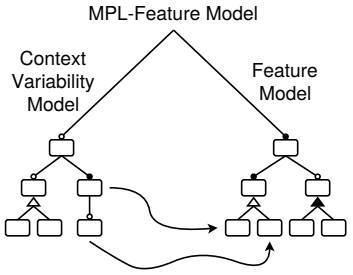

Figure 2: Illustration of the Multiple-Product-Line-Feature Model, adapted from Hartmann and Trew [17]

3.3 Context-oriented programming

Now that we have discussed how to model contexts and features, we still need to discuss how to program

context-oriented systems that can adapt to changes in their surrounding environment.

Context-oriented programming (COP) is a paradigm that provides dedicated programming language abstractions to adapt the behaviour of a software system dynamically upon changing contexts. The paradigm was introduced about a decade ago by Hirschfeld et al [19]. In COP, contexts and behavioural adaptations (modelled as features in this paper) are first-class language entities. The behavioural adaptations get (de)activated in the code whenever their corresponding contexts become (de)activated. Nowadays many different implementations of COP languages exist [3, 15, 18, 12, 2, 13, 26, 34, 36, 35, 32, 14, 24]. Most of them are extensions of existing programming languages, often object-oriented. However, many of these implementations do not clearly distinguish contexts from features. To address this issue, we propose a context-oriented software architecture [30] in which we separate contexts and features in different architectural layers. Fig. 3 provides a high-level overview of this architecture.

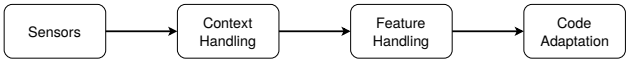

Figure 3: Our Context-Oriented Software Architecture

This architecture senses the surrounding environment in which the system executes. Whenever changes in the environment are detected, the *Context Handling* layer interprets and reasons about the raw data representing these changes, in order to reify them into contexts and (de)activate them. After the contexts are (de)activated, the *Feature Handling* layer (un)selects and (de)activates the features corresponding to the (de)activated contexts. Finally, the *Code Adaptation* layer (un)installs the code corresponding to these features to adapt dynamically the system behaviour.

3.4 Feature visualiser

In earlier work, we proposed a first visualisation tool [11] for the context-oriented software architecture described in the previous section. An excerpt of this visualisation tool, applied to the Risk Information System case, is shown in Fig. 4.

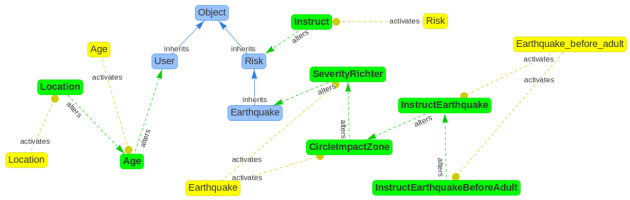

Figure 4: Illustration of our Feature Visualiser [11]

This tool depicts the currently active dependencies between the contexts, features and (object-oriented) classes of the system. For example, when an earthquake emergency is detected, the `Earthquake` context gets activated, and appears in the visualisation as a yellow rounded rectangle. This context activation then causes the selection and activation of the corresponding features `SeverityRichter` and `CircleImpactZone`, which contain specific functionality to display information about an earthquake emergency. When these features are activated, they get displayed as green rounded rectangles. Finally, these features adapt the code of some classes in the system, depicted as blue rounded rectangles (in this case, it is the *Earthquake* class that gets altered).

However, as can be seen from Fig. 4, this visualisation does not represent the contexts and features as separate hierarchical feature models, but rather as a single large graph of active contexts, features, classes and their dynamic intra- and inter-dependencies.

3.5 Context and feature models

One of the contributions of our current work is to model the possible contexts, features they trigger, and classes they adapt, as separate but interconnected hierarchical models, as depicted schematically in Fig. 5. This approach combines our context-oriented software architecture with the multiple-product-line-feature modelling approach of Hartmann and Trew [17]. We explore our dedicated visualisation tool support for this combined approach in the next section.

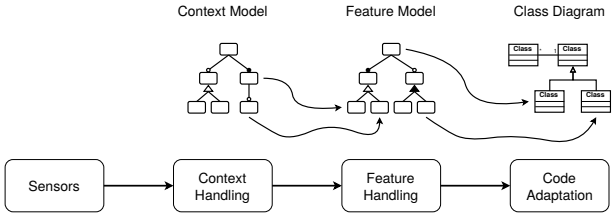

Figure 5: Overview of our feature-based context-oriented programming approach

4 Tool

Now that we have introduced our feature-based context-oriented programming approach, in this section we present the tool we built to help programmers visualise and animate the underlying hierarchical context and feature models and their interdependencies, allowing them to better understand and handle the complexity and dynamicity of systems built using that approach. We will describe the visualisation tool through different usage scenarios from a programmer's perspective. Before doing so, we briefly revisit and expand upon the case study.

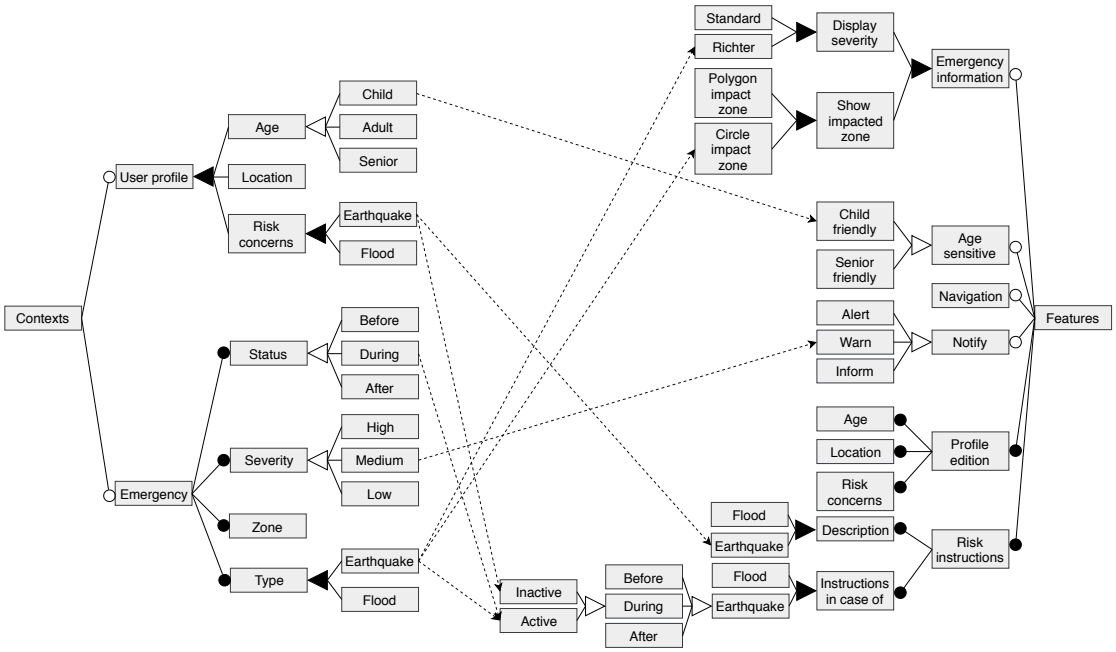

Figure 6: A context model (left) and feature model (right) for the risk information system and their inter-dependencies (dotted arrows).

Fig. 6 depicts both a context model (on the left) and a more complete feature model (on the right) of the risk information system introduced in Section 2. The functionalities (features) of this version of the system include the ability for a citizen to edit his profile, receive different kinds of notifications about emergencies depending on their severity, consult instructions for different kinds of risks, see the characteristics and instructions of ongoing emergencies, and be guided by the system to reach a safe point. Contexts that influence what features get activated at runtime include the user profile or the status, severity or type of an emergency. These dependencies are depicted as dotted arrows from the context model to the feature model.

To keep the image readable we omitted some information, such as certain dependencies from contexts to features, or the child features for Flood instructions. Despite these omissions, the models already become quite large and one can easily understand the need for a tool to visualise and explore such diagrams. On top of that, it would be useful to visualise what contexts and features are currently active, and how the diagram changes dynamically as contexts get (de)activated at runtime and trigger the (un)selection of features and their (un)installation in the code.

Our visualisation tool, depicted in Fig. 7, allows to 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 of Fig. 5. In the tool snapshot shown in Fig. 7, 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 during this emergency.

Below, we focus on different aspects of the visualisation tool, according to different usage scenarios corresponding to a programmer building a context-oriented system using this approach.

4.1 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. The *Context and feature model* pane of Fig. 7 shows what such a visualisation looks like in our tool. Such a static overview can show all contexts, features and classes of interest, regardless of whether they are currently active or not: the context model shown in Fig. 7 contains both active contexts (colored in green) and inactive ones (colored in red). The idea

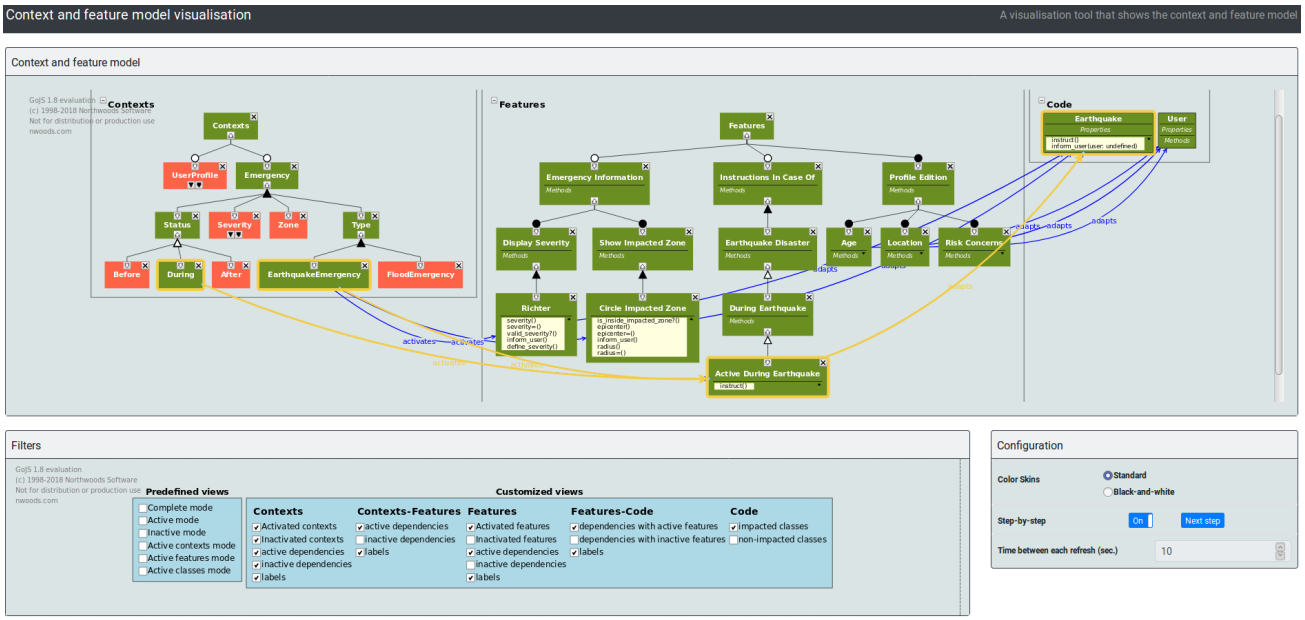

Figure 7: Snapshot of our visualisation tool applied to the risk information system. Three panes compose this tool: *Context and feature model*, *Filters* and *Configuration*. Red (resp. green) rectangles represent inactive (resp. active) contexts, features or classes in the *Context and feature model* pane. To keep the picture readable, we deliberately hid some information such as all child contexts of the `UserProfile` context, all inactive features and all non-impacted classes.

of including classes in the visualisation as well is inspired from our *Feature Visualiser* [11]. Furthermore, our visualisation tool allows a programmer to inspect in more detail the actual behaviour of features and classes. For example, Fig. 7 shows yellow boxes inside some of the features and classes, listing the methods they implement.

4.2 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 feature 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.

The visualisation tool provides several ways of exploring the system dynamics. A first one, which was already mentioned above, is the use of colouring to show active contexts, features and classes in green. A second one, which will be explained in the next subsection, is to use particular filters to show only activated contexts, features, dependencies, and currently adapted classes. The final and probably most powerful functionality provided by the tool is to show changes to the diagrams as they occur. To explore these dynamic changes, the tool provides the ability to replay the changes step by step (by activating the *Step-by-step* mode and using the *Next step* button in the Configuration pane, as showed in the bottom right of Fig. 7), so that a programmer can inspect the state of the diagrams after each change.

4.3 Filtering and predefined views

To help programmers manage the complexity of understanding big systems consisting of many different context and features, the tool comes 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. 7.

These filters (called 'Customized views') 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. The filters can be combined in many different ways. In addition to that some 'Prede-

fined views' are provided, which are predefined combinations of filters, often selected together. For example the "Active mode" shows all entities and dependencies that are currently 'active', as depicted in Fig. 8.

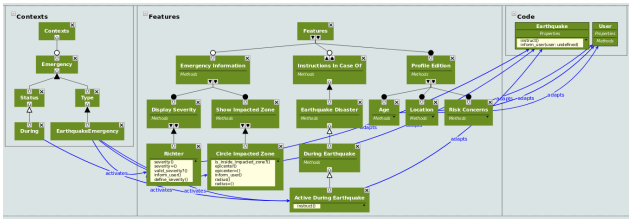

Figure 8: Visualisation of contexts and features using the "Active mode" predefined view.

4.4 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. Suppose for example that a programmer is trying to understand why a particular feature, say the `Active During Earthquake` feature, does not seem to exhibit the expected behaviour. By simply clicking on that feature, it will be highlighted in yellow, together with the contexts that triggered its selection (by following the dependencies that have this feature as target) and the classes it adapts (by following the dependencies that have this feature as source). In this example, the contexts `During` and `EarthquakeEmergency` will be highlighted, as well as the *Earthquake* class. This highlighting is illustrated by the yellow borders and yellow arrows in Fig 7.

4.5 Hiding and collapsing information

Finally, a programmer can customise his visualisation at an even more fine-grained level. For example, Fig. 9 shows a reduced feature model obtained by *hiding* and *collapsing* particular features, using the corresponding buttons to hide an element, collapse all elements above, or all elements below.

5 Validation

In this section, we describe and analyse the user study we conducted to assess 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

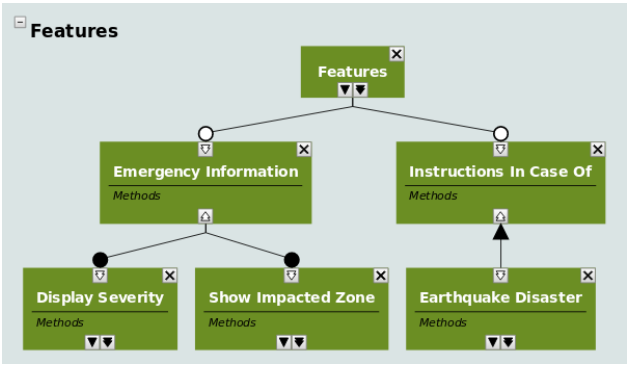

Figure 9: Simplified feature model obtained by hiding and collapsing some features.

different technologies used in the project, they participated in two preparatory sessions before the actual user study. Since the study was carried out during a course, in order not to bias the results 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. In the remainder of this section, we first describe the preparatory sessions preceding the study, then describe the user study itself, present the results we gathered from the study, and finally briefly discuss some threats to validity.

5.1 Preparation

Before conducting the user study, we initiated our master-level students with two sessions of two hours each. The first session was an introduction to the Ruby programming language, the language used to develop our feature-based context-aware approach. The second session was to explain our approach in which we exemplified it with an earthquake-specific risk information system.

5.2 User study

After these preparatory sessions, during a third two-hour session we performed the actual user study, where our students were asked to assess the usability and usefulness of our visualisation tool. To help them in their evaluation task, 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: 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 provide some information about themselves (age, knowledge of object-oriented programming, context-oriented programming and so on). They had to evaluate their knowledge on a five-level Likert scale, where a value of 1 meant they had no expertise in the field up to a value of 5 meaning they considered themselves as an expert in the field. Then we split the students/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. 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. Finally, all subjects were asked to answer some questions regarding how they perceived the usability and usefulness of our visualisation tool. We also asked them to provide some open feedback on how we could improve the visualisation tool.

5.3 Results and discussion

Despite the complexity of our feature-based context-oriented programming approach, the participants in our study seemed to agree that our visualisation tool is interesting for developers when learning our approach or during debugging.

Fig. 10 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-based context-oriented 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 the two-hour preparatory session) in Ruby.

Nevertheless, despite the difficulty of our approach, our participants do seem to be interested by the visualisation tool when they must develop a context-oriented system using our approach. Fig. 11 depicts their opinions about the tool. The first two questions concern whether they believed the static or dynamic representation of the models and their dependencies to be 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 representations 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. This result could be explained by the complexity of our 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. Finally, in our open question about which functionality is most useful, several participants answer that the ability to replay changes dynamically using the *Next step* button is really useful. In the received feedback, two main requests may be noted: the addition of a *previous step* button to step back in the process and better support for visualising larger context and feature models.

5.4 Threats to validity

The case study performed in this paper should not be considered as a full-fledged in-depth user study but rather as an initial exploratory study to help us identify the main strengths and weaknesses of the proposed approach and visualisation before developing it further. In particular we wanted to find out if the proposed visualisation tool could help programmers in coping with the inherent difficulty of building context-oriented programs of which the behaviour can change dynamically upon changing contexts. Although our initial findings were promising they are still premature and a more rigorous validation study of the form of a controlled experiment with proper task completion metrics would be needed to avoid biased conclusions. In particular, the study should be set up in such a way that the opinions of the subjects are not influenced by possible bias implanted by the context or preparation of the user study.

6 Related work

In Section 3 we already presented some background work on feature and context modelling, in order to introduce our feature-oriented context-aware approach. In this section we will explore some further related work, in particular on context-oriented and feature-oriented programming approaches, as well as on other visualisation tools related to these approaches.

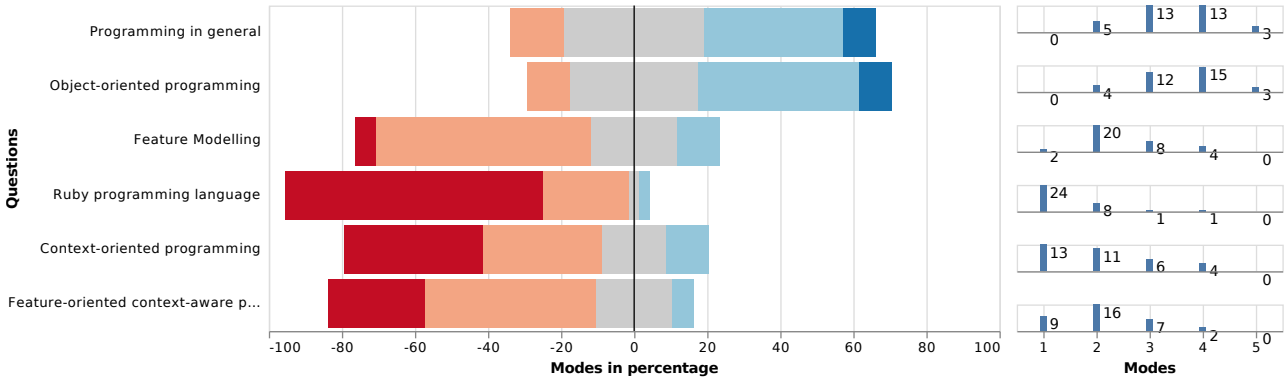

Figure 10: On the left, a divergent stacked bar diagram summarising the results of the background knowledge of our participants. On the right, a bar chart shows the frequencies of each answer for each of the questions.

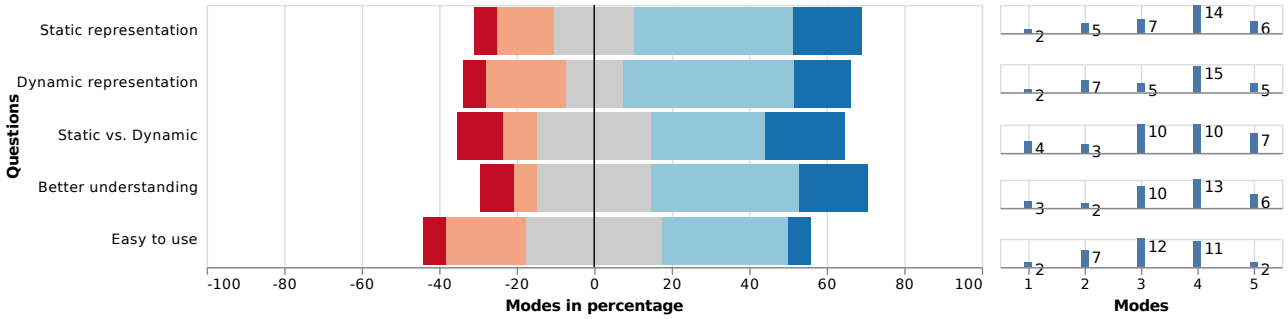

Figure 11: On the left, a divergent stacked bar presenting the results of the usability and the usefulness of our visualisation tool. On the right, a bar chart showing the frequencies of each answer for each corresponding question.

6.1 Context- and feature-oriented programming

Whereas feature-oriented programming [33] was designed with software product lines in mind, context-oriented programming [19] was designed with the purpose of creating dynamically adaptive software systems. After a closer comparison of these paradigms, Cardozo et al. [6] concluded that both approaches are quite similar, their main difference being that context-oriented programming is typically more dynamic (focussing on runtime adaptations) than feature-oriented programming (typically focussing on compile-time adaptation), even though more dynamic approaches to feature-oriented programming exist as well [16].

To create dynamic software systems sensitive to their environment, several approaches exist. One approach is to extend feature models with contextual information, so that the system can be reconfigured by selecting features at runtime depending on sensed context information [27]. A second approach is to make use of context-oriented programming languages. Some researchers have started to explore introducing the notion of features into that paradigm. For example, on top of an existing context-oriented programming language, Cardozo et al. [7] proposed a way of building applications that are composed dynamically from a set of available fine-grained features, according to contextual information. Costanza and D'Hondt [8] also proposed an extension of context-oriented programming with explicit feature descriptions that is quite similar to the feature-based context-oriented approach upon which we rely in this paper.

However none of these approaches separate clearly the features from the contexts, which is why we proposed a novel feature-based context-oriented approach based on top of our earlier context-oriented software architecture [30].

6.2 Visualisation tools

Several visualisation tools have been created for visualising different aspects of the context- and feature-oriented modelling or programming approaches that have been mentioned either above or earlier in Section 3. Many of these works state that 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* [23], 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.* [28], a web-based system that represents feature models in a much more compact tree-like structure. Illescas et al. [20] 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. [37] 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.

Nieke et al. created a tool suite for integrating modelling in context-aware software product lines [27]. This tool helps developers to model the three dimensions (spatial, contextual and temporal) of the variabilities of such approaches.

In Section 3, we showed our *Feature Visualiser* [11] tool on top of our context-oriented software architecture [30]. In addition to that tool, we also developed a *COP simulator* [10] to simulate context-oriented systems implemented with this architecture. Both of these tools can be seen as complementary to the visualisation tool introduced in Section 4.

7 Conclusion

To create context-oriented software systems, in this paper we presented a feature-based context-oriented approach, where both the contexts and features are modelled in terms of feature diagrams. Managing these different models and their dependencies 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 created a dedicated visualisation tool for this feature-based context-oriented approach, which confronts two hierarchical models (the context model and feature model) and highlights the dependencies between them. In addition, it shows the dependencies from the feature model to the code (i.e., the classes of the system). This visualisation tool not only allows programmers to inspect the models and their dependencies statically, but also to explore what happens dynamically as contexts and features are being (de)activated during system execution. For understanding and manipulating larger models, the tool also comes with filters, predefined views, and functionality to highlight, collapse or hide specific elements, allowing programmers to focus the visualisation only on specific parts of interest.

To assess the usefulness and usability of our visualisation tool, we conducted an initial user study 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 complexity tool is strongly linked to the complexity of the 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 will address this issue by relying on other visualisations such as for example a more compact tree-like view à la *S.P.L.O.T.* [28] or alternatively using hyperbolic trees [25] or a 3D representation such as GEF3D [38]. We will also conduct more rigorous user studies in order to get more conclusive results.

Acknowledgements

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

References

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proc. HUC 99*, pages 304–307. Springer, 1999.

[2] T. Aotani, T. Kamina, and H. Masuhara. Featherweight eventcj: A core calculus for a context-oriented language with event-based per-instance layer transition. In *Proc. COP 11*, pages 1:1–1:7. ACM, 2011.

[3] M. Appeltauer, R. Hirschfeld, and T. Rho. Dedicated programming support for context-aware ubiquitous applications. In *Proc. UBICOMM 08*, pages 38–43. IEEE, Sept 2008.

[4] R. Capilla, M. Hinchey, and F. J. Díaz. Collaborative context features for critical systems. In *Proc. VaMoS 15*, pages 43:43–43:50. ACM, 2015.

[5] R. Capilla, O. Ortiz, and M. Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, Feb. 2014.

[6] N. Cardozo, S. Günther, T. D'Hondt, and K. Mens. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *Proc. ICSEA 11*, pages 130–135. IARIA, 2011.

[7] N. Cardozo, K. Mens, P.-Y. Orban, S. González, and W. De Meuter. Features on demand. In *Proc. VaMoS 14*, pages 18:1–18:8. ACM, 2014.

[8] P. Costanza and T. D´Hondt. Feature descriptions for context-oriented programming. In *Lero Int. Science Centre*, pages 9–14, 2008.

[9] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt. Context-oriented domain analysis. In *Modeling and Using Context*, pages 178–191. Springer, 2007.

[10] B. Duhoux. L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte. Master's thesis, UCLouvain, Belgium, 2016.

[11] B. Duhoux, K. Mens, and B. Dumas. Feature visualiser: An inspection tool for context-oriented programmers. In *Proc. COP 18*, pages 15–22. ACM, 2018.

[12] C. Ghezzi, M. Pradella, and G. Salvaneschi. Programming language support to context-aware adaptation: A case-study with erlang. In *Proc. SEAMS 10*, pages 59–68. ACM, 2010.

[13] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-c: Bringing context to mobile platform programming. In *Proc. SLE 10*, pages 246–265. Springer, 2011.

[14] S. González, K. Mens, M. Colacioiu, and W. Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. In *Proc. AOSD 13*, pages 209–220. ACM, 2013.

[15] S. González, K. Mens, and A. Cádiz. Context-oriented programming with the ambient object system. *J.UCS*, 14(20):3307–3332, nov 2008.

[16] S. Günther and S. Sunkle. Dynamically adaptable software product lines using ruby metaprogramming. In *Proc. FOSD 10*, pages 80–87. ACM, 2010.

[17] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proc. SPLC 08*, pages 12–21. IEEE, 2008.

[18] R. Hirschfeld, P. Costanza, and M. Haupt. Generative and transformational techniques in software engineering ii. chapter An Introduction to Context-Oriented Programming with ContextS, pages 396–407. Springer-Verlag, 2008.

[19] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *JOT*, 7(3):125–151, 2008.

[20] S. Illescas, R. E. Lopez-Herrejon, and A. Egyed. Towards visualization of feature interactions in software product lines. In *Proc. VISSOFT 16*, pages 46–50. IEEE, Oct 2016.

[21] Z. Jaroucheh, X. Liu, and S. Smith. Mapping features to context information: Supporting context variability for context-aware pervasive applications. In *Proc. WIIAT 10*, volume 1, pages 611–614, Aug 2010.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, CMU, November 1990.

[23] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. Featureide: A tool framework for feature-oriented software development. In *Proc. ICSE 09*, pages 611–614. IEEE, 2009.

[24] A. Kühn. Reconciling context-oriented programming and feature modeling. Master's thesis, UCLouvain, Belgium, 2017.

[25] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proc. CHI 95*, pages 401–408. ACM, 1995.

[26] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *SCP*, 76(12):1194–1209, Dec. 2011.

[27] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context aware reconfiguration in software product lines. In *Proc. VaMoS 16*, pages 41–48. ACM, 2016.

[28] M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t.: Software product lines online tools. In *Proc. OOPSLA 09*, pages 761–762. ACM, 2009.

[29] K. Mens, R. Capilla, H. Hartmann, and T. Kropf. Modeling and managing context-aware systems' variability. *IEEE Software*, 34(6):58–63, Nov. 2017.

[30] K. Mens, N. Cardozo, and B. Duhoux. A context-oriented software architecture. In *Proc. COP 16*, pages 7–12. ACM, 2016.

[31] A. Murguzur, R. Capilla, S. Trujillo, O. Ortiz, and R. E. Lopez-Herrejon. Context variability modeling for runtime configuration of service-based dynamic software product lines. In *Proc. SPLC 14*, pages 2–9. ACM, 2014.

[32] T. Poncelet and L. Vigneron. The phenomenal gem: Putting features as a service on rails. Master's thesis, UCLouvain, Belgium, 2012.

[33] C. Prehofer. Feature-oriented programming: A new way of object composition. *CC-PE*, 13:465–501, 2001.

[34] G. Salvaneschi, C. Ghezzi, and M. Pradella. Javactx: Seamless toolchain integration for context-oriented programming. In *Proc. COP 11*, pages 4:1–4:6. ACM, 2011.

[35] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *JSS*, 85(8):1801 – 1817, Aug. 2012.

[36] G. Salvaneschi, C. Ghezzi, and M. Pradella. Contexterlang: Introducing context-oriented programming in the actor model. In *Proc. AOSD 12*, pages 191–202. ACM, 2012.

[37] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser. A visual support for decomposing complex feature models. In *Proc. VISSOFT 15*, pages 76–85. IEEE, Sept 2015.

[38] J. von Pilgrim and K. Duske. Gef3d: A framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *Proc. SoftVis 08*, pages 95–104. ACM, 2008.