Feature Visualiser: an Inspection Tool for Context-Oriented Programmers

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

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

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

ABSTRACT

As part of our ongoing research on context-oriented software technology, we propose a feature-oriented programming approach to context-oriented programming. Behavioural variations are implemented as fine-grained features that can be installed and activated dynamically, upon changing contexts. Given the highly dynamic nature of such a programming approach, and to cope with the complexity of many behavioural variations, that can depend on many varying contexts, developers could benefit from visual inspection tools to analyse what contexts and features are currently active, in which order they have been activated, and what code they adapt. We present a prototype of such a visualisation tool, and discuss potential improvements to that tool.

CCS CONCEPTS

• Human-centered computing → Ubiquitous and mobile computing systems and tools; **• Software and its engineering → Feature interaction**; **Integrated and visual development environments**;

KEYWORDS

Context-oriented programming, feature-oriented programming, dynamic software adaptation, software visualisation, Ruby programming language.

ACM Reference Format:
Benoît Duhoux, Kim Mens, and Bruno Dumas. 2018. Feature Visualiser: an Inspection Tool for Context-Oriented Programmers. In *10th International Workshop on Context-Oriented Programming (COP'18), July 16, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242921.3242924

1 INTRODUCTION

Building upon earlier exploratory work by Poncelet and Vigneron [27], Cardozo et al. [6] and Kühn [23], we have started to explore a novel context-oriented programming approach and software architecture [26] that reconciles the ideas of feature-oriented and context-oriented programming [5, 16, 28]. In this approach, behavioural variations of a software system are implemented as fine-grained feature modules that can be installed or uninstalled dynamically. The selection of features to be (un)installed dynamically depends on what contexts are currently active. Contexts are defined as separate modules, the activation or deactivation of which depends on sensory and other information detected from the surrounding environment in which the system executes. As such, the system becomes highly adaptive, changing its behaviour dynamically to contextual changes.

Given the highly dynamic nature of such a programming approach, it is hard for developers to keep track of what is going on. They need to cope with the complexity of many behavioural variations, that can depend on many varying contexts, each of which may be subject to certain declared dependencies. To support them in better understanding the complexity of such systems, we are prototyping a visualisation tool to inspect what is going on during program execution, i.e. what contexts and features get (de)activated and what existing or previously installed features they adapt.

Section 2 sets the scope of this paper by introducing a motivating example of a risk information system and summarising our previous approach and architecture for which we build this visualisation tool. Section 3 then presents the Feature Visualiser tool and gives some hints, based on the feedback received from actual programmers, on how we could further enhance this tool. Related work is presented in Section 4 and future work in Section 5, before presenting our conclusion in Section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
COP'18, July 16, 2018, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5722-7/18/07…$15.00
https://doi.org/10.1145/3242921.3242924

2 SETTING THE SCOPE

In this section, we describe a case study that will serve as motivating and running example illustrating the need for context-oriented software systems composed of many fine-grained features that can be activated and deactivated dynamically to adapt the system's behaviour to the current context of execution. Next, we describe an approach for building such systems, thus setting the scope for the work in this paper. Whereas part of this context-oriented software technology has been explored in previous work, this paper focuses in particular on a dynamic visualisation tool that can help developers inspect how the contexts and features interact and whether the implemented software system behaves as intended.

2.1 Motivating example

Our case study, which serves as running example throughout this paper, is strongly inspired by two existing applications deployed by the *crisis centre*[1] of the Belgian government. The goal of this centre

[1] https://crisiscentrum.be

and of both applications is to raise awareness, prepare, inform and alert citizens about possible risks and emergency situations.

The first system, *R!SK-iNFO.be*[2], is a web application whose main purpose is to inform citizens about possible risks, hazards and emergencies (terrorism, heat waves, industrial calamities, earthquakes, fire), and to provide generic advice on how to protect themselves and others in such situations. In what follows, we will refer to such a system as a *Risk Information System*, or RIS for short. The current implementation of this RIS provides up-to-date advice to citizens, to enhance their self-protection, by informing them about actions they can initiate before, during or after some emergency situation, in order to better protect themselves and their families.

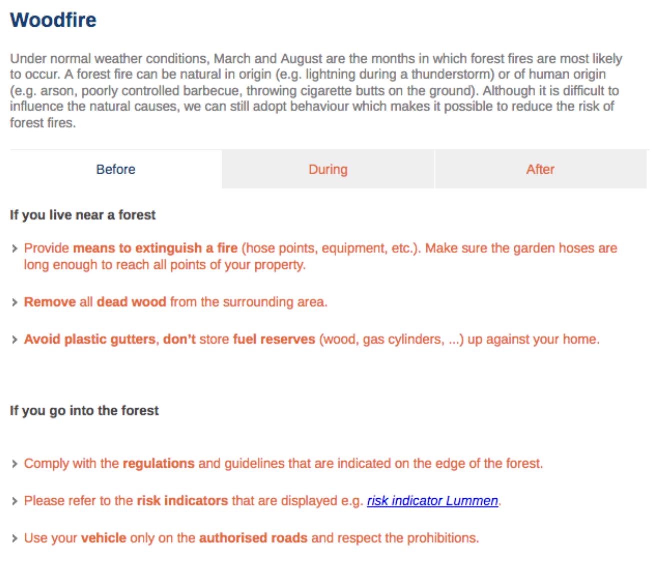
The second system, *.be alert*[3], is the alert system of the Belgian government. Citizens subscribe to this service to receive direct notifications through SMS, e-mail, Twitter, or voice-recording on their fixed phone, when an emergency occurs near their premises. For example, upon detection of a wood fire, the government would alert all citizens registered to the service and that either live or work in the affected area, with instructions about what measures they need to take to remain safe (shut windows, evacuate, ...), through these different communication channels.

Even though both systems are complementary and together address a relevant problem, something is still missing to reach the full potential that such RIS systems could offer. We believe that such systems could provide enhanced services if they would be made more dynamic and context-aware. The current *R!SK-iNFO.be* system is conceived mostly as a static website that displays a variety of information about possible risks. However it does not really adapt its content *dynamically* to contextual information such as the user, her age or location. For example, Fig. 1 shows two different sets of instructions depending on whether a citizen is *near* a forest or *in* a forest. Also, it shows three different panes depending on the status of the risk: *before*, *during* or *after*.

Other parts of the website, such as the page showing information about a heat wave, of which an extract is shown on Fig. 2, offer additional information to specific categories of users, depending on their age (senior or child) or role (working or not). However, for each of these cases it requires the user to navigate manually to the corresponding buttons or panes containing that information. The information is not selected nor filtered automatically based on the user's preferences or context. Another interesting observation is the presence of meteorological information on the bottom right of Fig. 2. That information is present to inform the citizen, but is not actually used by the system. In an ideal system, information about meteorological conditions should be able to influence instructions issued by the system. (E.g., wind strength and direction are important factors that could influence evacuation instructions given to citizens in case of woodfire.)

As for the *.be alert* system, its current version also remains rather static and mainly propagates alerts about certain emergencies to citizens living in affected zones, but does not take into account the actual location of a user [4] or its age. One could also wonder why the

[2]http://www.risico-info.be
[3]http://be-alert.be
[4]As a consequence, people present in an affected area but not registered as living or working there may not get informed, whereas people who are registered as such will get informed regardless of whether they are actually present there or not.


Woodfire

Under normal weather conditions, March and August are the months in which forest fires are most likely to occur. A forest fire can be natural in origin (e.g. lightning during a thunderstorm) or of human origin (e.g. arson, poorly controlled barbecue, throwing cigarette butts on the ground). Although it is difficult to influence the natural causes, we can still adopt behaviour which makes it possible to reduce the risk of forest fires.

Before | During | After

If you live near a forest

› Provide **means to extinguish a fire** (hose points, equipment, etc.). Make sure the garden hoses are long enough to reach all points of your property.

› **Remove** all **dead wood** from the surrounding area.

› **Avoid plastic gutters, don't** store **fuel reserves** (wood, gas cylinders, ...) up against your home.

If you go into the forest

› Comply with the **regulations** and guidelines that are indicated on the edge of the forest.

› Please refer to the **risk indicators** that are displayed e.g. *risk indicator Lummen*.

› Use your **vehicle** only on the **authorised roads** and respect the prohibitions.

Figure 1: Information on how to act before, during or after a *woodfire*, depending on the user's location w.r.t. that fire.

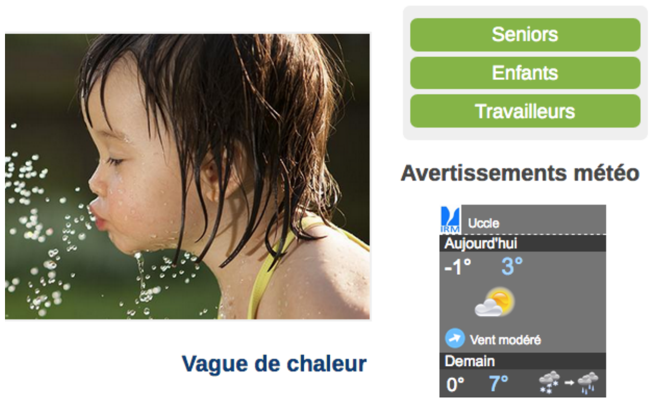


Figure 2: Information about a *heat wave*, with access to specific information for specific categories of users.

R!SK-iNFO.be and *.be alert* systems have been created as separate, disjoint systems while they cover a same goal, with their only difference being the platforms used for information diffusion. We thus conclude that, though useful and relevant, these systems are currently not optimal to provide the most relevant information to a citizen, depending on his state (age, role), current context (location), the current state of an emergency (kind, activity, severity, affected zone) and other factors that influence it (meteorological conditions).

2.2 Case study

Whereas Subsection 2.1 presented some of the features and limitations of an existing RIS, in this section we discuss the requirements for a new kind of RIS that can adapt its behaviour, in a fine-grained way, to the user and its surrounding execution context. The main purpose of our RIS remains to inform users about how to protect themselves against certain risks, but also to alert them with specific information when certain emergencies occur. The system distinguishes risks, which provide passive information on possible risks, from active emergencies, which correspond to events that are currently occurring. An emergency can have a status representing its progress, i.e. when the emergency is announced and very likely to happen (before), when the emergency is happening (during) or during its aftermath (after).

When no emergencies are actually installed, the system just provides generic information to users on how to protect themselves against certain risks. The user has the possibility to flag what risks she is more interested in, for example because she works close to a nuclear power plant or lives in a zone prone to earthquakes.

When a potential emergency is imminent (e.g., heightened earthquake risk based on seismological data), the emergency is installed with status 'before'. When the emergency eventually occurs, its status changes to 'during' and the user is provided with specific instructions on how to act during this emergency, taking into account its severity and its location. If the user is located inside an affected zone, the system gives more specific instructions depending on the status of the emergency and the user's age (child, adult or senior).

Each of these fragments of information (generic, specific, risk, emergency, for children, for seniors, during, before or after) are programmed as separate features that are installed or uninstalled as needed, thus making the system adapt dynamically to the changing context and to its users. This idea is illustrated in Fig. 3. Information from the surrounding context (e.g., information about the user, or emergency warnings issued by the government) can trigger certain features. Essentially, features are small units of functionality that can extend or override the functionality already provided by the base system. For example, when an earthquake alert is issued, one or more features describing different aspects of that earthquake, such as its severity and its impact zone, are installed. These features adapt the generic information that is shown to users about an earthquake emergency, with specific information about this particular earthquake, such as its severity (on a Richter scale) and its impact zone (described by an epicentre and a radius). To achieve this, the installed features override an existing method `inform_about_risk` to print additional information, while still calling the original method. The details of this feature-based context-oriented programming approach will be explained further in Subsection 2.4.

Since the complexity of a software system consisting of many such fine-grained features that can be (un)installed dynamically can quickly become overwhelming, we developed a prototype of a tool to help programers visualise in real-time the interplay between features, in a way similar to our diagram of Fig. 3. Fig. 4 shows a sneak preview of a dynamic visualisation rendered by our Feature Visualiser tool. Blue nodes represent Ruby classes; green nodes represent dynamically installed features and what classes or features they adapt; and yellow nodes represent active contexts and the features they triggered.

The visualisation depicted in Fig. 4 corresponds to a scenario where an *Earthquake* emergency has been issued and the earthquake is currently ongoing. The *SeverityRichter* and *CircleImpactZone* features provide specific information about this particular emergency. Also note the more basic *Instruct* feature which gets composed into the *Risk* class when a user wants to be informed about certain risks. This feature provides generic advice such as to remain calm and help others in case of emergency. This generic advice gets overridden with emergency-specific advice whenever an emergency has been issued.

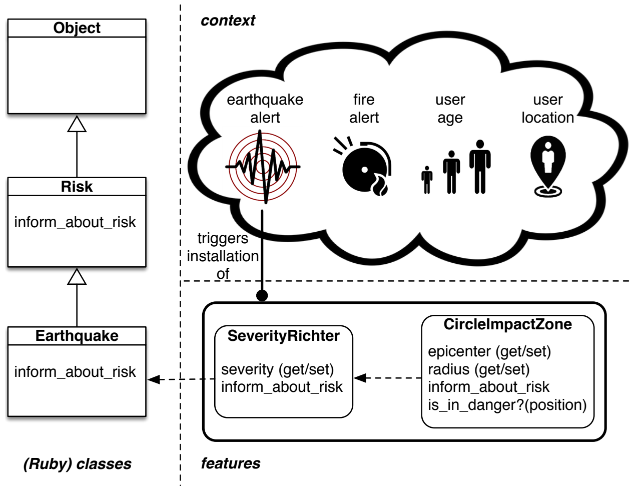


Figure 3: Schematic diagram of our approach.

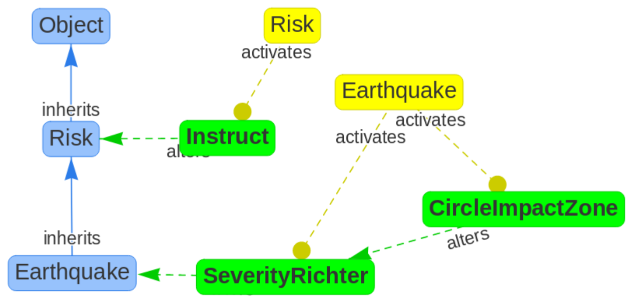


Figure 4: Sneak preview of our Feature Visualiser tool.

2.3 Context-oriented software architecture

The feature visualisation tool discussed in this paper fits into our context-oriented software architecture published earlier [26].

Fig. 5 depicts a simplified version of that architecture. It consists of three layers: the Interaction, Discovery and Handling layer, that manage the interpretation, reification and activation of contexts, respectively. The Handling layer is also responsible for the selection, activation and execution of features specific to the active contexts.

The workflow of this context-oriented architecture starts when a new event occurs that represents a change in the context surrounding the application. For instance, suppose that a user sets the age in his profile to 12. As soon as this change is detected by a sensor from the Interaction layer, the value is interpreted by the Discovery layer, who interprets this as the user being a child. This information is then reified as a context named *Child*, which is a subcontext of a more abstract *Age* context, based on a context model of possible

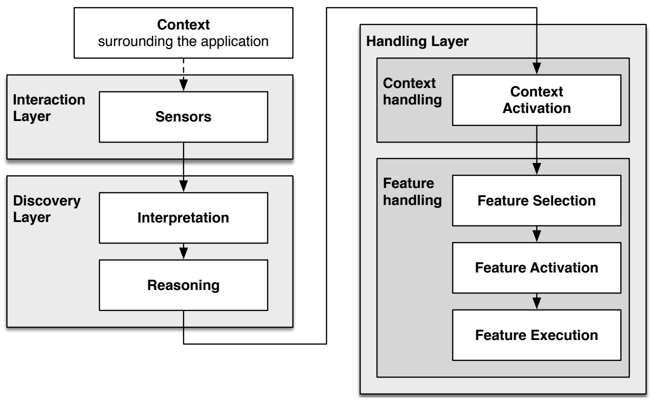


Figure 5: Our context-oriented software architecture.

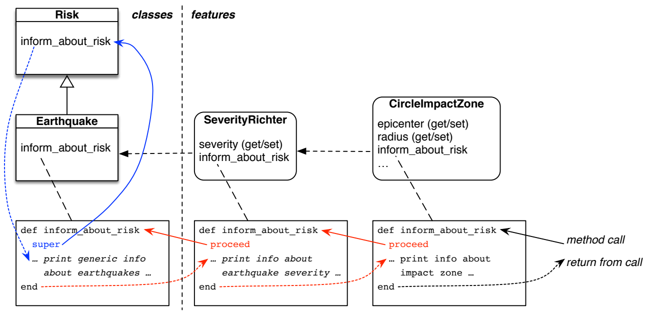
contexts and their interdependencies declared by the developers of the software system. Once the context *Child* has been reified, the Context Activation component of the Handling layer comes into play to verify whether this context can be activated or not taking into account the declared context dependencies. (E.g., an exclusion dependency could prohibit the activation of a context, if a context is already active with which that context is mutually exclusive. An implication dependency, on the other hand, may trigger the activation of a dependent context as soon as a certain context gets activated.) Finally, the Feature handling sublayer decides what features to select, activate and execute depending on the currently active contexts. More specifically, based on the activation of the *Child* context, the Feature Selection will select those features dedicated to child users. In our running example, rather than providing detailed instructions in textual format about what to do in case of an emergency, for children the text should be kept more simple or pictograms could be used. As was the case for the Context Activation component, a Feature Activation component will decide which of the selected features can be activated. To make this decision it makes use of a feature model, declared by the developers of the software system, describing the possible features and their interdependencies. Finally, after activation (resp., deactivation) of the features, the Feature Execution component adapts (resp., unadapts) the code at runtime to install the code of the newly actived features (resp., to remove the code of the deactivated features).

2.4 Feature Execution

The Feature Execution component provides a context-oriented language, the semantics of which is inspired by Hirschfeld et al. [17]. It allows a programmer to define fine-grained features, consisting of a few methods and attributes only, that can be applied to existing Ruby classes, to alter their existing methods and attributes or add new ones. In this programming approach, the application core consists of a set of predefined classes that define the application's *default behaviour* (i.e., its behaviour when no context-specific features are installed). In Fig. 3, the application core consists of a class *Risk* and some subclasses for different kinds of risks such as *Earthquake* or *Fire*, providing a generic `inform_about_risk` method to inform users about such risks.

This default behaviour can be *adapted dynamically* with fine-grained features that can either add new methods and attributes, or replace or override existing ones from the core. What behaviour goes into the core and what goes into the dynamic features is up to the developer to decide. As a rule of thumb, features should contain information and behaviour specific to particular contexts, whereas the application core should correspond to default behaviour and information of generic nature. Also, to promote reuse, we prefer small fine-grained features that can be composed together easily, rather than huge monolithic features consisting of many classes, methods and attributes. In Fig. 3 for instance, we decided to have a separate feature for each attribute of an emergency. Taking into account that most emergencies have both a severity and an impact zone, the intuition behind our guideline would be to have a separate feature for each of those attributes. So we have a feature *CircleImpactZone* for describing the impact zone of an *Earthquake* emergency. This feature may be reused for other emergencies having circular impact zones. Similarly, we have a feature for the severity level of the emergency. In case of an *Earthquake* emergency, this is provided by the feature *SeverityRichter* which defines the severity of an earthquake using the Richter magnitude scale. Whereas that feature is quite specific to an earthquake emergency, other emergencies like *Fire* may be adapted with a more generic *SeverityLevel* feature, described in terms of a simple value (low, middle or high) that can be applied more easily to other types of emergencies.

The dynamic feature adaptation mechanism also supports *multiple features adapting a same class*. Features are then applied one by one in order of activation, so that the last one activated gets installed last. This order is important to know in case different features override a same method. Indeed, it is possible for features to *refine* methods defined in the application core or introduced by previously installed features, using the so-called *proceed* mechanism [16]. More specifically, methods defined by a feature can make use of a special keyword *proceed*, which will call a previously installed method with the same name. This powerful mechanism allows features to incrementally modify previously defined behaviour. This notion of *proceed* is inspired by how aspect-oriented [21, 24] and context-oriented programming languages [12, 16, 17] allow aspects or contextual adaptations to dynamically extend the application core or previous adaptations of that core.



```
def inform_about_risk
  super
  … print generic info
    about earthquakes …
end
```

```
def inform_about_risk
  proceed
  … print info about
    earthquake severity …
end
```

```
def inform_about_risk
  proceed
  … print info about
    impact zone …
end
```

Figure 6: Method call semantics in presence of the proceed mechanism.

Fig. 6 illustrates the semantics of this *proceed* mechanism on our running example. The class *Earthquake* as well as the two features *SeverityRichter* and *CircleImpactZone* that adapt it, each implement a method named `inform_about_risk`. Whereas the version of this method defined on the *Earthquake* class just provides generic information about an earthquake risk, the methods defined on each of the two features that adapt it, incrementally refine this method to provide additional information. The combined effect of the interplay of these methods is to print the default information implemented by the `inform_about_risk` method defined on the class *Earthquake*, followed by information provided by the `inform_about_risk` method of the *SeverityRichter* feature which was installed next, followed by the `inform_about_risk` method of the *CircleImpactZone* feature which was installed last. The control flow starts with the latter one, but the proceed immediately delegates to the previously installed feature *SeverityRichter*, whereas the proceed there delegates to the default version of the method defined on the *Earthquake* class. This one makes a super call, which redirects to the `inform_about_risk` method defined on the superclass *Risk*. After that the control flows back in opposite order to execute the remainder of each of those methods. The result of this execution thus prints some generic information about earthquake risks, followed by the severity of that risk, followed by its impact zone (epicenter and radius).

Whereas this example still remains relatively easy to visualise mentally, things quickly get more complicated when more features come into play, because of the complex interplay of this powerful *proceed* mechanism combined with the dynamic activation and deactivation of features. Therefore we propose the programmers a visualisation tool to support the need to manage this complexity.

3 APPROACH

As stated previously, the focus of this paper is on a visualisation tool helping developers to get a better understanding of how features execute and interact within a feature-based context-oriented software system. Due to the high number of features and the complexity of dealing with many such fine-grained context-specific features, we created a feature visualisation tool to complement our context-oriented architecture and programming approach. After describing the features of our visualisation tool, we discuss some possible improvements based on the feedback we received from actual programmers.

3.1 Feature Visualiser

The Feature Visualiser tool allows to visually inspect how features dynamically adapt a running software system. This tool is dedicated to developers building highly dynamic context-aware applications using our approach, to understand the interactions between the features, or to debug the system if it does not seem to behave as desired. The visualisation shows what existing Ruby classes the features adapt, for what contexts, in what order, and can also show more details on the different methods provided by the features. Fig. 7 shows the Feature Visualiser tool at work. It consists of four widgets: a *Visualisation*, *Output console*, *Legend* and *Configuration* widget.

The *Legend* widget explains the visual notation through a simple meta-model. Classes are denoted by blue rounded rectangles and can inherit from other classes. Contexts are denoted by yellow rounded rectangles connected to the features they activate. Features are depicted as green rounded rectangles that alter either existing classes, or other features that were previously installed.

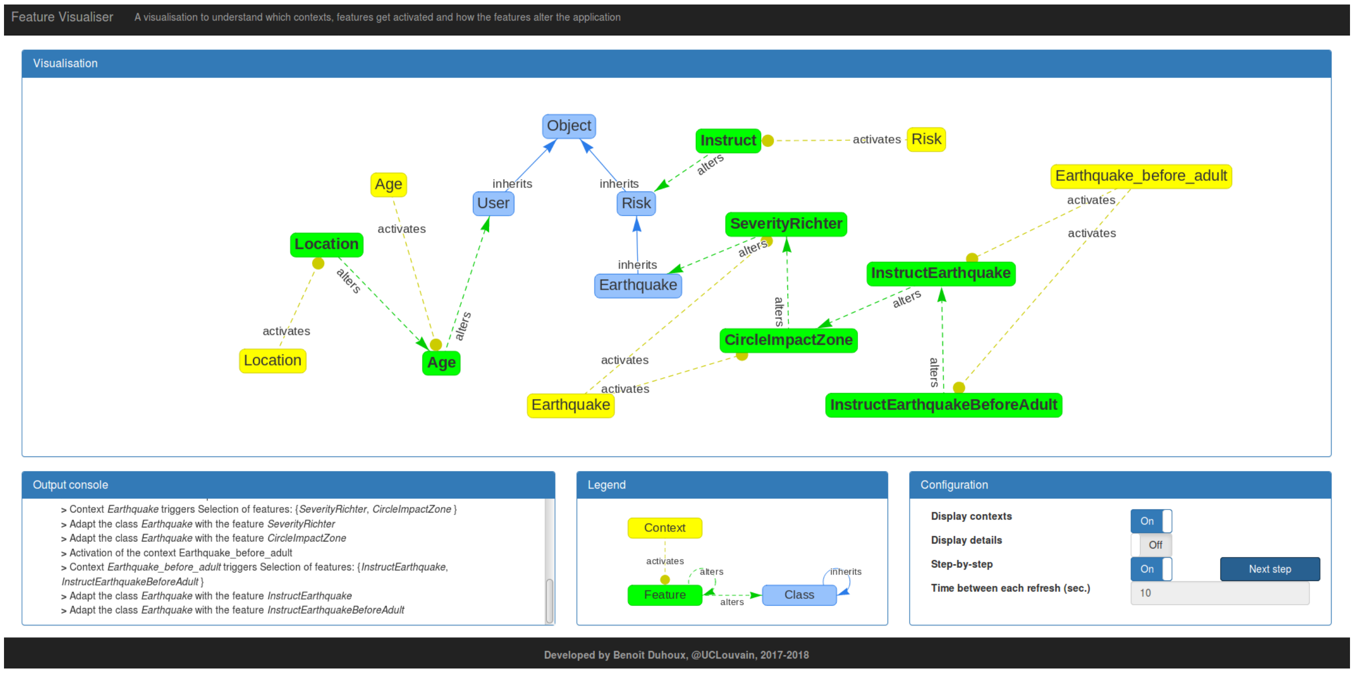
The *Visualisation* consist of a more passive part, i.e. the declared classes and their inheritance relationships, and a more dynamic part showing the active contexts, the features triggered by those contexts, and how these features alter existing classes and other features. This visualisation is highly dynamic and updated continuously as contexts get (de)activated and features get added to or removed from the application. To avoid making the visualisation too complex, it only shows those classes that get altered by features, together with their direct ancestors. Similarly, no other contexts and features are shown than those that get activated at runtime by the context-oriented architecture.

This visualisation is coupled to and triggered by the architecture, which sends key messages to the Feature Visualiser about what to visualise. For example, it sends a message ('adapt' or 'unadapt') whenever due to a context change an alteration adapts or removes a feature from the running code, and passes along key data such as the name of the feature, the class the feature applies to, the superclass of that class, the methods the feature alters, and so on. All this information is used by the Feature Visualiser to provide essential information to developers on what is going on and thus improve their understanding of the running system.

The *Output console* outputs a log of all the actions triggered by the architecture. In a sense it is a textual version of the visualisation widget, with the added benefit that it keeps a chronological trace of what contexts and features were activated or deactivated before. This can provide valuable information to the developer when debugging, for example to understand in what order certain features were added or removed and upon what contexts.

Finally, the *Configuration* widget allows a developer to configure the visualisation tool as desired. Enabling the Step-by-step functionality makes the different contexts and features appear and disappear dynamically, whenever the 'Next step' button is pressed. When disabled, the visualisation tool plays all actions automatically, though a certain timing delay can be set to slow down this automatic visualisation. Another configuration option is to show the contexts too, or only the features. A final configuration flag is to show more details about the methods provided by each class and feature. Enabling this option yields a visualisation like the one in Fig. 8. Note that, in the classes being altered by features, each method name is prepended with the name of the last (i.e., most specific) feature that overrode this method. This information is very valuable for debugging purposes to better understand the dynamic behaviour of the program.

The Feature Visualiser thus enables developers to easily visualise what is going on and whether the developed language abstractions and their behaviour exhibit the right semantics that corresponds to a developer's intuition. As a developer's support tool, it helps to spot early on conceptual or implementation errors in programs developed using this approach. It can thus serve as a live feedback and debugging tool.



```
> Context Earthquake triggers Selection of features: {SeverityRichter, CircleImpactZone }
> Adapt the class Earthquake with the feature SeverityRichter
> Adapt the class Earthquake with the feature CircleImpactZone
> Activation of the context Earthquake_before_adult
> Context Earthquake_before_adult triggers Selection of features: {InstructEarthquake,
InstructEarthquakeBeforeAdult }
> Adapt the class Earthquake with the feature InstructEarthquake
> Adapt the class Earthquake with the feature InstructEarthquakeBeforeAdult
```

Display contexts: On
Display details: Off
Step-by-step: On
Time between each refresh (sec.): 10

Figure 7: Snapshot of the full Feature Visualiser tool

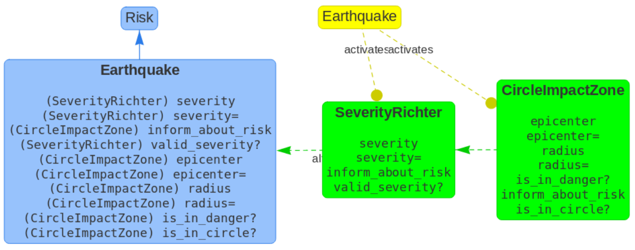

Risk
Earthquake
(SeverityRichter) severity
(SeverityRichter) severity=
(CircleImpactZone) inform_about_risk
(SeverityRichter) valid_severity?
(CircleImpactZone) epicenter
(CircleImpactZone) epicenter=
(CircleImpactZone) radius
(CircleImpactZone) radius=
(CircleImpactZone) is_in_danger?
(CircleImpactZone) is_in_circle?
Earthquake
activatesactivates
SeverityRichter
severity
severity=
inform_about_risk
valid_severity?
al
CircleImpactZone
epicenter
epicenter=
radius
radius=
is_in_danger?
inform_about_risk
is_in_circle?

Figure 8: Feature visualisation in detailed mode.

3.2 Possible improvements

To assess the understandability and usability of our Feature Visualiser tool, we conducted two different validation experiments with master-level students in computer science and in computer science engineering. For each experiment, we had around 20 students playing the role of real programmers evaluating our visualisation tool. During these initial experiments we received interesting feedback on how to enhance our visualisation tool. Some programmers proposed to add some filters to clean the visualisation to display only some specific behaviour or only a specific set of features or contexts. Another request was the addition of a *previous* button to enable step-by-step replaying of previous feature or context activations. The automated layout algorithm could also be improved. The ability to see the details of only some features or classes was a final requested improvement.

4 RELATED WORK

This related work section will be split in two parts exploring, on the one hand, the programming paradigms of feature-oriented and context-oriented programming and how they are related. On the other hand, we discuss alternative visualisation and other tools helping developers manage the complexity of having many fine-grained features or contextual adaptations.

FOP versus COP. Kang et al. define the notion of a feature as "a user-visible aspect or characteristic of the domain" [19], as for example our feature for managing the severity of an earthquake (*SeverityRichter*). Such features are typically used in the domain of software product lines which are composed of many distinct features which can be of fine granularity [20]. Different programming approaches exist to build a software product line, of which *Feature-Oriented Programming* (FOP) [28] is closest to the approach used in this paper. Whereas traditionally FOP is static, in the sense that the composition of features is done at compile-time [20], recent approaches often take a more dynamic stance allowing features to be deployed at runtime [13].

When building context-aware systems, Coutaz et al. state that "context is key" [7], and that developers must take into account the user, the external and physical environment as possible sources of contextual information. Whenever the context changes, the system may discover new contextual information which may trigger the system to adapt its behavior consequently [26]. The paradigm of *Context-Oriented Programming* (COP) [16] was conceived as a new programming paradigm able to deal with the dynamicity of such context changes and how they trigger dynamic adaptations of the system. Many implementations of COP exist, as extensions to a variety of programming languages like Erlang [9, 30], Java [1, 3, 29], JavaScript [11, 25], Smalltalk [15], Lisp [12], Objective-C [10], Ruby [8, 27] and so on.

When comparing both paradigms, Cardozo et al. [5] came to the conclusion that they are very similar, their main difference being the core concept on which they rely. While FOP is based on features, COP is based on context. Starting from that observation, some approaches proposed to reconcile both paradigms into a single one [6, 23]. From the perspective of modeling context-aware systems, Hartmann and Trew [14] and Capilla et al. [4] also confirmed that the notions of contexts and features seem complementary and could be reconciled into a single modeling approach.

Visualisation and other tools. The visual inspection tool presented in this paper is only one of many possible visualisations that could be of value to developers. Starting from the same observation that the large number of features and the complex nature of their interactions can make software development and maintenance tasks challenging, Illescas et al. [18] put forward four different visualisations that focus on features and their interactions at source code level, and evaluate them with four case studies. In the context of software product lines, Urli et al. [31] present a visual and interactive blueprint that enables a software engineer to decompose a large feature model in many smaller ones while visualising the dependencies among them. They too, claim that such visual support is essential especially for large and complex feature models. The efficiency of their visualisation with respect to maintainability is studied on two different real case studies. Still in the context of software product lines, 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. In the domain of context-oriented programming, Duhoux [8] also argued for the need for tool support when dealing with many context-specific adaptations and proposed tool support to simulate the execution of a context-oriented system. All these tools and visualisations can be seen as complementary to the one proposed in this paper and stress the importance of tool support to manage complexity and increase understandability.

5 FUTURE WORK

In this section, we discuss some avenues of future work related to our Feature Visualiser and how it fits into a bigger vision of context-oriented software development. In addition we want to extend our feature-based context-oriented programming approach to support user interface adaptation.

Improving the Feature Visualiser Taking into account the feedback received from developers in our initial experiments, we can further enhance the proposed visualisation tool with their suggestions as mentioned in Section 3.2.

Other visualisation idioms Whereas our Feature Visualiser shows the activation and deactivation of contexts and features during program execution, other complementary visualisation idioms could be used, as was already mentioned in the related work section. A visualisation of particular value to developers would be one that shows the detailed execution path of a method call taking into account the *proceed* mechanism (cf. Fig. 6).

Additional visualisation tools We introduced a visualisation tool helping programmers to see what contexts and features get activated and how features alter classes or other features. To improve the developer's experience further, we could create a entire suite of tools including this visualisation tool. For example, we could build a simulator tool to simulate such systems, like the one presented by Duhoux [8].

Further validation After having improved our visualisation tool, we need to carry out more experiments to collect evidence on the usability and appropriateness of the proposed visualisation tool, following Ko et al.'s practical guide [22] to conduct controlled experiments of software engineering tools with real participants.

User interface adaptation The feature-based context-oriented programming approach mentioned in this paper is only a first step in our quest for a novel highly adaptive context-oriented software development approach, with dedicated support for user interface adaptation as well. Our current notion of features should thus be extended so that they are no longer purely behavioural but include user interface aspects as well.

6 CONCLUSION

In this paper, we briefly discussed our feature-based context-oriented programming approach to build highly adaptive context-oriented systems. However, it is not always trivial for developers to understand the run-time semantics of such a program as contexts and features get incrementally activated and deactivated. This problem comes from the fine granularity and high dynamicity of the feature composition mechanism in this approach. To address that complexity, we proposed a visual inspection tool to help programmers understand how programs are constructed as a dynamic composition of many fine-grained features and to inspect which features adapt which methods of what classes, upon what contexts, in what order, and at what moment. From initial experiments conducted with programmers to assess the understandability and usability of our Feature Visualiser tool, we received useful suggestions to further improve the visualisation. Taking into account these suggestions we will further improve the visualisation tool and complement it with additional visualisation and development support tools. We will integrate these tools into an existing context-oriented development framework to provide run-time support for dynamic software adaptation and to be able to simulate and debug such programs under realistic conditions. Finally, we will extend the approach and framework with dedicated support for fine-grained dynamic user interface adaptation.

REFERENCES

[1] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. 2011. Featherweight EventCJ: A Core Calculus for a Context-oriented Language with Event-based Per-instance Layer Transition. In *Proceedings of 3rd International Workshop on Context-Oriented Programming (COP '11)*. ACM, Article 1, 7 pages.

[2] Sven Apel and Dirk Beyer. 2011. Feature Cohesion in Software Product Lines: An Exploratory Study. In *Proceedings of 33rd International Conference on Software Engineering (ICSE '11)*. ACM, 421–430.

[3] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. 2008. Dedicated Programming Support for Context-Aware Ubiquitous Applications. In *The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE, 38–43.

[4] Rafael Capilla, Oscar Ortiz, and Mike Hinchey. 2014. Context Variability for Context-Aware Systems. *Computer* 47, 2 (February 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 *International Conference On Software Engineering Advances (ICSEA'11)*. IARIA, 130–135.

[6] Nicolás Cardozo Alvarez, Kim Mens, Pierre-Yves Orban, and Wolfgang De Meuter. 2014. Features on Demand. In *Proceedings of 8th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 18, 8 pages.

[7] Joëlle Coutaz, James L. Crowley, Simon Dobson, and David Garlan. 2005. Context is Key. *Commun. ACM* 48, 3 (March 2005), 49–53.

[8] Benoît Duhoux. 2016. *L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte*. Master's thesis. Université catholique de Louvain, Belgium.

[9] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. 2010. Programming Language Support to Context-aware Adaptation: A Case-study with Erlang. In *Proceedings of 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*. ACM, 59–68.

[10] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. 2011. Subjective-C: Bringing Context to Mobile Platform Programming. In *Proceedings of 3rd International Conference on Software Language Engineering (SLE'10)*. Springer, 246–265.

[11] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. 2013. Context Traits: Dynamic Behaviour Adaptation Through Run-time Trait Recomposition. In *Proceedings of 12th Annual International Conference on Aspect-oriented Software Development (AOSD '13)*. ACM, 209–220.

[12] Sebastián González, Kim Mens, and Alfredo Cádiz. 2008. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science* 14, 20 (nov 2008), 3307–3332.

[13] Sebastian Günther and Sagar Sunkle. 2010. Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming. In *Proceedings of 2nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, 80–87.

[14] Herman Hartmann and Tim Trew. 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Proceedings of 12th International Software Product Line Conference (SPLC '08)*. IEEE Computer Society, 12–21.

[15] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. 2008. Generative and Transformational Techniques in Software Engineering II. Springer, Chapter An Introduction to Context-Oriented Programming with ContextS, 396–407.

[16] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich* 7, 3 (2008), 125–151.

[17] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. 2011. ContextFJ: A Minimal Core Calculus for Context-oriented Programming. In *Proceedings of the 10th International Workshop on Foundations of Aspect-oriented Languages (FOAL '11)*. ACM, New York, NY, USA, 19–23.

[18] Sheny Illescas, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Towards Visualization of Feature Interactions in Software Product Lines. In *IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 46–50.

[19] 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. Carnegie-Mellon University.

[20] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of 30th International Conference on Software Engineering (ICSE '08)*. ACM, 311–320.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer, 327–353.

[22] Andrew J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (February 2015), 110–141.

[23] Alexandre Kühn. 2017. *Reconciling Context-Oriented Programming and Feature Modeling*. Master's thesis. Université catholique de Louvain, Belgium.

[24] Ramnivas Laddad. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications.

[25] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming* 76, 12 (December 2011), 1194–1209.

[26] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. 2016. A Context-Oriented Software Architecture. In *Proceedings of 8th International Workshop on Context-Oriented Programming (COP'16)*. ACM, 7–12.

[27] Thibault Poncelet and Loïc Vigneron Vigneron. 2012. *The Phenomenal Gem: Putting Features as a Service on Rails*. Master's thesis. Université catholique de Louvain, Belgium.

[28] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. (1997), 419–443.

[29] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2011. JavaCtx: Seamless Toolchain Integration for Context-oriented Programming. In *Proceedings of 3rd International Workshop on Context-Oriented Programming (COP '11)*. ACM, Article 4, 6 pages.

[30] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. ContextErlang: Introducing Context-oriented Programming in the Actor Model. In *Proceedings of 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, 191–202.

[31] Simon Urli, Alexandre Bergel, Mireille Blay-Fornarino, Philippe Collet, and Sébastien Mosser. 2015. A visual support for decomposing complex feature models. In *IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 76–85.