École polytechnique de Louvain

Visualisation of Contexts and Features in Context-Oriented Programming

Author: **Hoo Sing LEUNG**
Supervisors: **Kim MENS, Benoît DUHOUX**
Reader: **Jean VANDERDONCKT**

Academic year 2018–2019
Master [120] in Computer Science and Engineering

Abstract

Over the last few years, the concept of using context-oriented programming to implement context-dependent systems is more and more investigated. The RELEASeD research group in UCLouvain is exploring a context-oriented software development approach that combines ideas from context-oriented programming and feature-oriented modelling. In this approach, contexts reify situations in the environment to which the software can adapt. Features encode how the system should adapt to such situations. Feature modelling can be used to model both contexts and features. However, when developing such systems, keeping track of all possible contexts, features, their activation, inter- and intra-dependencies quickly becomes a daunting task. In this master's thesis, we develop a prototype of a visualisation tool that can help developers of such systems understand, even during program execution, what contexts get activated, what features they trigger, and what code the features adapt at runtime.

Acknowledgements

I would like to first thank my supervisor, Prof. Kim Mens from the EPL faculty for his time and all his advices during this semester. Even tough, he had quite a busy semester, he still took time for scheduling a meeting every two weeks to follow my progress. I would also like to thank my other supervisor, Ph.D. student Benoît Duhoux for his patience when explaining the code and concepts of this master's thesis. Almost every week a meeting was scheduled with him. Without these sessions, I would not be sure to be able to finish all the work in the due time. I would like to also thank them for giving me the opportunity of taking part in the writing of a scientific paper.

I would also like to thank all my friends that support me during this period.

And last but not least, I would like to thank my family, especially my parents for their supports during all these years of study. It was not easy but thanks to your constant encouragements, I managed to overcome every difficult period.

Contents

1 Introduction **4**

2 Related work **6**
2.1 Feature-Oriented Software Development 6
2.1.1 Feature 6
2.1.2 Software Product Lines 7
2.1.3 Feature modelling 7
2.2 Context-Oriented programming 8
2.2.1 Context 9
2.2.2 Context-aware systems 10
2.3 Contexts versus Features 11
2.4 A Context-Oriented Software Architecture 14
2.5 Visualising tools 17
2.5.1 Feature model 18
2.5.2 Tools for the architecture 19

3 Case study **24**
3.1 Risk Information System 24
3.2 Functionalities 25
3.3 Contexts 26
3.4 Features 27
3.5 Dependencies 29
3.6 Running example 29

4 Problem statement **32**

5 Solution **33**
5.1 Choice of the programming language 33
5.2 GoJS 34
5.3 The visualisation tool 37
5.3.1 Context and feature model 38

5.3.2 Filters 50
5.3.3 Configuration 51

6 Validation **54**
6.1 Study 54
6.1.1 First session 54
6.1.2 Second session 55
6.2 Results 57

7 Future work **60**
7.1 Class diagram 60
7.2 Scalability of the context-feature model 60
7.3 Step-by-step 61
7.4 Combination with other tools 62

8 Conclusion **63**

Chapter 1

Introduction

Nowadays, people use more and more their mobile phone to get news over the world. As the world is increasingly facing natural disasters because of global warming, it would be interesting to have an application that can inform and instruct citizens in case of emergencies. This application could give specific instructions depending on the location, the profile and other information of a user. Such an application is called a *Risk Information System* (RIS). A solution for implementing this kind of system is the use of the concept of context.

A system that uses contexts to adapt its behaviour is called *Context-aware system*. The context-oriented programming paradigm (COP) is a solution for programming such systems. Researches in this domain is evolving over the years. In the last decade, several researchers have explored the relation between feature modelling and context-oriented software development. Amongst others, feature modelling has been proposed to model context variability in context-aware systems and context-oriented programming language have started to converge with feature-oriented programming. In such context-oriented approaches, the (de)activation of certain contexts can dynamically trigger the (de)activation of correspoding features, causing the application to exhibit behaviour adapted to the current context.

Following this idea, a research group in UCLouvain started to explore COP languages having as first class entities a kind of features, describing how the application should adapt its behaviour to changing contexts. They proposed an implementation architecture for such languages, in which the notion of contexts is clearly separated from the notion of features. Based on sensory input received from the surrounding environment, contexts get activated and deactivated, which in turn triggers activation or deactivation of corresponding features, using a context-feature mapping. These features then adapt existing code, causing the application to change its behaviour at runtime. Moreover, they represent both contexts and

features in terms of dynamic feature models, to model, respectively, both the context variability and behavioural variability of the application.

However debugging such an application seems complicated for programmers because they must visualise which contexts and features are active and how they interact with the system. They could face an exponential number of combinations between these entities. To help the developers in this task, this master's thesis proposes a visualisation tool displaying the declaration of the context and feature models as well as their inter- and intra-dependencies. To handle the runtime debugging, this tool also offers the visualisation of the dynamic configuration on these models. In addition, we extend this visualisation with the notion of classes (representing the system itself) to show the programmers which features adapt which classes in order to increase the comprehension of the programmers. Some filters are also added to improve the developer's view when applications become complex.

The structure of this master's thesis is as follow: Chapter 2 will describe the background material and the related work. Then Chapter 3 will expose a case study of a RIS application, including the context and feature models that need to be shown in the tool. Chapter 4 will emphasise the problem based on the case study. Afterwards, Chapter 5 will present our visualisation tool. Following it, an explanation of the validation study done will be presented in Chapter 6. Finally, Chapter 7 will give the possible improvements of the tool and Chapter 8 will conclude this master's thesis.

The codes of the visualisation tool may be found on the following private bitbucket repository: `https://bitbucket.org/benoitduhoux/rubycop/src/master/`. To have access on it, please contact the author or the supervisors of this master's thesis.

Chapter 2

Related work

In this chapter, we will present the background material and the related work of this master's thesis. The main purpose of our visualisation tool consists of helping a developer to debug a context-aware application based on a feature-oriented context-aware approach. In order to have a better understanding of this approach, we will start our chapter by discussing about **Feature-Oriented Software Development** (FODA) which is a research domain used in the framework of the approach. Afterwards, the main programming paradigm used in the implementation of the framework will be explained. This paradigm called **Context-Oriented Programming** (COP) allows us to build a particular class of systems called *Context-aware systems*. An example of an implementation architecture for this kind of systems will be discussed. This specific architecture is the one used in the programming approach. Finally, we will show several visualisation tools that are related to ours.

2.1 Feature-Oriented Software Development

Feature-Oriented Software Development is *"a programming paradigm for the construction, customization, and synthesis of large-scale software system"* [1]. This paradigm uses the notion of feature as its core.

2.1.1 Feature

The notion of feature is fundamental in this master's thesis. It is defined as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system"* [2].

2.1.2 Software Product Lines

Software Product Lines (SPLs) are an interesting type of software systems as it builds a family of systems from several software assets instead of a unique system composed of all these assets [3]. Basically, each system of the family has several of these assets. Assets that are shared for all the systems are called *commonalities*, whereas, assets that are partially shared or unique are called *variabilities*. Let us take as an example, a car factory. A car can have different assets such as a navigation system (maps), a radio, a USB connector and a communication protocol. If we imagine that all the cars have the same communication protocol, this asset is a commonality. If we state that a car, depending of its region, will have specific maps, then navigation system is a variability.

Basically, the process of SPLs starts from a set of software assets and a set of models. Then, each model will mix different assets together so as to obtain a software system. These models are called *Variability Models* [3]. In FOSD, these assets are called features.

2.1.3 Feature modelling

Kang et al. were the first to use the concept of the feature to represent the variability models [2]. They introduced the notion of *feature model* to show the dependencies and relationships of a set of features.

Feature modelling is a key concept in software product line engineering. It is used to describe the variability in a family of products (variability models), for requirements engineering and for configuring a product [4]. A feature model is typically described in terms of a feature diagram which is itself composed of two sets of elements: a set of nodes representing the features and a set of links representing relations between nodes. For example, Figure 2.1 shows a feature model that describes the variability of a car. **Car** is the root node, it can be considered as the name of the family of software systems. **Body**, **Transmission** and **Engine** are the features that are mandatory for a car. **Pulls Trailer** is an optional feature. **Automatic** and **Manual** has a XOR constraint which means that only one of them can be selected. Whereas, **Gasoline** and **Electric** has an OR relationship which means either one of them can be selected or both. The selection of a parent node implies the selection of at least one child. Similarly, the selection of a child (i.e. **High**) requires the selection of its parent (i.e. **Security**). At last, the selection of **Automatic** implies the selection of **Electric**.

A configuration of a feature model is a subset of features. It is valid if the combination of the selected features is allowed by the feature model. An example of valid configuration for Figure 2.1 is {Car, Body, Transmission, Automatic, Engine,

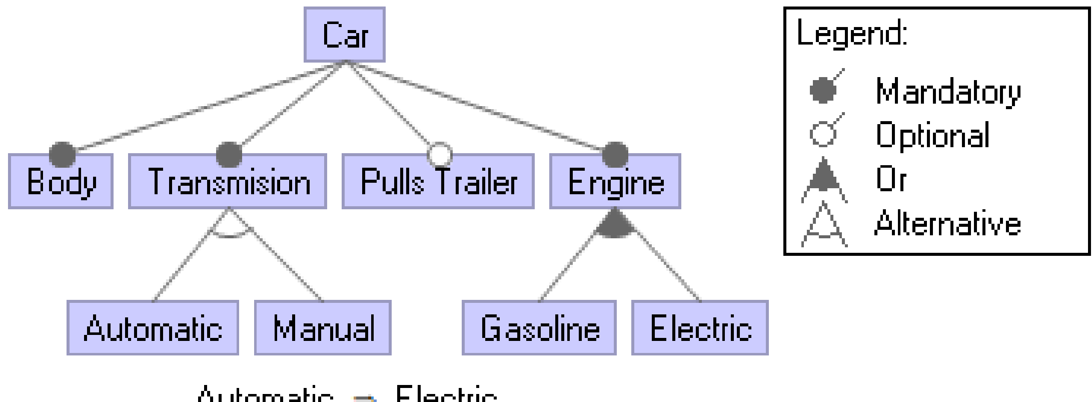


Figure 2.1: Example of feature diagram using FeatureIDE, inspired from [1]

Electric}. {Car, Body, Transmission, Automatic, Manual, Engine, Electric} is an invalid configuration since the feature model does not allow both the selection of Automatic and Manual.

A tool for creating such diagrams (FeatureIDE [2]) is shown in Section 2.5.1.

Feature modelling is a domain analysis technique that can be combined with other programming paradigms. Schlee et al. presented a combination between feature modelling and generative programming [5]. This last one is a computing paradigm allowing the creation of software families given particular requirements specifications. They tested their approach in a project called MiniABA [3], which is a software that allows the generation of calculators depending on a selection of features. Figure 2.2 depicts the part of the software where the user has to select the desired features. Whereas, Figure 2.3 shows the created project.

2.2 Context-Oriented programming

Context-oriented programming is a programming paradigm that allows developers to implement context-dependent systems. These software applications have their behaviour vary depending on their surrounding environment. Indeed, this paradigm provides mechanisms to dynamically adapt behaviour depending on changes in context, even after system deployment [6].

[1] `http://www.jot.fm/issues/issue_2009_07/column5/`

[2] `https://featureide.github.io/`

[3] The video on `https://www.youtube.com/watch?v=P5jhgl5EQ3A` shows the creation of calculators depending on the choice of features.

Figure 2.2: Screenshot of the software MiniAba.

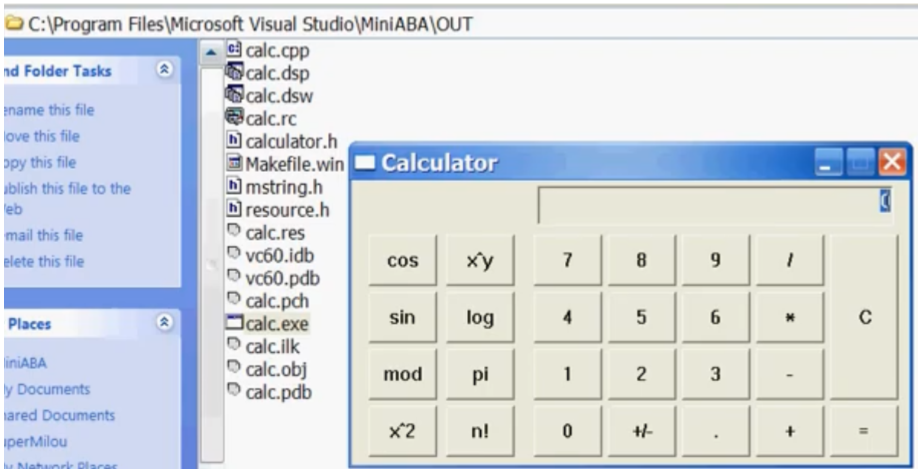


Figure 2.3: Screenshot of the calculator created with MiniABA.

2.2.1 Context

Hirschfeld et al. defined context as *"Any information which is computationally accessible may form part of the context upon which behaviour variations depend"* [6].

However, Salvaneschi et al. think that this definition is vague. Nevertheless, they affirm that practice with context-aware systems confirms its validity [7]. For them, this definition does not limit the context to the information reaching the system from its external environment, but it also includes information coming from inside the system boundaries. Furthermore, this general approach does not restrict the level of abstraction through which the context is represented in the system. They give the example of bandwidth consumption in a network. It can be quantified as 100Mb/s or we can observe that the processor is busy 95% of the time. Here the obtained information is numerical (obtained through sensors). These values are often abstracted. As an example, the measured processor usage can be

abstracted into a symbol stating that the processor is in *heavy load* condition.

2.2.2 Context-aware systems

As described in [8], a context-aware system is able to adapt its operations to a specific context without explicit user intervention. It uses environmental information in order to adapt its functionalities. By doing so, it allows an increase of usability and effectiveness. The best example to illustrate such a system is the use of a mobile device. For a lot of mobile applications, we often need them to react specifically to their current location, time and other environment attributes. As context data tends to change, we also want these applications to adapt their behaviour according to these changes. The information about the context can be provided through different ways, such as sensors, network information, and so on.

One easy but not optimal solution for implementing such systems is the use of conditional statements. For instance, depending on the current level of battery of a mobile phone, this one has a specific mode of luminosity (Listing 2.1). If the phone has less than 5% of battery, it will enter in `no_luminosity` mode. If the phone has 15% or less of battery, it enters in `low_luminosity` mode. In other cases, its luminosity will automatically update depending on the external brightness.

```
1 class Phone
2   def luminosity()
3     if battery_level < 5 then phone.no_luminosity
4     elseif battery_level <= 15 then phone.low_luminosity
5     else phone.auto_luminosity
6   end
7 end
```

Listing 2.1: Phone luminosity changes depending on the battery level

A far better solution is the use of context-oriented programming. In Figure 2.4, we can observe the general architecture for context-aware systems presented by Gonzales et al in [9]. If we continue with the example of battery level of a mobile phone, the `Context Discovery` module may discover that the battery level is at 3% thanks to the sensors. The `Context Management` knows that this information is linked to the *Low Battery* context and activates it. As a result, the application updates its behaviour by going into `no_luminosity` mode to save battery power.

In fact, `Context Management` does not only reify the sensed information to specific contexts. But it also checks whether these contexts can be activated or not based on the current application behaviour. Another aspect it needs to check is whether the activation of a selected context activates another one or not. Indeed, contexts have dependencies and constraints between them. For example, if the current behaviour has a context that does not allow the activation of the *Low*

Battery context. This last one will not be activated. Another example is the activation of the context *No network* due to the activation of *Low Battery*.

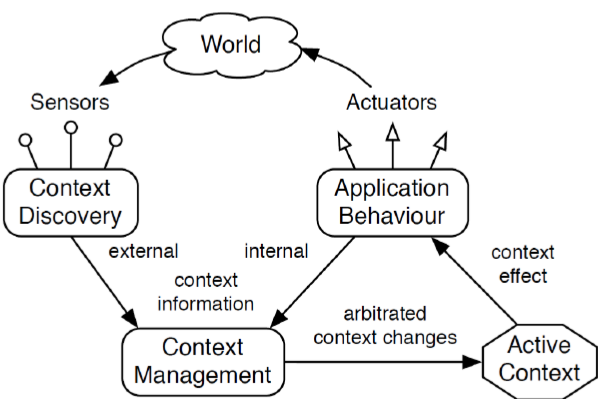


Figure 2.4: General architecture for context-aware systems, extracted from the paper [9]

As stated in [10, 11], the application part of a context-aware system is important, however, the user interface part is just as relevant. One problem of this last part is the constant evolution of the available platforms for this type of applications. While in the mobile platform we have principally IOS and Android, in the computer platform, we have Linux, Windows and MacOS. Therefore, having a user interface fitting all currently available platforms is a significant challenge. Despite the significance of this problem, this master's thesis will focus mainly on the application part.

2.3 Contexts versus Features

While a context can be regarded as any kind of information that can be used to characterize the situation of an entity (which can be a person, place or object) [12], a feature is considered as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system"* [2]. Context-aware applications adapt their behaviour according to the surrounding environment on which they run. This environment is modelled as a set of contexts representing the user with his preferences, sensed data from internal sensors (e.g. the battery level, ...) and external information (e.g. the weather, an emergency situation, ...). Because of the difference in their semantics, contexts and features are complementary in context-aware applications. In such applications, the (de)activation of contexts may trigger the (de)activation of some features which results in an adaptation of

the application behaviour. To contrive such applications, Mens et al. proposed a software architecture with a four tiered approach able to discover, (de)activate contexts and features and finally adapt the behaviour [13]. The complementarity between the two notions is even stronger thanks to their similarities in the modelling domain [14, 4, 15, 16, 17, 18] and in the programming domain [19, 20, 21, 22, 23].

In the modelling domain, Desmet et al. started to use the feature modelling notation to express context-aware software requirements [14]. Their approach is called *Context-Oriented Domain Analysis* (CODA). This model enforces engineers to compare a context-aware systems to pieces of basic context-unaware behaviour which can be refined. In Figure 2.5, we can see an example of the CODA approach applied to the case study of a context-aware cell phone.

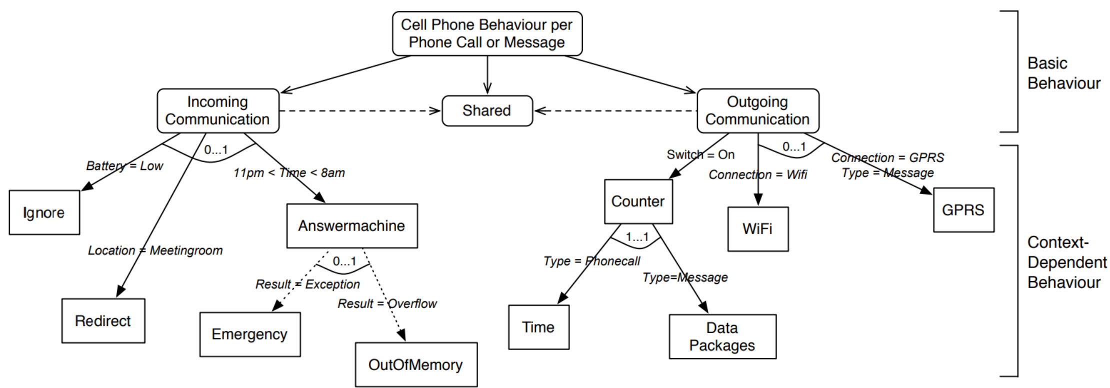


Figure 2.5: Example of CODA diagram, extracted from the paper [14]

Hartmann and Trew [4] also used feature modelling to describe a context and feature model for Software Supply Chains. This modelling approach (Figure 2.6) uses two branches: one for the *Context Variability Model* and the other one for the *Feature Model*. The combination of both results in a model of multiple product lines (MPL-Feature Diagram) supporting several dimensions in the context space. The *Context Variability Model* consists of general classifiers (set of requirements or constraints for that context) of the context in which the product could be used. In short, it tracks down the commonality and variability of the context. An example of classifier could be a geographic region. If some features are needed in Europe, then all of these features will have to be activated if the product is designed for Europe. In order to link the branches, they propose several dependencies between the *Context Variability Model* and the *Feature Model*.

Capilla et al. state in [16] that this strategy requires a lot of dependencies between the two models. Nevertheless, they recognize that this modelling is the

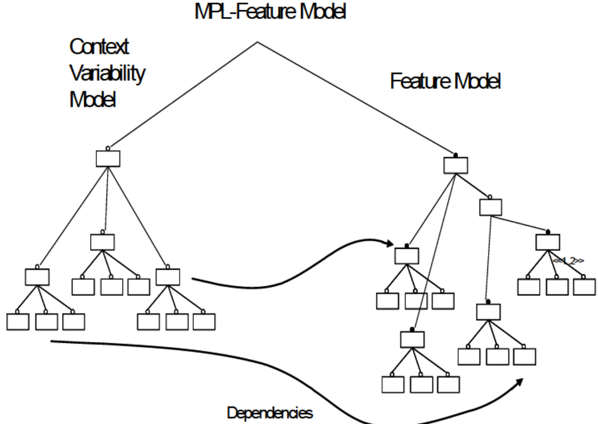


Figure 2.6: Hartmann and Trew MPL-Feature Model, extracted from the paper [4]

most adapted for application with a high amount of context changes.

Regarding the programming domain, Cardozo et al. compare contexts and features in their programming languages approaches [20], Context-Oriented Programming (COP) destined to the contexts and Feature-Oriented Programming (FOP) for the features. To find out the differences between the two paradigms, they implemented a common case study with *rbFeatures* [24] as a FOP language and *Subjective-C* [9] as the COP language. They concluded that there are 6 common characteristics shared between the implementations and 6 differences as depicted in Figure 2.7. The main difference is that FOP is static and COP is dynamic.

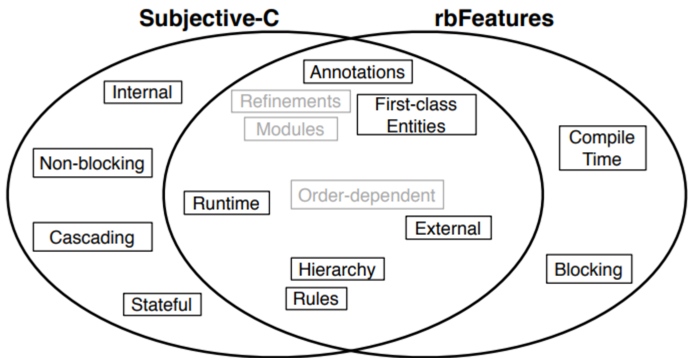


Figure 3: rbFeatures and Subjective-C characteristics.

Figure 2.7: rbFeatures and Subjective-C characteristics, extracted from the paper [20]

2.4 A Context-Oriented Software Architecture

As already mentioned in Section 2.3, Mens et al. presented in [25, 13] a new layered software architecture that uses features to describe how the application should adapt its behaviour to changing contexts. In this architecture, the notion of contexts is clearly separated from the notion of features. Based on sensory input received from the surrounding environment, contexts get activated and deactivated, which in their turn trigger the activation or deactivation of corresponding features, using a mapping similar to the one proposed by Hartman et al. [4]. These features then adapt existing code, which cause a change of behaviour of the application at runtime. In addition to that, they represent both contexts and features in terms of dynamic feature models, to model, respectively, both the context variability and behavioural variability of the application as presented in Kuhn's master's thesis [22].

To resume, this architecture combines FOSD and COP and has as main differences from traditional COP languages:

- the incorporation of features as first-class entities;
- the modelling of context and features with dependency and hierarchical relations.

Figure 2.8 depicts the proposed architecture. It is a layered architecture composed of 4 layers. In order to explain the purpose of each layer, we will use the following example: a *Risk Information System* (RIS). This system informs a user about the status of an active risk (e.g. the location and the severity of an earthquake) and instructs the citizens on how they must react before, during and after the risk.

The `interaction layer` gathers information from the system's surrounding environment (information coming from the physical environment, the user or the system's computing platform). The information is collected by the user input or sensors. For example, this layer has the information about the detection of an earthquake.

Thanks to a set of listeners provided by the `application layer`, and the context declarations specified by the programmer, the second layer entitled `discovery layer` interprets and reasons over the information gathered by the `interaction layer`. In our example, `discovery layer` interprets the information and reifies it into two contexts: `Earthquake` and `During`. This reification is obtained thanks to the context model defined by the user.

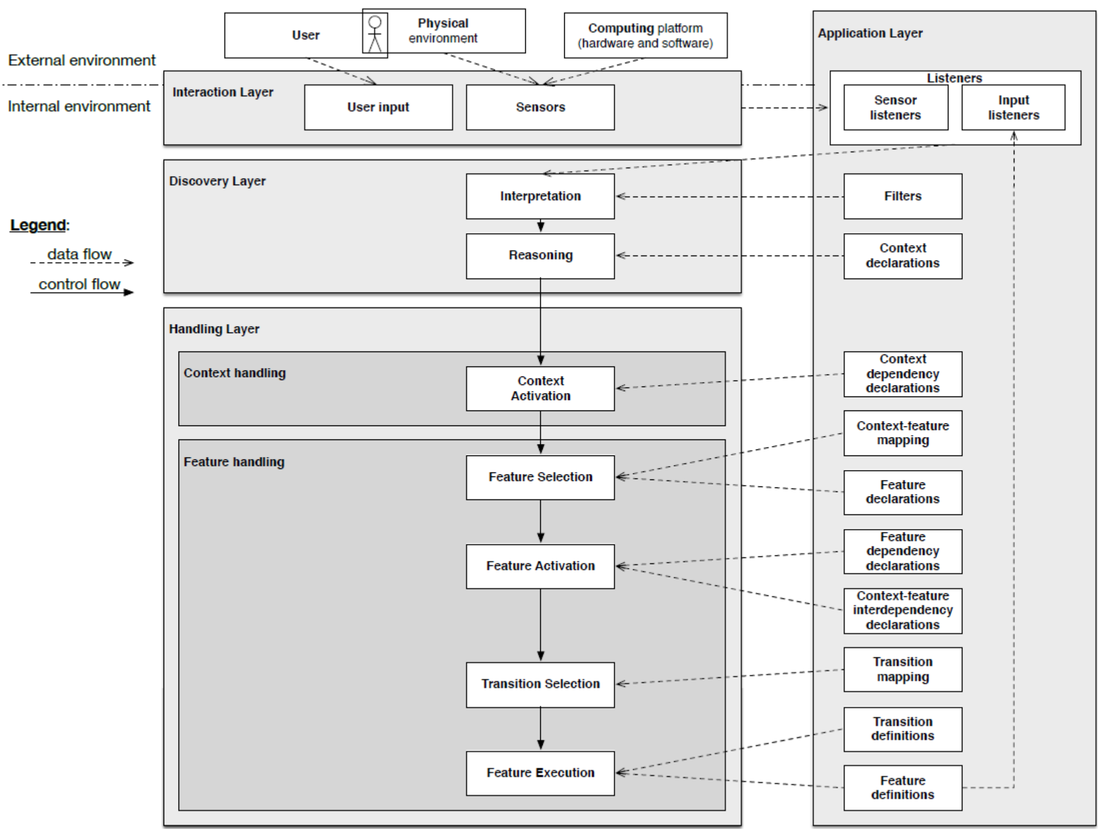


Figure 2.8: A Context-Oriented Software Architecture, extracted from the paper [13]

The third layer called `handling layer` will first check whether it can activate the selected contexts (thanks to the `Context Activation` component) based on the intra-contexts dependencies declared by the user. As a reminder, the activation of a context is valid if the context model (represented by a feature diagram) keeps having a valid configuration after its activation (based on the notion of configuration defined in Section 2.1.3).
If the activations are allowed, it will be the turn of the sublayer `Feature handling` to continue the work. It will first select the corresponding features to the (de)activation of the selected contexts thanks to the context-feature mapping entered by the user. In our example, let us imagine that the contexts `Earthquake` and `During` are bound to the feature `DuringEarthquakeRisk`.
Once the feature selected, the component `Feature Activation` will do a similar task as `Context Activation`, only this time it will be the configuration of the feature model that will be checked. In addition to this verification, the inter-dependencies between the context and feature model will also be verified (based on

the context-feature inter-dependency declared by the user).
If everything checks out, `DuringEarthquakeRisk` is activated and based on the purpose of this feature, the application behaviour will be adapted thanks to the `Feature Execution` component. Users will, for example, receive a notification with instructions they need to follow during the earthquake.

These three layers form the `adaptation framework`, which the authors describe as *"an implementation framework containing the machinery upon which context-oriented program(mer)s can rely"* [13].

The last layer called `Application layer`, specifies the different implementation components to be provided by an application developer to customize the `adaptation framework` into an actual context-oriented application (e.g. context declarations, feature declarations, mapping between contexts and features,...).

Dependencies between contexts

As presented by Duhoux [25], several dependencies between contexts is possible. Five types of dependencies is described based on [9, 25]:

- ***Xor*** This dependency assures that only one child context is active at a time. Each time that a new child context is activated, the other child context is deactivated.
  For instance, a disaster has a type, it can either be a natural one or a domestic one. It can be represented as a context `Disaster` having two children `Natural` and `Domestic`.
  - if neither child is active then `Natural` or `Domestic` can be activated;
  - if `Natural` (resp. `Domestic`) is activated and we want to activate `Domestic` (resp. `Natural`), the dependency refuses its activation as long as `Natural` (resp. `Domestic`) was not explicitly deactivated by someone;
  - `Natural` or `Domestic` can be deactivated.
- ***Exclusion*** This dependency excludes that two contexts are active at the same time.
  An earthquake has a location (e.g. Europe, America, ...). If an earthquake is located in `America`, it can not be in `Europe`. These two locations are mutually exclusive.
  - `America` (resp. `Europe`) can be activated if and only if `Europe` (resp. `America`) is not active;

  - `America` (resp. `Europe`) can be deactivated.

- ***Requirement*** This dependency implies that a context needs another one to be already activated.
  For example, a user of the *RIS* application receives notifications of the application only if the `4G` service of his mobile phone is activated. This is possible only if the phone is in `HighBattery`, which means that `4G` requires `HighBattery`.
  - `4G` can be activated if and only if `HighBattery` is active;
  - `HighBattery` can be activated;
  - `4G` can be deactivated;
  - `HighBattery` can be deactivated but `4G` will need to be deactivated as well.

- ***Causality*** This dependency is a weak inclusion.
  For instance, the `Emergency` context weakly implies the presence of `Emergency services` context. If there is an emergency (i.e. `Robbery`), it means that there is an intervention of emergency services (i.e. `Police`). But the presence of the police does not necessarily mean that there is a robbery.
  - The activation (resp. deactivation) of `Robbery` will lead to the activation (resp. deactivation) of `Police`;
  - `Police` can be activated or deactivated independently of `Robbery`.

- ***Implication*** This dependency is a logical implication.
  For instance, if the current location of an earthquake is `Brussels`, then it strongly implies that the earthquake is in `Belgium`.
  - The activation (resp. deactivation) of `Brussels` will lead to the activation (resp. deactivation) of `Belgium`;
  - Th deactivation of `Belgium` will automatically lead to the deactivation of `Brussels`.

2.5 Visualising tools

In this section, we will describe several tools that are related to this master's thesis.

2.5.1 Feature model

As one of the goals of this master's thesis is to visualise the context and feature models, several tools that allow the creation of features models will be described in this section.

- `FeatureIDE` [26, 27, 28] is an extensible framework for feature-oriented software development (FOSD). It is an Eclipse-based IDE that supports all phases of feature-oriented software development and is currently still under constant development since 2004. As explained in Section 2.1, FOSD is a paradigm for the construction, customization, and synthesis of software systems. Its idea is to decompose software systems into features in order to have configuration options. Depending on the set of features that is chosen, it will generate a different software system. A configuration is a subset of features and is valid if the combination of features is allowed by the feature model.

  With this IDE, it is possible to have a clean visualisation of a feature model. An example is shown in Figure 2.1 of Section 2.1.3.

  As this IDE is mainly designed to represent feature models, it is difficult to use it as a solution for this master's thesis. However, FeatureIDE has some interesting representation of relations between nodes such as the representation of the constraints (e.g. a black filled circle for a mandatory constraint, ...) and the Top-Down ordered tree structure. It will be taken into account in the solution.

- `S.P.L.O.T.` (**S**oftware **P**roduct **L**ines **O**nline **T**ools) is a Web-based reasoning and configuration system for Software Product Lines [29]. Developed in Java, it uses an HTML template engine to construct highly-interactive Ajax-based reasoning and configuration interfaces. Thanks to its web-based system, sharing knowledge (e.g. with the feature model repository, ...) is easier and software updates are no longer required.

  In Figure 2.9, a feature model created with S.P.L.O.T. is shown. As we can observe, it has an original representation. Whereas usually a feature model is shown with a horizontal tree graph similar to Figure 2.1, in S.P.L.O.T. we have a vertical one. Another interesting point to notice is the possibility of collapsing and expanding nodes. This option can help improving the scalability of the graph. In a case where a model has lots of features, the addition of this option will allow to shrink the graph on nodes that is not useful. Finally, this uncommon representation uses less places than the horizontal structure.

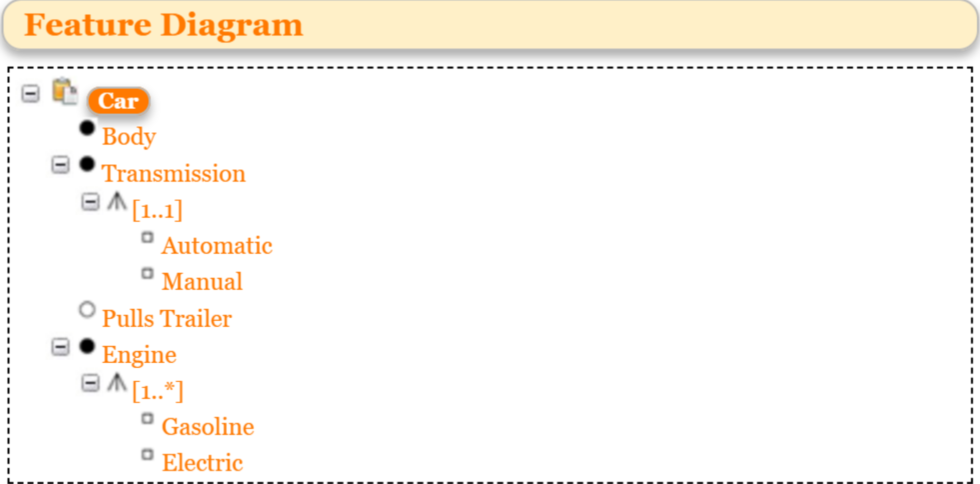


Figure 2.9: Feature model modelling an SPL (software product line) of a car with S.P.L.O.T.

Due to the large scale and complexity of today's software-intensive system, one of the most essential problems in software product lines is the complexity of variability management. One solution that has gained momentum over the last few years is tool support. Bashroush et al. conducted a survey on variability management more specifically on all the available tool support [30]. They concluded in their survey that researchers needs to focus more on several points:

1. Scalability and Complexity
2. Consistency Checking and Model Correctness
3. Traceability from Problem to Solution Space
4. Maintenance and Evolution

In the conception and implementation of our solution described in Chapter 5, we tried to focus more on the first and fourth point.

2.5.2 Tools for the architecture

In this section, we will describe tools that are related to the architecture presented in Section 2.4.

Feature visualiser

As described in Section 2.4, Mens et al. presented an architecture that supports a feature-based context-oriented programming approach [13] for context-aware systems. Due to the highly dynamic nature of such a programming approach, developers can have a hard time keeping track of the (de)activations of contexts and features as well as their impacts. Indeed, the (de)activation of contexts triggers the (de)activation of corresponding features which will adapt the application behaviour. In order to help developers debugging such applications, Duhoux et al. [23] presented a visualisation tool.

This tool depicted in Figure 2.10 consists of four widgets: *a Visualisation widget, Output console, Legend and Configuration widget*. The *Legend* widget explains the visual notation with a little example. The *Visualisation* widget shows the declared classes and their inheritance relationships. It also has a more dynamic part showing the active contexts, the features triggered by those contexts, and how these features alter existing classes and other features. The *Output console* shows a log of all the actions that have been triggered by the architecture (for example, the activation of the context `Earthquake_before_adult`,...). At last, the *Configuration* widget allows the developer to change the configuration of the visualisation tool.

Thanks to these four widgets, developers can easily visualise what is going on and whether the developed language abstractions and their behaviour match the developer's intuition. It can thus serve as a live feedback and debugging tool. However, this tool still lacks a global vision of all existing contexts and features (more generally, the static part of the application).

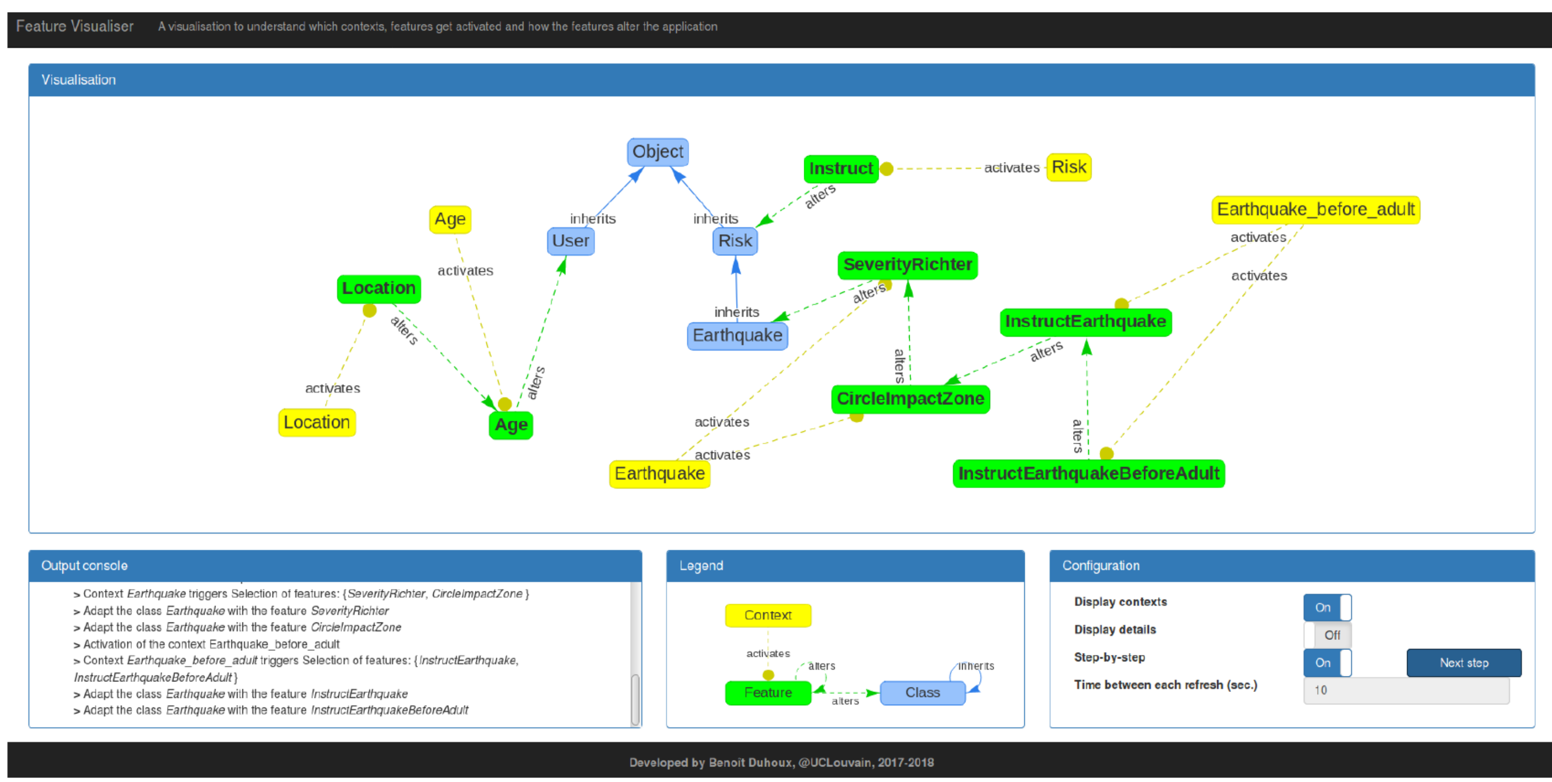


Figure 2.10: Snapshot of the Feature Visualiser tool, extracted from the paper [23]

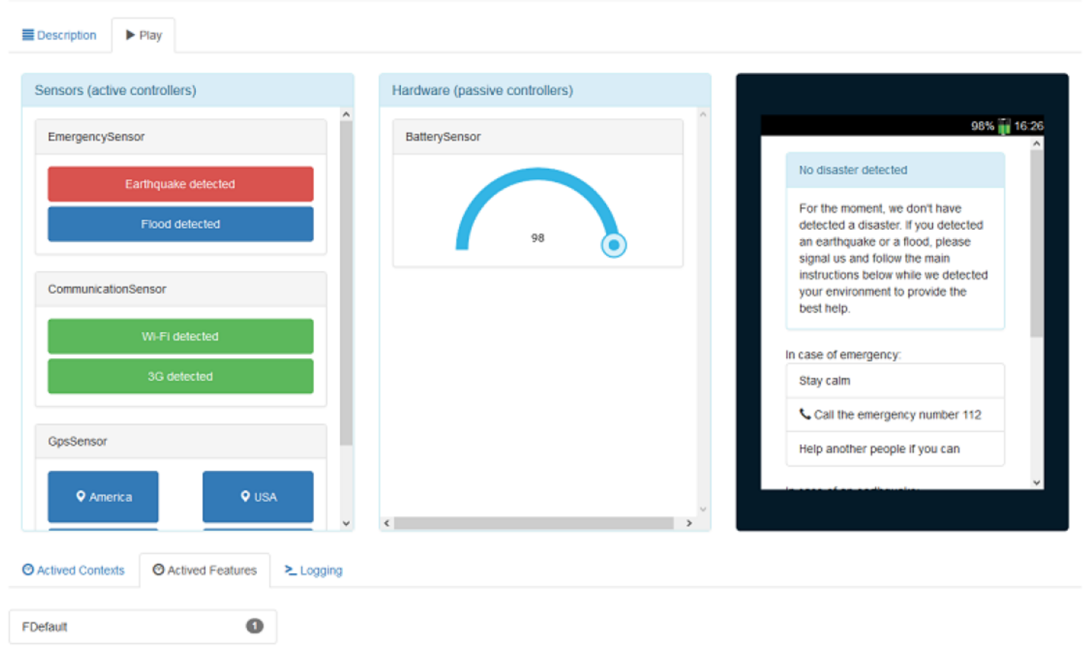


Figure 2.11: Snapshot of the simulator, extracted from Duhoux's master's thesis [25]

A simulator of context-oriented applications

In addition to the feature visualiser tool that helps a developer visualise the adaptations of the code, Duhoux presented in his master's thesis [25] a web application (Figure 2.11) that simulates a test environment for context-oriented application built using the architecture from Section 2.4. The user can reproduce environments in which the application will execute. With it, he can also modify the current situation so that it simulates a change of context and triggers the adaptation of the application behaviour. Thanks to these functionalities, it can, by consequent, serve as a debugger.

Figure 2.12 depicts how the simulator works:

1. when the user changes a context, the web browser will send a request to the server;
2. when the information is received, it will notify the framework thanks to the sensors of the application;

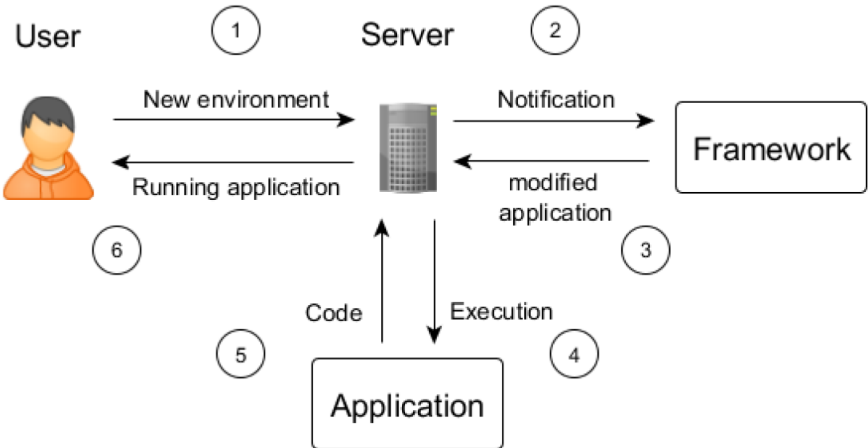


Figure 2.12: Working process of the simulator, extracted from Duhoux's master's thesis [25]

3. once the framework finishes treating the information, it will send back a modified application to the server with the activated features and needed transitions;

4. the server will execute the application. This execution will create the entire application;

5. the application will send back its source code;

6. the server responds to the client by sending the resulting execution.

Chapter 3

Case study

A case study will be explained in this chapter in order to help understanding context-aware applications. The examples in Chapter 5 will be based on this specific case study.

Our chosen case study is an application for a *Risk Information System* (RIS). The goal of such a system is to inform citizens about possible risks, hazards and emergencies (earthquakes, floods,...). Another purpose is to provide generic advice on how to protect themselves and others in case of such situations. This case study was mainly inspired by a Belgian government website [1] and will be designed to fit the architecture presented in Section 2.4, and to introduce some more context-oriented aspects to it.

First of all, I will give a more specific definition of *RIS*. Afterwards, I will describe the functionalities that the application needs to have. Then, I will expose the different contexts and features. Following that, a list of dependencies between contexts and features will be given so that I can conclude with a running example.

3.1 Risk Information System

A *RIS* is a system that informs when a risk (earthquake, flood, woodfire, ...) is detected and instructs the citizens on which actions they must undertake before, during and/or after this risk, in order to guarantee their safety. The instructions must be dependent on the user age, the weather, the status of the risk (before, during, after) and other contexts.
A risk can be active or passive. An active risk is when a risk appears, for instance, when an earthquake is detected. A passive risk is when the citizens want to

[1] `https://www.info-risques.be/en`

look for some instructions for self-protection, for instance, they search for instructions in case of earthquake but there is no earthquake sensed or expected. To simplify, we will use the word *emergency* for an active risk and *risk* for a passive risk.

3.2 Functionalities

Here you can find a list of functionalities of a *RIS*. Obviously, this list is not exhaustive, and many more functionalities can be added to it (for example, we can add more types of risks).

- Every user has a profile where he can enter his age, location, risk concerns and vicinity
- The application has instructions, descriptions and information for the following risks: earthquake, heat wave, wood fire, gas leak and domestic fire
- Every emergency has a status (before, during, after), an impact zone, a severity (high, medium, low) and a type (earthquake, heat wave,...)
- Depending on the type of risk, the impacted zone can be represented by a polygon or a circle (the epicentre of an earthquake + a radius)
- Depending on the type of risk, the severity can be represented by different scales. Richter's scale can be used for earthquakes and a standard scale (low, medium, high) for other types of risks.
- Instructions given to the users may depend on the type of user (child, adult or senior)
- Instructions given to the users may depend on the status of the risk/emergency
- Instructions given to the users may depend on whether it is a risk or an emergency
- Instructions given to the users may depend on the vicinity of the user (e.g. near a river, forest, ...)
- Instructions given to the users may depend on the availability of rescue services (e.g. availability of ambulance, police and fire brigade)
- The method of notification of an emergency depends on the severity of this emergency. For example, if a user is facing an earthquake with a high severity, then he will receive a notification written in red that takes all the screen.

- The application can give a navigation route to users so that they can reach a safe zone in case of emergencies
- The navigation itinerary may depend on the weather

3.3 Contexts

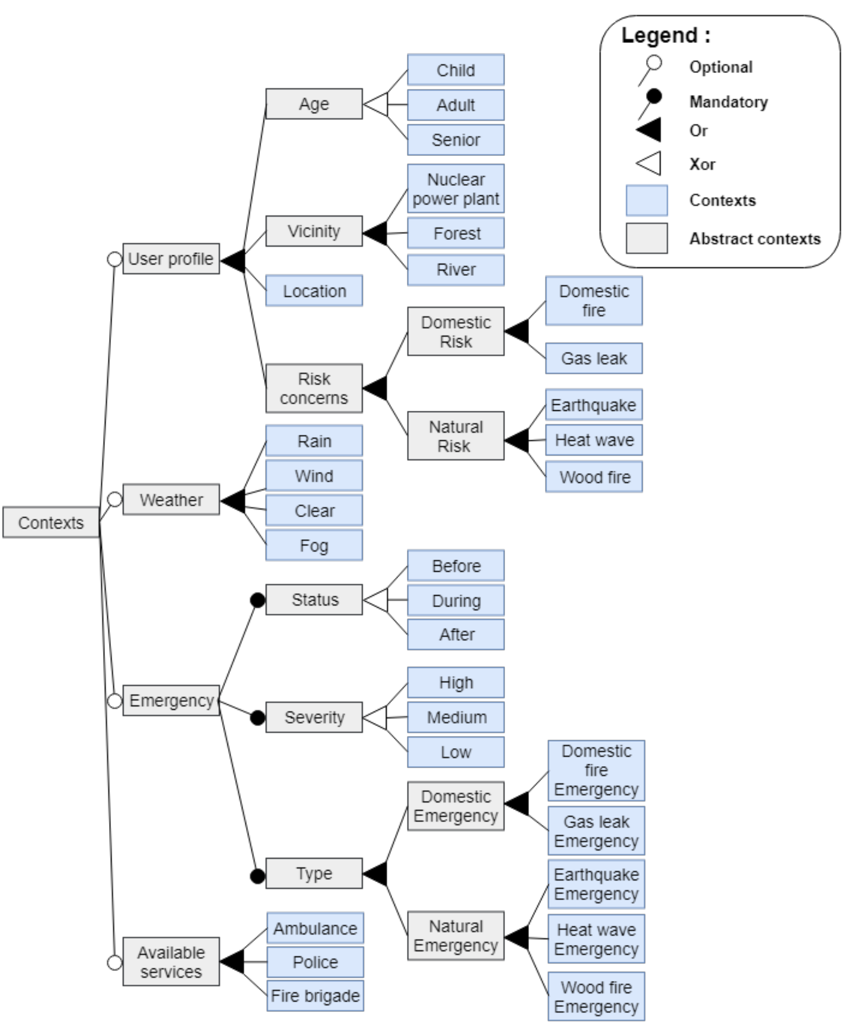


Figure 3.1: Context model of the case study

Figure 3.1 depicts the context model of the *RIS*. In this model, all grey nodes represent abstract contexts (they do not add new functionality), whereas blue nodes represent concrete contexts (they do affect the system behaviour). We can observe that this model has 4 big parts:

1. *User profile* represents the profile of a user who is using the application. With this branch of the model, the application is capable of determining what kind of user is using it and thus, provide instructions depending on the type of the user. As an example, let us imagine that the user is 25 years old, lives in Brussels and has a forest in its vicinity. He is also concerned about domestic fire. The following contexts will be activated: *Adult*, *Forest*, *Location* and *Domestic fire*. In addition to these contexts, their supercontexts will also be activated: *Contexts*, *User profile*, *Age*, *Vicinity*, *Risk concerns* and *Domestic risk*. As the user is concerned only about domestic fire, he will only be able to consult information about this specific risk (but he will still receive instructions about an emergency).

2. *Weather* is an abstract context that allows the system to know the current weather and consequently, the system will adapt the navigation instructions.

3. *Emergency* will be activated when an emergency is sensed. It has a status, a severity and a type. For instance, if an earthquake is happening and has a medium severity, three different contexts of this branch will be activated: *Earthquake Emergency*, *Medium* and *During*. Obviously, their supercontexts will also be activated: *Contexts*, *Emergency*, *Status*, *Severity*, *Type* and *Natural Emergency*. Regardless of the user's declared risk concerns, every user will receive specific instructions about sensed emergency situations.

4. *Available services* is an abstract context that allows the system to know what kinds of services are available. For example, if the application receives the information that no ambulance is available immediately, the *Ambulance* context will be deactivated. On the contrary, if it has the information that an ambulance is free, this context will be activated, as well as its ancestors *Available services* and *Contexts*. Such information may affect specific instructions given to particular users.

3.4 Features

Figure 3.2 shows the feature model of the case study. As for the context model, grey (resp. blue) nodes represent abstract (resp. concrete) features. Abstract features help categorize the features whereas concrete ones adapt the behaviour of the application. The model is subdivided in 8 parts:

1. *Emergency information* contains two abstract children. *Display severity* is the action to display the severity with a specific scale (Standard or Richter). *Show impacted zone* is the action of showing the impacted area in a specific

shape (circle, polygon or unknown). An example is when an earthquake happens, it will activate the *Richter* feature to show its severity with the Richter scale. The *Circle impact zone* will also be activated since an impact zone of an earthquake is given by an epicentre and a radius.

2. *Intervention sensitive* represents the action of updating the instructions according to the availability of rescue services. If no ambulance is available, the intervention time will be longer than usual. Consequently, instructions have to be updated.

3. *User sensitive*: the activation of the concrete features in this branch will update the given instructions. For example, if the *Senior friendly* feature is activated, the given instructions will be specific to a senior person.

4. *Navigation* is the feature that gives the possibility to have a navigation route in case of emergency: instructions on how to reach the nearest safe zone, a map that shows the route,...

5. *Notify*: depending on the severity of the emergency, a child of this abstract feature will be activated. The format of the notification will differ from one child to another. For example, if the *Alert* feature is activated, it means that a very serious emergency is happening. The format of the notification will be a message written in red that covers the entire screen.

6. *Profile edition* is the action of updating the profile of a user. This abstract feature and all its children are mandatory.

7. *Risk description*: this group of features allows to show the description of a risk.

8. *Instructions in case of*: activating one of the concrete feature will give the possibility of displaying instructions in specific cases. Here, we separate each disaster into two sub-branches to achieve a clean separation between emergency and risk (instructions could be the same but it is not for sure). An example of instruction for a *During Earthquake Risk* can be "During an earthquake, it is safer to hide below a table.", whereas an instruction for a *During Earthquake Emergency* can be "Hide below the table but be aware that there is also a storm". In fact, instructions for a risk are quite general. On the contrary, instructions for an emergency are more precise and specific. In order to improve the readability of the model, we decided to show only one risk completely (earthquake). Obviously, other risks also have the same kind of structure. Notice that *Instructions in case of* is a mandatory abstract feature since it is always possible to have instructions about a risk.

Earthquake Risk has an `OR` relation with its children as a user can see all the instructions about a risk independently of the status. On the contrary, *Earthquake Emergency* has a `XOR` relationship with its children considering that an emergency can only have one particular status at a time.

3.5 Dependencies

Given that this application is designed based on a context-oriented feature oriented programming approach (Section 2.4), the (de)activation of contexts will trigger the (de)activation of features. Figure 3.3 depicts the mapping between contexts and features. A dependency should be read as follow: context *Child* implies the activation of the feature *Child friendly*. Concerning the contexts describing the weather (*Rain*, *Wind*, *Clear* and *Fog*), in the case where they are inactive, the activation of *Location* and *During* will still trigger the activation of the *Navigation* feature.

3.6 Running example

If we link this application to the *adaptation framework* described in Section 2.4, we will obtain a context-aware application built using a context-feature oriented approach. The running example used in Chapter 5 is about the detection of an earthquake.

When a earthquake is detected, the `Discovery context` component translates the sensed data into several contexts: *Earthquake Emergency*, *During* and *Medium*. The framework can interpret this information thanks to the context model (Figure 3.1). Then, the framework tries to activate the newly selected contexts with the `Context Activation` component. Based on the notion of configuration in the feature modelling, the selected contexts is activated if all constraints and all their intra-dependencies are satisfied. Since these contexts has no intra-dependencies and no special constraints. They will be activated. The newly activated contexts are sent to the `Feature Selection`. This component checks and selects the corresponding features: *Richter*, *Circle impact zone*, *During Earthquake Emergency* and *Warn* thanks to the mapping between contexts and features (Figure 3.3). Similar to the `Context Activation` component, the `Feature Activation` determines the satisfiability of the new configuration of the feature model 3.2. Finally, if the feature model is in a valid configuration, the framework will alter (change a part of the code) the behaviour of the program thanks to the `Feature Execution` component. Users will receive a notification with the specific instructions to follow.

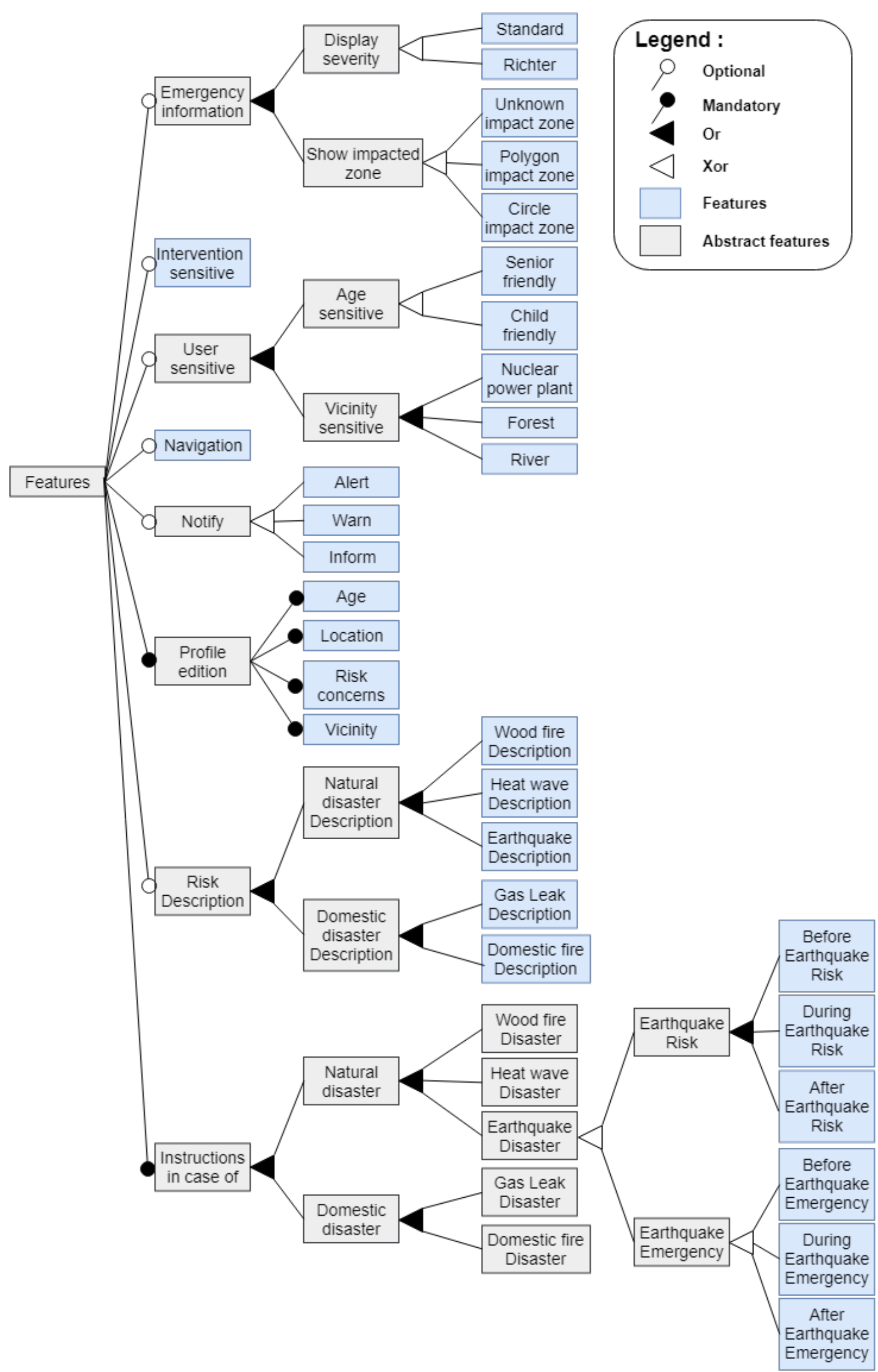


Figure 3.2: Feature model of the case study

Figure 3.3: Mapping between contexts and features of our *RIS* application

Chapter 4

Problem statement

As we can notice in the case study of Chapter 3, using an implementation architecture (Section 2.4), in which the notion of contexts is clearly separated from the notion of features brings a lot of dependencies. As the number of contexts and features can increase depending on the need of the developer (for example, the types of risks he wants his application to cover, the number of characteristics of an emergency, the depth of an user profile,...), we are facing an exponential number of combinations of these entities (contexts and features). Therefore debugging such an application seems complicated for programmers because they must visualise which contexts and features are (de)activated and how they interact with the system.

As an example, a developer has implemented the case study of Chapter 3 but he is not sure of his implementation. When his application has detected an earthquake, he finds out that the given instructions are not the expected ones. The only way for him to find out the bug is by going into his code, and by verifying the declaration of contexts and features, the mapping between contexts and features as well as the dependencies between features and the code. The more contexts and features are declared in his application, the more it will be hard for him to keep track of every (de)activation and dependency.

Chapter 5

Solution

In order to solve the problem described in Chapter 4, this master's thesis proposes a visualisation tool displaying the declarations of context and feature models as well as their inter- and intra-dependencies. To handle the runtime debugging of context-aware applications (that use the programming approach described in Section 2.4), the tool also offers the visualisation of the dynamic configuration of these models. In addition, this visualisation can show to the programmers which features adapt which classes in order to increase the comprehension of the programmers. Finally, some filters are also added to improve the developer's view when applications become complex.

To begin, we will discuss about the choice of the programming language for implementing this tool. Following this, a description of the chosen language GoJS will be given. Finally, we will talk in more detail about the visualisation tool itself.

5.1 Choice of the programming language

In order to select the ideal programming language for our tool, we have taken into account several points:

- The RELEASeD research group headed by Prof. K. Mens at UCLouvain has already implemented a first tool (Section 2.5.2) to help developers debugging applications that uses the context-aware feature-oriented programming approach. The language used at this moment is JavaScript, more precisely, the library `vis.js` [1]. This library has the particularity of being dynamic and having a browser based visualisation;

[1] `http://visjs.org/`

- As one of the main requirements of this visualisation tool is to represent the context and feature models, the chosen language must have a feature to implement tree graphs.

From these points, we decided to continue with the use of JavaScript to implement this web visualisation tool. We also retained 6 JavaScript libraries: `vis.js`, `Treant.js` [2], `Cytoscape.js` [3], `D3.js` [4], `GoJS` [5] and `jsPlumb` [6].

`Treant.js` is a library specialized in creating tree structure charts, however, these charts are static and are not suited for complex tree graphs. `Cystoscape.js`, `jsPlumb` and `vis.js` are three libraries designed for graphs, nevertheless, they are limited in options for tree graphs. Moreover, representing feature model constraints seemed difficult. Finally, the two remaining libraries (`D3.js` and `GoJS`) are two big libraries and have all the options that we needed to design our models. We decided to opt for the `GoJS` library since this library is specially designed for diagrams. In addition, it is well documented [7] and has a lot of samples that are very interesting in our case. Even tough `D3.js` has all these characteristics, it contains much more than what we needed, and is not specially designed for tree graphs which would lead to more work from our part.

5.2 GoJS

GoJS is a JavaScript library containing a big number of features for implementing custom interactive diagrams and complex visualisation across web browsers and platforms. Thanks to its customisable templates and layouts, the difficulty for constructing diagrams with complex nodes, links and groups is quite low.

Tutorial on how to make a diagram

Every graph in GoJS is based on an object called Diagram. Every Diagram is hosted by an HTML Div element, GoJS will manage the contents of that Div element. Information that we need to provide are the position, size and style of the Div (as it is done for every HTML element). This Diagram will add a Canvas element where everything will be drawn inside. This Canvas element is automatically sized based on the Div. One of the biggest problems of GoJS is the impossibility to have a dynamic height for the Canvas. Indeed, changing the size of the web browser

[2] `http://fperucic.github.io/treant-js/`
[3] `http://js.cytoscape.org/`
[4] `https://d3js.org/`
[5] `https://gojs.net/latest/index.html`
[6] `https://jsplumbtoolkit.com/`
[7] GoJS has its own forum for helping new developers `https://forum.nwoods.com/c/gojs`

will not update the height of the Diagram (nevertheless, it is possible to have a dynamic width).
We can create a Diagram in JavaScript with a reference to that Div element. An example is given in Listing 5.1 where we declare a GoJS diagram linked to the HTML Div `myDiagramDiv`.

```
1 var diagram = new go.Diagram("myDiagramDiv");
```

Listing 5.1: creation of a GoJS diagram

Note that all references in JavaScript code to GoJS types are prefixed with "`go.`".

Several properties of a GoJS diagram

In the implementation of the solution several properties of a GoJS Diagram are used. Here are some of the most important ones:

- **layout**: This property allows to get or set the `Layout` used to position the nodes and links in the `Diagram`. Several layouts are already defined by the authors including a tree layout.
- **model**: Gets or sets the `Model` holding data corresponding to the nodes and links of the `Diagram`. Replacing this property causes a change of all the `Nodes` and `Links` (deletion + re-creation). In a `Model`, it is possible to define different templates of `Nodes`, `Links` and `Groups`.

In Listing 5.2, we have a code that creates a tree graph with two shapes of node possible (circle and rectangle) and two types of links (the end of the link can either be a triangle or a circle). The resulting graph is shown in Figure 5.1. If there is no category specified for a node or a link, a default one is chosen. For example, for the last declared link, there is no category specified which results in a basic link. As we can observe in this code, there is a possibility of declaring bindings. For example, in the node template for a circle node, we have bound the attribute "name" of the node data to "text" of a node object. In a similar way, we could have added a binding for the shape of the node so that we do not have to create another node template.

```
1 var maker = go.GraphObject.make;
2 // linking the div to the diagram
3 var myDiagram = maker(go.Diagram, "myDiagramDiv");
4
5 // define a TreeLayout that flows from top to bottom
6 myDiagram.layout = maker(go.TreeLayout, { angle: 90, layerSpacing:
    50 });
```

```

// circle node
myDiagram.nodeTemplateMap.add("Circle",
  maker(go.Node, "Auto",
    maker(go.Shape,
      {
        figure: "circle",
        fill: "white"
      }),
    maker(go.TextBlock,
      { margin: 5 },
      new go.Binding("text", "name"))
));

// rectangle node
myDiagram.nodeTemplateMap.add("Rectangle",
  maker(go.Node, "Auto",
    maker(go.Shape,
      {
        figure: "rectangle",
        fill: "white"
      }),
    maker(go.TextBlock,
      { margin: 5 },
      new go.Binding("text", "name"))
));

// adding different link template
myDiagram.linkTemplateMap.add("Triangle",
  maker(go.Link,
    maker(go.Shape),  // the "from" end arrowhead
    maker(go.Shape,   // the "to" end arrowhead
      { toArrow: "Triangle", fill: "black", scale: 2 })
));

myDiagram.linkTemplateMap.add("Circle",
  maker(go.Link,
    maker(go.Shape),  // the "from" end arrowhead
    maker(go.Shape,   // the "to" end arrowhead
      { toArrow: "Circle", fill: "white", scale: 2 })
));

// adding graph data
var model = maker(go.GraphLinksModel);
model.nodeDataArray =
[
  { key: "1", name: "rectangle1", category: "Rectangle" },
  { key: "2", name: "rectangle2", category: "Rectangle" },
  { key: "3", name: "Circle1", category: "Circle" },
```

```
56   { key: "4", name: "Circle2", category: "Circle" }
57 ];
58 model.linkDataArray =
59 [
60   { from: "1", to: "2", category: "Triangle" },
61   { from: "1", to: "3", category: "Circle" },
62   { from: "2", to: "4" },
63 ];
64 myDiagram.model = model;
```

Listing 5.2: Creation of a tree graph with different categories of nodes and links

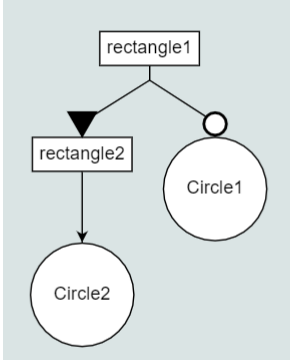


Figure 5.1: Resulting graph from the code in Listing 5.2

5.3 The visualisation tool

For the implementation of context-aware applications that uses the programming approach described in Section 2.4, application developers need to provide several components of the application layer such as `Context declarations`, `Context dependency declarations`, `Context-feature mapping`, `Feature declarations`, `Feature dependency declarations`, `Context-feature interdependency declarations` and `Feature definitions`.

Each time the user declares an entity (context or feature), information is sent to the tool thanks to a server. The same goes for the declaration of dependencies and constraints between entities. Once information is received by the tool, it creates the entity immediately.

Each time the `Context Activation` and the `Feature Activation` component of the `Handling layer` (de)activates an entity, the information is sent to the tool

and stored into a queue. The need of this queue will be explained in Section 5.3.3.

In order to illustrate our tool, an application of our running example respecting the application layer described in Section 2.4 has been implemented. The Figure 5.2 depicts the tool in a state where all information is received from the *RIS* application. The tool contains three main parts: *Context and feature model*, *Filters* and *Configuration*. This section will describe these three parts in detail.

5.3.1 Context and feature model

As is shown in Figure 5.2, the *Context and feature model* part is divided into three components: the context model, the feature model and a class diagram component.

Context model

This model allows us to have a clear vision of all the different contexts that have an impact on the application. Figure 5.3 shows the resulting context model based on the running example given in Section 3.6 (An earthquake is detected by sensors). As depicted in the figure, a node can have two different colours. If the context is activated (resp. deactivated), its colour will be green (resp. red). Furthermore, a node has several properties. Depending on the action done on the graph or information given by the server, these properties can change. In Listing 5.3, we have the representation of the *Emergency* abstract context node. Each context node has a key property which is the concatenation of its name and type. The *group* property allows GoJS to know to which graph this node belongs. The attribute *isParentPresent* tells us if its parent node is hidden in the graph or not. Whereas *isExpanded* tells us about the presence of its children. Their utility will be explained afterwards. The boolean *root* indicates whether the node is the root or not. As a node is represented by a GoJS node, other properties can be found in the API [8] of this library. One important property coming from this API is *visible*, it gives us the possibility to (un)hide a node.

```
1 {
2   key: "EmergencyContext",
3   name: "Emergency",
4   group: "Contexts",
5   active: true,
6   isParentPresent: true,
7   isExpanded: true,
8   root: false
9 }
```

Listing 5.3: Declaration of a context node

[8] https://gojs.net/latest/api/symbols/Node.html

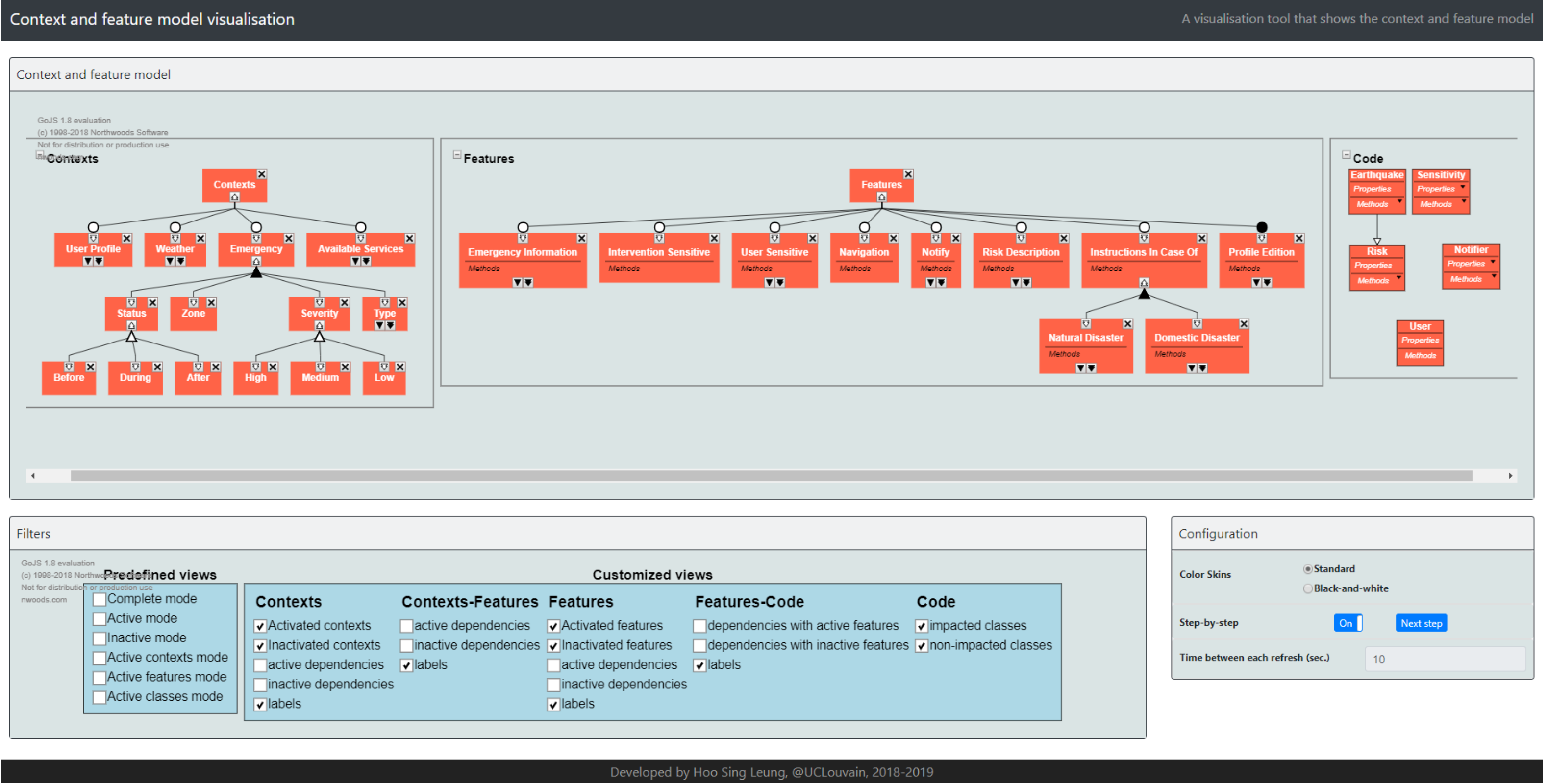


Figure 5.2: Snapshot of the full visualisation tool

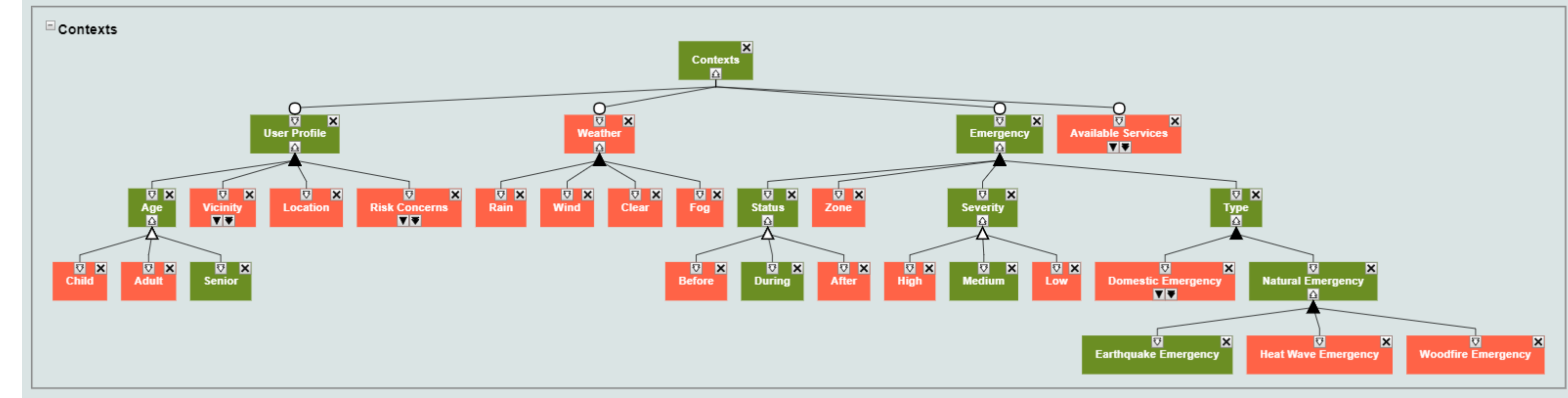


Figure 5.3: Resulting context model of the tool based on the running example

In the same figure, we can observe different kinds of buttons. These are designed to help with the scalability of the graph.

- **RemoveButton** allows us to remove a node that we are not interested in at the moment. By clicking on this button, the property *visible* is set to false and consequently the node will become hidden. The property *isParentPresent* of its first level children will be set to false. In a similar way, the property *isExpanded* of the direct parent is also set to false.

- **ParentsCollapserButton** allows us to hide all the upper part of the graph. The node on which this button is clicked on will become the top of the graph. In fact, it will set the property *visible* of all the nodes except the current node and all its descendants to false. The property *isParentPresent* of this context is set to false.

- **DirectParentExpanderButton** allows us to unhide the parent of the node and all the other children of this parent node (by changing the value of the property *visible*). The property *isParentPresent* of the parent node is set to false.

- **AllParentsExpanderButton** allows us to unhide all the ancestor nodes until the root is reached. The property *isParentPresent* of this node is set to true.

- **ChildrenCollapserButton** allows us to collapse all the current node's children. The property *isExpanded* is set to false. Every children nodes have their property *visible* set to false.

- **DirectChildrenExpanderButton** allows us to expand the direct children of the node (first level of children). The property *isExpanded* of this node is set to true, whereas its direct children will have theirs set to false.

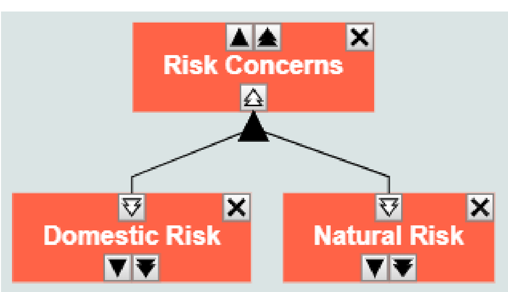
- **AllChildrenExpanderButton** allows us to expand all the direct and indirect children of the nodes until the leaves are reached (all levels of successors). The property *isExpanded* of this node is set to true.

Setting the value of the property *isParentPresent* to true (resp. false) will display **ParentsCollapserButton** (resp. **AllParentsExpanderButton** and **DirectParentExpanderButton**). Similarly, setting the value of the property *isExpanded* to true (resp. false) will display **ChildrenCollapserButton** (resp. **AllChildrenExpanderButton** and **DirectChildrenExpanderButton**).

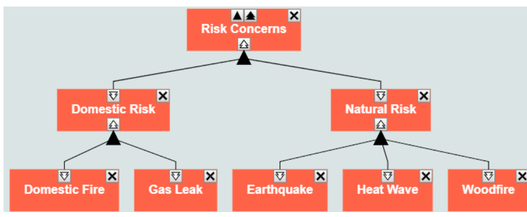
The position of these buttons on a node gives us information. If the button is on the upper (resp. lower) side of the node, it means that the button will trigger changes on the upper (resp. lower) part of the graph. The shape inside the button also gives information. A single triangle means a change on one level of parents or children, whereas a double triangle means all levels of parents or children. The Figure 5.4 shows different states of the node *Risk concerns* obtained by clicking on the different buttons. Figure 5.4a is the result of both clicking on the **ParentsCollapserButton** and **ChildenCollapserButton**. As a result, the only context that is shown in the graph is *Risk concerns*. From this figure, if we click on **DirectChildrenExpanderButton**, only the direct children *Domestic Risk* and *Natural Risk* are expanded (Figure 5.4b). From this state, if we want to unhide more level of children, we will have to use the buttons from these two contexts. In the case where we would have clicked on **AllChildrenExpanderButton** instead of **DirectChildrenExpanderButton** on the *Risk concerns* node, the resulting context model would be the one shown in Figure 5.4c. Concerning the **DirectParentExpanderButton** and **AllParentsExpanderButton**, the result of clicking on these buttons are shown respectively in Figure 5.4d and Figure 5.4e. The case where the context *Risk concerns* is removed, is depicted in Figure 5.4f. We can observe that its direct children (*Domestic risk* and *Natural Risk*) are separated from the rest of the graph. In order to re-show the removed node, we can either do it by using the parent expander buttons of the children (*Domestic risk* and *Natural Risk*) or the children expander buttons of the parent node (*User profile*). Consequently, the children will be re-attached to the rest of the graph.

Different type of links are designed to represent the constraints and dependency of a context (feature) model. Currently, 4 constraints are implemented: Or, Xor, Mandatory and Alternative. Their representation are shown in Figure 5.5. A Mandatory (resp. Alternative) constraint is represented by an arrow with a black (resp. white) filled circle as the "to" end arrowhead. Whereas an Or (resp. Xor) constraint has a black (resp. white) triangle as its "from" end arrowhead. Adding new constraints will not be a problem since we will just need to add the lines of codes shown in Listing 5.4. The tool also gives the possibility of creating dependencies (requirement, inclusion,...), an example is shown in Figure 5.6. In contrast to the constraints, we do not need to declare each kind of dependency. Indeed, all the

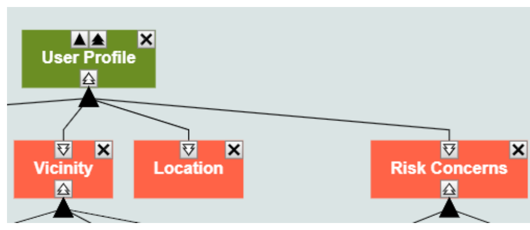
(a) All parents and children collapsed



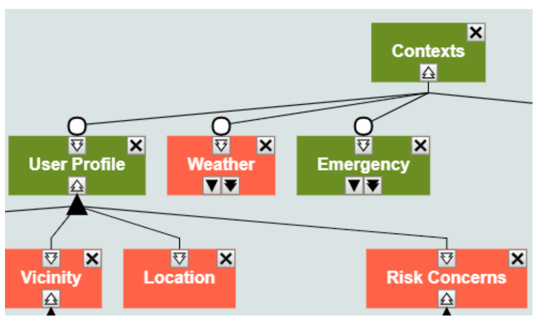
(b) Direct children expanded



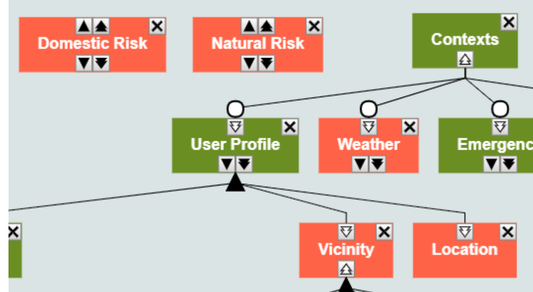
(c) All children expanded



(d) Direct parent expanded



(e) All parents expanded

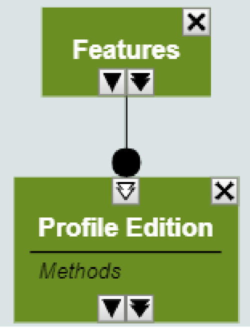


(f) *Risk concerns* node removed

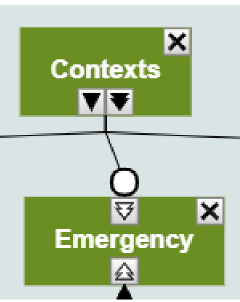
Figure 5.4: Different states obtained by playing with the buttons of the abstract context *Risk concerns*

dependencies are represented in the same way. They only differ in their label which is the name of the dependency. An example of declaration of constraints and dependencies is shown in Listing 5.5. The code shows the declaration of the Xor constraint between *Age* and *Child* as well as the dependency shown in Figure 5.6.

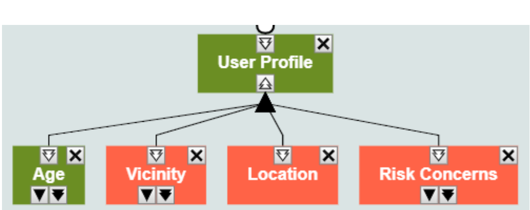
```
1
2 myDiagram.linkTemplateMap.add("newConstraint", // constraint name
3   maker(go.Link,
4   maker(go.Shape),   // the "from" end arrowhead
5   maker(go.Shape,    // the "to" end arrowhead
6     {
```



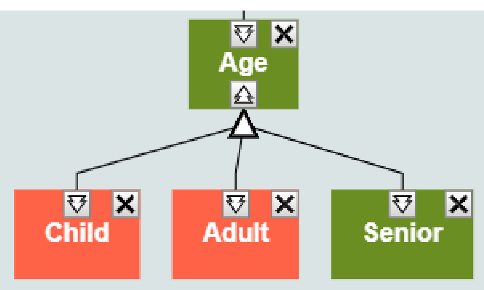
(a) Mandatory constraint between *Features* and *Profile Edition*



(b) Alternative (Optional) constraint between *Contexts* and *Emergency*



(c) Or constraint between *User Profile* and its children



(d) Xor constraint between *Age* and its children

Figure 5.5: Different types of constraints represented in the tool

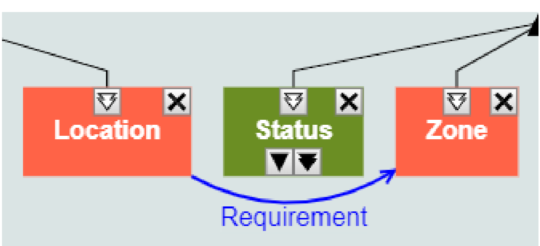


Figure 5.6: Example of intra-dependencies

```
7         toArrow: "Triangle", // shape to represent the constraint
8         fill: "black",      // colour to represent the constraint
9         scale: 2
10      })
11
12      // Depending on the wanted representation, we will either
           change the from end arrowhead or the to.
```

```
13 ));
```

Listing 5.4: Lines of code for creating a new type of constraint

```
1  { // Declaration of a constraint
2    key: AgeChild,
3    from: AgeContext,
4    to: ChildContext,
5    group: Contexts,
6    category: Xor  // Or, Mandatory, Alternative
7  }
8
9  {
10   // Declaration of a dependency
11   key: LocationZone,
12   from: LocationContext,
13   to: ZoneContext,
14   group: Contexts,
15   category: "Dependency",
16   text: "Requirement"     // text is bound to the label of the
       link. The content of text is the name of the Dependency
17 }
```

Listing 5.5: Declaration of a constraint and dependency

Feature model

Figure 5.7 shows the resulting feature model of the running example. For a better readability, several inactive nodes are hidden. The feature model is basically designed the same way as the context model. Every aspect described for the context model can be used for the feature model. The only differences between these two kinds of nodes are the *group* property and a new property entitled *methods* (Listing 5.6). This last property allows the developer to have a vision on the methods the feature alters (modify in a class). Two very similar representations of a feature is shown in Figure 5.8. The figure on the left depicts a feature that has no methods to alter. Generally, it represents an abstract feature. We can deduce it thanks to the fact that it has no little triangle button on the right of *Methods*. The figure on the right side, on the contrary, shows a feature that has four methods to alter. The class on which the feature applies is given by the dependencies between features and the class diagram. This kind of dependencies will be explained in Section 5.3.1.

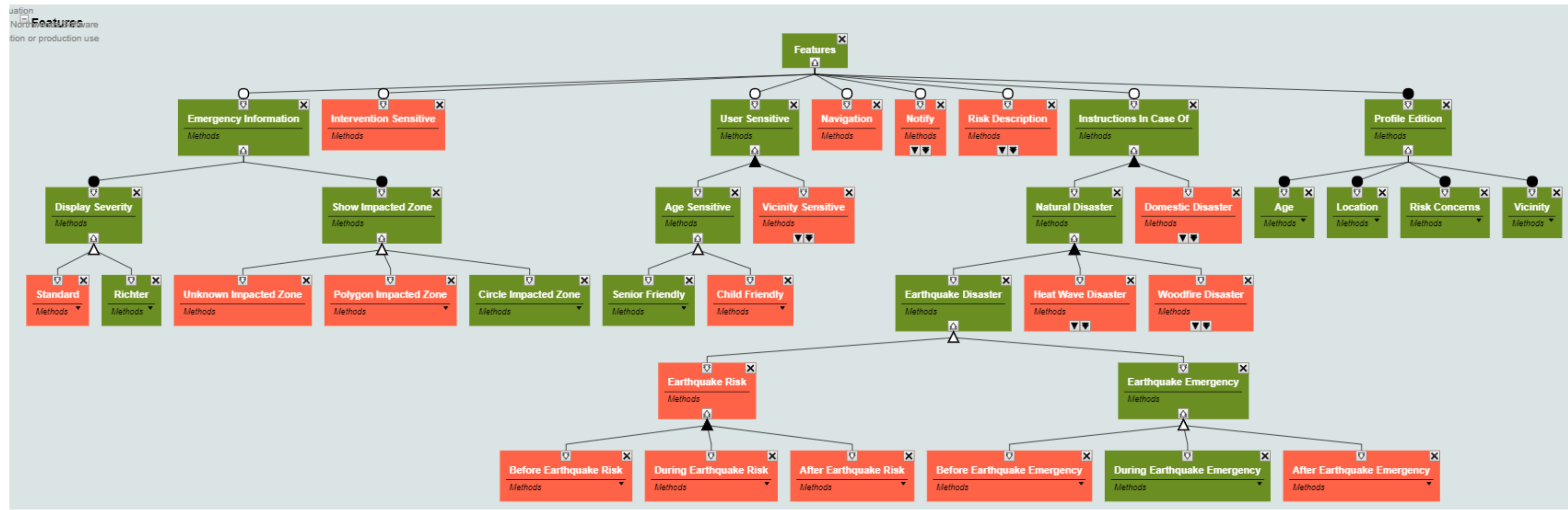
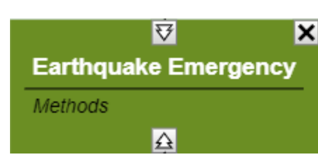


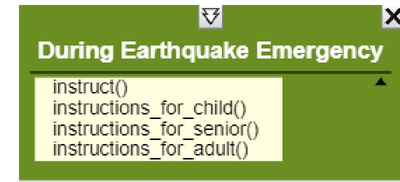
Figure 5.7: Resulting feature model of the tool based on the running example

```
1  {
2    key: "DuringEarthquakeEmergencyFeature",
3    name: "During Earthquake Emergency",
4    group: "Features",    // gives information that it is for the
        feature model
5    category: "Feature",  // gives information that we do not use
        the default type of node anymore
6    active: true,
7    isParentPresent: true,
8    isExpanded: true,
9    root: false,
10   firstChild: false,
11   methods: [
12         {name: "instruct", type: "String", visibility: "public"},
13         {name: "instructions_for_childt", type: "String",
              visibility: "public"},
14         {name: "instructions_for_senior", type: "String",
              visibility: "public"},
15         {name: "instructions_for_adult", type: "String",
              visibility: "public"}
16       ] // list of methods that will adapt a class
17 }
```

Listing 5.6: Declaration of a feature node



(a) Representation of an abstract feature that has no methods to alter



(b) Representation of a feature with the methods it alters

Figure 5.8: Representation of features

Class diagram

In Figure 5.9, we have a representation of the application class diagram. Similarly to a feature node, there is a possibility to see the properties (instance variables) and instance methods of a class. The *Sensitivity* class have four instance variables and two instance methods, whereas the *Notifier* class has its properties and methods hidden.

For this diagram, we tried to reproduce every aspect of an UML class diagram [9]. Currently, this diagram only shows the Ruby classes that are adapted by a feature.

[9] https://en.wikipedia.org/wiki/Class_diagram

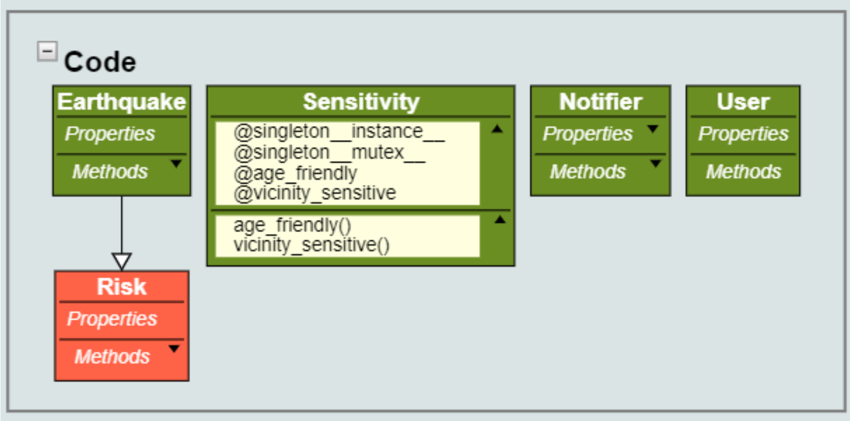


Figure 5.9: Representation of the application class diagram

It will be subject to a future work to have a complete class diagram. As a matter of fact, several important classes are lacking in this diagram for the comprehension of the application and for debugging. As a example, a class *Point* is created in order to represent a location. An earthquake uses this object in order to represent its epicentre. However, this class is not depicted in the diagram. In fact, the diagram lacks all the classes that would not be in any case adapted by a feature.

The declaration of this kind of node is shown in Listing 5.7. Whereas, Listing 5.8 shows how to declare a relationship between two classes.

```
1  {
2    key: "EarthquakeCode",
3    category: "code",        // specific type of node
4    group: "Code",       // for the class diagram
5    active: true,
6    name: "Earthquake",
7    properties: [
8        { name: "prop1", type: "String", visibility: "public" },
9        // example of property declaration
10     ],
11   methods: [
12       { name: "instruct", type: "String", visibility: "public" },
13       {
14         name: "inform_user",
15         type: "String",
16         parameters: [{name: "user", type: "User"}],
17         visibility:"public"
18       }
19     ]
```

```
20 }
```

Listing 5.7: Declaration of a class

```
1 {
2   from: "EarthquakeCode", // key of Earthquake class
3   to: "RiskCode",      // key of Risk class
4   relationship: "generalization",  // type of relationship, could
       be Association, Realization, Dependency, Aggregation or
       Composition.
5   category: "UML"
6 },
```

Listing 5.8: Declaration of UML relationships

Context-Feature and Feature-Code dependencies

As described in Section 2.4, the (de)activation of contexts will trigger the (de)activation of some features which will at their turn adapt different classes. In order to visualise these kind of dependencies, two types of links are designed in the tool: *Contexts-Features* and *Features-Code*. Their structure is nearly the same as the dependency represented in Listing 5.5. The difference lies in the removal of the *group* property since they do not belong to a specific graph. the property *category* is a string representing the name of these new kinds of dependencies. For a *Contexts-Features* (resp. *Features-Code*), the origin node is a context (resp. feature) whereas the destination node is a feature (resp. class). The label of these links tells the action that the origin node will do to the destination node. In Figure 5.10, an example for each of these types of dependencies is shown. This figure tells us that the activation of the context *Senior* will trigger the activation of the feature *Senior Friendly*. This same feature adapts the class *Sensitivity*.

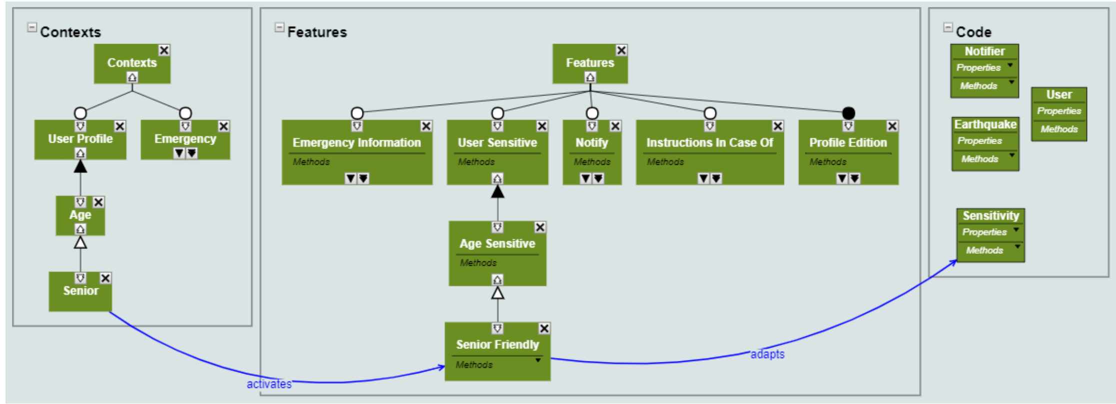


Figure 5.10: Example of *Contexts-Features* and *Features-Code* dependencies

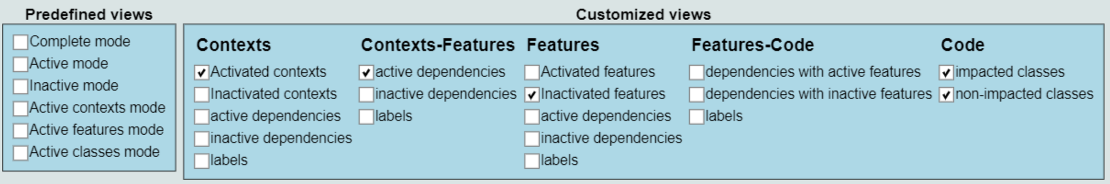


Figure 5.11: Filters for the context and feature model

5.3.2 Filters

If we combine the complete graph of every components described in Section 5.3.1, the resulting graph will be too large. Therefore, we will have to play with the different buttons of the context and feature nodes so as to obtain a readable graph that fits the browser window. Moreover, it is unusual that a developer needs to work on the complete graph. For these reasons, we decided to add several sets of filters (Figure 5.11).

To customize views, each component has a set of filters. A checkbox that is checked means that the elements it designate are shown in the graph. For example, the customised configuration shown in the figure tells that only activated contexts, inactivated features, all classes of the class diagram and active *Contexts-Features* dependencies are shown in the tool. An active dependency means that either the origin node or the destination node is activated, whereas inactive dependencies means that both of these nodes are deactivated.

In addition to these filters, several predefined views can be chosen.

- Complete mode: every element is shown in the graph.
- Active mode: only active elements are shown in the graph.
- Inactive mode: only inactive elements are shown in the graph.
- Active contexts mode: every activated context is shown as well as features depending on them. Moreover, classes that are adapted by the selected features are also shown.
- Active features mode: Every activated feature is shown as well as contexts and classes that are linked to them.
- Active classes mode: Every adapted class is shown as well as features that cause their adaptation. Contexts that can activate these features are also shown.

Figure 5.13 shows an example of a predefined view (i.e. Active mode). This selection triggers the selection of a corresponding set of filters in the customized part. The selection of an additional filter (for example, inactivated features) will automatically uncheck the predefined view. Moreover, it is impossible to select more than one predefined view at a time.

In addition to these filters, a system of highlighting is implemented. If we click on a node (context, feature or class) and this one has shown dependencies, every node that are reachable from it will be highlighted. In the figure, *Senior* context was clicked on, if we follow the dependencies, *Senior Friendly* feature and *Sensitivity* class are reachable. Consequently, they are highlighted. This functionality allows us to visualise immediately the impact of a node.

5.3.3 Configuration

This part of the tool depicted in Figure 5.12 is separated in two different functionalities. A first one is the possibility to change the skin of the tool. By doing so, it triggers a change of colour for the tool. Figure 5.14 shows the selection of the black-and-white skin. Adding a new skin or modifying a skin will not be a problem since we will just have to add (or modify) an option in the updateColor function shown in Listing 5.9.

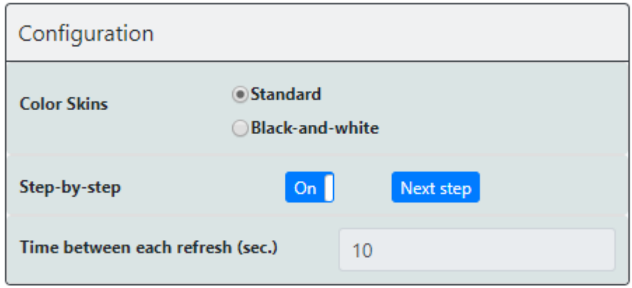


Figure 5.12: Configuration part of the tool

The other functionality is more essential to the developer. It is a widget that allows the programmer to follow each (de)activation of entities (contexts and features). As introduced in the beginning of Section 5.3, every data of entities (de)activation is stored into a queue. If the option step-by-step is selected, every time that we click on the next step button, an element of this queue is popped, consequently, the (de)activation of an entity is depicted in the graphs. On the contrary, if the step-by-step option is not selected, a timer is launched. Every time the timer reaches zero, an element of the queue is popped and the timer is refreshed. Once the queue is empty, a pop-up message will appear.

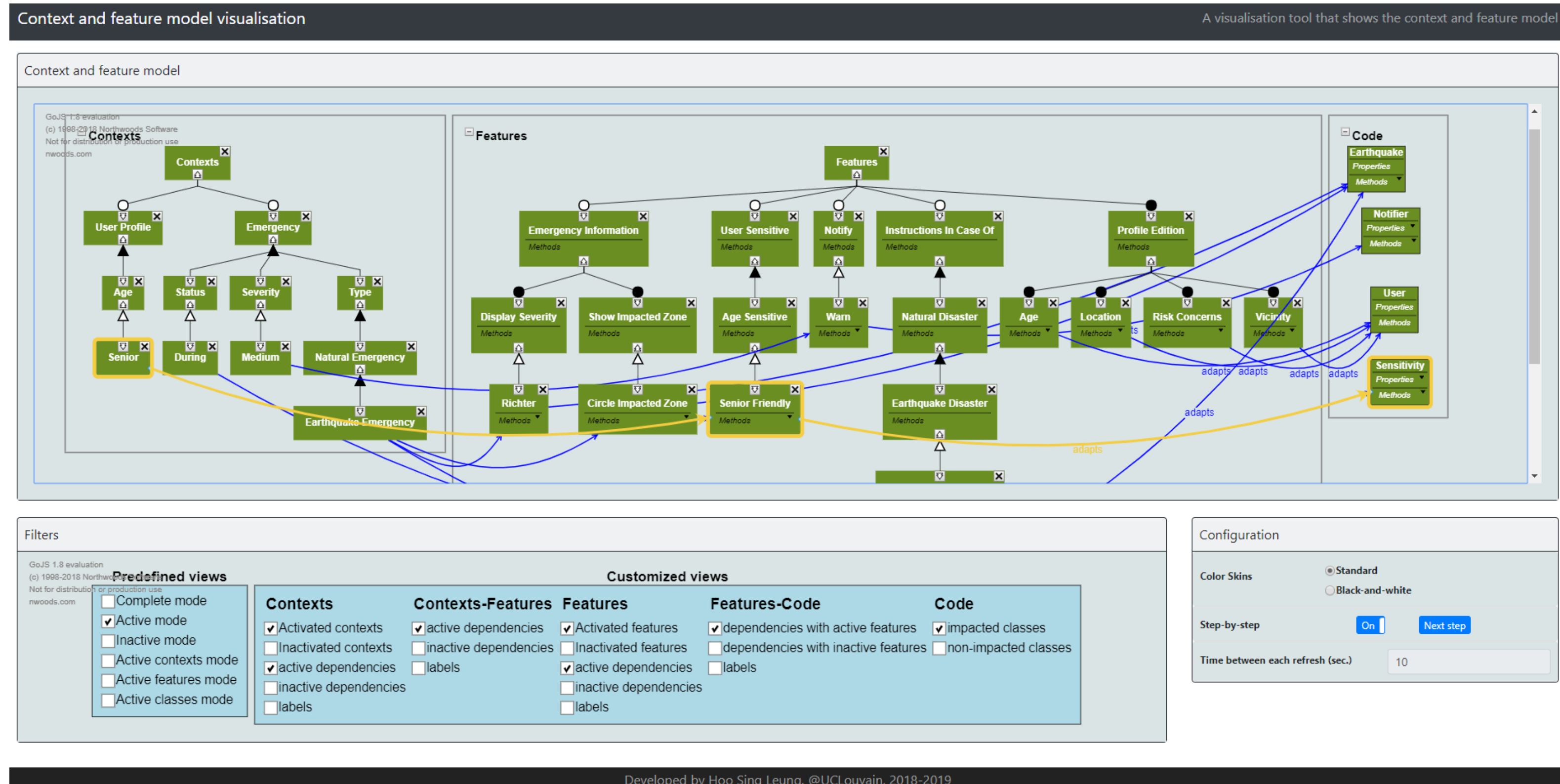


Figure 5.13: Snapshot of the tool in active mode

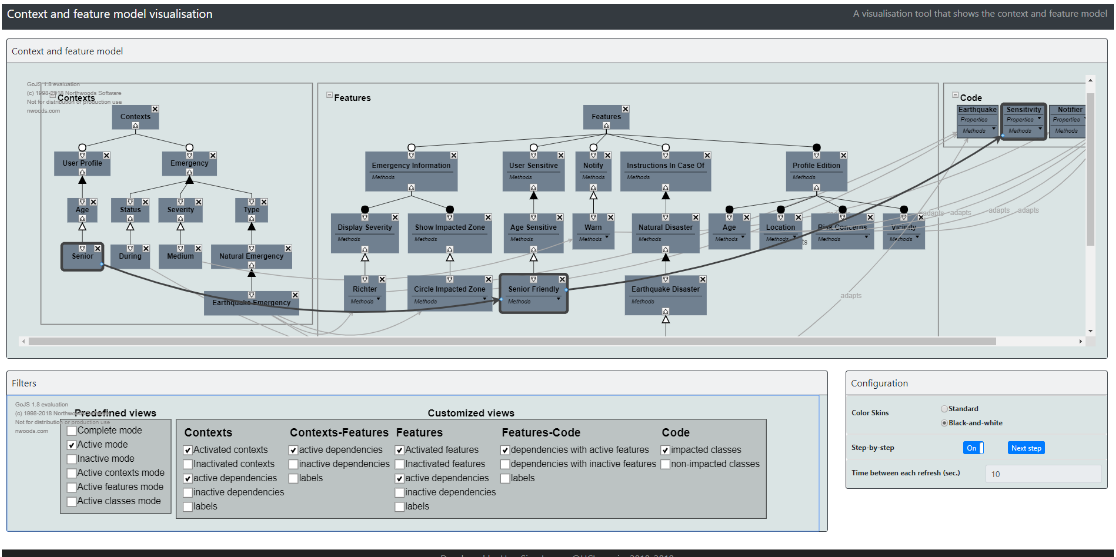


Figure 5.14: Tool in the black-and-white skin

```
1  function updateColor(type) {
2    switch (type) {
3      case "standard":
4        ACTIVE = "#6B8E23";
5        INACTIVE = "#FF6347";
6        BACKGROUND = "lightblue";
7        STROKE = "white";
8        SELECTED = "#f4cd41";
9        NOTSELECTED = "blue";
10       break;
11     case "black-and-white":
12       ACTIVE = "#708090";
13       INACTIVE = "#f2f1ef";
14       BACKGROUND = "rgba(128,128,128,0.33)";
15       STROKE = "black";
16       SELECTED = "#424344";
17       NOTSELECTED = "#a8a8a8";
18       break;
19   }
20 }
```

Listing 5.9: function updateColor

Chapter 6

Validation

To assess the usefulness and the understandability of the visualisation tool proposed in Chapter 5. We conducted a validation study with the students of the course LINGI2252 Software maintenance and evolution teached by Prof. K. Mens at UCLouvain. This study took place during two times of two-hour classroom sessions, after an introduction session of the programming language: Ruby.
In this chapter, we will first describe the conducted study and then expose the different results.

6.1 Study

The study was separated into two slots of two-hour session with the same students. These ones have significant programming experience. They are either in their first year or second year of their master in computer science or computer science engineering. In the first session, we gave an introduction of the feature-oriented context-oriented approach such as to make them ready for implementing a little context-aware system. For the second session, we asked the students to add some contexts and features to an already implemented *RIS* application. In this same session, they could use the visualisation tool to help themselves. To avoid bias, the students were told that these classroom sessions did not make part of the course evaluation but were only dedicated for research. In addition, all answers from the students were anonymous.

6.1.1 First session

In order to introduce the feature-oriented context aware approach, the following schedule had been followed during the first session:

1. We introduced the architecture presented in Section 2.4.

2. We then described how they can create a context-aware application using the feature-oriented context-oriented programming language. We first showed them a `hello world` COP application and explained the structure of the project. Based on their knowledge of Context-Oriented Programming and the overview of the architecture from the previous point, they needed to try understanding how the application is developed.
3. Following this, we introduced the case study from Chapter 3 and asked them to design and implement a short version of this system. For this version, we considered the earthquake as the only risk and emergency. The user could see all instructions of an earthquake. These instructions were the same independently of its type (risk or emergency). However, the dedicated instructions still needed to depend on the status of the emergency (before, during or after). For example, if an earthquake is planned, the system will only display the *before* instructions. Every instruction is devoted to an adult. For a last feature, we told them that an earthquake (regarded as an emergency) has several characteristics such as a severity based on a Richter's scale and an impact zone defined by an epicentre and a radius.
4. In order to help them, some hints were given. As a first step, they were asked to design the context and feature model.
5. After that, we corrected it together such as everyone had the same ones.
6. At last, they started the implementation with some technical concepts described in a PDF as aid.

Unfortunately, not every student has achieved to implement the small application. This was partly due to some technical errors such as a bad installation of the Ruby language. To address this issue, a corrected version of this application had been given to the students during the second session.

6.1.2 Second session

In this session, 34 students were asked to play the role of potential programmers using the feature-oriented context-aware programming approach and evaluate the usefulness and understandability of the visualisation tool. For this experience, students were split into two groups (A and B) and were asked to conduct two different tasks (task 1 and 2). Group A (resp. B) had to handle task 1 (resp. 2) before handling task 2 (resp. 1). The first task of every group was done without using the visualisation tool, whereas, the second task was done with the help of the tool. After every task, a list of questions were asked regarding that task. The main

purpose of this experiment was to find out whether the tool helped the student in completing such task easily or not. The different steps of this experiment are detailed below.

1. A first set of questions gathered information on the students' software development skills, the group he belonged to and his level of comprehension of the approach described during the first session.

2. We assigned them an initial task depending on their group. For this first task, they were not allowed to use the visualisation tool.
   The tasks were based on a case study. This one was a Risk Information System established from the final result that the students were supposed to achieve at the end of the first session (*RIS* with one type of risk/emergency for an adult user). The goal of these tasks were to extend the application for another kind of risk/emergency: floods. Instructions for this new risk are different from those of an earthquake. Moreover, floods has a standard scale of severity (low, medium, high) and a polygonal impacted zone. For task 1, we asked the students to implement the new characteristics (standard severity and polygonal impacted zone). And for task 2, students needed to implement flood-specific-instructions (for an emergency and a risk).

3. After this initial task, three sets of questions related to their task were asked. The first set was about knowing whether the student achieved the task or not and how much time did he take (in the case were he did not finish, how much more time does he need). The second set was about testing his solution by asking him which contexts and features were activated depending on a specific situation. The last set of questions were the same as the second one, however, students were asked to answer it by using the visualisation tool.

4. Once the students were finished answering the questions, they received their second task to do. This time, they were demanded to use the visualisation tool.

5. After that, the first and third sets of questions of the third point were asked again (still related to their task).

6. Finally, we asked them for their feedback on the visualisation tool. Students were to rate the understandability, the usefulness and the ease of use of the tool on a five-level Likert scale. Moreover, we asked the students what particular functionalities of the tool can improve a developer's experience with the programming approach and what could be improved in the tool.

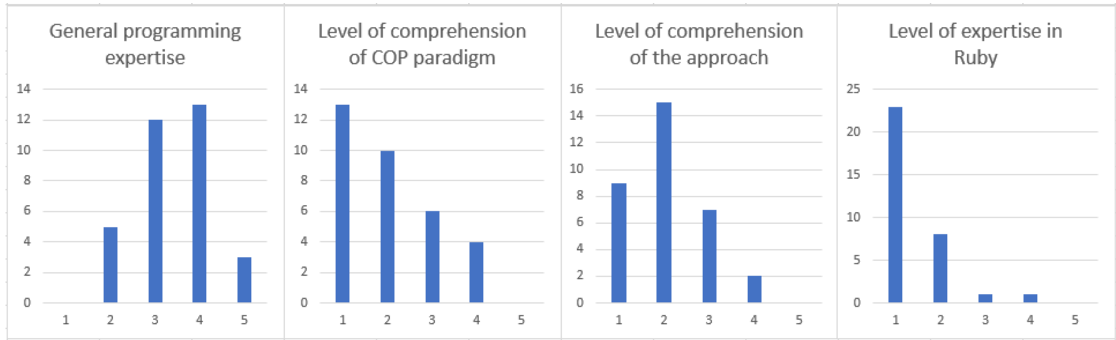


Figure 6.1: Students' comprehension of the COP paradigm and the programming approach

6.2 Results

Unfortunately, this study did not provide any conclusive evidence that our visualisation tool improved developers in their tasks. This is mainly due to the complexity of the programming approach. Even tough the students thought they have a quite good general programming expertise, they were not at ease with the context-oriented programming paradigm and the feature-oriented context-aware programming approach as it is depicted in Figure 6.1. The graphs represent the number of answers received (y-axis) using a five-level Likert scale (x-axis). 1 represents "no expertise" and 5 means "expert". In addition to this reason, most of the students did not have lots of experience with the Ruby programming language which could interfere with the experiment. Due to these reasons, many students did not have enough time to finish their tasks. Nevertheless, we observe a slight improvement in this domain for the second task with respect to the first one (See Figure 6.2). An explanation to this improvement could be the use of the tool or a better understanding of the framework after the first task.

In order to improve this study, we think that a better introduction and explanation of these concepts would be required. The participants need to have more time to get acquainted with the paradigm and the proposed programming approach. A suggestion could be the explanation of these concepts during four hours of course and then the development of a project using these concepts. The students have a deadline of two weeks to work on this project. After these two weeks, we introduce the visualisation tool and give them one more week to improve their implementation. This way students will have a better understanding of the approach and will no longer be a novice in this domain.

Despite these non-conclusive evidences, the study still provided some interesting insights. When the question of what particular functionalities of the tool can

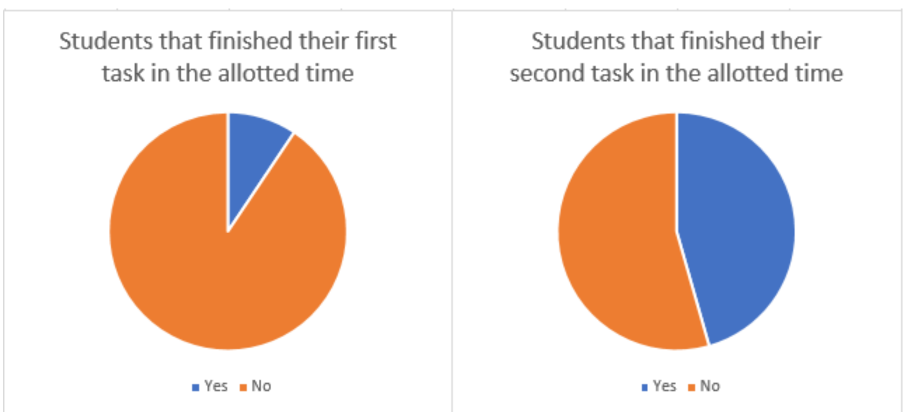


Figure 6.2: Percentage of students that finished their task

improve a developer's experience with the programming approach, was asked, most of the answers were the expected ones. Indeed, some students said that the tool allows to verify if the feature and context models are the ones they wanted. Others said that it helps for debugging since he can have a vision of the (de)activation of entities one by one. Moreover, as the Figure 6.3 shows it, most of the students think that the representation of the models in the visualisation tool (in a static and dynamic way) is understandable. A majority also thinks that the tool helps to understand the programming approach and is easy to use. 1 in the Likert scale of these graphs means "not at all" whereas 5 means "absolutely".

In addition to these results, several feedbacks of potential improvements were received. We will discussed about them in the next chapter.

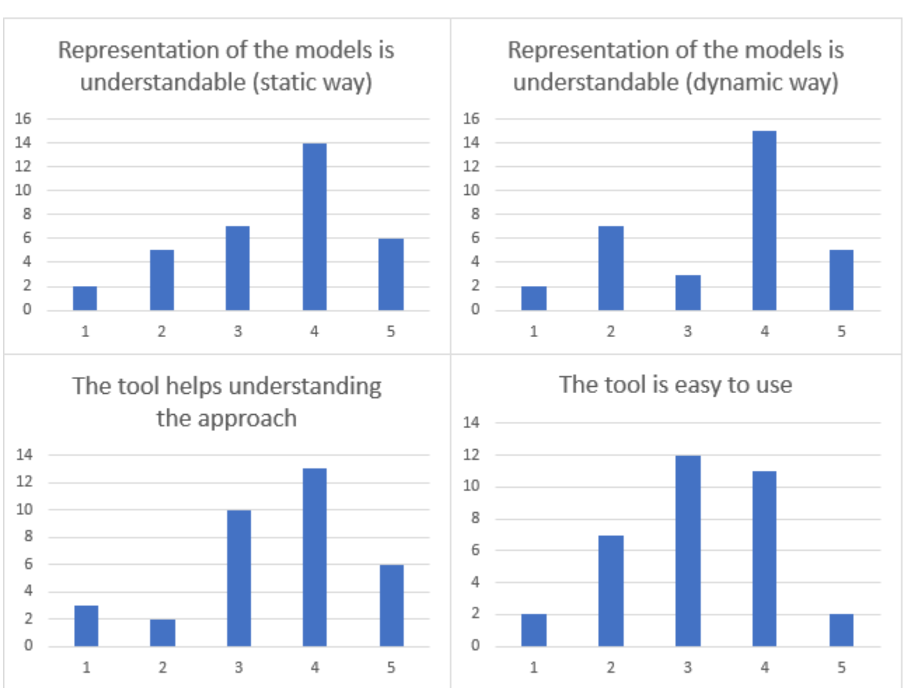


Figure 6.3: Students' feedbacks on the tool

Chapter 7

Future work

In this chapter, we will present several ideas of improvement of the tool including some new functionalities and combinations with other tools.

7.1 Class diagram

As already mentioned in Section 5.3.1, the class diagram in the tool is not a complete version. It still lacks the classes that are not adapted by any feature. This breach is quite problematic as it removes several important pieces of information for the comprehension of the COP application. We have concluded that it is not a problem coming from the implementation of the tool but from the link between the framework and the tool. Currently, information about a class for the tool is sent when a declaration of feature occurs. Indeed, every feature is represented by a Ruby module and this specific module adapts a class. Thus, retrieving information about the class at this moment is the easiest. We have not found an optimal solution yet, but one solution could be to add a getter and a setter to the singleton *RootFeature* of the framework. As this class represents the root of the feature model, it will always be instantiated. Thanks to the getter and setter, we can ask the application developer to give a list of names of the classes he implemented. This way, once we send data about the creation of the root feature to the tool, we can also send all the information about the classes. This solution is not optimal as it is weird to ask for a list of class names. Indeed, it is not intuitive to ask information just for a tool to work.

7.2 Scalability of the context-feature model

One critic that was raised by the students in the validation study is the scalability of the graph. The more the number of contexts and features increases the more

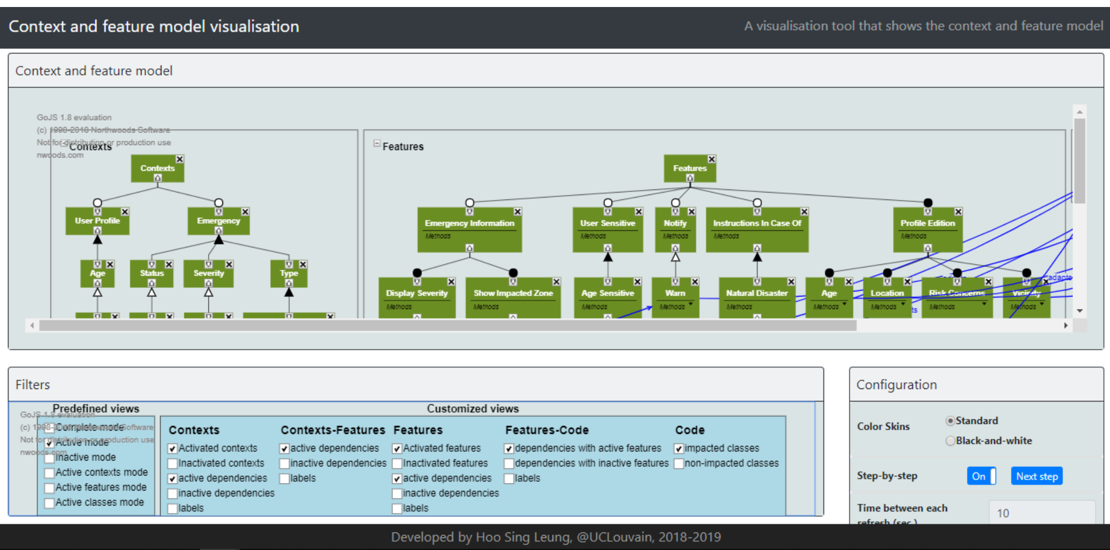


Figure 7.1: Snapshot of the tool having large models.

the graphs become hard to handle. Indeed, even with the help of filters and the designed buttons, a case like Figure 7.1 can happen. The active mode is selected but we still can not have a full vision of the graphs. We are forced to play with the scrollbars. A solution to this problem can be to have all the graphs scaled to fit the size of the HTML Div, and then have the functionality of a magnifying glass by holding the left click and moving the mouse. The problem with this solution is the very small and unreadable resulting graphs. Therefore, having a clean look of a context and a feature for example, at the same time will nearly be impossible.
A better solution is to change the representation of the graphs. Instead of using a classic structure of the feature (context) model, we can exploit the vertical structure used by S.P.L.O.T. (Section 2.5.1). This way, we will have two vertical models and a class diagram, and consequently, we will be able to save some width spaces. As a result, we will only have to play with the vertical scrollbar in case of big models. The optimal solution will be to add a functionality that allows the user to switch between these two representations.

7.3 Step-by-step

Currently, the step-by-step option only gives the user the possibility to see the next step. What if we click unintentionally two times in a row and missed the piece of information that we were looking for? Two additional options need to be added in order to have an optimised step-by-step debugger. The first one obviously is the possibility to make a step back. However, implementing it is not such an easy task

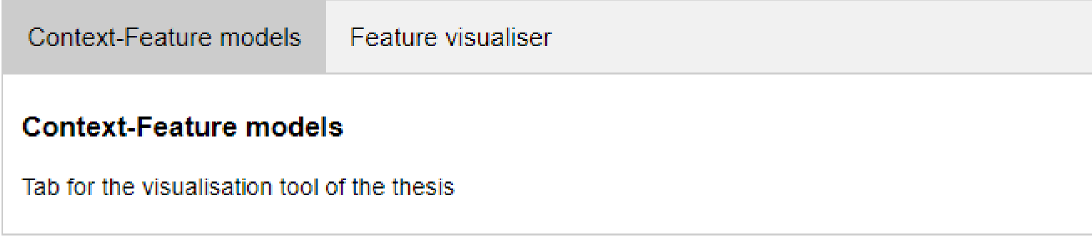


Figure 7.2: Example of HTML tab with the two tools.

since we need to make the reverse move of what we just did. That means we have to remember the done steps and have a mechanism to deduce the counter move of a step. The second option that needs to be added is a reset button. This task is easier since we can save the model data (nodes and links) before any (de)activation, and once the reset button is clicked, we will just have to re-affect the saved model to the GoJS diagram.

7.4 Combination with other tools

Since the tool presented in Section 2.5.2 gives a more precise visualisation of the interaction between features and the classes of the application, grouping it to our solution can be an interesting idea. The idea of having a web application with each tool as an HTML Tab (Figure 7.2) can help a developer in his debugging task. As an example, the developer spots an error in the adaptation of a class due to the activation of a certain feature. The possibility of switching to the feature visualiser at this specific state can help the developer, as the feature visualiser has a more precise visualisation of this kind of information.

Another tool that can be added to this web application is the simulator described in Section 2.5.2. The solution of this master's thesis can show each step of (de)activation done by the simulator.

Chapter 8

Conclusion

The purpose of this master's thesis was to create a visualisation tool in order to help a developer debug an application built using a context-aware feature-oriented approach. The tool had to allow the visualisation of which contexts and features are active and how they interact with the system.

This thesis brings a new visualisation tool subdivided into three parts:

1. **Context and feature model** shows the context and feature models as well as a class diagram that represents the context-aware system. The two models contains several buttons that allow a better scalability of these graphs. The nodes of the feature diagram also gives the names of the classes they adapt. The selection of an entity of these graphs will highlight all the nodes that depend on it and that can trigger its (de)activation. Obviously, the inter- and intra-dependencies are also shown in this part.

2. **Filters** improves the developer's view when applications become complex. Some predefined views are defined, in addition to a set of filters allowing the creation of customised views.

3. **Configuration** gives the possibility to follow each (de)activation of entities (contexts and features). It also allows to change the skin of the tool.

For a better understanding of its functioning, the explanation of the tool was done based on a case study, concerning a *Risk information system*. In order to assess its usefulness and its understandability, the tool had been subject of a validation study.

Obviously, this tool can still be improved as explained in the future work.

Finally, the redaction of a scientific paper for *The 11th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, about this visualisation tool is currently under way. The authors are Benoît Duhoux, Bruno Dumas, Kim Mens and myself.

Bibliography

[1] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July 2009.

[2] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University, 1990.

[3] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54, Aug 2001.

[4] Herman Hartmann and Tim Trew. 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, pages 12–21. IEEE Computer Society, 2008.

[5] Max Schlee and Jean Vanderdonckt. Generative programming of graphical user interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 403–406, New York, NY, USA, 2004. ACM.

[6] Pascal Costanza Robert Hirschfeld and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008.

[7] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801 – 1817, 2012.

[8] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4):263–277, June 2007.

[9] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, pages 246–265, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] Ricardo Tesoriero, José A. Gallud, María D. Lozano, and Víctor M. R. Penichet. Cauce: Model-driven development of context-aware applications for ubiquitous computing environments. *Journal of Universal Computer Science*, 16(15):2111–2138, jul 2010.

[11] Jabier Martinez, Jean-Sébastien Sottet, Alfonso García Frey, Tewfik Ziadi, Tegawendé Bissyandé, Jean Vanderdonckt, Jacques Klein, and Yves Le Traon. Variability management and assessment for user interface design. In Jean-Sébastien Sottet, Alfonso García Frey, and Jean Vanderdonckt, editors, *Human Centered Software Product Lines*, pages 81–106. Springer International Publishing, Cham, 2017.

[12] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In Hans-W. Gellersen, editor, *Handheld and Ubiquitous Computing*, pages 304–307, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[13] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 7–12, New York, NY, USA, 2016. ACM.

[14] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, pages 178–191, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Z. Jaroucheh, X. Liu, and S. Smith. Mapping features to context information: Supporting context variability for context-aware pervasive applications. In *International Conference on Web Intelligence and Intelligent Agent Technology WIC2010*, volume 1, pages 611–614, Aug 2010.

[16] Rafael Capilla, Óscar Ortiz, and Mike Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, Feb 2014.

[17] Rafael Capilla, Mike Hinchey, and Francisco J. Díaz. Collaborative context features for critical systems. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems VaMoS2015*, pages 43:43–43:50, New York, NY, USA, 2015. ACM.

[18] Kim Mens, Rafael Capilla, Nicolás Cardozo, and Bruno Dumas. A taxonomy of context-aware software variability approaches. In *Companion Proceedings of*

the 15th International Conference on Modularity, MODULARITY Companion 2016, pages 119–124, New York, NY, USA, 2016. ACM.

[19] Pascal Costanza. Feature descriptions for context-oriented programming. In *Lero Int. Science Centre, University of Limerick, Ireland*, pages 9–14, 2008.

[20] Nicolás Cardozo, Sebastian Günther, Theo DHondt, and Kim Mens. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *International Conference On Software Engineering Advances (ICSEA'11)*, pages 130–135. IARIA, 2011.

[21] Nicolás Cardozo, Kim Mens, Pierre-Yves Orban, Sebastián González, and Wolfgang De Meuter. Features on demand. In *Proceedings of 8th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 18:1–18:8. ACM, 2014.

[22] Alexandre Kühn. Reconciling context-oriented programming and feature modeling. Master's thesis, Université catholique de Louvain, Belgium, 2017.

[23] Benoît Duhoux, Kim Mens, and Bruno Dumas. Feature visualiser: An inspection tool for context-oriented programmers. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*, COP '18, pages 15–22, New York, NY, USA, 2018. ACM.

[24] Sebastian Günther and Sagar Sunkle. rbfeatures: Feature-oriented programming with ruby. *Science of Computer Programming*, 77(3):152 – 173, 2012. Feature-Oriented Software Development (FOSD 2009).

[25] Benoît Duhoux. *L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte*. PhD thesis, UCL - Ecole polytechnique de Louvain, 2016.

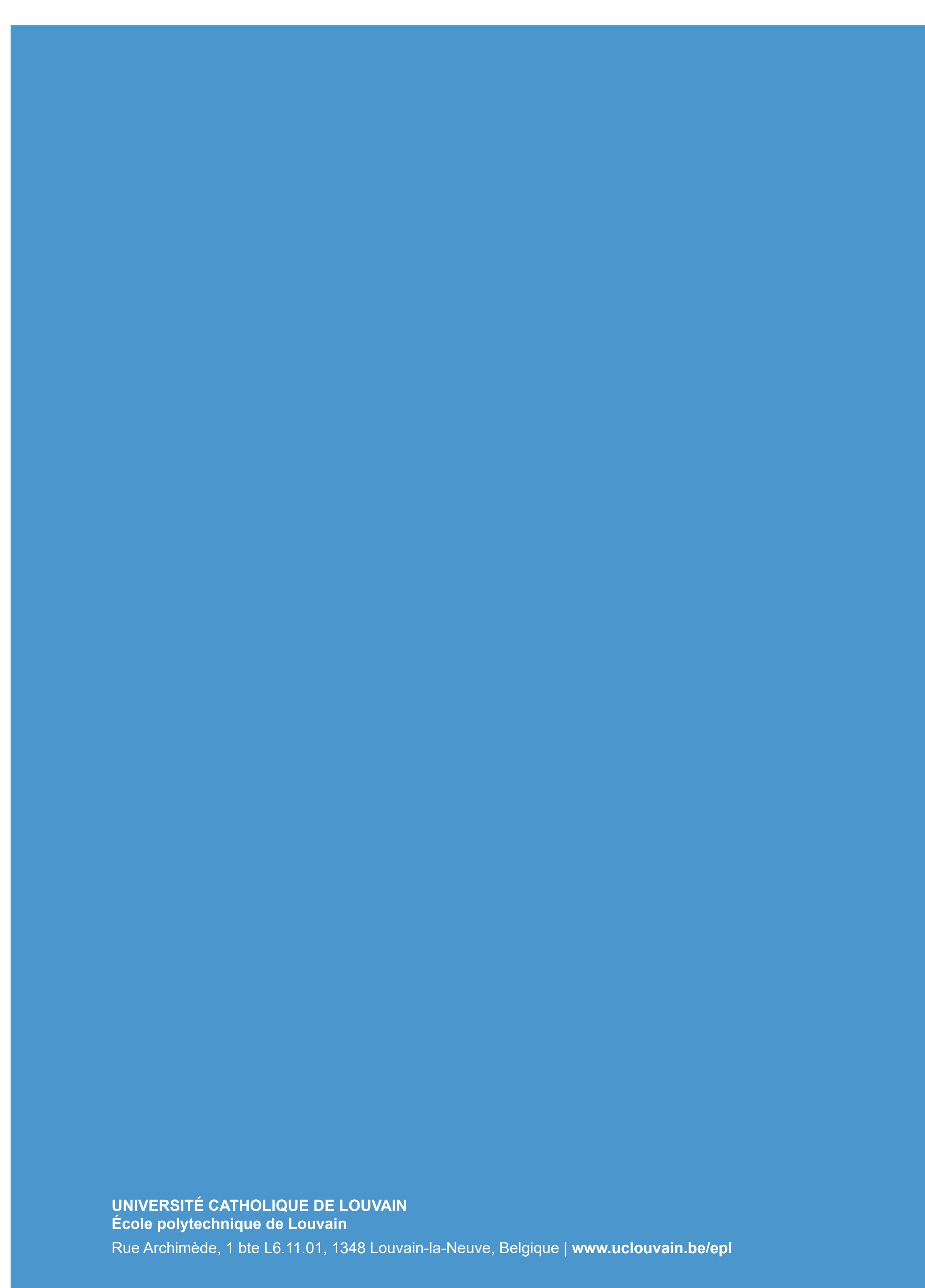
[26] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool support for feature-oriented software development: Featureide: an eclipse-based approach. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 55–59, New York, NY, USA, 2005. ACM.

[27] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70 – 85, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[29] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[30] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. Case tool support for variability management in software product lines. *ACM Comput. Surv.*, 50(1):14:1–14:45, March 2017.


UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**