Implementation of a Feature-Based Context-Oriented Programming Language

Benoît Duhoux
Université catholique de Louvain
Belgium
benoit.duhoux@uclouvain.be

Kim Mens
Université catholique de Louvain
Belgium
kim.mens@uclouvain.be

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

ABSTRACT

We implemented a feature-based context-oriented programming language, which clearly separates the notion of contexts from the notion of features. Contexts reify particular situations occurring in the surrounding environment, to which a software system can adapt. Features reify the system's behaviour; they are the language components that describe the system's functionality at a fine-grained level. Contexts are mapped to features, such that, when certain contexts become active at run-time, the corresponding features get selected and activated, thus adapting the system's functionality to those particular contexts. In this paper we show the object-oriented architecture, design and implementation issues of such a feature-based context-oriented programming language, which we implemented on top of the Ruby programming language as an application framework for context-oriented programmers. An important part of our language design is the explicit representation of contexts and features in terms of hierarchical tree structures that capture the structural constraints to be maintained at runtime. We illustrate our language design with a small example of a feature-based context-oriented program written in this language.

ACM Reference Format:
Benoît Duhoux, Kim Mens, and Bruno Dumas. 2019. Implementation of a Feature-Based Context-Oriented Programming Language. In *Workshop on Context-oriented Programming (COP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3340671.3343357

1 INTRODUCTION

An explicit representation of, and adaptation to, context is key to facilitate the building of context-aware applications [1] where the user's or system's surrounding context is changing rapidly. Context-oriented programming languages [16] were designed to build software systems of which the behaviour can be adapted at runtime in response to context changes. In layer-based context-oriented programming languages [16], the behavioural variations are grouped into layers. When a context changes, the corresponding layer gets deployed to refine the current behaviour.

In our feature-based context-oriented approach, we regard and represent contexts and features as separate dual entities, related through an explicit mapping. While a context represents a particular situation of the surrounding environment in which the system executes, a feature is *a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems* [18]. In what follows, when we use the word feature, we mean some fine-grained functionality of the system, such as a single method or a set of related methods. Despite their different definitions, these two kinds of entities (contexts and features) bear similarity. They can both become active or inactive at runtime, and many structural constraints can be declared among them that need to be maintained upon (de)activation. Furthermore, the kinds of constraints that can be declared among them (mandatory, optional, or, alternative) are similar.

For that reason, in earlier work we proposed a context-oriented software architecture [22], where these two notions are clearly represented separately, yet modelled in a similar way. Our main inspiration for this architecture came from the field of feature and software product line modelling [4, 5, 17, 21], and Hartmann and Trew's multiple product-line feature modelling approach in particular [14]. In their approach, both contexts and features are part of a single model that is explicitly separated into a context variability diagram and a feature diagram, each with their own internal structural constraints, and connected by explicit dependencies between both diagrams.

In the implementation of our feature-based context-oriented language too, we represent contexts and features as separate models, connected through an explicit mapping. We implemented a prototype of such a language as an application framework in the Ruby programming language. The purpose of this paper is to present the overall object-oriented implementation architecture and design issues underlying that implementation. This information may be relevant to potential developers using our language to write context-oriented applications, or for researchers in context-oriented programming who want to take inspiration from our language design to implement their own language. We also illustrate how to build a context-oriented application in our language, using the example of a risk information system which we already introduced in a previous paper [9].

In Section 2 we introduce our feature-based context-oriented approach in more detail. Section 3 describes the case study, a risk information system, that we will use in this paper. Section 4 presents the design and implementation choices of our programming language, as well as how to write a context-oriented application in this language. Finally Section 5 exposes the related work and Section 6 concludes our paper and presents some future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
COP'19, July 15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6863-6/19/07…$15.00
https://doi.org/10.1145/3340671.3343357

2 APPROACH

An important implementation choice of our feature-based context-oriented language is the explicit representation of both contexts and features in terms of hierarchical tree structures, i.e. feature models.

Feature model

Feature models are diagrams that capture the possible functionalities of systems in a software product line [18]. They model the main commonalities and variabilities of such systems. In our approach, we use feature models to represent the commonalities and variabilities of a single software system of which the functionality can change at runtime depending on changing contexts. (We even use them to model the possible contexts and how these can change at runtime, but more about that soon.) Commonalities represent the features that need to be present in each instance of the system, while the variabilities are features that may be present in some instances of the system only.

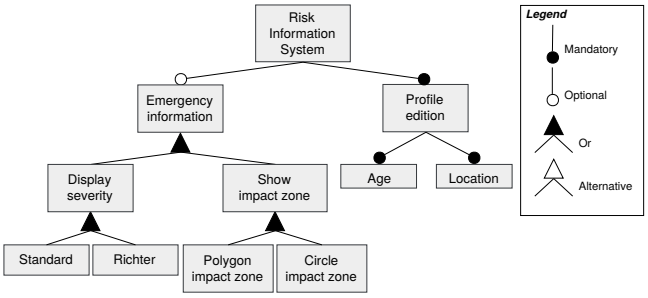

Figure 1: An excerpt of a feature model for our risk information system.

As depicted in Fig. 1, a feature model can be represented as a tree, where the nodes represent features and its hierarchical edges declare constraints between a feature and its subfeatures. Such constraints can be *mandatory* (when the feature is present its subfeature should be as well), *optional* (when the feature is present its subfeature may be present), *or* constraints (at least one of the subfeatures of a given feature should be present when the parent feature is present), or an *alternative* constraint (exactly one of the subfeatures should be present when its parent feature is). In a feature model, we can also distinguish abstract features from concrete ones. Whereas abstract features are mainly infrastructural and help programmers group related finer-grained functionalities, concrete features contain code that may affect existing system behaviour when they get selected. For an instance of a feature model to be valid, its set of selected features must satisfy all the constraints declared by the feature model.

Feature-based context-oriented approach

Fig. 2 presents an overview of our feature-based context-oriented approach. It is an extension of our earlier context-oriented software architecture [22] where we already proposed handling contexts and features in separate architectural layers. In our current implementation, inspired by Hartmann and Trew [14], we extend this idea further by representing contexts and features explicitly in terms of two separate feature models. One feature model, the *Context Model*, declares the possible contexts (and their constraints) to which the system may adapt, whereas the *Feature Model* declares the system's behavioural variations. In addition to these two models, a *Mapping Model* declares the dependencies from contexts to features. With these dependencies, the system knows what features must be installed in or removed from the current instantiation of the system, depending on the current state of the contexts.

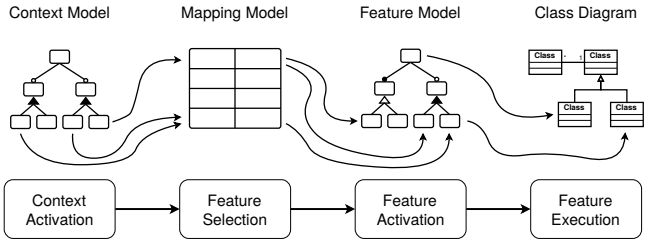

Figure 2: Overview of our feature-based context-oriented programming approach.

The control flow of our approach goes as follows: when a context changes, the Context Activation component gets informed about this new discovery. It will then attempt to activate this context, taking into account the constraints imposed by the *Context Model* declared by the programmer. Once the context activation is done, the Feature Selection component takes over, selecting the features corresponding to the newly activated context(s) based on the *Mapping Model*. The Feature Activation component, as its homologue Context Activation, will then try to activate the selected features taking into account the constraints imposed by the *Feature Model*. Finally, the Feature Execution component will deploy the activated features in the classes of the currently running system. Similarly, when certain contexts become inactive, the different components in the architecture will take care of the necessary deactivations of contexts and their corresponding features and code.

3 RUNNING EXAMPLE

Before explaining the design and implementation issues encountered in conceiving our programming language, we first introduce the running example used in this paper: a risk information system (*RIS* for short) [9]. It aims to notify citizens about the characteristics of emergencies such as earthquakes or floods, and to provide them with instructions on how to act in case of such emergencies.

The characteristics are the information about an ongoing emergency, such as for example its severity and its location. For example, the severity for an earthquake emergency is computed using the Richter scale and its location is defined as a circular impact zone with a certain epicentre. For a flood emergency, the severity may be indicated as *low*, *medium* or *high* and its location may be described by a polygon impact zone.

The instructions given to citizens in case of an emergency can depend on many contexts: the user's age, the weather, the status of the emergency and so on. The status of an emergency describes

whether the emergency is about to happen, is ongoing or is finished. According to this status, an instruction could for example be "Limit journeys and avoid the danger zones" if a flood is happening or "Ventilate rooms adequately, but also heat them adequately to dry off the moisture" when the flood is over. [1] Such instructions could be given in textual form for adults, or using pictograms when the citizen is a child.

At all times, the citizens can also consult the instructions on actions they must undertake in case an emergency occurs, even if there is currently no emergency warning. In that case, the system provides all instructions for the selected emergency type.

In the next section, we will present the design and implementation choices of our programming language, and illustrate with code fragments how a programmer can create a context-oriented risk information system using our language.

4 DESIGN AND IMPLEMENTATION

Fig. 3 shows the implementation architecture of our feature-based context-oriented programming language. This architecture is organised in three parts. The Entities part reifies the different kinds of entities that need to be declared and defined by the programmer, i.e. the contexts, features, as well as the mapping between them. The Modelling part reifies the hierarchical tree structure representing the context and feature models and their internal constraints and dependencies. Finally, the Architecture part implements the different components of our context-oriented software architecture [22] as explained in the last paragraph of section 2.

Each of these parts will be explained in more detail below. We also show how our implementation architecture allows us to easily create tools (e.g., visualisation tools), to support programmers writing programs in our feature-based context-oriented programming language.

4.1 Entities

Fig. 4 shows a more detailed version of the Entities part of our implementation architecture. It captures the main kind of entities that programmers need to declare and define, such as the contexts to which their context-oriented program can adapt, the features describing the application behaviour specific to certain contexts, and the mapping from context to features declaring what specific behaviour is triggered by what particular contexts.

Contexts and features as entities. Together with classes (representing the system's base functionality that can be adapted dynamically), contexts and features are the primitive building blocks of our language. Due to their different nature, each of these notions is reified by a different class in our implementation architecture: the class `Context` and the class `Feature`. While a context is just identified by a name, a feature is characterised by a name and a targeted class, that is, the class of which this feature will adapt the behaviour. Each of these two classes inherit, respectively, from their more abstract counterpart `AbstractContext` and `AbstractFeature`. This abstraction comes from the fact that we can have both abstract features and concrete ones in our feature model (and similar for contexts). Abstract features are just there for helping to structure the tree hierarchy, but are not attached to a targeted class that they can alter, as opposed to concrete features. Similarly, abstract contexts are just infrastructural entities to help structuring a context model, whereas concrete contexts correspond to an actual situation that may occur in the environment surrounding the application. Finally, both classes `AbstractFeature` and `AbstractContext` inherit from a common class `Entity`. This allows us to model the fact that both features and contexts are a special kind of entities among which similar kinds of constraints and dependencies can be declared, as we will see next.

Context and feature declarations. One of the first things a programmer building a feature-based context-oriented application needs to declare is the context and feature model of the application. To do so, the programmer needs to extend, respectively, the classes `ContextDeclaration` and `FeatureDeclaration` provided by our implementation architecture. The declaration of the context (resp. feature) model starts by the declaration of an abstract context (resp. feature) as root entity of the context (resp. feature) model.

Just like the classes `Context` and `Feature` in our architecture inherited from a common parent class `Entity`, for consistency, the classes `ContextDeclaration` and `FeatureDeclaration` inherit from a common parent class `EntityDeclaration`. This parent class contains some infrastructural scaffolding code, such as the method *generate_all_getters* to automatically generate accessors for all entities. These accessors are useful for a programmer to access the entities when (s)he needs to declare the mapping from the contexts to the features.

A code example of a context declaration and a feature declaration are shown in Listings 1 and 2, with the classes `RISContext-Declaration` and `RISFeatureDeclaration`, respectively.

```
1  class RISContextDeclaration < ContextDeclaration
2    include Singleton
3    def initialize()
4      super()
5      ...
6      self.define_emergency_contexts()
7      @root_context.add_relation(:Optional, [
       @user_profile_context, @emergency_context])
8    end
9    def define_emergency_contexts()
10     @emergency_context = AbstractContext.new('
       Emergency')
11     ...
12     @zone_context = Context.new('Zone')
13     @type_context = AbstractContext.new('Type')
14     @earthquake_em_context = Context.new('
       EarthquakeEmergency')
15     @flood_em_context = Context.new('
       FloodEmergency')
16     @type_context.add_relation(:Or, [
       @earthquake_em_context, @flood_em_context])
17     @emergency_context.add_relation(:Or, [
       @status_context, @severity_context,
       @zone_context, @type_context])
18   end
19 end
```

Listing 1: Snippet of the context declaration of our *RIS*.

[1] https://www.info-risques.be/en/hazards/naturals-hazards/floods

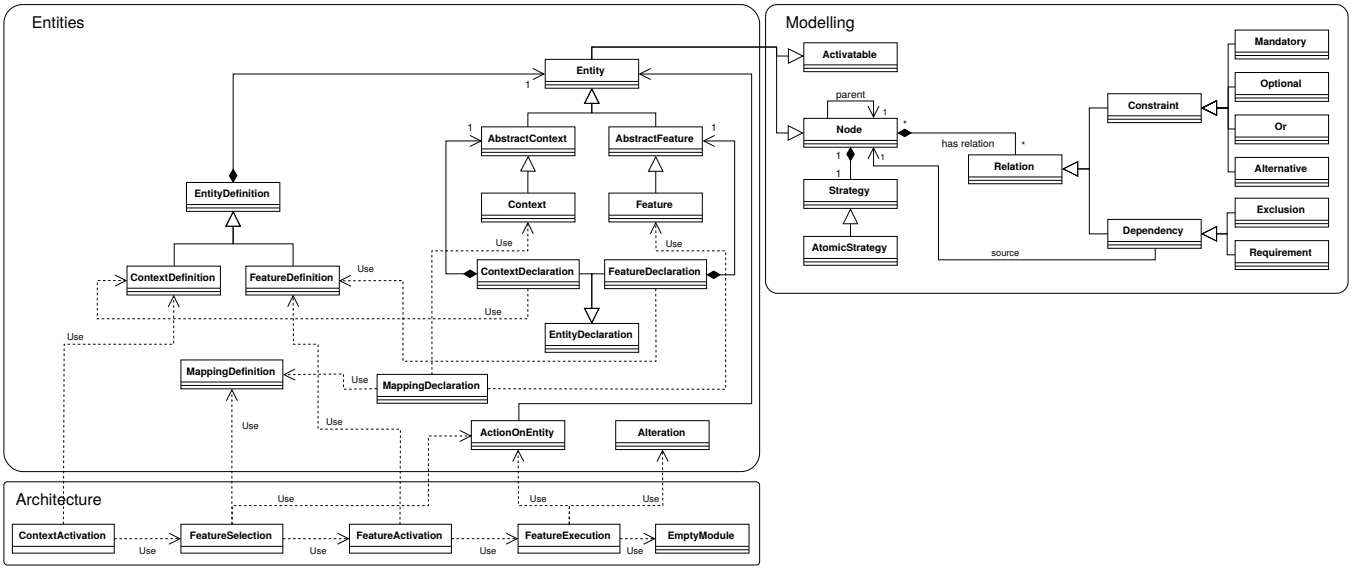

Figure 3: Implementation architecture of our feature-based context-oriented programming language.

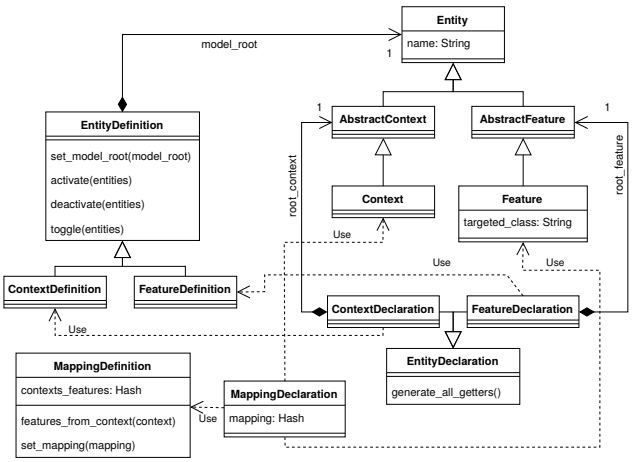

Figure 4: Class diagram of the ENTITIES part of our implementation architecture.

A programmer creating these classes must implement the method *initialize*. This method must first call the super method to initialise the root context (resp. feature), then declare the other contexts (resp. features), and finally attach them to the root context (resp. feature).

```
class RISFeatureDeclaration < FeatureDeclaration
  include Singleton
  def initialize()
    super()
    self.define_emergency_info_features()
    ...
    @root_feature.add_relation(:Optional, [
    @emergency_info_feature,
    @age_sensitive_feature, @navigation_feature,
    @notify_feature, @risk_description_feature,
    @instructions_in_case_of_feature])
    @root_feature.add_relation(:Mandatory, [
    @profile_edition_feature])
  end
  def define_emergency_info_features()
    @emergency_info_feature = AbstractFeature.new(
    'EmergencyInformation')
    @display_severity_feature = AbstractFeature.
    new('DisplaySeverity')
    @standard_feature = Feature.new('Standard', '
    Flood')
    @richter_feature = Feature.new('Richter', '
    Earthquake')
    @display_severity_feature.add_relation(:Or, [
    @standard_feature, @richter_feature])
    ...
    @emergency_info_feature.add_relation(:
    Mandatory, [@display_severity_feature,
    @show_impacted_zone_feature])
  end
end
```

Listing 2: Snippet of the feature declaration of our *RIS*.

Context and feature definitions. From the context and feature declarations provided by the application programmer, the implementation architecture automatically creates corresponding context and feature definitions. These definitions represent the entities that will be used at runtime and contain the necessary code to manage the activation and deactivation of these entities. In other words, interpreting the programmer's declared subclasses of `ContextDeclaration`

(resp. `FeatureDeclaration`), implies the initialisation of the context (resp. feature) model in the class `ContextDefinition` (resp. `FeatureDefinition`) thanks to the method *set_model_root(model_root)*. Because the entire behaviour of these `*Definition` classes is similar, again they inherit from a common parent class `EntityDefinition` that contains the implementation of this behaviour. These definition classes serve as a bridge with the Architecture and Modelling parts of our implementation architecture. Whenever some component of our implementation architecture wants to activate or deactivate an entity, it sends a message (e.g. with a call to the method *activate(entities)*) to the definition of this entity, which then delegates that action to the entity itself.

Mapping contexts to features. Once the programmer has declared a context and feature model with all the context and feature entities known to the application, (s)he still needs to declare how the system should react to contextual changes, i.e. what context activations trigger what feature activations. To declare this mapping from contexts to features, the programmer extends the class `MappingDeclaration` for which he initialises the *mapping* instance variable.

This *mapping* is a hash data structure where the key is a list of contexts and the value is a list of features. The reason for using a list structure for the keys and the values is that the programmer cannot only create simple mappings (from one context to one feature) but also more complex ones (from many contexts to many features, indicating that if a set of contexts is simultaneously active, this would trigger activation of all corresponding features listed).

Listing 3 gives an example of such a mapping declaration.

```
class RISMappingDeclaration < MappingDeclaration
  include Singleton
  def initialize()
    @mapping = {...
      [
        RISContextDeclaration.instance.
          during_context(),
        RISContextDeclaration.instance.
          flood_em_context()
      ] => [
        RISFeatureDeclaration.instance.
          active_during_flood_feature()
      ],
      [
        RISContextDeclaration.instance.
          flood_em_context()
      ] => [
        RISFeatureDeclaration.instance.
          standard_feature(),
        RISFeatureDeclaration.instance.
          polygon_impacted_zone_feature()
      ]
    }
  end
end
```

Listing 3: Snippet of a mapping declaration of our *RIS*

In this declaration, the programmer uses the accessors of the context and feature declarations to create the mapping from contexts to features. In this particular mapping, when a flood emergency is currently occurring, two contexts (*during_context* and *flood_em_context*) would be active, which would imply the selection of the feature *active_during_flood_feature*, dedicated to provide instructions to citizens on how to act when a flood is occurring.

As for the context and feature definitions, when this mapping declaration gets interpreted, the mapping is set in the class *MappingDefinition* by calling the method *set_mapping(mapping)*.

4.2 Modelling

Now that we have explained the different classes in the Entities part of our implementation architecture, we turn our attention to the Modelling part. The main purpose of this part is to provide a generic implementation for building entity models (i.e., either context or feature models depending on the kinds of entities contained in them). Fig. 5 shows the detailed class diagram for the Modelling part of our implementation architecture. We will present this part from two different angles: the creation of an entity model and the (de)activation of entities in an entity model.

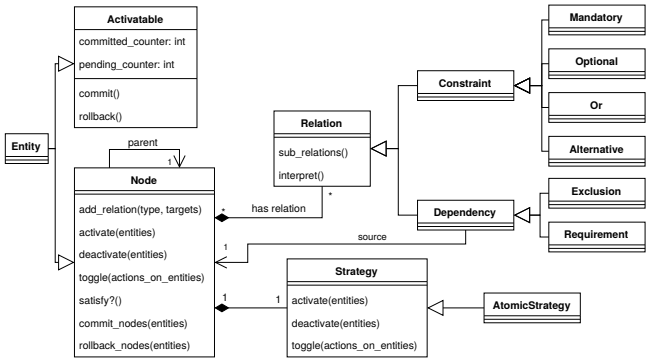

Figure 5: Class diagram of the Modelling part of our implementation architecture.

Creation of an entity model. An entity model is a tree structure where the nodes are entities and the edges are relations between those entities. To allow having entities as nodes, the class `Entity` inherits from the trait [2] `Node`. Thanks to this trait, an entity can have many relations (instances of the class `Relation`) to compose the entity model. These relations can be either constraints or dependencies.

Constraints, which are instances of the class `Constraint`, represent a hierarchical constraint between an entity and its child entities. These constraints can be `Mandatory`, `Optional`, `Or` or `Alternative`. The meaning of these constraints have already been discussed in Section 2.

Dependencies, which are instances of the class `Dependency`, on the other hand, represent non-local dependencies between nodes in the tree that do not necessarily have a direct hierarchical relation. Two types of such dependencies are currently available in our programming language: `Exclusion` and `Requirement`. An exclusion dependency between two nodes implies that their corresponding entities cannot be active simultaneously at runtime. A requirement

[2] Implemented as a module in the Ruby programming language.

dependency between some source and target node implies that the source entity cannot be activated if the target entity was not already activated before. Other types of dependencies could also be implemented in the future, such as for example the causality dependency or implication dependency suggested by Cardozo et al. [6].

Activation of entities. The activation (or deactivation) of contexts or features is less trivial than it may seem, because we must always ensure that the constraints and dependencies declared by their entity model remain satisfied at all times. To verify this, the system calls the method *satisfy?()* on the root node and goes through the entire model to check if the satisfiability of the new configuration is valid. This dynamic satisfiability checking was implemented for the context and feature models but is not (yet) implemented for the mapping model because our mapping model is (currently) less complex that our entity model. When an entity model is discovered not to be valid (i.e. not satisfying the constraints), we must provide appropriate strategies to resolve the context/feature interaction problem [23]. Therefore, in addition to the different relations a node has, each node also has a strategy to (de)activate its entities. When the system attempts to (de)activate an entity, the *Node* trait delegates the action to this strategy by calling the methods *activate(entities)* or *deactivate(entities)*. We defined a default `AtomicStrategy` that tries to (de)activate all the entities in a single transaction. If this transaction cannot be completed correctly, the strategy rolls back these (de)activations. Otherwise it commits the transaction to yield a new stable configuration of the entity model. This transaction is implemented by the trait *Activatable*.

4.3 Architecture

The Architecture part implements the different components of our context-oriented software architecture [22], from the activation of contexts, over feature selection and activation, to the deployment of the features in the running system. Fig. 6 depicts the class diagram of the Architecture part of our implementation architecture. Each component of our context-oriented software architecture [22] is reified as a class in this diagram.

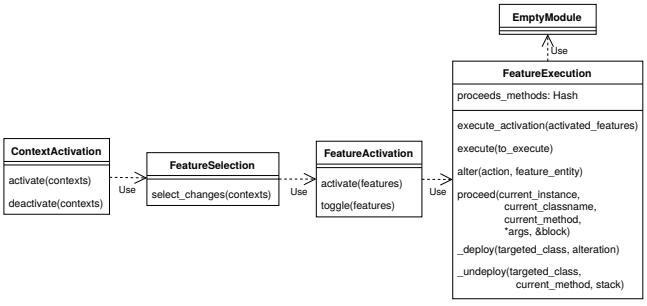

Figure 6: Class diagram of the Architecture part of our implementation architecture.

Control flow of the architecture. When a context change is detected in the environment surrounding the application, the system sends a message to the class `ContextActivation` with the method *activate(contexts)* or *deactivate(contexts)* as depicted in Listing 4. In this example, the system has detected a situation where a flood emergency is currently happening, so it tries to activate the contexts *during_context* and *flood_em_context*.

```
ContextActivation.instance.activate(
  context_declaration.during_context(),
  context_declaration.flood_em_context()
)
```

Listing 4: Activation of the contexts *during_context* and *flood_em_context* when these contexts are sensed.

Once these activations are finished, the contexts are sent to the class `FeatureSelection`. This class then selects the corresponding features from the mapping definition to transfer them to the class `FeatureActivation`. This class then attempts to (de)activate the features thanks to two methods *activate(features)* and *toggle(features)*. While the method *activate(features)* is called at the creation of the feature definition to activate the *mandatory* features, the method *toggle(features)* is used at runtime when a change is detected. Finally when the features are activated, these features must be added to the running system thanks to the class `FeatureExecution`. This class is in charge of deploying or undeploying the new features in the existing classes of the context-oriented program. For that the programmer must provide an object-oriented skeleton of base classes that can be extended at run-time, as well as the code of the features that will extend these base classes.

An example of such code fragments are presented in Listings 5 and 6.

```
class Flood < Risk
  def inform_user(user)
    super(user)
  end
  def instruct()
    puts "Instructions to follow when a flood is
    detected:"
  end

end
```

Listing 5: Snippet of the `Flood` base class.

```
module Standard
  adapt_classes :Flood
  ...
  def severity=(severity)
    self.valid_severity?(severity)
    @severity = severity
  end
  def inform_user(user)
    proceed(user)
    puts "Its severity is #{@severity}."
  end
end
```

Listing 6: Snippet of the `Standard` feature.

In this example, the class `Flood` will be adapted by the feature `Standard` to add the severity of a flood. The programmer needs

to declare in each feature [3] the classes that will be altered by that feature, by calling the method *adapt_classes*.

As for the class `FeatureActivation`, the class `FeatureExecution` implements a method *execute_activation(activated_features)* for the activation of the *mandatory* features at the launch of the application and a separate method *execute(to_execute)* dedicated for the runtime. Each of these methods delegates the change action to be performed to the method *alter(action, feature_entity)*. Depending on the action, the system will deploy (resp. undeploy) all the methods of the newly activated (resp. deactivated) features in the corresponding classes with the method *_deploy(targeted_class, alteration)* (resp. *_undeploy(targeted_class, current_method, stack)*).

Proceed mechanism. When a feature refines the default behaviour or that of a previous feature, it is useful to call that previous behaviour to enhance the power of dynamic adaptability. For that, Hirschfeld et al. suggested a *proceed* mechanism in context-oriented programming [16]. Our language also implements such a *proceed* mechanism, which we attached to the class `Object` provided by the Ruby programming language. With this addition to the class `Object`, each object can call this *proceed* mechanism. When the *proceed* statement is called, the class `Object` intercepts the call. It retrieves the method that called the *proceed* in the stack and delegates the rest of the behaviour to the class `FeatureExecution`. This class contains an attribute *proceeds_methods* to store all the applied adaptations for each method of each class. When the method *proceed* of this class is executed, the system gets the previous version of the adaptation, deploys it in the corresponding class, executes it, and then redeploys the last version of the adaptation to be sure to be consistent with the previous state of the system.

4.4 Tool support

So far we have discussed the implementation choices underlying our programming language for building feature-based context-oriented applications. However, when building such applications, some programmers can get lost in their development task due to the high complexity of creating such applications. The problem can come from the high dynamicity of such systems but also from the exponential number of combinations when designing the contexts, the features and the mapping between them. To help programmers in their development task, we believe there is a strong need to support them with dedicated programming support tools. For example, in previous work we have suggested and presented two different visualisation tools: a *Feature Visualiser* [9] and a *Context and Feature Modelling Visualiser* [8]. The first tool helps to visualise the more dynamic side of our approach, by displaying the active contexts, which features are activated depending on these contexts and how these features alter the classes of the system. The second visualisation tool focuses more on helping the programmer to inspect and display, either statically or dynamically, the context model and feature model as well as the mapping that the programmer must design to sketch which contexts imply which features. While the first tool mainly focusses on the classes in the Architecture part of our implementation architecture, the second one is more related to the Modelling part.

[3] implemented as a trait in Ruby

Without showing what these visualisations look like (for more information see [9] or [8]), we focus here on how we extended our implementation architecture to make it easy to connect such tools to our language. We added a system of proxies so that our implementation architecture can communicate easily through communication channels with a server that maintains these different communication channels. As Fig. 7 depicts, each class of the Architecture part has a corresponding trait to send relevant information to the server. For example, the class `FeatureExecution` has a trait `FeatureExecutionProxy` which provides the server with information about what feature alters which classes of the system. The visualisation tool then interacts with the server to visualise this information independently from the actual language implementation. All the proxies in the *Architecture* component send information to our *Feature Visualiser* tool [9]. For the *Context and Feature Modelling Visualiser* [8] tool, on the other hand, the class `Node` has his own trait `ModellingProxy` which sends the server information about what contexts or features are (in)active, which mapping is activated as well as some other relevant information.

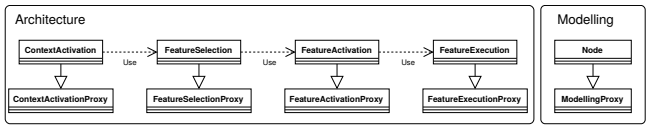

Figure 7: A proxy-based architecture for programming support tools to communicate with our programming language.

5 RELATED WORK

Many other context-oriented programming language implementations exist [2, 3, 7, 10–13, 15, 20, 24–27]. However, most of these implement either a layer-based approach or do not explicitly implement contexts and features in terms of an explicit context model and feature model, respectively. This lead us to explore this new implementation architecture where a clear separation distinguishes the contexts from the features and where both contexts and features are stored in feature-like models of which the consistency is maintained at runtime.

6 CONCLUSION

In this paper, we presented the implementation architecture of a feature-based context-oriented programming language and approach as a Ruby application framework, which separates the notions of contexts, features and the classes they adapt. Inspired by research in the field of software product line modelling, both contexts and features are represented explicitly in a feature-like diagram, with an explicit mapping from the contexts to the features.

This implementation architecture is subdivided in three main parts: Entities, Modelling and Architecture. The Entities part reifies the main building blocks of our language, used either by the context-oriented application programmer, or by the system itself. The Modelling part reifies the notion of feature modelling used to represent both the context model and feature model of a feature-based context-oriented program. Finally the Architecture part implements the different components implementing the overall

control flow of our context-oriented software architecture [22]. We also included some code fragments of a context-oriented risk information system to illustrate how a programmer can use the different building blocks of our programming language to build a context-oriented application. In addition to these parts, we showed how we implemented a mechanism of proxies communicating with a server, to make it easy to build programming support tools such as visualisation tools on top of our programming language.

Several paths of future work can still be explored to extend our proposed programming language. For example, as stated by Cardozo et al. [6], we can extend the kind of dependencies that can be declared between the different features or contexts in a context-oriented application. Or to extend our implementation with alternative strategies that a programmer can use to solve some of the problematic context and feature interactions (s)he may encounter [23]. We can also enhance the mapping model to increase the programmer's expressiveness when describing the mapping from contexts to features. Note that this improvement also implies the need for implementing a consistency checking mechanism for the mapping model. Also, in the current version of our implementation the features dynamically adapt the classes of the system, but we could study how we could allow adaptations at the instance level as well [20].

Another extension which we intend to explore is to go beyond adaptations with mere behavioural features, but to explore as well dynamically adaptive features that focus more on the user interface or the data handling of the application and how these features could vary with changing contexts.

Finally, in addition to these language extensions, we need to carry out larger experiences to explore whether our language and tools satisfy the needs of real developers [19] when building context-oriented applications.

REFERENCES

[1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. 1999. Towards a Better Understanding of Context and Context-Awareness. In *Handheld and Ubiquitous Computing*. Springer, 304–307.

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

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

[4] Rafael Capilla, Mike Hinchey, and Francisco J. Díaz. 2015. Collaborative Context Features for Critical Systems. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '15)*. ACM, Article 43, 8 pages.

[5] Rafael Capilla, Óscar Ortiz, and Mike Hinchey. 2014. Context Variability for Context-Aware Systems. *Computer* 47, 2 (Feb. 2014), 85–87.

[6] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, and Theo D'Hondt. 2013. Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets. In *2013 International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 191–198.

[7] Pascal Costanza and Theo D'Hondt. 2008. Feature Descriptions for Context-oriented Programming. In *Lero Int. Science Centre*. 9–14.

[8] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. 2019. Dynamic Visualisation of Features and Contexts for Context-Oriented Programmers. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '19)*. ACM, 6.

[9] Benoît Duhoux, Kim Mens, and Bruno Dumas. 2018. 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)*. ACM, 15–22.

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

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

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

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

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

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

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

[17] Zakwan Jaroucheh, Xiaodong Liu, and Sally Smith. 2010. Mapping Features to Context Information: Supporting Context Variability for Context-Aware Pervasive Applications. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 1. 611–614.

[18] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

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

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

[21] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. 2017. Modeling and Managing Context-Aware Systems' Variability. *IEEE Software* 34, 6 (Nov. 2017), 58–63.

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

[23] Kim Mens, Benoît Duhoux, and Nicolás Cardozo. 2017. Managing the Context Interaction Problem: A Classification and Design Space of Conflict Resolution Techniques in Dynamically Adaptive Software Systems. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)*. ACM, Article 8, 6 pages.

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

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

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

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