UCL
Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Programming Context-Aware Features in Ruby

Dissertation presented by
David Sarkozi

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Kim Mens

Reader(s)
Guillaume Maudoux, Bruno Dumas , Anthony Cleve

Academic year 2015-2016

Abstract

Observing the evolution of hardware and software technologies over the years, the programming tendency smoothly switched from declarative and static approaches, where a given program should ensure the same input-output pairs regardless of its environment, to more dynamic and context-dependent behaviours such as in mobile platforms, where for example applications are able to adapt to the output of their sensors (accelerometer, GPS, proximity, ...). The purpose of this thesis is to provide an insight on several techniques and means that were developed over the years regarding Context-Oriented Programming and behavioural variations. It is illustrated via a self-adaptive framework architecture where features and contexts are treated separately, but are encouraged to collaborate. This context-aware framework implemented in Ruby with the newest features that the language can provide is taking the challenge of unifying features and contexts under the same self-adaptive system, where only an object-oriented application needs to be settled. Especially, this thesis is focusing on the integration of features dynamically in a running application, where Ruby's metaprogramming capabilities truly shine.

Acknowledgements

I would like to warmly thank my promoter Kim Mens for his wonderful support, openness and availability over the past few months. I learned a lot through him and our discussions about Context-Oriented Programming. He is a wise man who can achieve many things in life, and I am looking up to him.

I would also like to thank the readers of this thesis, namely Guillaume Maudoux, Bruno Dumas and Anthony Cleve, for their patience and their effort.

I would like to thank my colleague Benoît Duhoux for his patience, effort and coding talent, which were roughly challenged throughout this thesis. I am sure he will become someone important someday, and I hope he will keep his sense of humour and his personality.

Finally, I would like to thank my family and friends who supported and will surely continue to support me along my way.

Contents

1 Introduction **5**
1.1 Context 5
1.2 Contributions 5
1.3 Roadmap 6

I Background **7**

2 Self-Adaptive Software Systems **8**
2.1 Introduction 8
2.2 Concepts 8
2.2.1 Variability 8
2.2.2 Management 9
2.3 Conclusion 10

3 Context-Oriented Programming **11**
3.1 Introduction 11
3.2 Concepts 11
3.2.1 Context 11
3.2.2 Variation 12
3.3 Context-Aware Systems 12
3.3.1 Variations management 13
3.3.2 Context management 14
3.4 Conclusion 16

4 Feature-Oriented Programming **17**
4.1 Introduction 17
4.2 Concepts 17
4.2.1 Feature 17
4.2.2 Feature modeling 19
4.3 Comparison with COP 19
4.4 Conclusion 19

5 Ruby **21**
5.1 Introduction 21
5.2 Ruby 2.0 21
5.2.1 Refinements 21
5.2.2 TracePoint 22
5.2.3 Module prepending 22
5.3 Conclusion 22

6 Related work 24
6.1 Introduction . . . 24
6.2 Phenomenal Gem . . . 24
6.3 rbFeatures . . . 25
6.4 ContextErlang . . . 25
6.5 Context Traits . . . 26

II Framework 28

7 Self-Adaptive Framework 29
7.1 Introduction . . . 29
7.2 Global architecture . . . 30
7.3 Interaction layer . . . 31
7.4 Discovery layer . . . 31
7.4.1 Interpretation . . . 31
7.4.2 Reasoning . . . 33
7.5 Handling layer . . . 34
7.5.1 Context handling sublayer . . . 35
7.5.2 Feature handling sublayer . . . 36
7.6 Conclusion . . . 37

8 Features integration 43
8.1 Introduction . . . 43
8.2 Principles . . . 43
8.3 Feature installation . . . 45
8.3.1 Principles . . . 45
8.3.2 Addressing native limitations . . . 47
8.4 Conclusion . . . 48

9 Benchmarks 50
9.1 Introduction . . . 50
9.2 Base . . . 50
9.3 Results . . . 50
9.4 Conclusion . . . 51

III Conclusion 54

10 Future work 55
10.1 Introduction . . . 55
10.1.1 Self-Adaptive Framework . . . 55
10.1.2 Features integration . . . 56

11 Conclusion 57
11.1 Contributions . . . 57
11.2 Objectives . . . 57
11.3 Final words . . . 58

Acronyms

`JSON`	JavaScript Object Notation.
AOP	Aspect-Oriented Programming.
API	Application Programming Interface.
CAAF	Context-As-A-Feature.
CAS	Context-Aware System.
CIL	Class-In-Layer.
COP	Context-Oriented Programming.
DSL	Domain-Specific Language.
FAAS	Feature-As-A-Service.
FODA	Feature-Oriented Domain Analysis.
FOP	Feature-Oriented Programming.
FOSD	Feature-Oriented Software Development.
LIC	Layer-In-Class.
OOP	Object-Oriented Programming.
SAF	Self-Adaptive Framework.
SASS	Self-Adaptive Software System.
SPL	Software Product Lines.
VM	Virtual Machine.
XOR	eXclusive-OR.

Chapter 1

Introduction

Contents

1.1 Context 5
1.2 Contributions 5
1.3 Roadmap 6

1.1 Context

Observing the evolution of hardware and software technologies over the years, the programming tendency smoothly switched from declarative and static approaches, where a given program should ensure the same input-output pairs regardless of its environment, to more dynamic and context-dependent behaviours such as in mobile platforms, where for example applications are able to adapt to the output of their sensors (accelerometer, GPS, proximity, . . .).

Although environment-aware programs are becoming more and more present for everyday use, proper and standard semantics have not yet been established and there is plenty of room for researching purposes in that direction. The unavailability of such standards and the inability for most popular languages to provide an idiomatic way to make context-aware applications and dynamically adaptable behaviours leaves the developers willing to do so, toolless, leading to unmaintainable and inefficient code.

Many researches have already been conducted to look into that problem and gathered their findings around the same banner : Context-Oriented Programming (COP) [RN08].

1.2 Contributions

The purpose of this thesis is to provide an insight on several techniques and means that were developed over the years regarding COP and behavioural variations. It will start with an overview of its surroundings in terms of paradigms, techniques, mechanisms, abstractions, . . .

And it will follow with the presentation of a self-adaptive framework architecture where features and contexts are treated separately, but are encouraged to collaborate. This framework implemented in Ruby with the newest features that the language can provide is taking the challenge of unifying features and contexts under the same self-adaptive system, where only an object-oriented application needs to be settled.

Especially, this thesis will focus on the integration of features dynamically in a running application, where Ruby's metaprogramming capabilities truly shine.

The implementation of a COP-capable Ruby has already been done in [Thi12], but with contexts and features mixed without the ability to distinguish them anymore. Also, the behaviours integration was mainly based on a pre-dispatching process, where the methods were manually looked up in order to provide the correct behaviour.

From this base, even though the framework has been implemented from scratch, the purpose of the solution is to implement a self-adaptive framework working with an object-oriented application alongside that is unaware of it, using only native Ruby code.

Both objectives are in mind. One is the implementation of a self-adaptive framework that separates the concepts of context and feature efficiently. The other is the implementation of a behaviours integration mechanism that installs features at runtime in an application seamlessly where the application is completely unaware of contexts and features for compatibility purposes. Both objectives should be achieved using native Ruby language.

1.3 Roadmap

The paper will be mainly divided in three parts.

Part I will present all the background work needed to embrace this thesis, with the concepts of self-adaptability, an insight on some paradigms such as COP itself, but also Feature-Oriented Programming, a quick note about Ruby and its characteristics, and eventually some related works.

Part II will provide a description of the framework's architecture and design, a focus on the integration of features in a running application and some benchmarks to compare with a naive implementation.

Finally, Part III will conclude the thesis by presenting some future work, and the current issues and possible improvements of the framework.

Part I

Background

Chapter 2

Self-Adaptive Software Systems

Contents

2.1 Introduction **8**
2.2 Concepts **8**
2.2.1 Variability 8
2.2.2 Management 9
Design patterns 9
Dependency injection 9
Event-driven behaviour 9
Metaprogramming 9
Reactive programming 9
Aspect-Oriented Programming 10
2.3 Conclusion **10**

2.1 Introduction

In this chapter, the concept of self-adaptability of software systems in its most generic form will be presented in order to grasp the challenges it generates. The concepts of variability and management will be presented, as well as some existing means to provide both of them in a dynamically adaptable system.

2.2 Concepts

2.2.1 Variability

To tackle the challenges generated by implementing self-adaptive systems, we have to define first what goes behind the concept of variability.

A Self-Adaptive Software System (SASS) has the ability to adapt its general behaviour* depending on changes in its close environment, the user's requirements and/or in the characteristics of the system itself [And+09]. Many software systems claim to fall into that category, and with the growth of distributed and mobile platforms, it is important to model such systems properly.

Variability in its rawest form is inspired from software product lines where a variety of software functionalities is used to cover a broad range of contexts. The main goals to be achieved

*Which can be described in many ways, such as through user interface, database updates or common implemented behaviours.

are to ensure both space and time variability management, where a software is highly variable if it can be used in several contexts (space variability) and if it supports evolving requirements in each of them (time variability) [GB03].

Software variability can be achieved at several stages of the implementation, according to [CBK13], and variation installations are mostly decided statically, namely before the execution of a self-adaptive program†. However, we will see later in this thesis that our main concern and challenge is for a system to achieve the highest variability at runtime, while the software program is executing.

2.2.2 Management

When dealing with SASS, we also have to consider variability management. One of the toughest challenges it is facing is the ability to provide behaviour consistency and predictability, as providing more flexibility during execution leads to management difficulties, such as deciding which variations are actually conflicting and how to resolve such conflicts.

Several ways to achieve manageability exist, and taken from [Car13], the following sections of this chapter are here to give a quick insight on what are the possibilities at hand for it.

Design patterns

The establishment of well-chosen design patterns inside the architecture of a SASS is useful to provide a structure to manage the variations. Especially, the use of dynamic proxies, the strategy pattern or even the observer pattern ensures variability to some extent. However, used alone, they tend to not provide enough flexibility, as it has to be fully determined beforehand.

Dependency injection

Although the concept of dependency, which will be depicted later, and the ability to separate variations from the base architecture are important, it is not suitable on its own because it also has to be decided beforehand, and it is not suited to provide variation composition.

Event-driven behaviour

The ability for a system to react to events is promising, as external events triggered from the surrounding environment give an opportunity to structure them and assign a variation to each occuring event. However, it provides little flexibility in terms of variation complexity, as it is mainly achieved through callbacks, and as such, it complexifies the maintenance and debugging processes.

Metaprogramming

Moving to language-specific solutions, and as it will be seen later in Part II, metaprogramming is surely promising in providing nearly absolute flexibility and control for variations installation, at the cost of consistency and predictability. Especially, the intercession property, the ability to capture and modify the state of a system, will prove to be central in this thesis.

Reactive programming

Seen as a refined combination of dependency injection and event-driven programming, reactive programming gives event detection and management at a reduced cost. However, the variation

† At (pre)compilation time, at link time or at start-up for example.

definitions are so fine-grained, that the bigger the multiplicity of the variations is, the more difficult it is to combine the event conditions, leading to maintenance and performance difficulties.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [Kic+97] represents the different variations of a system as aspects and makes it possible to structure them for better manageability. It will prove to be quite useful to formalize variations in subsequent chapters, as the architecture depicted in Part II is partly inspired from this paradigm.

2.3 Conclusion

The purpose of this chapter was to introduce the reader to elements of self-adaptive, and more generally dynamically adaptive software systems, in terms of variability [And+09; GB03; CBK13] and management [Car13]. The basic idea is to give a system means to adapt its behaviour depending on its surroundings, specifically at runtime, and to give means to the developer to manage the adaptable system in terms of flexibility, control, predictability and consistency mostly.

Chapter 3

Context-Oriented Programming

Contents

3.1 Introduction **11**
3.2 Concepts **11**
3.2.1 Context 11
3.2.2 Variation 12
3.3 Context-Aware Systems **12**
3.3.1 Variations management 13
Management through layers 13
Variation selection 14
3.3.2 Context management 14
Context relationships 14
3.4 Conclusion **16**

3.1 Introduction

COP[RN08] is a paradigm on top of Object-Oriented Programming (OOP) that shares some properties with AOP [Kic+97] in the sense that it separates the structure of the base architecture of a system from the structure of its behavioural variations, providing a mechanism to intall and/or remove them in a way that is easy to maintain and control, while preserving consistency across executions [Car13]. It is accomplished through the establishment of a hierarchy of contexts from which it is decided which variation should be applied to the base system. Those contexts describe the environment in which the software system is evolving and executing, and under what situations it is confronted, and such systems are called context-aware systems.

The following sections will illustrate the key concepts COP and context-aware systems in general.

3.2 Concepts

3.2.1 Context

A context is a first-class entity that describes one or several environmental properties. It is an abstraction of the external world that represents in programming terms one or several aspects of the surroundings of a system. [Col12; Gon+10].

It can be used especially to give a meaning to sensors output and to assign a certain behavioural variation when triggered, but it can also be related to the system's user and its

requirements, as it is linked to the concept of variability addressed in Chapter 2. An example can be seen in Table 3.1.

Sensor output	Output meaning	Context triggered
+50.715896, +4.612808	Latitude and Longitude	In Belgium
4.0	CPU base frequency (GHz)	High-end CPU
100	Battery level (mAh)	Low battery
10	Ambient temperature ($^\circ$ C)	Cold

Table 3.1: Mapping between environmental data and contexts.

Contexts are modelled in order to assign them different system variations, depending on their activation state, such that the context-aware system behaves as intended by the user or as it should, provided its environment. It is thus a way for the user to influence the behaviour of a system meaningfully without actually changing the program himself, with the hypothesis that the developer of the system assigned a variant to that influence.

A schematic way to represent contexts in a context-aware system can be seen in Figure 3.1.

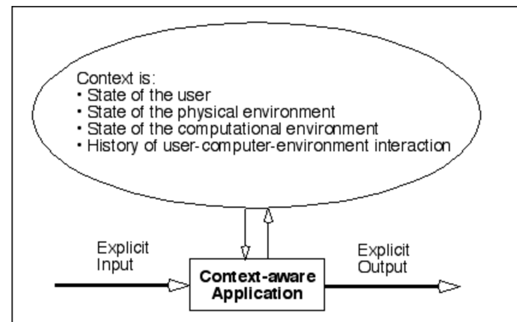

Figure 3.1: Schematic view of the meaning of contexts in a context-aware application. From [LS00].

3.2.2 Variation

A behavioural variation, as used several times in the self-adaptive context, represents a change, a *delta*, that is applied to the base system to provide, adapt and/or remove execution behaviours. Such variation is for example represented as an aspect in AOP or a callback procedure in event-driven programming systems, and a variant represents the result of applying different variations to a system [Col12].

These variations in COP are context-dependent, so that when a context condition is satisfied, it activates and triggers the mapped variations. A very simple example of such behaviour is given in Listing 3.1.

3.3 Context-Aware Systems

A Context-Aware System (CAS) brings the notions of context and variation to it, in order to make his multiple behaviours context-dependent [Tan+06]. Context-awareness leads to the needs of proper context and variation management [Raf14]

```
# Given a class User having a feeling method depicting its state
# regarding the ambient temperature
class User
  def feeling
    print "I am feeling normal"
  end
end

# This variation of the feeling method triggers and replaces
# the initial behaviour when the context Cold activates
def feeling
  print "I am cold"
end

# This variation triggers and replaces the previous behaviour,
# whether the context was Normal or Cold, when the new context
# is Hot
def feeling
  print "I am warm"
end

# This variation is triggered when the context is Freezing.
# Because Freezing also ensures Cold, the subsequent variation can
# call the variation that is in the supercontext
def feeling
  super.feeling
  print ", even freezing !"
end

user = User.new

# When the context Normal is active (by default)
user.feeling  # "I am feeling normal"

# When the context Cold is active
user.feeling  # "I am cold"

# When the context Hot is active
user.feeling  # "I am warm"

# When the contexts Cold and Freezing are active
user.feeling  # "I am cold, even freezing !"
```

Listing 3.1: Given the contexts `Normal, Cold, Hot` and their different activation states, the corresponding variations are applied to a given base class.

3.3.1 Variations management

Several implementations of variations management at language level were conducted to explore different ways to handle them in various aspects (performance, composition, structure, selection, . . .) [Car13; App+09; Nic15; Thi12; GPS10; Lin+11; Col12].

Management through layers

One of the early implementations at language level was *ContextL* [App+09], where behavioural variations were structured in layers in which partial definitions were included. These layers were stateless and were made to be composed with each other, so that the resulting multi-layered system would behave completely and accurately according to its surroundings. Layers are equivalent to contexts in the sense that they represent a certain state of the environment, but the main difference is that contexts are stateful and they can be reasoned with to ensure a more intelligent activation mechanism [Car13].

When declaring layers, two types of declarations have been defined [Car13; App+09], namely Class-In-Layer (CIL) declarations and Layer-In-Class (LIC) declarations. Both are about dealing with the lexical scope of the layers definition.

In CIL, the layers are defined completely apart from the base system, respecting the *separation of concerns* principle. It also provides a better scalability to the system, as it is possible to create new layers without interfering or redefining existing ones. However, it makes the maintenance process more difficult, as it is more cumbersome to track the execution of a certain behaviour when it has to go through several layers, because the base system and the variations are loosely coupled.

In LIC, the layers are defined inside the base system, providing access to the internal state of the base classes, as opposed to CIL. It provides better understandability and less scattered definitions, as the layers are tightly coupled with the classes they adapt, at the cost of losing scalability, as adding new layers in classes may mess with the existing ones.

Variation selection

The selection of the variations to install or remove depending on the currently active contexts also varies from one implementation to the other [App+09]. Most of them manipulate the method lookup chain of the language extended with COP in order to interpose their behavioural variations. The structure of the whole context-aware system is preserved through proxies, `proceed` methods or even dynamic dispatch when languages provide good meta-level functionalities.

Other implementations, such as Subjective-C [Gon+10], proposed another way for variations establishment. They provided a mechanism called `predispatching`, in which the definitions of the methods themselves were determined at runtime, in order to ease the lookup process.

Both of those techniques are in the end a matter of a tradeoff analysis between methods calls and variation selection frequency, as when the former is more frequent than the latter, the way proposed by Subjective-C offers better performance compared to method lookup chain manipulation and vice-versa. Also, some provide a way to preserve consistency through asynchronous, thread-based or scoped selection [App+09; Kam+15], some others through concurrency techniques in order to not disrupt the system's execution when new selections need to take place [GPS10; SGP15].

3.3.2 Context management

Bringing the notion of context management is one of the contributions to COP [RN08] that makes it different from AOP which considers stateless layers as aspects. It provides the ability to consider relationships between contexts in order to better grasp the physical environment or the user's requirements at some point in time, and having a complete context model inside a context-aware system makes it more prone to inconsistencies and better ensures predictability, allowing to define conflict resolution mechanisms in order to provide the right adaptation chain for the right composition of contexts.

It is also a way to better link the virtual world of implemented behaviour to the tangible real world for maintenance and monitoring purposes, as it offers a better way to track and tackle context-dependent implementation problems [Raf14].

A schematic view of context management inside a context-aware system can be seen in Figure 3.2.

Context relationships

Assigning relationship constraints between contexts allows to structure them between each other and is used for conflict resolution policies [Thi12]. It is part of context modeling in which contexts can be represented as directed graphs, where the arrows are the relationships or dependencies [Car13].

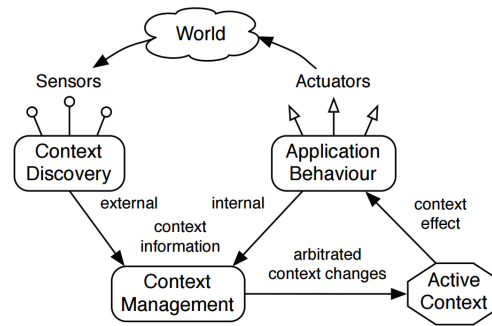

Figure 3.2: Schematic view of the links between the physical world, its translation into contexts and the application behaviour. From [Thi12; Gon+10].

Examples of relationships can be seen in Figure 3.3 and an illustrative example of a complete context graph is shown in Figure 3.4.

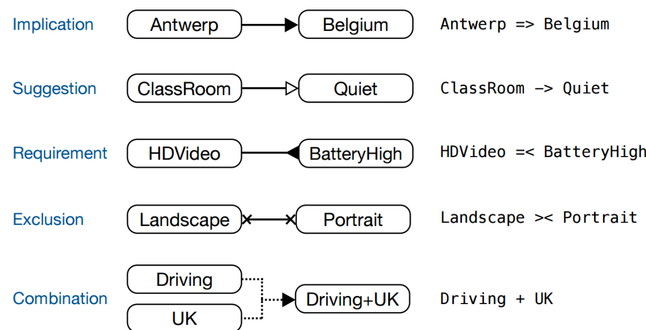

Figure 3.3: Examples of context relationships used in context management. From [Men15].

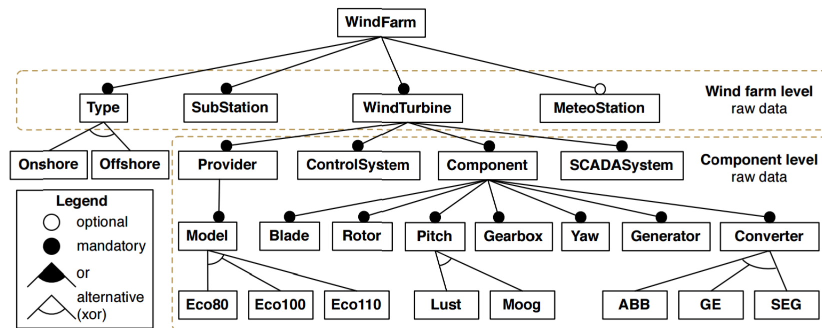

Figure 3.4: An illustrative example of a wind farm context dependency graph. From [Mur+14].

From Figure 3.3, the different kinds of relationships are as follows [Thi12; Car13], for two generic contexts `A` and `B` :

Implication `A` *implies* `B` if and only if the activation of `A` provokes the activation of `B`. `A` is tightly coupled to `B` in the sense that `A` is constrained to have the same activation state as `B`. It shares the same properties as the mathematical implication between two sets. From the example in Figure 3.3, if `Antwerp` activates, `Belgium` has to activate as well, otherwise it would not make sense to be in a city of a country, but not in the country itself. Likewise, if `Belgium` deactivates, `Antwerp` also deactivates for the sake of consistency. It is also called a **strong inclusion** [Car13].

Suggestion `A` *suggests* `B` if and only if the (de)activation of `A` triggers the (de)activation of

B. The difference with an implication is that both contexts are loosely coupled in the sense that there can be an activation state where A is active and B is not*. Hence, the (de)activation of A is not prevented, regardless of the state of B, but B can always change its state subsequently from A. It is also called a **weak inclusion** [Car13]. From Figure 3.3, the activation of ClassRoom suggests the activation of Quiet, but it is not mandatory for Quiet to be active to have ClassRoom also active.

Requirement A *requires* B if and only if the activation of A is only possible if B is already active. It is similar to the implication constraint, but differs from the fact that it imposes B to be active before trying to activate A†. From Figure 3.3, HDVideo requires BatteryHigh in the sense that BatteryHigh has to be active first before activating HDVideo. Even though the difference with an implication is subtle, it better represents the environment, as BatteryHigh should not be subsequently activatable obviously‡. This subtlety is more a matter of design and dealt environment, as both constraints could be interchangeable, but one can be more accurate than the other in certain situations.

Exclusion A *excludes* B if and only if the activation of A is only possible if B is currently inactive and vice-versa. It means that both activation states must be opposite to each other, or both be set to inactive. It is the exact opposite of a requirement constraint, as the activation state of B is checked before trying to activate A. From Figure 3.3, Landscape can only be activated if Portrait is currently inactive, as both contexts cannot be active at the same time by any means§.

Combination (optional) AB is the combination of A and B if and only if the activation of A and B triggers the activation of AB. It is mostly used to provide a behavioural variation that would only make sense if both A and B are active at the same time. But it is marked as optional, as it is equivalent to two requirement constraints and one could simply create a subcontext of A and B with two requirements, and store the variation in it. From Figure 3.3, the activation of Driving+UK is possible only if Driving and UK are both currently active.

3.4 Conclusion

The purpose of this chapter was to present the concepts of COP to the reader, especially the notions of context as opposed to layers, and variation which was already used when introducing self-adaptive systems.

This chapter also gave an insight on context-awareness and the need to provide a proper context and variation management to a COP-extended language. For variations, the concept of selection was addressed, and for contexts, the notion of relationships and dependency graphs was illustrated.

*For example when the activation of B failed when activating A for some reason related to other dependency checks.

†As opposed to implication, where the activation of A can trigger the activation of B if it is not active.

‡Because it can be defined for example by physical sensors measuring the battery level of a smartphone.

§As in a smartphone, the screen cannot be in landscape mode and in portrait mode at the same time.

Chapter 4

Feature-Oriented Programming

Contents

4.1 Introduction . . . 17
4.2 Concepts . . . 17
4.2.1 Feature . . . 17
4.2.2 Feature modeling . . . 19
4.3 Comparison with COP . . . 19
4.4 Conclusion . . . 19

4.1 Introduction

Feature-Oriented Programming (FOP) [Pre97] is also a paradigm like COP that is on top of OOP. It has been created to represent a system's functionalities as features where code composition and reuse is key. In OOP, inheritance and subclassing are the key concepts to enhance software reusability, but suffers from limitations such as multiple inheritance or flexible composition, and FOP was built to address them.

Although COP brings fine-grained behavioural variations, FOP brings a more coarse-grained approach through features, which inspired several researchers to combine features and contexts in a similar fashion in order to build context-aware, functionality-oriented software systems [Raf14; Thi12; Ach+09; Car+11].

The following sections will describe the concept of feature and compare the FOP paradigm with COP.

4.2 Concepts

4.2.1 Feature

"*A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option*" [AK09].

Similar to contexts, features aim at representing the user's functionality requirements and are structured to describe the different behaviours of a system. In FOP, and more generally in Feature-Oriented Software Development (FOSD) [AK09], a system can be implemented completely from a set of features representing all its functionalities, following an incremental approach.

Features can be coarse-grained, as they can represent a complete package of functionalities, but they can also be fine-grained, as they are also similar to the aspects of AOP, providing only small bits of functionalities, like partial classes or methods [Car+11].

Hence, features provide flexibility and reusability, as they are intended to be composable and commutable. These properties are thus quite useful when considering SASSs, as a whole dynamic system can be built upon features acting as behavioural variations [GS10], even though features setup is meant to be done at compilation time through annotations in the source code, special structures provided by the FOP-extended language such as modules in Ruby or traits in Scala, or refinements such as aspects from AOP [Car+11; Thi12].

A simple example of a coarse-grained feature in Ruby can be seen in Listing 4.1.

```
# An example of a coarse-grained feature represented by a module
# containing a Counter implementation that can be reused for other purposes
module Counter
  def initialize
    @count = 0
  end

  def reset
    @count = 0
  end

  def inc
    @count += 1
  end

  def dec
    @count -= 1
  end

  def size
    @count
  end
end

# An example of a class where the objects are able to count cars
# that go through a tunnel
# It is simply able to count cars, reset the counter and retrieve
# the current value
class CarsCounter
  include Counter

  def count_car
    inc
  end

  def reset_counter
    reset
  end

  def cars_counter
    size
  end
end

# An example of execution
c = CarsCounter.new

c.count_car
print c.cars_counter    # => 1
c.reset_counter
print c.cars_counter    # => 0
```

Listing 4.1: Example of a coarse-grained feature applied to a given class.

4.2.2 Feature modeling

As for contexts in COP, features in FOP are subject to be structured via a feature model [AK09].

Because of their incremental nature, a badly designed model can result in inconsistencies or unexpected behaviours, which are addressed in the same way as with contexts, using dependencies and constraints similar to the ones already depicted in Chapter 3.

An example of a feature diagram can be seen in Figure 4.1.

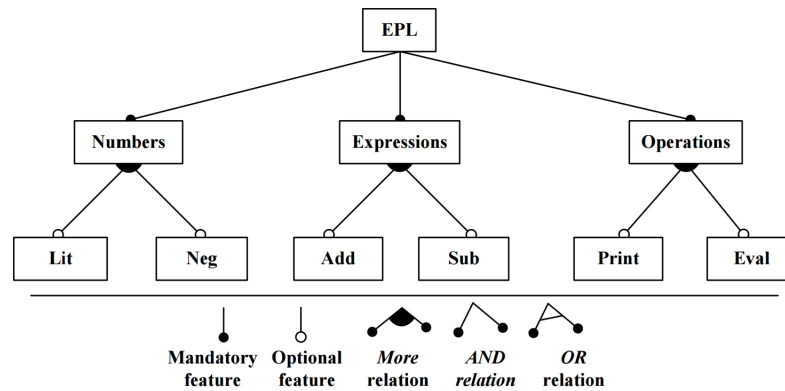

Figure 4.1: Example of a feature graph, taken from the Expression Product Line case study of [Car+11; GS09].

4.3 Comparison with COP

From [Car+11] addressing exactly the similarities and differences between COP and FOP via two languages (respectively Subjective-C [Gon+10] for COP and rbFeatures [GS09; GS10] for FOP), we can conclude that :

- Features represent the bits of functionalities of a software system, as contexts represent the link to the external world. Both describe different environments, but they can be associated to serve the main purpose, which is applying behavioural variations to a SASS at runtime.
- Both entities are first-class, in the sense that they can be manipulated and reasoned upon to preserve consistency and predictability, via similar context and feature graphs.
- Even though they can be treated with similar techniques, they tend to grow apart from each other in the litterature, as for contexts, researches try to dig further through how to incorporate more and more the environment into a CAS and act based on it [LS00], and for features, there is the need to pursue the idea of FOSD through Feature-Oriented Domain Analysis (FODA) to further develop the generalization and parametrisation aspects, as well as the need to represent non-functional requirements and propositional constraints [AK09; Thi12] not covered in COP. But this can only improve their collaboration in the future.

An interesting view of the characteristics of both languages reviewed can be seen in Figure 4.2.

4.4 Conclusion

The purpose of this chapter was to present the concepts of FOP, especially the notions of feature and modeling through a graph. From these concepts, a comparison with COP was presented in order to understand and see how they can work together or at least how they can benefit from each other.

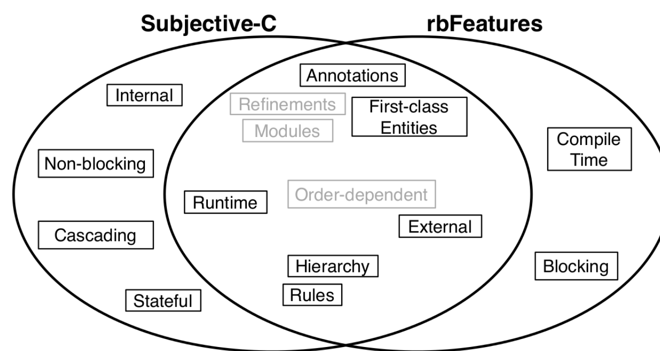

Figure 4.2: Comparison of the characteristics of Subjective-C and rbFeatures in terms of context-awareness. From [Car+11]

Chapter 5

Ruby

Contents

5.1 Introduction **21**
5.2 Ruby 2.0 . . . . **21**
5.2.1 Refinements 21
5.2.2 TracePoint 22
5.2.3 Module prepending 22
5.3 Conclusion **22**

5.1 Introduction

This chapter presents an overview about Ruby and its capabilities in terms of metaprogramming, and the major contributions of Ruby version 2.0 that provides several useful tools for the purpose of this thesis.

5.2 Ruby 2.0

Ruby is a language inspired from Lisp, Smalltalk and Perl, especially for metaprogramming capabilities [Per14; Thi12], where everything is an object. Released in 1995, Ruby version 2.0 was released in February 2013, with a bunch of new features.

Also, as far as popularity is concerned, Ruby is currently ranked eighth [TIO16] as of May 2016, showing that it is quite a stable language with an important fan base.

5.2.1 Refinements

Refinements are a way to refine a defined class at a lexical scope [Per14]. An example is given in Listing 5.1. The ability to reopen classes (even system ones) is one of the strengths of Ruby's metaprogramming capabilities, as it is possible to redefine objects on the fly anytime anywhere in the code.

Refinements were added in Ruby 2.0 to limit classes opening in a lexical scope, in order to not harm the whole system by doing it globally. It provides a temporary way to refine classes by adding a layer in the definitions lookup chain [Per14]. When a refinement is active, it takes precedence when calling a method for example, and it is disabled when reaching the end of a class definition or a file if it was activated at top-level with `using`.

```ruby
module ExtString
  refine String do
    def reverse
      "Hello Kim"
    end
  end
end

class A
  using ExtString
  def message(arg)
    arg.reverse
  end
end

a = A.new
a.message("Hello") # => "Hello Kim" instead of "olleH"
```

Listing 5.1: Example of a refinement.

5.2.2 TracePoint

TracePoint was introduced with Ruby 2.0 [Kon14] in order to replace the `set_trace_func` function that was known to be slow, with an object-oriented Application Programming Interface (API). It provides the ability to execute a given procedure when a particular event triggers, like a hook method. It is mainly used for profiling code, as the nature of the events can be from at each method call, to at each class declaration [Kon14]. Inside the procedure definition, several properties are available for use, such as the class that triggered the event, or the method that is being called for example.

5.2.3 Module prepending

Before Ruby 2.0, two ways of dealing with modules were available : `include` and `extend` [Per14]. When a module is included into a class, it is part of the ancestors of the class, in the same way as a superclass, such that it is possible to call a defined method in a module included with `super`. Extending a class with a module is roughly the same, except that it is part of the ancestors of the singleton class instead.

A singleton class in Ruby is a class that holds all the class variables, constants and methods of a given class [Per14]. Including a module holds for instance definitions, and extending holds for class definitions.

Ruby 2.0 brought the ability to prepend modules to classes [Per14]. Prepending a module is roughly the same as including, but instead of going up the ancestors chain of a class, it goes right below, in order to take precedence over the class in the definitions lookup chain. It was mainly released to solve the problem of chain aliasing, as before the version 2.0, the only way for a module to take precedence over a class, was to use aliases to redefine methods and variables [Per14]. This led to maintenance and debugging difficulties, as well as less readability, making the system more unstable.

An example of module prepending can be seen in Listing 5.2.

5.3 Conclusion

Ruby is recognized for its exhaustive metaprogramming capabilities and the version 2.0 brought even more features. Although refinements are a great way to control the scope of a feature, the fact that it is a lexical scope limits the ability to use it extensively in the concern of this thesis.

```ruby
module PrepA
  def message(arg)
    "Hello Kim " + super(arg)
  end
end

class A
  prepend PrepA
  def message(arg)
    arg.reverse
  end
end

a = A.new
a.message("Hello") # => "Hello Kim olleH" instead of "olleH"
```

Listing 5.2: Example of a module prepending.

TracePoint and module prepending however provide both native ways to use event-driven programming and to deal with definitions lookup chains. As prepended modules take precedence, it provides a native way to integrate behavioural variations in a base application, which will be intensively used to integrate features.

Chapter 6

Related work

Contents

6.1 Introduction 24
6.2 Phenomenal Gem 24
6.3 rbFeatures 25
6.4 ContextErlang 25
6.5 Context Traits 26

6.1 Introduction

This chapter will present past works that influenced the development of this thesis and provided useful insights on what is possible to achieve and how.

6.2 Phenomenal Gem

Phenomenal Gem [Thi12] is one of the works that inspired the realization of the framework the most, even though nothing was taken from their implementation, because the approach is completely different. They implemented the concept of Context-As-A-Feature (CAAF), where contexts and features are mixed into the same concept, making it difficult to know what is exactly a context and how to distinguish it from a feature.

The framework depicted in Chapter 7 started from there where the main idea was to separate completely the concepts of contexts and features and to make them work side by side instead of mixed up, in order to clearly see the parts where COP is involved and the parts where FOP is involved.

Phenomenal Gem is a COP framework handling contexts as first-class entities, where adaptations can be defined via a Domain-Specific Language (DSL) in order to make context-aware applications. It represented an attempt to merge completely COP and FOP into one single paradigm. Also, in order to target mobile platforms and devices, it was implemented with the notion of Feature-As-A-Service (FAAS) in mind, and built in Ruby on Rails, the web framework of Ruby.

They also provided a viewer showing context graphs for debugging purposes, and they made a website which was completely implemented with Phenomenal, their COP Ruby gem [Thi12].

The global architecture of their work can be seen in Figure 6.1

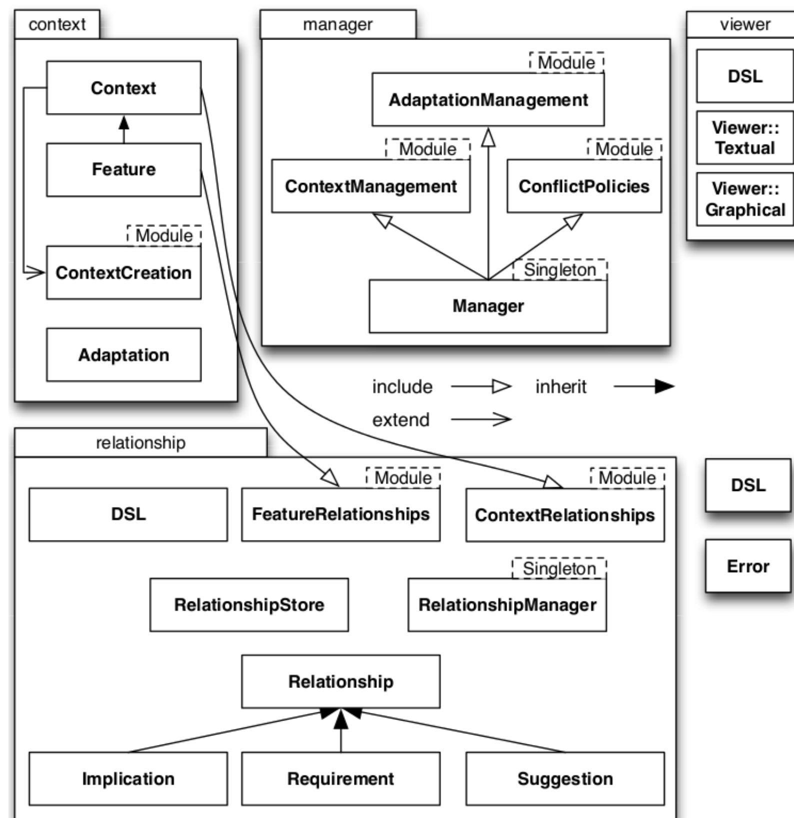

Figure 6.1: Global view of the architecture of the Phenomenal Gem framework [Thi12].

6.3 rbFeatures

rbFeatures [GS10; GS09] is an extension of Ruby providing FOP capabilities to the language, especially for Software Product Liness (SPLs). It allows features to be defined using annotations to have the freedom of defining coarse-grained or fine-grained features on the fly. The contents of a feature defined in rbFeatures can be seen in Figure 6.2. The core defines the fundamental methods that are needed to handle a feature, such as being able to (de)activate it. The operations are used to provide feature composition capabilities, as a feature code is defined in a containment with an activation condition, it is used in order to assign a code block to one or several features at once. And the method added hooks are meant for consistency purposes, such as defining an error if a method from a disabled feature is called or applying the visibility on a method whose body has been changed by a feature.

The way rbFeatures provides features at source code level inspired the work of this thesis, especially for the dynamicity that rbFeatures offers [GS10], as they also define a feature graph for features management.

6.4 ContextErlang

ContextErlang [SGP12; SGP15; GPS10] is an extension of Erlang providing COP capabilities to the language. The purpose of developing in Erlang was to use its distributed and concurrency capabilities in order to provide concurrent behavioural variations. This work focuses more on consistency preservation, which turned to be very useful for this thesis, as the framework has self-adaptive capabilities and runs concurrently with a running application attached to it.

ContextErlang relies on context-aware reactive agents whose behaviour can support context-

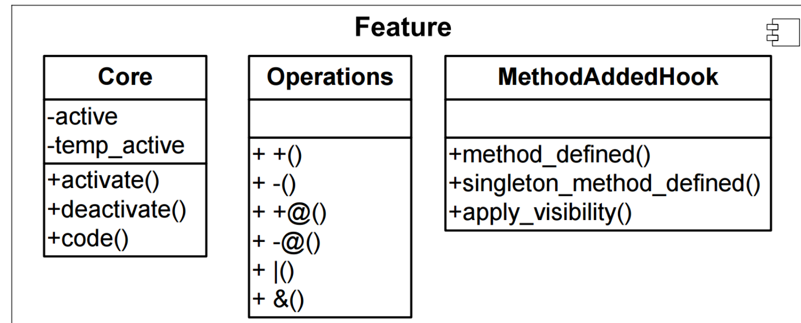

Figure 6.2: Feature model of rbFeatures [GS09].

dependent variations. A variation is activated by overriding methods definition. It offers formal semantics in order to provide context-awareness to any application implemented in Erlang.

Code snippets of the callback module and an example of a variation can be seen in Figure 6.3 and Figure 6.4 respectively.

```
-module(user).
-behavior(context_agent).
-include("context_agent_api.hrl"). % contextual API
% API
receive(AgentId, Source, Msg) ->
context_agent:cast(AgentId, {receive_msg, Source, Msg}).
add(AgentId, Dest, Msg) ->
context_agent:cast(AgentId, {send_msg, Dest, Msg}).

handle_cast({receive_msg, Source, Msg}, State) ->
    % ... forward to my client
    {noreply, State}.
handle_cast({send_msg, Dest, Msg}, State) ->
    % ... forward to dest client
    {noreply, State}.
% startup, shutdown and other auxiliary functions
```

Figure 6.3: Callback module of a context-aware chat in ContextErlang [SGP12].

6.5 Context Traits

Context Traits [Gon+13] is another extension providing COP to JavaScript, focusing especially on composition aspects of behavioural variations using traits. Whereas ContextErlang was focused on consistency and concurrency aspects, Context Traits is focused on traits composition and means to resolve potential conflicts via conflict resolution policy definitions, and to invoke the redefined behaviour from a variation through a proceed mechanism.

The context detection mechanism is mainly based on event triggers and listeners, and each variation is defined in a trait. It is also possible to define composition policies in order to indicate for example which trait has precedence.

As opposed to this thesis' framework which uses explicit context dependency relationships

```
—module(offline).
—context_cast([receive_msg/2]). % Contextual dispatch
    ...
handle_cast({receive_msg, Source, Msg}, State) —>
    store_chats:store_message(Source, Msg),
    {noreply, State}.
```

Figure 6.4: Example of a variation in ContextErlang [SGP12].

as part of a context definition, in Context Traits, it is done manually through the composition policies that define the relationships between traits.

An example of traits composition can be seen in Figure 6.5 and an example of a trait definition in Figure 6.6

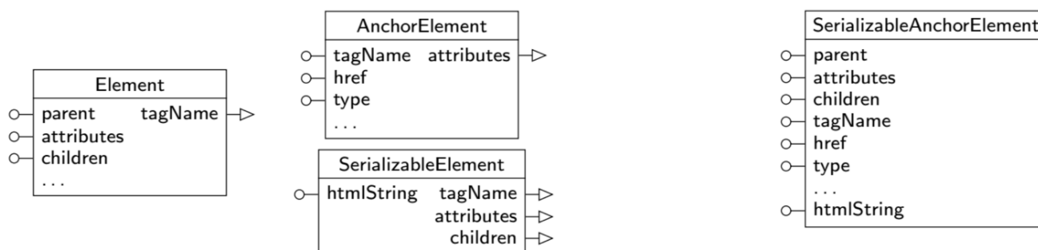

Figure 6.5: Example of traits composition for an HTML DOM library [Gon+13].

```
1  FastSerialization = Trait({
2    htmlString: function() {
3      var result = this.retrieve(); // probe cache
4      if (result == undefined) {    // cached?
5        result = this.proceed();    // get serialisation
6        this.store(result); }       // cache it
7      return result; }});           // return stored value
8
9  Production.adapt(HTMLElement, FastSerialization);
```

Figure 6.6: Example of a trait definition [Gon+13].

Part II

Framework

Chapter 7

Self-Adaptive Framework

Contents

7.1 Introduction **29**
7.2 Global architecture **30**
7.3 Interaction layer **31**
7.4 Discovery layer **31**
7.4.1 Interpretation 31
Filters 32
7.4.2 Reasoning 33
Context declarations 33
7.5 Handling layer **34**
7.5.1 Context handling sublayer 35
Context Activation 35
Context dependency declarations 36
7.5.2 Feature handling sublayer 36
Feature Selection 37
Feature Activation 37
Feature Execution 37
7.6 Conclusion **37**

7.1 Introduction

The implementation of the framework and the making of its design have been realized by and with the student Benoît Duhoux in Ruby version 2.3.0, where he focused more on contexts and user interface transition aspects, while I focused my work on features and behavioural variations. His work can be found in [Duh16], especially for all the transition-specific properties and units of the framework that will not be detailed here, as they treat user interface issues and challenges that go beyond the scope of this thesis.

Also, even though most of the work depicted here was inspired by Phenomenal Gem [Thi12], everything was implemented from scratch, as the approach is completely different. The sources of this implementation can be found on `https://github.com/dsarkozi/COP2016`.

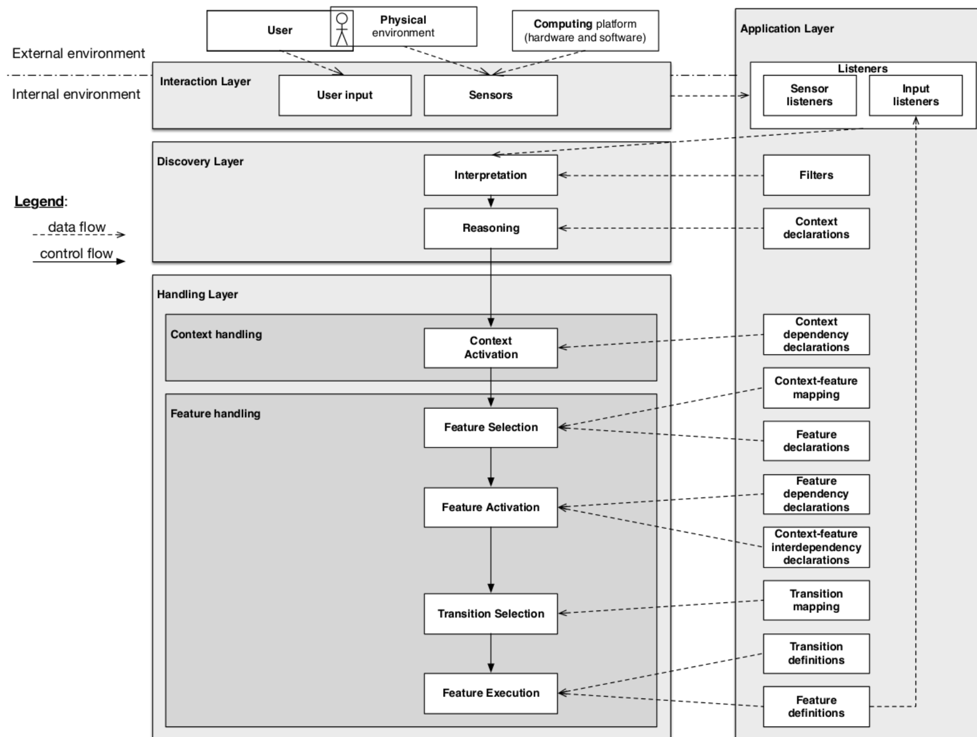

Figure 7.1: Global view of the architecture of the framework.

7.2 Global architecture

This section will present the general architecture and design of the Self-Adaptive Framework (SAF), before diving into the core concepts and functionalities.

Figure 7.1 show the global architecture of the framework. It is composed of several layers of abstraction, implemented in a top-down fashion. Each layer passes the control flow from the top one all the way to the bottom one, and each unit inside each layer represented by a box is passed relevant data flows depending on its use and role.

The top end of the framework architecture is responsible for dealing with the external environment, in order to translate it to sensors and user input abstractions. Those units forming the **Interaction layer** will represent the explicit input of the framework, like in Figure 3.1.

Since the architecture represents a SASS, an **Application layer** is established in order to provide all the application-specific functionalities and properties to the SAF. For the output of the sensors, listeners are provided in order to pass raw but useful data to the internals of the framework.

This raw data is interpreted and reasoned upon in the **Discovery layer**, where it is transformed into fully expressed contexts like in Table 3.1, based on the provided context declarations and data filters from the Application layer. It is also to determine which context(s) should be (de)activated based on current sensors data flows.

This information is delivered to the **Handling layer** which distinguishes two types of handling, namely *Context* handling and *Feature* handling. The intent is to separate completely the processing of context-specific properties and issues from the feature-specific ones.

The role of the **Context handling sublayer** is to process and manage the activation states of the contexts, especially to (de)activate contexts based on the data flows coming from the sensors and the output of the reasoning process, but also based on the context dependencies

defined by the Application layer*. The result is then transferred to the Feature handling sublayer.

The **Feature handling sublayer** is intended to select the set of features that need to be activated in order to provide the functionalities to the running application that match best its current surroundings. Especially, its role is to adapt the activation states of every feature of the application from the current state of the contexts, in order to produce in the end an execution that is expected and consistent with the current state of the external environment.

It is done through a feature selection process which selects the relevant features from the currently active contexts based on a context-feature mapping provided by the Application layer. Then, a feature activation process is attempted with this information, which tries to (de)activate features based on the feature dependencies defined in the Application layer. From this, a set of relevant features is produced and retrieved to the transition and execution processes, where a transition selection is realized for interface variations purposes [Duh16], and the feature execution is intended to install and execute the activated features in the running application and also to uninstall the deactivated features from it, making them unavailable.

The main idea of the SAF is to provide a self-adaptive framework with complex context and feature decision-making processes, which results are applied to a given complete application. That application has to provide its properties to the SAF in order to make the decision-making processes work as intended by its developer.

7.3 Interaction layer

The Interaction layer represents an abstraction of the external world, mainly user input and sensors. Since the SAF was not implemented on a machine that is equipped with readable sensors, the interactions between the sensors, the listeners and the Interpretation process have been replaced by mockups for experimental purposes.

Most of the data that is flowing throughout the SAF is encoded in JavaScript Object Notation (`JSON`) for easy processing. The sensor abstractions are built using a custom pattern Observer mixin, where each sensor class is observable and the `notify` method is launching the Interpretation process of the sensor instance passed as a `JSON` object.

The implementation of the mixin can be seen in Listing 7.1.

Although it is possible to create its own sensor classes via dynamic creation, the SAF provides default classes for common sensors, such as GPS, temperature or even brightness. An example of a sensor class, such as `TemperatureSensor` can be seen in Listing 7.2.

7.4 Discovery layer

The Discovery layer mainly encapsulates the Interpretation of the raw data produced by sensors, filtered by dedicated filters provided by the Application layer.

7.4.1 Interpretation

The code snippet of the Interpretation process can be seen in Listing 7.3.

The `interpret` function takes a `JSON` object as argument, that contains the `JSON` representation of a sensor object. It parses its input, then filters it via the filters provided by the Application layer, to eventually send it to the Reasoning process if it passes the filters.

*As two different applications with the same set of contexts could define different dependency relationships for each pair of contexts.

```ruby

# Mixin Observable (pattern-like Observer)
module Observable
  def set_data(data)
    self.update_state(data)
    notify
  end

  def notify
    Discovery::Interpretation.instance.interpret(self.to_json)
  end

  def to_json
    hash_json = Hash.new()

    regexClassNameSensor = /(.+::)*(.+)/
    classNameSensor = self.class.name
    classNameSensor = regexClassNameSensor.match(classNameSensor).captures.last
    hash_json['sensor'] = classNameSensor
    regexClassName = /(.+)Sensor/
    hash_json['class_name'] = regexClassName.match(classNameSensor).captures.last

    regex_inst_var = /@(.+)/
    inst_vars = self.instance_variables
    inst_vars.each {
    |inst_var|
        name_var = regex_inst_var.match(inst_var).captures.first
        hash_json[name_var] = self.instance_variable_get(inst_var)
    }

    hash_json.to_json
  end
end
```

Listing 7.1: Mixin implementing the Observer pattern.

There is also a function `intepretDefault` that is used mainly for bootstrapping purposes, in order to start the SAF from a default context and feature, as the whole framework is a self-adaptive process.

Filters

Filters are one of the first units that are application-specific, as they provide ways to tamper with sensors output. They are useful for delaying sensors data reasoning if for example, an application does not need to get GPS coordinates every second, but only every minute or so. As one reasoning leads to running the whole framework until it produces relevant features, for the sake of performance, applications would want to limit that, and filters provide such tampering behaviour.

A `Filter` class contains simply a `filter(o)` method that must be overriden for every kind of sensor object `o`. It is a boolean operator that returns true if `o` has to be sent to the reasoning process, and false otherwise. Also, the class implementing the `filter` method must inherit from a `Filter` class type, and must be called with the same name as the sensor class, concatenated with `Filter`, for easier processing by the SAF.

Two kinds of filters are natively supported by the SAF, namely time filters and including filters. Time filters are used for delaying sensor data reasoning and including filters are used to selectively reason on a subset of sensors data.

The implementation of the time filter class can be seen in Listing 7.4, and the including filter class in Listing 7.5.

```
# Abstract sensor class
class ApplicationSensor

  include Observable

  def self.factory(data)
    object = self.new(data)
    app = object.notify
    return object, app
  end

end

class TemperatureSensor < ApplicationSensor

  attr_reader :degree

  def initialize(data)
    update_state(data)
  end

  def update_state(data)
    @degree = data['degree'].to_f
  end
end

class GpsSensor < ApplicationSensor

  include Observable

  attr_reader :latitude, :longitude

  def initialize(data)
    update_state(data)
  end

  def update_state(data)
    @latitude = data['latitude'].to_f
    @longitude = data['longitude'].to_f
  end
end
```

Listing 7.2: Example of a sensor class.

7.4.2 Reasoning

The Reasoning process takes place right after the Interpretation of sensors data, and its role is to execute a depth-first search on a context graph based on the context declarations provided by the Application layer (in a `JSON` format) to retrieve all the relevant contexts based on activation conditions. It can also yield the default context when bootstrapping the SAF.

The code snippet of the Reasoning process can be seen in Listing 7.6.

Context declarations

A set of context declarations has to be provided by the running application in order to create the context graph. The graph includes for each context all its subcontexts, the sensors to which it is applied if relevant, its activation condition(s) and its abstract property. A context is abstract if its activation only makes sense if at least one of its subcontexts is also activated.

Taking the example from Listing 7.7, the abstract context `Temperature` is active only if at least one from `Hot`, `Normal`, `Cold` or `Freezing` is also active. `Temperature` is also abstract because it does not describe a concrete state, as opposed to a subcontext like `Hot`.

```
class Interpretation

  include Singleton

  def interpret(json)
    hash_data = JSON.parse(json)
    class_name = hash_data['class_name']
    object = parser(hash_data)

    if filter(class_name, object)
      Reasoning.instance.reason(object)
    else
      raise Exceptions::FilterException, "No data filter due to filters"
    end
  end

  # Bootstrapping method
  def interpretDefault
    Reasoning.instance.reasonDefault
  end

  private

  def filter(class_name, o)
    appName = Application::Config.instance.appName
    filter_class_name = class_name << "Filter"
    filter_class_exists = Application.const_get(appName).const_defined?(filter_class_name
    )

    if filter_class_exists
      filter_class = Application.const_get(appName).const_get(filter_class_name)
      return filter_class.instance.filter(o)
    else
      raise Exceptions::FilterException, "No filter existing."
    end
    return false
  end

  def parser(hash_data)
    sensor = hash_data['sensor']
    hash_data.delete('sensor')

    class_name = hash_data['class_name']
    hash_data.delete('class_name')

    create_instance(sensor, class_name, hash_data)
  end
end
```

Listing 7.3: Interpretation of sensor raw data (JSON).

7.5 Handling layer

The Handling layer represents the main process of the SAF, as it includes both context and feature decision-making processes or *managers*, and it contains one of its most important contributions compared to a CAAF approach like PhenomenalGem [Thi12], because contexts and features are managed in a completely separated fashion, and it provides new ways to consider both concepts as parts of a same system, like COP and FOP merged in the same paradigm.

Hence, it contains two sublayers, one for context management and one for feature management, and both are highly dependent on application-specific properties, in order to provide finer options to the SAF on a per-application basis.

```
class TimeFilter < Filter

  def initialize
    @wait_to_filter = 0
    @new_wait_to_filter = 0
    @time_now = Time.now
  end

  def filter(o)
    if Time.now >= @time_now + @wait_to_filter
      @wait_to_filter = @new_wait_to_filter
      @time_now = Time.now
      return true
    else
      return false
    end
  end

end
```

Listing 7.4: Time filter class, used to delay the reasoning of sensors data.

```
class ExistsFilter < Filter

  def initialize
    @list = []
  end

  def exists?(attribute)
    @list.include?(attribute)
  end

end
```

Listing 7.5: Including filter class, used to select a subset of sensors data to send to the reasoning process.

7.5.1 Context handling sublayer

The context handling sublayer comes right after the reasoning process explained above. It is mainly composed of a single processing unit, namely the Context Activation process, which provides a decision-making system in which contexts retrieved by the reasoning process above are changing their activation state, based on the context dependency declarations provided by the Application layer.

Context Activation

The code snippet of the Context Activation process can be seen in Listing 7.8.

Each context has an activation counter, such that if one is activated several times through dependencies or other means, an activation counter gives the ability to make a difference between a deactivation that only decrements the counter and a true deactivation, where the context must be really disabled afterwards.

As such, given a set of relevant contexts retrieved by the reasoning process above, the context activation process first gets the activation counters object that handles all the currently active contexts, and checks if the default context is active. If it is, it should be deactivated as an activation process is currently running[†].

Then, for each relevant context, the process checks if they can be activated or if a dependency

[†]As the default context is only a placeholder waiting for true contexts to activate and trigger behavioural variations subsequently.

```ruby
class Reasoning

  include Singleton

  def reason(o)
    nameClass = o.class.name

    currentContextsGraph = Application::ContextDefinition.instance.getEntitiesGraph(
    nameClass)

    if currentContextsGraph
      admissibleContexts = getAdmissibleContexts(currentContextsGraph, o)

      if admissibleContexts.empty?
        admissibleContexts = [Application::ContextDefinition.instance.getEntitiesGraph]
      end

      return Handling::ContextActivation.instance.activate(o, admissibleContexts)
    end
  end

  # Bootstrapping method
  def reasonDefault
    defaultContext = Application::ContextDefinition.instance.getEntitiesGraph
    return Handling::ContextActivation.instance.activateDefault(defaultContext)
  end

end
```

Listing 7.6: Reasoning on the interpreted data.

exception should be raised, as if conflicts were detected. The same process is run upon deactivation, despite the naming of the methods of Listing 7.8. Eventually, the subset of the relevant contexts that were actually impacted in their activation state are sent to the feature handling layer.

Context dependency declarations

Apart from the context declarations, a running application also has to provide a set of context dependencies, such as the ones shown in Chapter 3. These dependencies are defined in two ways :

- A context contains a relationship type that lists all the contexts it is related to.
- A context contains only the relationship type with an empty list, which means that the relationship must be applied between all its subcontexts.

Also, the SAF contains an additional dependency similar to the exclusion relationship, which can be called as a *toggle* relationship. It has the same effect as a logical eXclusive-OR (XOR), where a context `A` *toggles* a context `B` if and only if the activation of `A` triggers the deactivation of `B` if it is active, and vice-versa. It is useful when the SAF should switch from a certain state to another.

Taking the example from Listing 7.9, a context `Temperature` containing `Hot`, `Normal`, `Cold` and `Freezing` as subcontexts has a XOR dependency between all of its children so that the SAF can switch from a `Hot` state to a `Cold` state for example. Also, the temperature `Normal` requires `Europe` to be active to be able to activate itself for example.

7.5.2 Feature handling sublayer

The feature handling sublayer, similarly to the context handling sublayer, contains a decision-making process that manages all the feature-specific properties of the SAF, based on the contexts retrieved by the context activation process and the different feature properties of the running application.

Feature Selection

The code snippet of the Feature Selection process can be seen in Listing 7.10.

Similarly to contexts, features are also structured in a dependency graph, in which they also share the same relationship types. The idea is that even though contexts and features are treated separately, they are both considered as entities in the SAF and thus share some properties, like both having a graph or both having an activation counter[‡].

Given the contexts that changed their activation state during the previous context activation process iteration, newly (de)activated contexts are determined in order to get the relevant features that are associated to them. Then, the selection process begins where the features to (de)activate are determined based on a context-feature mapping file (in `JSON`) provided by the Application layer. This step is done in order to act only on features that are concerned by the changing state of contexts, to prevent any unnecessary overhead of having to rewrite the state of all the features of the SAF.

Then, the features to actually (de)activate are retrieved from the feature graph that was produced based on the feature declarations also provided by the running application. They are eventually sent further to the feature activation process. Also, the default feature is activated when bootstrapping the SAF, in order to provide a default behaviour to the running application, which is simply the application's class hierarchy without any behavioural variations applied to it, and it is possible to fall back to it if needed, so the default feature can represent the core functionalities of an application.

An example of a context-feature mapping file is given in Listing 7.11, and feature declarations are mostly identical in shape to context ones in Listing 7.9.

Feature Activation

The code snippet of the Feature Activation process can be seen in Listing 7.10.

The feature activation process is quite similar to the context one, as both are subject to dependency conflicts that are raised when needed, and (de)activation attempts are made, based on the selected features given from above. There is also a bootstrapping method that activates the default feature, and it is deactivated when a feature activation process is in progress.

When the (de)activation checks have been made via the (context-)feature dependency declarations provided by the Application layer, the relevant features are sent to the transition selection process that is detailed in [Duh16].

Likewise, the dependency files have the same shape as the context dependency files in Listing 7.9.

Feature Execution

The purpose of the Feature Execution process is to first launch the application by default, then to retrieve the new state of features to be installed, with its definitions provided by the Application layer. This part will be the subject of Chapter 8

7.6 Conclusion

This concludes the description of the SAF. Its purpose is to represent a self-adaptive system in which contexts and features are dealt with separately, while offering means to make them work

[‡]Although features do not behave the same way regarding their activation counter as contexts, because it is sufficient to consider features to be in an active or an inactive state, regardless of their counter.

together. The motivation of choosing a self-adaptive system over a DSL was because hardware and software systems nowadays are more and more aware and influenced by their surroundings, and providing a framework that could self-adapt to its environment proved to be challenging but really interesting for the future.

The Interaction layer provides a way to translate the real physical world into programmable and adaptable units, that can be interpreted and reasoned upon via context discovery.

The complex Handling layer, dealing with both contexts and features provides the management needed to deal with dependencies and conflict resolution policies to match the environment best and make the system predictable and consistent regarding its surroundings. Although both are separated, there is a clear translation process between contexts describing the current state of the environment and features that provide the functionalities to better tackle it. For example, a GPS sensor is active and providing data. The sensor data is interpreted and the reasoning gives sense to it by linking to the corresponding `GPS` context. It tries to activate in order for the SAF to be aware of those data, then selects and activates the corresponding features that provide GPS functionalities. And those features are installed seamlessly into the currently running application.

Since most of this whole process is application-dependent, the SAF has an Application layer to provide all the properties of the running application from the sensors filters to feature definitions in order to act on almost all the stages of the framework, for the sake of flexibility.

Although the SAF can still be improved, it provides a good start to a potential collaboration between contexts and features, and more generally, between COP and FOP.

```
{
  "contexts": {
    "Default": {
      "isAbstract": false,
      "subentities": {
        "Temperature": {
          "isAbstract": true,
          "sensors": [
            "TemperatureSensor"
          ],
          "subentities": {
            "Hot": {
              "condition": "(TemperatureSensor.degree > 25)"
            },
            "Normal": {
              "condition": "((TemperatureSensor.degree <= 25) & (TemperatureSensor.degree >= 15))"
            },
            "Cold": {
              "condition": "(TemperatureSensor.degree < 15)",
              "subentities": {
                "Freezing": {
                  "condition": "(TemperatureSensor.degree < 0)"
                }
              }
            }
          }
        },
        "Gps": {
          "isAbstract": true,
          "sensors": [
            "GpsSensor"
          ],
          "subentities": {
            "Europe": {
              "condition": "(((GpsSensor.latitude < 67.190407) & (GpsSensor.latitude > 34.857088)) & ((GpsSensor.
     longitude > -27.509766) & (GpsSensor.longitude < 30.673828)))",
              "subentities": {
                "Belgium": {
                  "condition": "(((GpsSensor.latitude < 51.5) & (GpsSensor.latitude > 49.5)) & ((GpsSensor.longitude <
     6.4) & (GpsSensor.longitude > 2.55)))"
                }
              }
            },
            "Unknown": {
              "condition": "(((GpsSensor.latitude >= 67.190407) | (GpsSensor.latitude <= 34.857088)) | ((GpsSensor.
     longitude <= -27.509766) & (GpsSensor.longitude >= 30.673828)))"
            }
          }
        },
        "Brightness": {
          "isAbstract": true,
          "sensors": [
            "BrightnessSensor"
          ],
          "subentities": {
            "High Brightness": {
              "condition": "(BrightnessSensor.intensity > 50)"
            },
            "Low Brightness": {
              "condition": "(BrightnessSensor.intensity <= 50)"
            }
          }
        }
      }
    }
  }
}
```

Listing 7.7: Example of a context declarations file.

```
class ContextActivation

  def activate(o, contexts)

    activedContextsCounters = Application::ContextDefinition.instance.
    activedContextsCounters
    activedContexts = activedContextsCounters.clone
    defaultContext = Application::ContextDefinition.instance.getEntitiesGraph
    if activedContexts.getActivedEntities.include? defaultContext
      activedContexts.decreaseCounter(defaultContext)
    end

    managedContexts = {}
    begin
      contexts.each {
        |context|
        context.canActivate(activedContexts, managedContexts)
      }
      Application::ContextDefinition.instance.setActivedContextCounters(activedContexts)
    rescue Exceptions::DependencyException => e
      raise Exceptions::DependencyException, e
    end
    activedContextsCounters = Application::ContextDefinition.instance.
    activedContextsCounters
    FeatureSelection.instance.select(o, managedContexts)
  end

  # Bootstrapping method
  def activateDefault(defaultContext)
    activedContexts = Application::ActivedEntitiesCounters.new()
    activedContexts.increaseCounter(defaultContext)
    Application::ContextDefinition.instance.setActivedContextCounters(activedContexts)
    activedContextsCounters = Application::ContextDefinition.instance.
    activedContextsCounters
    FeatureSelection.instance.selectDefault(defaultContext)
  end

end
```

Listing 7.8: Context activation, taking care of the contexts' activation state.


```
{
  "dependencies": {
    "Temperature": {
      "xor": []
    },
    "Gps": {
      "xor": []
    },
    "Brightness": {
      "xor": []
    },
    "Europe": {
      "exclusion": [
        "Freezing"
      ]
    },
    "Normal": {
      "requirement": [
        "Europe"
      ]
    }
  }
}
```


Listing 7.9: Example of a context dependency declarations file.

```
class FeatureSelection

  def select(o, managedContexts)
    newActivedContexts, newDeactivedContexts = getChangedContexts(managedContexts)

    featuresToActivate = selectFeatures(getMoreSpecificFeatures(newActivedContexts))
    featuresToDeactivate = selectFeatures(getMoreSpecificFeatures(newDeactivedContexts))

    featuresToActivate = Application::FeatureDefinition.instance.getListOfEntitiesGraph(
    featuresToActivate)
    featuresToDeactivate = Application::FeatureDefinition.instance.getListOfEntitiesGraph
    (featuresToDeactivate)

    featuresToDeactivate += selectParentsOfNewFeatures(featuresToActivate)

    FeatureActivation.instance.activate(o, featuresToActivate, featuresToDeactivate)
  end

  # Bootstrapping method
  def selectDefault(defaultContext)
    defaultFeature = selectFeatures([defaultContext])
    defaultFeature = Application::FeatureDefinition.instance.getEntitiesGraph(
    defaultFeature.first)
    FeatureActivation.instance.activateDefault(defaultFeature)
  end

end
```

Listing 7.10: Feature selection, selecting relevant features based on (de)activated contexts.

```
{
  "Hot": "FHot",
  "Normal": "FNormal",
  "Cold": "FCold",
  "Freezing": "FFreezing",
  "Gps": "FGps",
  "Unknown": "FUnknown",
  "Europe": "FEurope",
  "Belgium": "FBelgium",
  "Default": "FDefault"
}
```

Listing 7.11: Example of a context-feature mapping file.

```
class FeatureActivation

  def activate(o, featuresToActivate, featuresToDeactivate)

    activedFeaturesCounters = Application::FeatureDefinition.instance.
    activedFeaturesCounters

    managedFeatures = {}

    activedFeatures = activedFeaturesCounters.clone
    defaultFeature = Application::FeatureDefinition.instance.getEntitiesGraph
    if activedFeatures.getActivedEntities.include? defaultFeature
      activedFeatures.decreaseCounter(defaultFeature)
      managedFeatures = {:deactivated => [defaultFeature]}
    end

    begin
      featuresToDeactivate.each {
        |feature|
        feature.canDeactivate(activedFeatures, managedFeatures)
      }
      featuresToActivate.each {
        |feature|
        feature.canActivate(activedFeatures, managedFeatures)
      }
      Application::FeatureDefinition.instance.setActivedFeaturesCounters(activedFeatures)
    rescue Exceptions::DependencyException => e
    end

    activedFeaturesCounters = Application::FeatureDefinition.instance.
    activedFeaturesCounters

    TransitionSelection.instance.select(o, managedFeatures)
  end

  def activateDefault(defaultFeature)
    activedFeatures = Application::ActivedEntitiesCounters.new()
    activedFeatures.increaseCounter(defaultFeature)
    Application::FeatureDefinition.instance.setActivedFeaturesCounters(activedFeatures)
    activedFeaturesCounters = Application::FeatureDefinition.instance.
    activedFeaturesCounters
    logger.debug { "New activate features : #{activedFeaturesCounters}" }
    TransitionSelection.instance.select(nil, {:activated => [defaultFeature]})
  end

end
```

Listing 7.12: Feature activation, (de)activating selected features based on dependencies.

Chapter 8

Features integration

Contents

8.1 Introduction **43**
8.2 Principles **43**
Framework-centric 44
Application-centric 44
8.3 Feature installation **45**
8.3.1 Principles 45
8.3.2 Addressing native limitations 47
8.4 Conclusion **48**

8.1 Introduction

This chapter describes the interactions between the SAF and its running application, especially the Ruby specificities that provide the ability to change the application's features at runtime.

8.2 Principles

The SAF from Chapter 7 provides a self-adaptive system that updates the state of features every time there is a change in the state of the environment via context management. Considering that the surroundings are the input of the framework, the main output is the new state of features to be installed in the running application.

The purpose of features integration is to provide a way to update the state of the application by installing this new state of features. Several ways were considered in Ruby, mostly being possible because of the new functionalities that the version 2.0 has provided, especially in the metaprogramming field.

Ruby has well developed meta-level capabilities, as it is possible to open classes (even system ones) seamlessly and to update methods and variables at runtime. This property is the key to the success of features runtime integration, as it allows to update an application's functionalities without the need to stop and restart it, completely transparently.

The main principle is thus to have a running thread that executes the SAF in order to push feature state updates every now and then, when the state of the environment changes, and another thread to run the application concurrently. Hence, the main thread's purpose is to transfer data and control between both threads. From this principle, it is possible to implement it in two ways :

Framework-centric

Implementing the system to be framework-centric means that the application is paused in order to wait for the next iteration of the SAF. When the framework finished its iteration, it resumes the application with potentially new features. In this situation, the application is waiting for the SAF to get the most relevant features depending on the environment. But, as the SAF process is significantly slow, it proves to be impactful to the application.

Setting up a framework-centric system could be useful when reliability and consistency are key at the cost of the application running more slowly, especially when the environment is evolving and changing its state very often. But in most cases, especially at a time level of a microsecond, the environment does not change frequently enough to threaten reliability and a slower application is more harmful to the end user.

Application-centric

If the system is application-centric, the SAF is constantly pushing feature states, but the application execution is key, in the sense that every time it pauses its execution to check for new features, the main thread acts upon the last pushed feature states, even if the SAF is in the middle of an iteration, in order to resume the application as fast as possible. This principle is faster than the former, at the cost of reliability because of limited information on the true current state of the environment, which can lead to inconsistencies that need to be addressed at application-level.

In order to tackle inconsistency, the application has the ability to control its pausing process to seek for new feature states. It is possible due to the use of a fiber to run the application. A `Fiber` is a non-preemptive thread, ideal for dealing with concurrent executions, where the scheduling has to be done manually.

Hence, the application has the ability to pause its execution by calling `Fiber.yield`, in order to give the control back to the thread that initiated the fiber, namely the main thread. It is also possible to pass values back to the main thread in order to provide some information about the current state of the application. For example, if the application will enter in a critical part at the next resume, it would be possible to inform the main thread to wait for a complete iteration of the SAF before retrieving the potentially new features.

Also, when the main thread is resuming the application's execution, it can also provide useful information regarding the state of the SAF, if for example, the critical part that needs to be executed has become irrelevant in the current state of the environment. The application can then act upon this information by checking the return value of `Fiber.yield` and choose to terminate prematurely the current method call.

Even though it is possible for the application to pause its execution manually, it can become quite cumbersome if it has to deal completely with its own scheduling, especially when the application started with default features.

In order to do seamless feature integration, it is really important to be able to run the application with a SAF even if it never pushes new features without having to put framework-specific code in the base application. As applications retrieve the control back with `Fiber.yield`, Ruby provides a way to call it at certain events, like at every method call or return, or at every class declaration, via the `TracePoint` class.

The `TracePoint` class is a new feature of Ruby 2.0 which provides the ability to execute a procedure at defined events. It also retrieves specific information about the state of the event, such as the class in which it triggered or the method name that got called. However, it is quite

sensitive, as traces defined that way are valid every time the event fires if it is active. In the framework, it is decided to pause the application by default at every method return, in order to check for new features between two method calls. A code snippet can be seen in Listing 8.1, where a default `Fiber.yield` is executed, and the current trace is disabled in order for it to be active only inside the application, as it is irrelevant for the SAF and the main thread.

As a trace is putting its procedure call in the low-level Virtual Machine (VM) code, if a yield happens in the middle of a method call that gets changed or even removed by the SAF, the old method still runs for that iteration, as only further calls are affected by this change. It addresses inconsistency issues seamlessly, as the application does not need to be restarted, even in the middle of an irrelevant method, at the cost of treating the inconsistency manually in the application code if needed. A code snippet of a manual yield can be seen in Listing 8.2.

This principle has been inspired from [Car+11], in which there is a distinction between lazy and eager context (de)activation. It provides a way for the application to address consistency issues by manually querying the SAF to get the most relevant features. In this case, an execution is considered lazy if at a feature state change, the current method finishes its execution before depending on its new definition, and it is eager when the method is forced to return in order to depend immediately on its new definition.

```
trace = TracePoint.new(:return) do |tp|
  trace.disable
  Fiber.yield false
end
```

Listing 8.1: `TracePoint` use to pause the execution at each method return.

```
# Main thread: application resume
if fiber.alive?
  trace.enable
  # Inform the app if the state changed and if the current exec is critical
  returnval = fiber.resume (changed and returnval)
else
  break
end

# Application thread: manual yield inside a method definition
def critical
    returnval = Fiber.yield true
    # Force return the critical function if the state changed
    return if returnval
    critical_part
end
```

Listing 8.2: Manual `Fiber.yield` before executing a critical part.

8.3 Feature installation

The previous section detailed the control flow between a SAF and its running application, and this section is focusing more on ways to actually install the features as behavioural variations.

8.3.1 Principles

At system startup, the SAF bootstraps by creating the context and feature graphs, and launches the default feature by starting the base application. Any object-oriented application that has no variation whatsoever can execute with the SAF without framework-specific code in it, provided that it specified a `main` class to start from.

From there, the application should provide all the relevant properties needed by the SAF in the Application layer, in order to fully use its adaptability power. As to features, they need to be defined at a specific location with a specific name, in order to ensure easier processing in Ruby :

- Each feature file should be named with the same name as in its declaration (feature `FHot` has `FHot.rb` as file name).
- Each feature file contains all the modules that are offering variations to classes
- Each feature module is named after the class it adapts, following by its depth (if several features change the same class) and ending with an `S` if it is a singleton class.
- From the previous point, classes holding instance properties and singleton classes holding class properties must be defined separately for better maintenance purposes at the cost of less readability, and they are distinguished by the ending `S`.

An example of a feature file can be seen in Listing 8.4 and an example of a base class that has variations in Listing 8.3

```ruby
# Application base class
class Klass
  def feature
    print 'Feature'
  end

  def self.class_feature
    print 'Class feature'
  end
end
```

Listing 8.3: Example of a base class.

```ruby
# New feature adapting Klass instances
module Klass1
  def feature
    print 'New feature'
  end
end

# New feature adapting Klass itself
module Klass1S
  def class_feature
    print 'New class feature'
  end
end
```

Listing 8.4: Example of a feature file that contains behavioural variations to a given class.

The main principle of the process is to use the module prepending ability provided by Ruby 2.0. When a module is prepended to a class, it changes the lookup chain to be before the class itself, so that any definition is looked up first in the prepended module, then in the class, then it looks potentially in the superclasses. It provides a native way in Ruby to structure class features as layers, preserving the definition lookup chain. It also enables the ability to refer to other features prepending the same class, as a `super` call provides a way to run through the lookup chain of a given method.

A simple example of module prepending can be seen in Listing 8.5.

```ruby
# Base class
class Klass
  def feature
    print 'Feature'
  end
end

# Feature level 1
module Klass1
  def feature
    print 'New '
    super
  end
end

# Feature level 2
module Klass2
  def feature
    print 'Very '
    super
  end
end

k = Klass.new
k.feature                # => 'Feature'
print Klass.ancestors    # => [Klass, Object, Kernel, BasicObject]

Klass.class_eval "prepend Klass1"
k.feature                # => 'New Feature'
print Klass.ancestors    # => [Klass1, Klass, Object, Kernel, BasicObject]

Klass.class_eval "prepend Klass2"
k.feature                # => 'Very New Feature'
print Klass.ancestors    # => [Klass2, Klass1, Klass, Object, Kernel, BasicObject]
```

Listing 8.5: Example of module prepending.

8.3.2 Addressing native limitations

One of the main drawbacks in this native solution is that Ruby does not provide any way to remove prepended modules from the lookup chain, which removes a lot of potential from this implementation for dynamic runtime behavioural variations. In order to address this issue, a limiting way has been implemented as follows :

- Modules representing variations are only prepended once in the lookup chain.
- Each prepended module represents a depth level of a given class, so that every feature module has to be assigned to a specific depth level. The absolute flexibility of interchangeable features for the same class is lost, as it has to be determined statically by the application, which can lead to code duplication.
- At each feature deactivation, the concerned modules are emptied in order to be reused by other features. It explains why the features have to specify which class they adapt in the module name, in order to be easily retrieved, and to avoid unnecessary class lookup overhead.

In Listing 8.6, a regex matcher is used to retrieve the concerned class, the depth level and to know if it is a singleton class or not. The class definition is then retrieved based on its name, and the module gets prepended if it is not already. This ensures a lazy prepending, in order for a base application to not have a pre-prepending process for every class at startup. Only the once adapted classes get prepended this way.

Also, for the sake of little interchangeability, if a higher depth level is encountered first, all the subsequent levels are prepended in order to maintain and control the lookup chain order.

In Listing 8.7, the given prepended module is emptied by removing all its instance methods, variables and constants. By emptying a module, it is easier to repopulate it with new functionalities, as a simple load of the new module definition is enough, thanks to the metaprogramming capabilities of Ruby that allow to reopen and redefine classes on-the-fly.

In Listing 8.8, another `TracePoint` is used in order to retrieve all the defined modules of a feature file at loading. It is a neat way to get all the variations a given feature provides at once, as at every beginning of a module definition, the corresponding module is added to a list, and it is decided to prepend or to remove them, depending on the state of the feature (activating or deactivating).

```
1  def prepend_module(mod)
2    mod_name = mod.to_s
3    # (Classname)(Depth)(Singleton mark)
4    mod_name.match(/(^[a-z]+)([0-9]+)(S?)/i) do |m|
5      klass_name = m[1]
6      depth = m[2].to_i
7      is_singleton = m[3]
8
9      klass_def = Object.const_get(klass_name)
10     klass = is_singleton.empty? ? klass_def : klass_def.singleton_class
11     (1..depth).each do |level|
12       unless klass.included_modules.include?(mod)
13         klass.class_eval "prepend " + klass_name + level.to_s + is_singleton
14       end
15     end
16   end
17 end
```

Listing 8.6: Prepending a feature module.

```
1  def remove_module(mod)
2    mod.instance_methods(false).each do |meth|
3      mod.send(:remove_method, meth)
4    end
5    mod.instance_variables.each do |var|
6      mod.send(:remove_instance_variable, var)
7    end
8    mod.constants(false).each do |const|
9      mod.send(:remove_const, const)
10   end
11 end
```

Listing 8.7: Emptying a feature module.

8.4 Conclusion

This chapter was about the way feature definitions are seamlessly installed and integrated into a given running application with native Ruby tools. Compared to [Thi12] where they used a hack to proceed and maintain the definitions lookup chain, Ruby 2.0 provides natively ways to manage the chain by using prepending modules and traces in order to have a better control on a running application, without the cost of having to act directly on its execution, by restarting it or by making it visible to the user.

However, it suffers from deep limitations, as prepended modules are impossible to remove without messing with the Ruby VM and as such, absolute flexibility could not be achieved because for the sake of keeping maximum control over features integration for consistency, as per-class module precedence has to be determined statically, removing adaptability over corner

```ruby
def update_feature(feature, is_active)
  mods = []
  trace = TracePoint.new(:class) do |tp|
    # Add all the concerned modules
    mods << tp.self
  end

  trace.enable
  load feature.path
  trace.disable

  mods.each do |mod|
    if is_active
      # If the feature is activated
      prepend_module(mod)
    else
      # If the feature is deactivated
      remove_module(mod)
    end
  end
end
```

Listing 8.8: Activating or deactivating a feature.

cases where features could change their position in the depth level structure depending on the context.

Also, in order to maintain a standard way of dealing with feature definitions, module names and places are enforced with a naming convention to ease the prepending and removing process. Although it makes automatic feature creation easy as a future work, it puts unwanted constraints on the application developer.

Despite the limitations, with the advent of Ruby 2.0 and its new tools, Ruby became a very promising candidate for COP and FOP as module prepending allows through metaprogramming a lot more dynamicity in features integration than before.

Also, the alternative of using refinements instead of prepended modules was also quite promising, as the ability to control feature (de)activation by scoping is a step forward in feature fine-grained management, but as the scoping is lexical, it is currently impossible to control an application from outside with this technique.

Chapter 9

Benchmarks

Contents

9.1 Introduction **50**
9.2 Base **50**
9.3 Results **50**
9.4 Conclusion **51**

9.1 Introduction

In this chapter, some benchmarks are shown to compare the execution of features integration versus the execution of a naive implementation with conditional statements. As it is very difficult to benchmark the whole framework due to its self-adaptive nature, only the feature integration part from Chapter 8 is benchmarked compared to a naive implementation. This is especially to show how the algorithm behaves when the number of method calls increases and when the number of features increases.

9.2 Base

The initialization of the naive implementation and of the parameters are in Listing 9.1, and the class that was used is in Listing 9.4. The benchmarking process for the `if` statements are in Listing 9.2, and for the features integration in Listing 9.3.

The `SWITCH` constant is used as a parameter of the number of features that are all occuring during the benchmark, and the `CALL` constant is a parameter for the number of times the adapted method is called per feature. The class contains only one method that will be subsequently changed with features and a list of `if` statements is `eval`'d in the class to simulate a naive implementation.

The naive benchmark is simply calling the method containing the naive implementation, and the features benchmark simulates a running application and a SAF with two fibers in order to compare apples with apples. Originally, the SAF part is in a thread, but as the number of switches and the moment at which they occurs are controlled, it is replaced by a fiber in order to call its body only when the method was called `CALL` times. Also, instead of loading a path, the features state changer is simply `eval`'ing the given feature.

9.3 Results

The results can be seen in Figure 9.1 and in Figure 9.2.

```
SWITCH = 50    # Number of features
CALL = 1000    # Number of method calls

ifs = ""
SWITCH.times do |i|
  ifs += "if choice==#{i}
            return choice
          end
          "
end

Klass.class_eval "
  def if_method(choice)
    #{ifs}
    return 0
  end
  "

k = Klass.new
```

Listing 9.1: Initialization.

```
x.report("IF's:") do
  SWITCH.times do |i|
    choice = i % SWITCH
    CALL.times { k.if_method(choice) }
  end
end
```

Listing 9.2: `If` statements benchmark.

In Figure 9.1, the algorithm of features integration is slower over time than the naive implementation. This is mostly due to the fact that the system is not optimized to be efficient, because it is mostly focused on keeping behaviours consistent and executions predictable, which lead to a significant overhead.

Also, the use of traces and the fact that in the benchmark, the features integration process does not take the advantages of concurrency and multithreading at all, as everything is executed sequentially to match the behaviour of the naive implementation, justify the results.

Furthermore, while the features do not change, there is also a significant overhead of verifying if the current state of features has changed, all of which contribute to the general efficiency loss.

In Figure 9.2 however, the features integration mechanism becomes more efficient than the naive implementation when a high number of features is present. For 1000 calls per feature, both implementations meet at 500 features. Although it is promising for systems with a high number of features, it would not harm to optimize the process.

9.4 Conclusion

This chapter was about comparing the efficiency of the features integration of Chapter 8 with a naive implementation of `if` statements. It appeared that there is a significant overhead when the number of method calls between each feature switch is increasing, mostly because consistency is preserved at the cost of efficiency.

However, when there is a high number of features in the system, the integration is more efficient after a certain number, as expected.

```
x.report("Features") do
  trace = TracePoint.new(:return) do |tp|
    trace.disable
    Fiber.yield false
  end
  fiber = Fiber.new do
    SWITCH.times do
      CALL.times do
        k.feature_method
      end
      Fiber.yield 2
    end
  end
  th = Fiber.new do
    SWITCH.times do |i|
      current = ["#{i}"]
      Fiber.yield
    end
  end
end

# Feature state changer
def update_feature(feature, is_active)
  mods = []
  trace = TracePoint.new(:class) do |tp|
    mods << tp.self
  end

  trace.enable

  eval("
  module Klass1
    def feature_method
      #{feature}
    end
  end
  ")

  trace.disable
  mods.each do |mod|
    if is_active
      prepend_module(mod)
    else
      remove_module(mod)
    end
  end
end
```

Listing 9.3: Features integration benchmark.

```
class Klass
  def feature_method
    0
  end
end
```

Listing 9.4: Base class used for the benchmarks.

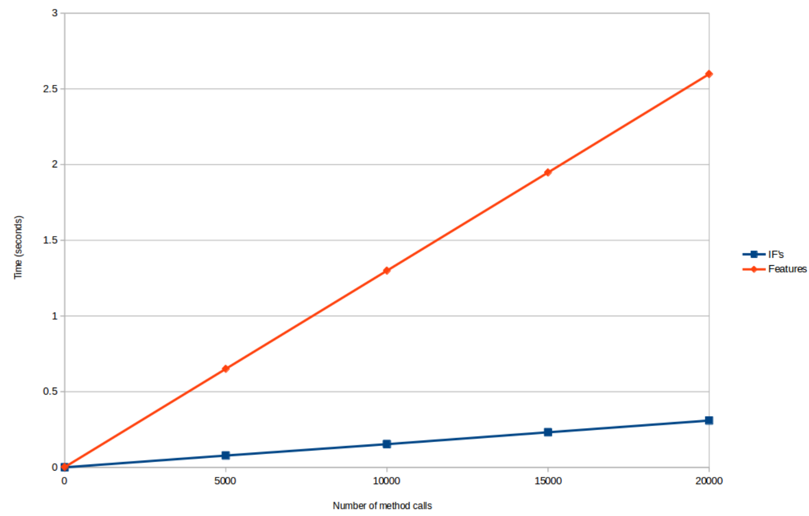

Figure 9.1: Benchmark with increasing number of method calls (`CALL` = [0:20000]) and a fixed number of features (`SWITCH` = 50)

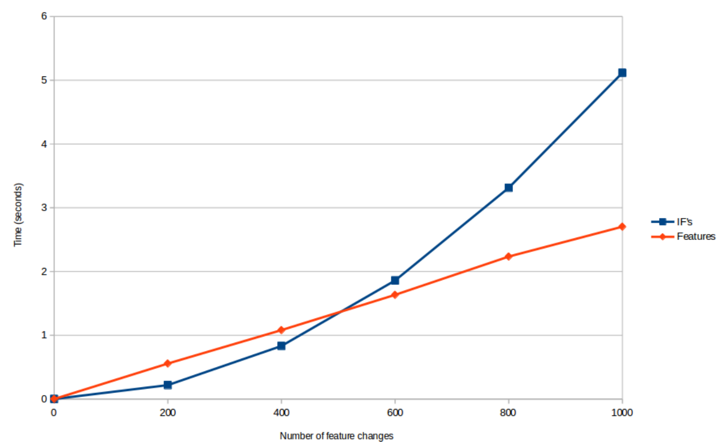

Figure 9.2: Benchmark with increasing number of features (`SWITCH` = [0:1000]) and a fixed number of method calls (`CALL` = 1000)

Part III

Conclusion

Chapter 10

Future work

Contents

10.1 Introduction **55**
10.1.1 Self-Adaptive Framework 55
Context data 55
Sensors 55
Graphs 56
Conflict resolution 56
Multithreading 56
Prompt notification 56
10.1.2 Features integration 56
Unprepend modules 56
TracePoint 56

10.1 Introduction

This chapter covers all the possible improvements of the SAF and of feature runtime integration.

10.1.1 Self-Adaptive Framework

Context data

Although it is possible to propagate a sensor object throughout the framework, there is at the moment no mean to explicitly deal with context data. As contexts are stateful and can contain meaningful information, explicit context data management would provide ways to act at context change, and not only before and after activation. For example, contexts could retain a previous state that would be recovered upon its reactivation.

Sensors

Currently, the sensors used in the SAF are mockups in order to simulate the behaviour of real ones. Implementing a way to provide sensors output to the SAF without having to make an abstraction of them would be useful at times when the SAF is integrated into a mobile device that provides all the sensor metrics needed natively for example.

Graphs

Although context and feature graphs are managed separately, it would be a step forward to be able to merge them into one complete graph such as in [HT08] that could be generated automatically for debugging purposes. It would also allow to deal better with potential dependencies between contexts and features.

Conflict resolution

Although the dependency conflicts are detected in the SAF, there is no complete conflict resolution manager at the moment. It would be quite useful, not only for dealing with contexts and features dependencies, but also it could detect integration conflicts, such as methods being redefined several times at once by different features. Policies could be implemented in order to automatically resolve conflicts by precedence, activation age or other customized policies.

Multithreading

Although the SAF uses multithreading to be able to iterate over the framework and also execute the application at the same time, but as there is no true multithreading in Ruby*, the whole system is slowed down naturally compared to a more straightforward approach.

Prompt notification

Currently, when a sensor produces some data, the SAF has to complete its full iteration before deploying the relevant features, but if a more critical environmental state change occurs (a sensor gets disabled abruptly for example), there should be a way to bypass the whole iteration to send a signal to the application that a major change in the system is about to happen. Also, because of the use of a fiber to control the flow of the application, it is impossible to stop its execution from outside. An application whose control flow would completely depend on another thread would be the key for absolute control.

10.1.2 Features integration

Unprepend modules

As said in Chapter 8, there is no way in Ruby at the moment to remove a prepended module from the lookup chain of a class. It is possible using non-native C code to do it manually, but it is not a really clean solution. The ability to prepend and unprepend modules on-the-fly would be a killer feature and a big step forward COP and FOP collaboration, as flexibility over interchangeability would be guaranteed.

TracePoint

Although putting traces reacting to certain events are useful, it produces a significant overhead to the application. Providing another more efficient way to deal with automatic control yielding would be better if performance is an issue.

*Threads are interleaved instead of using processor multithreading capabilities.

Chapter 11

Conclusion

Contents

11.1 Contributions **57**
11.2 Objectives **57**
11.3 Final words **58**

11.1 Contributions

Obviously, there is still a lot to do in the field of COP, but the implementation of a solution such as the SAF is very promising for the future. More and more hardware and software are connected, and software systems are even more influenced by their surrounding environment. Also, the success of being able to separate contexts and features while being able to make them work together to produce variability is key, especially when only minor changes in the attached application are needed.

However, the main drawback and limitation is the overhead it generates. Parsing sensor output into contexts and features, and gathering them with multiple decision-making processes is tough. This is why there should be a language that would be defined for COP, in order to be able to optimize the processes and to give abilities that current languages lack, such as a high level of metaprogramming capabilities, but also a high level of code and definitions interchangeability.

Although Ruby is very promising in that sense, its module prepending capability lacks the ability to install and remove on the fly, which is very impactful for flexibility. But even without it, it was possible to achieve a completely self-adaptive system, which only needs a context-unaware object-oriented application, and real sensors to work with.

11.2 Objectives

Implementing a self-adaptive framework with a running application unaware of it is almost a complete success. The application is not completely unaware of it, as the presence and absence of features can lead to inconsistencies that need to be addressed locally, but it is promising.

The implementation of features integration is also a success, but is mostly impacted by the limitations of the Ruby language, especially in terms of module prepending and management. But using non-native Ruby code would be an option to consider in order to go further.

11.3 Final words

The thesis started with some background work in Part I, where mostly the key concepts and useful works over the years were depicted. Then, the SAF was presented, with all the useful components it is equipped. A particular close-up was made on the end of the framework, namely the features integration and how to install new features into a running application inspired from FOP. Eventually, conclusions were drawn from the benchmarks and possible future works.

I am proud that I was able to have a role in discovering new ideas for COP, and I hope that one day, COP will emerge from the research works and impose itself on everyone, like OOP did in the past. And when that day will come, I will be proud to say that I was part of it.

Bibliography

[Ach+09] Mathieu Acher et al. “Modeling Context and Dynamic Adaptations with Feature Models”. In: *4th International Workshop Models@run.time at Models 2009 (MRT’09)*. United States, Oct. 2009, p. 10. URL: https://hal.archives-ouvertes.fr/hal-00419990.

[AK09] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8.5 (July 2009). (column), pp. 49–84. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.5.c5. URL: http://www.jot.fm/contents/issue_2009_07/column5.html.

[And+09] Jesper Andersson et al. “Modeling Dimensions of Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems* (2009), pp. 27–47.

[App+09] Malte Appeltauer et al. “A comparison of context-oriented programming languages”. In: *International Workshop on Context-Oriented Programming*. Ed. by New York ACM Press. 2009, pp. 1–6.

[Car+11] Nicolás Cardozo et al. “Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations”. In: *International Conference On Software Engineering Advances (ICSEA’11)*. IARIA, 2011, pp. 130–135.

[Car13] Nicolás Cardozo. “Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems”. PhD thesis. Université catholique de Louvain - Vrije Universiteit Brussel, Sept. 2013.

[CBK13] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer, 2013.

[Col12] Marius Colacioiu. “Context-Aware trait composition for mobile platforms”. PhD thesis. Università degli studi di Milano, 2012.

[Duh16] Benoît Duhoux. “L’intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte”. MA thesis. Université catholique de Louvain, June 2016.

[GB03] Jilles Van Gurp and Jan Bosch, eds. *Software Variability Management*. Feb. 2003.

[Gon+10] Sebastián González et al. “Subjective-C: Bringing Context to Mobile Platform Programming”. In: *SLE’10: Proceedings of the Third international conference on Software Language Engineering (SLE 2010)*. Lecture Notes in Computer Science 6563. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 246–265.

[Gon+13] Sebastian Gonzalez et al. “Context Traits: Dynamic Behaviour Adaptation Through Run-Time Trait Recomposition”. In: Accepted for publication in Proceedings of AOSD2013. 2013.

[GPS10] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. “Context-oriented programming in highly concurrent systems”. In: *2nd International Workshop on Context-Oriented Programming*. 2010.

[GS09] Sebastian Günther and Sagar Sunkle. "Feature-oriented Programming with Ruby". In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. FOSD '09. Denver, Colorado, USA: ACM, 2009, pp. 11–18. ISBN: 978-1-60558-567-3. DOI: 10.1145/1629716.1629721. URL: http://doi.acm.org/10.1145/1629716.1629721.

[GS10] Sebastian Günther and Sagar Sunkle. "Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming". In: *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*. FOSD '10. Eindhoven, The Netherlands: ACM, 2010, pp. 80–87. ISBN: 978-1-4503-0208-1. DOI: 10.1145/1868688.1868700. URL: http://doi.acm.org/10.1145/1868688.1868700.

[HT08] Herman Hartmann and Tim Trew. "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains". In: *Proceedings of the 2008 12th International Software Product Line Conference*. SPLC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 12–21. ISBN: 978-0-7695-3303-2. DOI: 10.1109/SPLC.2008.15. URL: http://dx.doi.org/10.1109/SPLC.2008.15.

[Kam+15] Tetsuo Kamina et al. "Method Safety Mechanism for Asynchronous Layer Deactivation". In: *7th International Workshop on Context-Oriented Programming (COP'15)* 6 (2015).

[Kic+97] Gregor Kiczales et al. "Aspect-Oriented Programming". In: *European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241*. June 1997.

[Kon14] Kashyap Kondamudi. *Ruby, Trace, Leave, Oh my!* Apr. 2014. URL: http://kgrz.io/2014/04/19/ruby-trace-leave-oh-my.html.

[Lin+11] Jens Lincke et al. "An open implementation for context-oriented layer composition in ContextJS". In: *Science of Computer Programming* (2011).

[LS00] Henry Lieberman and Ted Selker. "Out of Context: Computer Systems That Adapt To, and Learn From, Context". In: *IBM Systems Journal* 39.3-4 (2000), pp. 617–631.

[Men15] Kim Mens. *Context-oriented programming: Overview and Applications*. Sept. 2015. URL: http://www.slideshare.net/kim.mens/contextoriented-programming.

[Mur+14] Aitor Murguzur et al. "Context variability modeling for runtime configuration of service-based dynamic software product lines". In: *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*. Ed. by Stefania Gnesi et al. ACM, 2014, pp. 2–9. ISBN: 978-1-4503-2739-8. DOI: 10.1145/2647908.2655957. URL: http://doi.acm.org/10.1145/2647908.2655957.

[Nic15] Siobhan Clarke Nicolas Cardozo. "Context Slices: Lightweight discovery of behavioral adaptations". In: *COP'15 Workshop* (2015).

[Per14] Paolo Perrotta. *Metaprogramming Ruby: Program Like the Ruby Pros*. second. The Pragmatic Programmers, Aug. 2014.

[Pre97] Christian Prehofer. "Feature-Oriented Programming: A Fresh Look At Objects". In: Springer, 1997, pp. 419–443.

[Raf14] Mike Hinchey Rafael Capilla Óscar Ortiz. "Context Variability for Context-Aware Systems". In: *IEEE Computer* 47.2 (Feb. 2014), pp. 85–87.

[RN08] Pascal Costanza Robert Hirschfeld and Oscar Nierstrasz. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (Mar. 2008), pp. 125–151.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. “ContextErlang: Introducing Context-oriented Programming in the Actor Model”. In: *11th Annual International Conference on Aspect-Oriented Software Development*. Ed. by New York ACM Press. 2012, pp. 191–202.

[SGP15] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. “ContextErlang: A language for distributed context-aware self-adaptive applications”. In: *Science of Computer Programing 102* (May 2015).

[Tan+06] Éric Tanter et al. “Context-Aware Aspects”. In: *Software Composition*. Ed. by Welf Löwe and Mario Südholt. Lecture Notes in Computer Science 4089. Springer Berlin Heidelberg, 2006, pp. 227–242.

[Thi12] Loïc Vigneron Thibault Poncelet. “The Phenomenal Gem: Putting Features as a Service on Rails”. MA thesis. Louvain School of Engineering, June 2012. URL: `http://www.phenomenal-gem.com`.

[TIO16] TIOBE. *TIOBE Index for May 2016*. May 2016. URL: `http://www.tiobe.com/tiobe_index`.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve www.uclouvain.be/epl