Université catholique de Louvain
Louvain School of Engineering
Computing Science Engineering Department

The Phenomenal Gem

Putting Features as a Service on Rails

Thesis submitted in partial fulfilment of the requirements for the degree M.Sc. in Computer Science option Software Engineering

Authors: **Thibault Poncelet** & **Loïc Vigneron**

Promoter: Kim Mens
Co-advisor: Sebastián González
Reader: Nicolas Jacobeus

Louvain-la-Neuve

June 2012

Abstract

This thesis introduces the Phenomenal Gem, a Context-Oriented Programming framework for the dynamic programming language Ruby. With this framework, programmers can handle contexts as first-class entities allowing them to adapt the behaviour of their applications dynamically in a clean and structured manner. In addition to this COP framework, the thesis also introduces the notion of Context as a Feature that tries to merge the best of COP and Feature-Oriented Programming into a single new paradigm. From the point of view of usability in today's web application, this thesis builds the notion of Feature as a Service on top of CaaF, and integrates it in the Ruby on Rails web framework. The implementation and semantics of these concepts are presented in detail and validated by a real-world case study of a Software as a Service Enterprise Resource Planning application, developed by an industrial collaborator.

Acknowledgments

We would like to thank our promoter Professor Kim Mens and our co-advisor Doctor Sebastián González for their outstanding support throughout the development of this thesis. For the extensive time they took to review our work but also for all the meetings we had and which were most fruitful and motivating. We could not have had greater support and we are very grateful to both of them.

We would also like to acknowledge the support of Nicolas Jacobeus who agreed to be our reader and collaborated on our case study by giving us access to the source code of Benubo. In addition, his counsel on our findings enabled us to add an industrial aspect to this thesis.

Finally, we would like to thank our families and friends for supporting us throughout our studies and especially during the writing of this thesis.

Thibault Poncelet & **Loïc Vigneron**

Contents

Acronyms

API Application Programming Interface . . . 24
AST Abstract Syntax Tree . . . 28
CaaF Context as a Feature . . . 117
CoC Convention over Configuration . . . 71
COP Context-Oriented Programming . . . 117
CSV Comma-Separated Values . . . 95
DOM Document Object Model . . . 115
DSL Domain Specific Language . . . 118
DRY Don't Repeat Yourself . . . 71
ERB Embedded Ruby . . . 115
ERP Enterprise Resource Planning . . . 119
FaaS Feature as a Service . . . 118
FOP Feature-Oriented Programming . . . 117
FODA Feature-Oriented Domain Analysis . . . 15
HAML HTML Abstraction Markup Language . . . 115
HTML HyperText Markup Language . . . 74
HTTP HyperText Transfer Protocol . . . 124
JIT Just In Time . . . 18
JS JavaScript . . . 23
LAN Local Area Network . . . 108
LGPL GNU Lesser General Public License . . . 3
MRI Matz's Ruby Interpreter . . . 114
MVC Model-View-Controller . . . 89
OO Object-Oriented . . . 104
OOP Object-Oriented Programming . . . 66
OS Operating System . . . 72
RoR Ruby on Rails . . . 117
REST Representational State Transfer . . . 22
SaaS Software as a Service . . . 118
SME Small and Medium Enterprise . . . 88

Chapter 1

Introduction

Contents

To the man who only has a hammer in the toolkit, every problem looks like a nail.

Abraham Maslow

1.1 Motivation

In current applications, customisation has become a major issue. While our programming languages have been developed at a time when software was designed as a finished product and expected to behave exactly the same everywhere, today's applications are more and more connected to the external world. This connection is a huge potential source of adaptation to the context in which these applications are running.

We believe that tools matter and that the current tools do not put the programmers in the right frame of mind to build dynamically adaptable applications from scratch. For end users, the software unit is a feature (or functionality) that an application provides. They do not reason about object and classes. To overcome this distortion of approach,

several solutions already exist but most of them remain principally static and are not able to provide real run time software variability on the basis of the situation. However, new ways of selling applications like Software as a Service (SaaS) would really enhance their user experience by being able to achieve mass customisation in a straightforward way.

The Context-Oriented Programming (COP) paradigm emerged starting from these observations with the aim to solve the software adaptability headache. Several implementations already exist. A comparison of a number of them is presented in [AHH+09] but a lot of work is still needed before they are accepted by a wide community of developers. This low rate of adoption is due to the fact that most frameworks have a research approach with a heavy syntax, pre-processing steps, etc, which prevents them being used in industry.

1.2 Contribution

Starting from these observations, this thesis goes through and revises the concepts of COP while applying them to the Ruby programming language. This review of concepts is conducted keeping in mind the concepts of Feature-Oriented Programming (FOP). The goal is to bring together the best of both paradigms to provide a robust and natural way of developing dynamically customizable software. And this will lead to Feature as a Service (FaaS) software.

On the basis of these revised notions, which are clearly defined with the help of formal semantics for the core mechanisms, an implementation is proposed in the Ruby programming language. Contexts and features are reified as first-class entities and available for the developer as adaptable application building blocks.

Web applications are especially well suited for behavioural adaptation. A single code base can potentially serve millions of users in different places, different specific needs and through an uncountable number of means. In addition to the plain Ruby implementation, this thesis provides the glue to bind the developed concepts within the Ruby on Rails (RoR) web framework. Because one of our main goals is to provide something useful in real life, our test case is the refactoring of a SaaS implemented in RoR by Belighted, a web company based in Louvain-la-Neuve, Belgium.

Since real applications quickly involve many different contexts, features and relationships between them, a visualizer tool is also implemented and provided alongside the framework.

1.3 Approach

The COP notions are mainly derived from the one defined in Subjective-C [GCM+11] while the FOP ones from rbFeature [GF11]. In order to merge these two paradigms

and create the Context as a Feature (CaaF) concept, we started with a COP implementation and then searched for the interesting concepts in FOP to enhance this base implementation..

Being developer-friendly was a very important point for us, thus we went to great lengths to provide as clear a syntax as possible and to integrate it smoothly in Ruby. This was made possible thanks to the reflective Application Programming Interface (API) of Ruby that allowed us to extend the language without having to modify the interpreter or provide a code generator which has multiple drawbacks, such as a difficult maintainability and debugging.

Finally, we decided to integrate our ideas into the RoR web framework as smoothly as possible because, once again we wanted to build something usable by a large number of developers, and we think that Phenomenal Gem will be a great advantage for web applications.

1.4 Running Example

To illustrate the notions developed we will use a TO-DO list application as a running example that will be extended throughout the thesis. We will start with an off-line pure Ruby TO-DO list of several tasks. These tasks will then be extended with new capabilities such as adaptation to the Operating System (OS). The application will finally be put on the Web and become a RoR application.

1.5 Resources

The entire source code (released with a GNU Lesser General Public License (LGPL)) and documentation is available on `www.phenomenal-gem.com`. In addition to the resources linked to this thesis, this website has been developed with the framework itself. It serves as a live demo with several intercession and introspection capabilities, allowing the user to play with contexts and features.

1.6 Roadmap

Part I of this thesis presents the background on which it is based, namely the COP and FOP paradigms, as well as the Ruby and RoR framework used by the implementation. Next, an analysis of the related works describes first Subjective-C, an Objective-C based COP language for smartphones; then contextR, another COP implementation in Ruby from the Potsdam University; and finally, rbFeature, a FOP implementation in Ruby.

Part II presents the actual contributions made and developments achieved. It starts with the concepts and their formal semantics, followed by the core mechanisms presentation as well as the debugging tool presentation. It ends with the approaches used, and best

practices adopted during the development of the framework.

Part III validates the second part with a real world case study that consists in refactoring a SaaS Enterprise Resource Planning (ERP) provided by Beligthed, our industrial partner. Then several benchmarks assess the achieved performance and compare several applications implementations with and without Phenomenal Gem.

Finally, Part IV proposes possible future work and concludes by reviewing the contents covered.

Part I

Background

Chapter 2

Context-Oriented Programming

Contents

For me context is the key - from that comes the understanding of everything.

Kenneth Noland

2.1 Introduction

In this chapter, we will present the concept of Context-Oriented Programming (COP). We will use this paradigm later in our implementation to create a COP extension of the Ruby language. Once combined with the concept of Feature-Oriented Programming (FOP) presented in the next chapter, they become the building blocks of the Context as a Feature (CaaF) and Feature as a Service (FaaS) concepts.

A context can be seen as a set of relevant circumstances that can influence a behaviour of the application. It can be anything that may exert this influence, for example, the country, the battery level, the load of a server, etc. The global idea is that today software is more and more connected to the external world, and that current programming languages do not provide a good support for this problem. COP is a paradigm built on top of Object-Oriented Programming (OOP) by adding the concepts presented in this chapter.

We will start by presenting the problem on our running example from Section 1.4, and then present the different concepts of COP.

2.2 Motivation

As presented in Section 1.4, we want to develop a TODO-list application. Basically, we have a set of tasks and due dates for them. By default, the application will show an alert on the screen when the due date arrives. This behaviour is sufficient in most cases, but in order to provide a better user experience we would like to adapt it in the following cases:

- Presenter mode : Delay notification until full-screen is deactivated.
- Away mode : Send mail notification instead of the screen notification.
- Trial mode: Until the user has a valid licence, some functionalities are limited. For example, the number of tasks is limited.

This kind of application can of course be done with current programming languages, but it will probably lead to "messy" codes with many conditional statements. Furthermore, these behaviour adaptations can be fairly well implemented, for example, with the Strategy Pattern [Gam95], but they have to be foreseen.

2.3 Concepts

2.3.1 General Architecture

A COP framework architecture is designed on the basis of the one illustrated in Figure 2.1 Firstly, the *Context Discovery* module collects data from the external world in which the system is running and make sense of the logical environment. In addition to the *Context Discovery*, context change can also be triggered by the internal application. The *Context Management* on its side analyses the current situation and context change request and might choose to prioritize some changes, solve conflicts that might appear, drop some contexts. Once these choices have been made, a coherent set of changes is commited

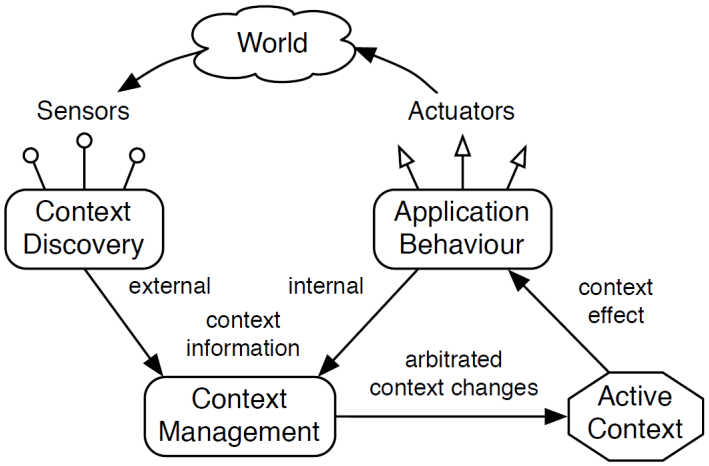


Figure 2.1 – Context aware system architecture.

on the *Active Context* representation which directly affects the *Application Behaviour* [GCM+11].

2.3.2 Adaptation

An adaptation is a first-class entity in a context-oriented system. It reifies a behavioural adaptation of a method in a class for a particular context or situation. Thus, the granularity of an adaptation is at the method level in OOP languages.

This adaptation of behaviour can be dynamically activated and deactivated at run time. During the execution, changes in the software environment will trigger changes of contexts which will lead to (de)activation of its associated adaptations specific to those contexts.

A method can be adapted several times in different contexts. These different adaptations can then be composed to provide a new behaviour. Active adaptations of a same method can be composed together to form a new behaviour and can vary at any time. Adaptations of a method in different contexts are added to the base behaviour and, can change dynamically of order following an ordering policy. This means that an adaptation do not shadow other ones.

2.3.3 Context

We firstly define a *situation* as a combination of relevant environmental facts for the application. We then define a *context* as the reification of a situation that can occur while an application executes. Applications might present behavioural adaptations to those particular situations that are reified. Behavioural adaptations associated to a context are

deployed in the system whenever that context becomes *active*. The context-dependent behaviour is no longer visible in the system when its associated context becomes *inactive*. [GCM+11]

As defined, a context is concerned with run-time behaviour modification. Other approaches such as Software Product Line (SPL) in FOP provide static adaptation of an application by pruning and adding content in the source code. This source code can then be compiled and shipped in its different versions to different types of stakeholders. However, in this case it will adapt the software functionalities dynamically on the basis of the current situation.

A set of adaptations can be associated to a context (providing the response to the new situation), and these adaptations are automatically (de)activated when the context itself is (de)activated.

2.3.4 Combined Context

A *Combined context* is a context built by composition of several other contexts. This means that it is automatically active only when all these contexts are themselves active, giving the capability to define behaviour specific to this very combination of contexts. The combined context behaviour always takes precedence over composing contexts (named sub-contexts) behaviour but can still be combined with it.

2.3.5 Context Relationships

Relationships enforce constraints between contexts. These constraints are specified by the application developer and allows him to be sure that the application behaviour will be consistent. They will ensure that two incompatible contexts are never active at the same time or that a context that relies on another for its implementation is never activated alone. Furthermore, it means that the (de)activation of a context involved in a relationship can trigger other (de)activations and its (de)activation may fail if the constraints cannot be met.

Implication

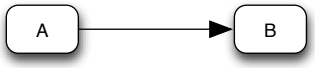


Figure 2.2 – Implication.

An implication (Figure 2.2) between contexts A and B means that the activation of A triggers the activation of B. If the activation of B fails, A cannot be activated either. Conversely, the deactivation of A or B triggers the deactivation of B or A, respectively.

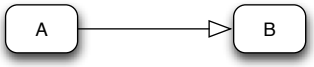


Figure 2.3 – Suggestion.

Suggestion

A suggestion (Figure 2.3) between contexts A and B means that the activation of A triggers the activation of B. However, if the activation of B fails, A is still activated. Conversely, the deactivation of A triggers the deactivation of B, if possible.

Requirement

Figure 2.4 – Requirement.

A requirement (Figure 2.4) between contexts A and B means that A can only be activated if B is already active. The deactivation of B triggers the deactivation of A. Furthermore, if A cannot be deactivated, neither can B.

Exclusion

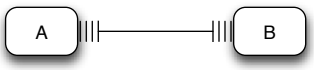


Figure 2.5 – Exclusion.

An exclusion (Figure 2.5) between contexts A and B means that A can only be activated if B is not already active and conversely.

2.4 Conclusion

In this chapter, we introduced the COP paradigm, which allows dynamic behaviour adaptations of software. In order to achieve this goal, it models situations by *contexts* that contain a set of method *adaptations*. These contexts can be *combined* and system coherency is enforced through *relationships*.

We will implement these concepts as an extension of the Ruby language and combine them with the ones of FOP presented in Chapter 3 to finally provide our new concepts of CaaF that will allow us to put FaaS on Rails in Chapter 10.

Chapter 3

Feature-Oriented Programming

Contents

One new feature or fresh take can change everything.

Neil Young

3.1 Introduction

In this chapter, we will present the main concepts of Feature-Oriented Programming (FOP) and identify which of these concepts are relevant to the new Context as a Feature (CaaF) concept that we will develop further in this thesis.

One of the reasons why we are interested in FOP is that, while current Context-Oriented Programming (COP) implementations provide behavioural adaptation in a fine-grained and dynamic approach, FOP for its part addresses the same class of problems but from a more static and structural approach. These two paradigms are quite similar [CGD11] and we think that the coarser-grained functionality-centred point of view from FOP is something currently missing in COP.

3.2 Motivation

In Section 2.2, we defined our TODO-list running example. We will use the same example here to express our need for features. To illustrate this, we will provide the following extensions to our base application:

- The detection of the environment, for example, if the application executes in full-screen mode, is handled by the feature EnvironmentSense.
- The detection of the current Operating System (OS) is handled by the feature OperatingSystemsSense. It can be used to adapt the notification behaviour.
- TODO-lists can involve several people. There is a need of sharing TODO-items either through social networks or email.

3.3 Concepts

3.3.1 Feature

The FOP paradigm regards end-user visible behaviour, identifying all the functionalities [CGD11] or the increments on top of the software [Ape08] that are relevant to a stakeholder. Usually, a feature is a cross-cutting concern not implemented by a single class only but by a set of collaborating classes.

The concept of feature is usually defined as follows in the literature: "The concept of features initially emerged with the goal of expressing distinct functionality that is targeted towards a specific stakeholder. This notion of a feature is called conceptual, because it only regards the end-user visible behaviour, but not its implementation."[CGD11] A feature can also be defined as "a unit of functionality that may be added to (or omitted from) a system"[PR01] or again "an optional or incremental unit of functionality"[Zav03].

If we apply these definitions on the TODO-list example, the sharing functionality fits perfectly the concept of feature. Indeed, this is clearly something that incrementally extends the base application and is visible for the end-user.

Two kinds of granularities characterize a feature. On the one hand, a feature can be fine-grained, in this case, its definition is some lines, e.g. methods in one or multiple files through the application. On the other hand, a feature can be coarse-grained, e.g. its definition is done through classes, modules. The granularity of the feature depends on its interactions with the application. Indeed, where many interactions occur at precise points through the entire application, the granularity tends to be finer, and inversely.

In FOP, there are three ways to implement feature, annotations, modules and refinements. Using the first one, the definition of feature is achieved using annotations in

the program's source code. The second way is by using the modularization mechanisms provided by the language like traits, mixins or classes. Finally, refinement is the use of an external feature representation that is added to the software.[CGD11]

Features are typically used in Software Product Line (SPL) in order to easily maintain the base-code of a multi-user application. Using features in this context helps to have a better modularization and separation of concerns. Thus facilitate development and maintenance. Using SPL, it becomes very easy to produce different applications for specific stakeholders.

3.3.2 Feature-Oriented Domain Analysis

Feature-Oriented Domain Analysis (FODA) is a domain analysis method to help analyse applications in order to develop domain-specific products that remain generic and widely reusable. Genericity is obtained by abstracting all factors that make different applications in that domain different from each other. Once those factors are identified, the produced parametrized domain models can be instantiated to produce domain-specific applications.[KCH+90]

In software engineering, almost no applications are built from scratch any more. They typically reuse some frameworks or other building blocks. FODA bases its method on multiples of those abstractions such as aggregation/decomposition, generalization/specialization and parametrisation. FODA uses aggregation and generalization in order to find commonalities and refinements for differences.[KCH+90]

Parametrization is a concept already known and applied on code, like subroutine or macro. FODA also applies this concept at the level of design and requirements. The identified factors responsible for the differences among applications are used in parametrization. Those factors can be classified into four categories, capabilities, operation environments, application domain technology and implementation techniques.[KCH+90]

The previous concepts are partially illustrated on the TODO-list example. Instead of designing a specific application for each OS, the OS specificities are abstracted from the base application to generalize it. The OS detection feature is added on top of the base application in order to automatically parametrize it.

3.4 Comparison with Context-Oriented Programming

The concept of FOP is closely related to that of SPL, the applications of those concepts usually taking place at compile time. The orientation of FOP is to provide a modular software that can easily be composed to provide other variants before execution. On the other hand, we have COP which is more aware of changes in the environment at run time and does not deal with SPL. Our CaaF concept tries to combine the concepts of COP and FOP into a full run-time approach.

3.5 Conclusion

FODA is useful in the context of our case study application: Benubo, in Chapter 10. It helps us to analyse our application and decompose the application into a base functionality that can be extended using additional features.

Other important points retained for CaaF are the methodology and implementation possibilities related to the FOP paradigm. One of our main concerns is to provide the developer with an intuitive process to highlight contexts and features from the application, and with the simplest way to implement contexts and features regardless of the granularity.

Chapter 4

Ruby

Contents

God is a Ruby programmer, after all :-)

Andy Hunt

4.1 Introduction

The first version of the Ruby programming language was released in 1995 by Yukihiro Matsumoto.

This language was mostly inspired by Perl, Python and Smaltalk and aimed to be a "scripting language that was more powerful than Perl, and more Object-Oriented (OO) than Python" [Ste11]. It has a pure OO approach, everything is an object, even basic types such as integers. It is a dynamic interpreted language with duck typing [TFH09], meaning that the objects do not have a static type but can be any type to which they respond[1].

The key factor in our choice was the great reflective capabilities Ruby provides. It gives introspective and intercessive access to the running application and we needed this in

[1] "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." [Hei07]

order to be able to adapt easily the behaviour at runtime [Bla09]. In addition to its reflective capabilities, also found in other languages like Common Lisp or Smalltalk, Ruby is used in industry and especially by our industrial partner.

4.2 Strengths and Weaknesses

Strengths

The large and growing community of this language is a great strength because it provides good feedback for the new gems that are developed, and therefore ensures the durability of the language. In our case, we wanted to develop an implementation of Context-Oriented Programming (COP) that was usable in real life. A first condition to this was to develop it in a language that is really used by industry. Another advantage of the Ruby community in our case is that the developers are familiar with reflection and meta-programming and so hopefully, will adopt COP more easily than others.

Ruby owns a packaging system named RubyGem. A RubyGem (commonly named a "gem") contains packaged Ruby applications that are easy to download, install and update in a system. It comes with dependencies and version management. Numerous of gems have been developed in open source and add many capabilities to the language. In our case, we package our implementation into a gem and it allows anyone to use our framework by only adding one line in their applications.

Finally, Ruby on Rails (RoR), which will be presented in Chapter 5, is the "killer gem" in Ruby and is heavily used in real-world projects.

Weaknesses

The Ruby programming language has a low support for multi-threading, actually, in the default interpreter (Matz's Ruby Interpreter (MRI)), a global lock prevents two threads from being executed at the same time.

Secondly, as it is an interpreted language, it leads to slower performances than compiled languages like C,.. or languages like Java with Just In Time (JIT) compilers.

Finally, although Ruby is now becoming popular this was not the case for several years and thus it has not reached the same maturity as other languages of the same generation like Java, for instance.

4.3 Why Ruby?

As presented in this chapter, we chose Ruby for this thesis firstly because it is a language where industrial partners are available. Belighted, in particular, is interested in COP

technology and this company has provided us with an interesting case study.

An other advantage is its great reflective Application Programming Interface (API). This allows us to extend this OO language with our Context as a Feature (CaaF) concepts without having to modify the interpreter or to use a pre-compilation step.

Finally, the facility provided by the RubyGem packaging facilitates the spreading of our gem that can be installed and ready to go using a single command[2].

[2] `gem install phenomenal`

Chapter 5

Ruby on Rails

Contents

Rails is the killer app for Ruby.

Yukihiro Matsumoto

5.1 Introduction

First released in 2004, Ruby on Rails (RoR) [Fer10] is an open source full-stack web framework[1] built on top of Ruby. Over the years, more and more people have become involved in its development through GitHub[2] and now use the framework in their applications. Thanks to this enthusiasm for RoR, the framework improves every day and is very responsive to the community's needs.

[1] A *full-stack web framework* provides an all-in-one solution including Uniform Resource Identifier (URI) routing, caching, tools, its own rules like MVC, for example, needed to develop a complete web application.

[2] `https://github.com/`

The framework contains a variety of tools to ease the developer's life. *Scaffolding* is used to automatically generate files related to a controller, model, database access, for example. *WEBrick* is a simple server written in Ruby that can run an application just by typing one command. *Rake* is a building system used in multiple contexts like managing databases, assets compilation, etc. The additional packages are all managed by the RubyGem tool of Ruby.

RoR uses the *Model-View-Controller (MVC)* as the main architectural pattern for a web application. The *Model* contains all the business logic of the application and is mapped to a database table by default. The *Controller* handles user requests by determining which file needs to be shown to the user from the *View*. Typically, a controller contains several simple actions accessible through a routing system.

The way RoR maps external requests through the appropriate controller is achieved by the routing system. In RoR, a route maps HyperText Transfer Protocol (HTTP) verbs and URI to controller actions. The routing system is based on the Representational State Transfer (REST) architecture that forces the developer to use routes like create, new, edit, update, destroy, show, and index. Those routes are automatically declared for a model and mapped to the right controller actions simply by adding a line in the routes file.

There are some concepts that are part of the RoR philosophy like Convention over Configuration (CoC) or Don't Repeat Yourself (DRY) which make the framework very pleasant to use. CoC is the fact that a lot of concerns that are generally defined by the programmer are now fixed by conventions. Configuration becomes necessary when something outside the predefined conventions needs to be defined. Furthermore, the conventions have to be verified because, the entire behaviour of the framework is based on those conventions, e.g. routing or database access. For example, the name of a model is in singular and the name of the associated table is in plural such as "User" model associated to the "users" table. If this convention is not verified, the framework will fail to access the database for this model. DRY states that all pieces of information have their unambiguous places which it avoids duplicating the same information in multiple places[HT99].

5.2 Concepts

5.2.1 Application Middleware

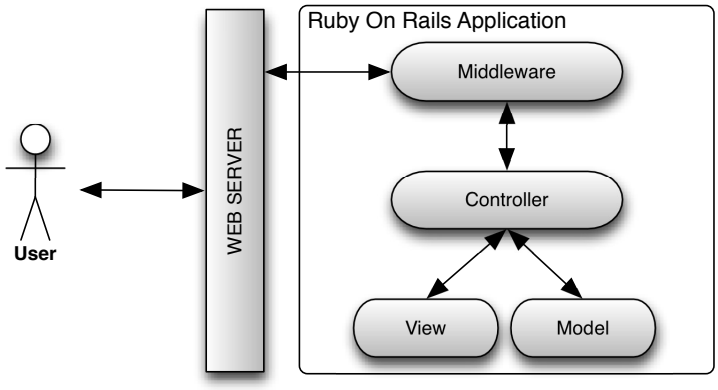


Figure 5.1 – Simplified application stack.

Figure 5.1 illustrates the simplified RoR application stack. The interesting part we will use during the implementation of our framework is the *middleware* layer. The middleware is an object (or multiple objects) that stands between the user (browser) and the controller of the application. Thus all requests go through the middleware before being processed by the controller and inversely after the processing.

5.2.2 Rendering Mechanism

The rendering mechanism is typically used by the controller to send a response back to the user, by passing control to the view. It can also be used inside views themselves to compose views with each other.

RoR can handle views of different format like Embedded Ruby (ERB), HTML Abstraction Markup Language (HAML) or JavaScript (JS). In order to convert those different formats into HyperText Markup Language (HTML), the framework uses builders. In addition, views can be either partial or template, a partial is part of a view built to be integrated in a template. To display a requested file to the user, the rendering mechanism checks if there exists a partial or template of the right name and "locale"[3]. Then, it checks thanks to the extension of the files if the corresponding builder is available for the file. Finally, if a file is selected, it is converted in HTML and displayed on the screen for the user.

[3]Locales are specific options like language, date or currency formats needed for internationalization.

5.2.3 Lazy Loading

Lazy loading is a pattern that aims to delay the loading of an object until it is really needed. This pattern is used to accelerate the loading of an application during its development. It is disabled in the production environment of RoR, in which case all classes are eager loaded at system startup.

5.3 Strengths and Weaknesses

Strengths

Thanks to the philosophy and all the tools provided by RoR, development is made very fast and easy. Its philosophy helps to organize the code and facilitate team work. With all the built-in tools and the Gem system, the required dependencies for the application can be installed easily.

Weaknesses

RoR provides many additions to help a developer, but those additions can also hinder performance. In fact, all methods are added using the reflective Application Programming Interface (API) of Ruby. Generally, many of them are not used by the developer but remain in the system which results in a lack of performance.

5.4 Why Ruby on Rails?

For the validation of this thesis, we wanted to apply our paradigm to a real life case study and were given the opportunity to work in partnership with the Belighted company in Louvain-La-Neuve specialized in RoR applications.

The RoR framework gives us a better chance to bring Context-Oriented Programming (COP) to the masses. Firstly, its community is used to meta-programming concepts. Secondly, the entire framework is open source, which makes it easy to understand how it works and to add many functionalities to the framework.

RoR seems to be very appreciated and used in real world projects like Groupon[4] or GitHub. Some people may think that RoR is not suited for heavy-load applications but Github is clearly the proof that RoR has nothing to envy from other big web frameworks.

[4] `http://www.groupon.com/`

Chapter 6

Related Work

Contents

If I had eight hours to chop down a tree, I'd spend six sharpening my axe.

Abraham Lincoln

6.1 Introduction

This chapter will analyse different papers that illustrate Feature-Oriented Programming (FOP) and Context-Oriented Programming (COP) paradigms. Those works deal with problems that we have in common, and give us the opportunity to improve our implementation. The two paradigms differ but our goal will be to select the most interesting concepts and combine them in order to build our new Context as a Feature (CaaF) concept.

6.2 ContextR

ContextR is a Ruby COP framework developed by Gregor Schmidt for his master thesis in 2008 at the University of Potsdam [Sch08]. The aim of this thesis was to design a COP framework and test it on a web application. The approach was to represent contexts as

layers that are added to classes of the base application and that can be (de)activated at run-time. The framework and application were developed on Ruby 1.8.7 and are no longer compatible with the current major version of Ruby (1.9).

In order to model layer behaviour, ContextR uses the module abstraction of Ruby. Layers provide the ability to define partial behaviour to be executed in a given context. Furthermore, layers are stateful, meaning that they can contain variables that will be restored between activations. Finally, they have the ability to add methods that are available only when a layer is active.

In order to define partial behaviour two syntaxes exist: either one use a plain-old module that can be reused through different layers or classes, or alternatively, one can define layer-specific behaviour directly in the classes that have to be adapted using the `in_layer` (See Figure 6.1) keyword. (This will then generate an anonymous module.)

```
1 class Religion
2  in_layer :location do
3   def to_s
4    "#{super} (#{yield(:receiver).origin})"
5   end
6  end
7 end
```

Figure 6.1 – Layer-specific behaviour definition.

In order to activate a layer, the framework provides the `with_layer` (See Figure 6.2) keyword:

```
1 christianity = Religion.new("Christianity", "Israel")
2  ContextR.with_layer :location do
3   christianity.to_s == "Christianity (Israel)" # True
4  end
```

Figure 6.2 – Layer activation.

The framework has the following restrictions:

- The framework does not provide support for anonymous contexts. If it is needed, the developer will have to generate new unique names himself, because all the layers have to be named uniquely.
- To call an adaptation that was previously deployed for a method, i.e the adaptation of the "super" context, the keyword `super` is used. A problem occurs when an inherited method is redefined in a subclass. In fact, the `super` keyword will not

call the inherited method implementation because it is used for other purposes by the framework.

- Some methods are not available for adaptations (`__id__`,`__send__`,...) because they are used internally by the gem.
- Because of the modularization approach of this framework, `self` is not available in the layers. You have to use `yield(:receiver)` instead, but you may only call public methods.

ContextR is relatively old (at least for the fast moving Ruby world) but has a great number of capabilities needed by a COP framework. What is missing are the relationships between contexts and the ability to define custom conflict resolution policies. Its most important strength is that the activation of layers are scoped and so support multi-threaded applications easily. Real multi-threading is not yet supported in the default interpreter of Ruby (Matz's Ruby Interpreter (MRI)), but other interpreters, like JRuby, support this feature.

Finally, although it works well, we think that the syntax provided could be enhanced in order to bring it closer to day-to-day programmers, and that it still contains too many uses of workarounds (like the `yield(:receiver)` needed to replace the `self` keyword in the layers (see line 4 in Figure 6.1).

6.3 rbFeature

rbFeature is a versatile Ruby implementation for FOP created by Sebastian Günther in the context of his PhD thesis [GF11]. The project was started with two motivations, the first one was to understand how to build a DSL from scratch, and the second was to see how to add a non-native paradigm to Ruby.

The framework allows the definition of features as first-class entities and allows to define features using annotations. We explain the different approaches to define feature in Section 3.3.1. In this framework, the annotations approach was preferred to the module or refinement approaches for its ability to define either coarse or fine-grained features. More precisely, rbFeature uses feature containments that are semantic annotations consisting of a condition and a body. In Figure 6.3, `(Print | ConsoleOutput)` is the condition and the block (closure) passed to the method `code` is the body. The latter is evaluated only if the condition is true, the evaluation of the block will define the method `print`.

In addition, rbFeature has two important capabilities which are the Variant Generator and *Feature Aggregator*. *Variant Generator* is a tool to pre-process the program and prune feature-specific code to provide a program with a fixed behaviour, like a Software Product Line (SPL). The *Feature Aggregator* allows the developer to extract a file with the feature-specific code that will give him a coherent view of the feature.

```
# Print | ConsoleOutput is the containment condition.
(Print | ConsoleOutput).code do
# The lines below define the containment body that is executed
# only if the containment condition is true.
 def print
  ...
 end
end
```

Figure 6.3 – Feature containment.

The technique used by rbFeature to generate the code using the *Feature Generator* is by modifying the Abstract Syntax Tree (AST). The first step is to convert the Ruby files using the Ruby2Ruby Gem. Then the AST is walked to find all the feature containments and the condition of the containment is evaluated to see if the body of the containment must be included or not. Finally, the modified AST is reconverted to Ruby using the RubyParser Gem.

The features are (de)activated by an external trigger. rbFeature does not use sensors to detect changes of the environment, all the features are only commanded by the user.

rbFeature is a complete implementation of the FOP paradigm in Ruby. It has some interesting capabilities like the principle of SPL (Variant Generator) which can create variants of the application according to the features selected at compile time. Another interesting capability is the ability to define dependencies between features.

6.4 Subjective-C

Subjective-C [GCM+11] is a COP framework designed on top of Objective-C, a language for mobile platforms. The main motivation was to bring COP to mobile platform because this had not been done before, although those platforms would probably be most exposed to context changes. Subjective-C was also the first to introduce a Domain Specific Language (DSL) to express contexts interdependencies in COP implementations.

The framework defines contexts as first-class entities and defines the adaptations using annotations in the code where the adaptations normally take place, like rbFeature.

```
#context Landscape
- (NSString*)getText() {
 return [NSString stringWithString:@"Landscape view"];
}
```

Figure 6.4 – Context-specific method definition.

To (de)activate contexts, Subjective-C uses internal triggers based on the system state, which gives the ability to react to a change in the environment. Contrary to the other COP implementations available at that time, the framework uses method pre-dispatch. The method pre-dispatch has the ability to change the method's implementation at each adaptation of the method, unlike other COP implementations which adapt the method lookup process.

Subjective-C is a great implementation of COP for mobile platforms. Its main drawback is the preprocessing required for the annotations. Furthermore, the preprocessing replaces annotations by generated code, which leads to some difficulties for debugging. In spite of this, it provides the ability to specify dependencies between contexts using relationships, exclusion or requirement, for example. The method pre-dispatch constitutes an advantage to COP applications whose context do not often change because the method implementation is adapted once during context (de)activation. Otherwise, the method pre-dispatch becomes a disadvantage if (de)activations occur more frequently than method calls.

Part II

Framework

Chapter 7

Phenomenal

Contents

If we knew what we were doing, it wouldn't be called research, would it?

Albert Einstein

7.1 Introduction

In this chapter, we will present the main contributions of this thesis in terms of concepts and implementation. We will start by presenting intuitively our vision of contexts, relationships and especially the brand new notion of Context as a Feature (CaaF). These concepts will then be formally defined through a semantics based on sets and functions. The actual implementation will then be discussed, and this chapter will end with the presentation of the current limitations of the framework.

Incidentally, we chose the name "Phenomenal Gem" because, like Perl, the Ruby language bears the name of a gemstone. A RubyGem is a software package, commonly called a "gem". The name "Phenomenal Gem" thus comes from the fact that the phenomenal gemstone is a gemstone that will change colour depending on external factors such as light or temperature. In our case, it is a gem that adapts software behaviour according to the surrounding situation.

7.2 Concepts

As presented in Part I Background, the goal of this thesis is to merge the paradigms of Context-Oriented Programming (COP) and Feature-Oriented Programming (FOP). The approach used is to first implement a COP framework and then bring in some interesting ideas of FOP that were missing. In fact, these two paradigms have more points in common than differences [CGD11]. Thus, we first implement all the regular COP concepts (contexts, relationships, adaptations, management,...), drawing inspiration mainly from Subjective-C [GCM+11]. Then we attempt to integrate the idea of feature into it in order to obtain the CaaF concept.

7.2.1 Context

We defined what a context is in Section 2.3.3. This definition is the one we use in the Phenomenal Gem framework. We take the concept of context and reify it in Ruby as

a class. Around our framework we defined a small Domain Specific Language (DSL) that provides a structured and easy access to the new concepts we add to Ruby. While defining it, we tried to stick to the Ruby conventions as much as possible.

Simple Context

```
1 context :Trial do
2  adaptations_for TodoList
3  adapt :add_task do |date,time,title,text|
4   ... # Behaviour of the adaptation
5  end
6 end
```

Figure 7.1 – Context definition. The *Trial* context is used in the TODO-List application to limit functionalities until the application has been bought. In this case, the adaptation will limit the number of tasks that can be added.

Figure 7.1 introduces the definition of a new context. Several points are highlighted in this simple example. First, syntactically the context definition is similar to a new class definition. Next, `adaptations_for` allows to specify the class to be adapted. Then, `adapt` provides the capability to adapt an instance method by passing a block (closure) as implementation.

A context can be reopened later. In the same way as open classes allow to add methods at any time in Ruby, open contexts allow to add adaptations at any time (see Figure 7.2).

```
1 context :Trial do
2  adaptations_for TodoList
3  adapt :get_tasks do |date,time|
4   ... # Behaviour of the adaptation
5  end
6 end
```

Figure 7.2 – Reopen the *Trial* context in order to add the adaptation that will limit the number of tasks that can be retrieved.

Combined Context

A context can be combined with others to provide behaviour specific to this combination of contexts. Our DSL offers two different ways of defining combined contexts, either by nesting the context definitions (Figure 7.3) or by passing multiple context names to the `context` keyword (Figure 7.4). The only distinction between the two notations is that the former sets the surrounding context as the parent of the other. We will see how it is used in Section 7.2.4.

A *sub-context* is one of the contexts that compose a combined context. In Figure 7.3, con-

```
context :Away
 context :Network do
  adaptations_for TodoList
  adapt :notify do
   ... # Behaviour of the adaptation
  end
 end
end
```

Figure 7.3 – Nested context definition. The combination of the *Away* and *Network* contexts expresses the behaviour that occurs when the user is away from his computer and the network is available.

text *Away* and *Network* are both *sub-contexts* of the combined context *[Away,Network]*.

```
context :Away, :Network do
 adaptations_for TodoList
 adapt :notify do
    ... # Behaviour of the adaptation
 end
end
```

Figure 7.4 – Combined context definition.

Activation Count

Each context contains a value called *activation count*. This value is incremented at each context activation and decremented at each deactivation. The main purpose of this value is the detection of the true deactivation. A true deactivation occurs when the *activation count* is equal to 0. In this case, all the adaptations of the context are removed.

The integer counter is needed because the activation of a context can be triggered by relationships, users, or automatically. Therefore, the deactivation is effective only if there are as many deactivation as activation. Otherwise the system would become inconsistent.

Adaptation Activation

The adaptation activation is roughly the deployment of the new implementation to the target method. In order to make the new implementation effective, there exist two ways, method pre-dispatch and modifying the method dispatching semantics. The former changes the target method body by the new implementation whenever the adaptation is deployed. The latter modifies the method call mechanism and adds some dynamic behaviour that looks for the deployed adaptation implementation at each method call. Method pre-dispatch is more efficient when there are many method calls and few context switches, and conversely for the modification of the method dispatch semantic.

In our case, we use method pre-dispatch in the same way as Subjective-C (see Section 6.4) to avoid adding a proxy method or modifying the method mechanism.

7.2.2 Proceed

As multiple adaptations can be deployed for the same method in a specific order, the language offers the *proceed mechanism* to be able to call the implementation of the adaptation previously deployed for this method. The previously deployed adaptation is the one just before the adaptation where the `proceed` call is made according to the conflict policy. This behaviour is illustrated by the Figure 7.5.

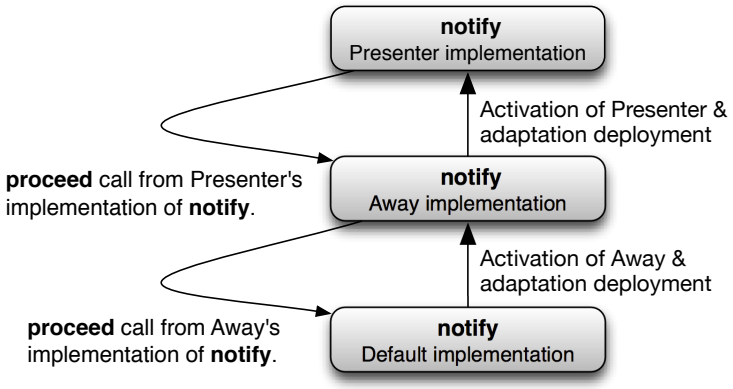


Figure 7.5 – Proceed example.

In Figure 7.5, after the activation of the context *Away*, the default implementation of `notify` in `TodoList` is replaced by the implementation of the adaptation defined for `notify` in the context *Away*. Subsequently, the same process is triggered by the activation of the context *Presenter*[1].

7.2.3 Relationships

In Section 2.3.5 we stated that relationships between contexts are subject to certain constraints to ensure overall system consistency. Our implementation provides a means of defining all these relationships.

The relationships between contexts are defined directly in the contexts bodies like in Figure 7.6.

We define a small DSL for the relationships introduced in Section 2.3.5. For example, a context that *requires* a context B will contain `requires :B` in its definition. A similar notation exists for `implies :B` and `suggests :B` for implications and suggestions relationships.

[1]This context reifies the situation where the user is doing something in fullscreen mode.

```
context :Mobile do
 requires :OperatingSystemsSense
end
```

Figure 7.6 – Relationships definition.

The syntax used in Figure 7.6 only works for asymmetric[2] relations. Otherwise, for symmetric[3] relations, the relationship definition is allowed outside the involved contexts using `requirement_for`, `implication_for` and `suggestion_for` as illustrated in Figure 7.9.

7.2.4 Context as a Feature

We presented the standard definition of feature in Section 3.3.1. However we have chosen to adapt it somehow and bring it into COP. From our point of view, a feature is a refinement of the notion of context. We see a feature as a coarser-grained context that provides particular functionality to stakeholders through its sub-contexts and sub-features. We therefore define the keyword `feature` which operates like `context` but defines a feature instead.

The main idea is that a feature is self-contained and that its (de)activation will (de)activate all behaviour and related relationships. This is why we decided to attach relationships to features to be able to (de)activate them when the feature itself is (de)activated.

Here, the nested definition syntax in Figure 7.7, means that the behaviour is defined for the combination of the feature and the nested context. Rather, the behaviour defined in a combined context is active only when all the sub-contexts are active.

```
feature :EnvironmentSense do
 context :Presenter do
  # Behaviour specific to :EnvironmentSense
  # and :Presenter combination
 end
end
```

Figure 7.7 – Nested definition.

The fact that relationships are attached to features means that, when you define a relationship in a context nested in a feature, this relationship will actually be stored in the feature, like in Figure 7.8. But this notation is only a shortcut, since relationships can also be defined immediately at feature level, allowing to define relationships between contexts that are not sub-contexts of the feature, like in Figure 7.9.

[2] One context plays a more important role than the other in the relation.
[3] Two or more contexts play the same role in the relation.

```
feature :OperatingSystemsSense do
 context :Android do
  implies :Mobile
 end
end
```

Figure 7.8 – Nested context relationships definition.

```
feature :OperatingSystemsSense do
 implications_for :Android, :on=>:Mobile
end
```

Figure 7.9 – Feature relationships definition. The ":on" is used to match with the RoR conventions and allows to read the DSL as plain English.

In COP implementations [GCM+11], a default context is defined which models the base application behaviour. In our case, we have a *default feature* which, in addition to being default context, stores the relationships defined at root level (in contexts like the ones in Figure 7.6 or directly in the *default feature* like in 7.10).

```
requirements_for :Mobile, :on=>:OperatingSystemsSense
```

Figure 7.10 – Default feature relationships definition.

The advantages of such a *default feature* are the following. First, it represents the default application as a feature. When a method is adapted, its default implementation is saved as an adaptation in the default feature. This allows to simplify the implementation by using the same mechanism for the adaptations and the default behaviour. Secondly, when relationships are defined outside a feature, instead of having special mechanism, the relationships are stored in the default feature.

Figure 7.11 provides a schematic representation of a system developed with the Phenomenal Gem. It shows the default feature which is the parent of all other features and contexts and stores the relationships of root contexts (this feature is always active and models the base application behaviour). On the left, the feature *OperationSystemSense*[4] is defined with some relationships and sub-contexts; on the right we have some simple contexts *Android,...,Mobile* which are combined with the default feature to form sub-contexts of it.

Conceptually, the CaaF also provides structural adaptations, meaning that a feature should be able to add/remove methods and classes through its (de)activations. We do not have support for this yet in the current implementation, but we think it could be an interesting area of research to complete the concept in this sense in the future.

[4]This feature reifies the behaviour needed to detect the operating system.

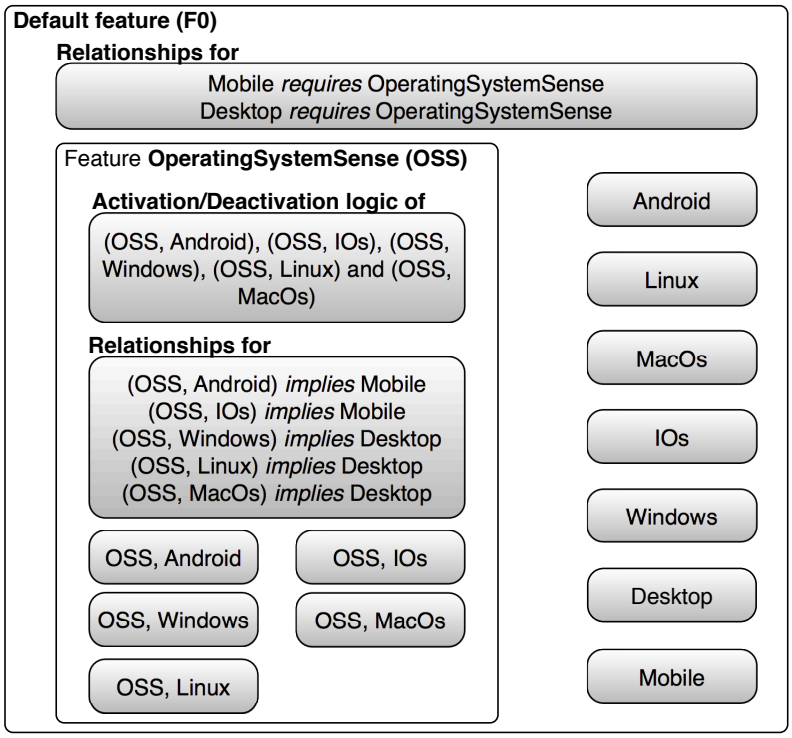


Figure 7.11 – Architecture of a Phenomenal application. The code used to draw this figure is in Appendix B.

7.3 Tools

7.3.1 Context Visualizer

Alongside the development of the framework, we felt the need for a debugging tool that would allow the developers to easily get a representation of their system state. System state is characterized by the defined contexts, features, relationships, and if they are activate or not.

Once an application contains several contexts and features having complex relationships between them, it can become really difficult to determine which contexts, relationships or features are active at any moment in time, and thus to debug the system.

The Phenomenal Gem framework comes with a tool which is able to automatically generate a graphical view of the system state at any time. Figure 7.12 shows the representation of the running example[5]. The squared boxes represent features and the rounded ones contexts. The relationship notation is defined in Section 2.3.5. The red boxes and arrows mean that the contexts or relationships are active at the time the image was generated.

[5]The code used to generate this graph is in Appendix B.

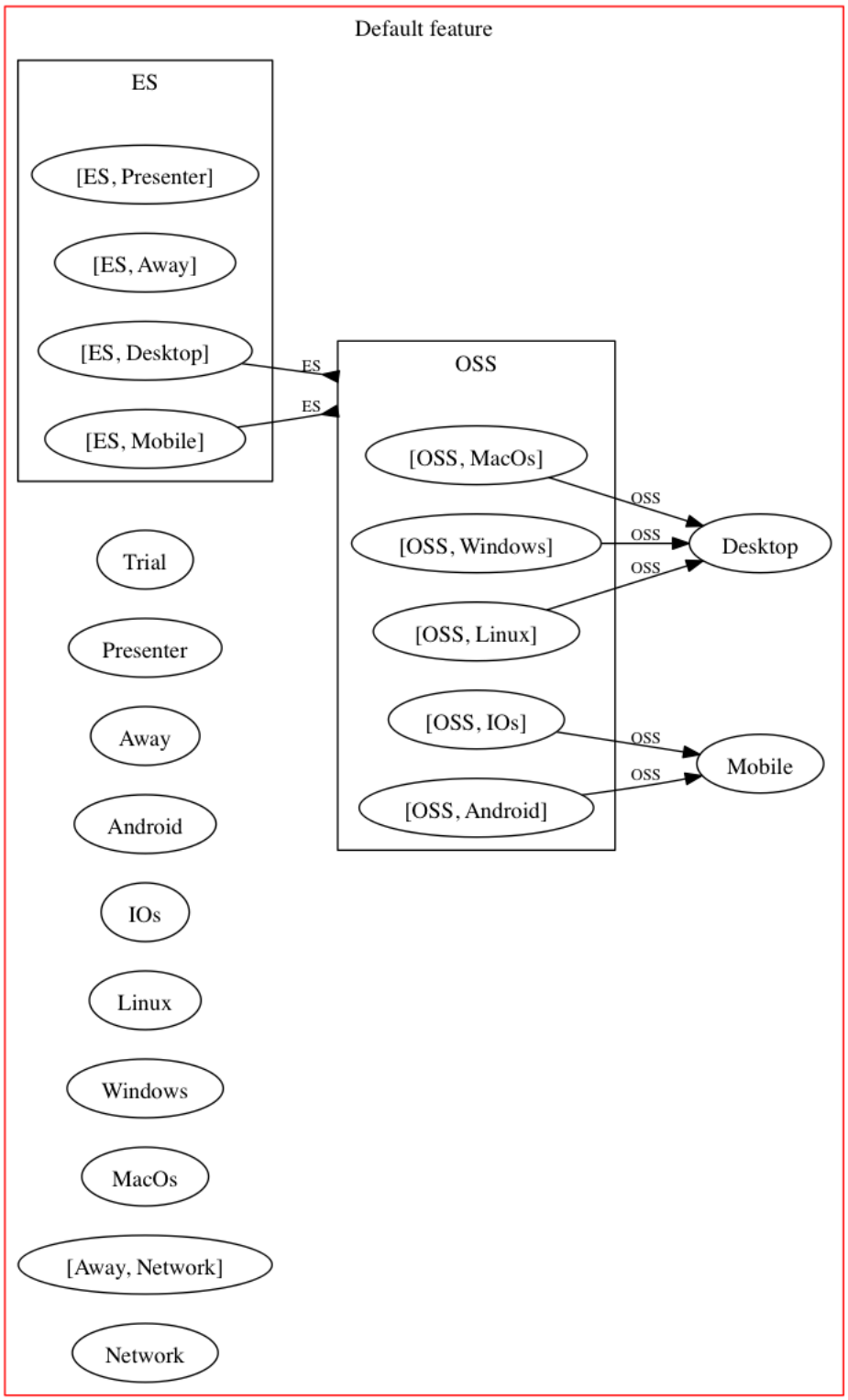


Figure 7.12 – Graphical View. For clarity, OSS stands for OperatingSystemSense and ES for EnvironementSense.

7.4 Semantics

The following formal semantics is based on sets and functions. Although, it nonetheless presents the main mechanisms and ideas of the framework in order to have a precise definition of them. These definitions only assume that the framework is build on top of an Object-Oriented (OO) programming language. This allows to be used for other implementations without requiring the reader to dig into our Ruby code to know what we have done. For the sake of simplicity, the handling of relationships and errors has been omitted.

7.4.1 Building Blocks

Particular Notations

$\rightarrow$	This arrow defines a total function.
$\nrightarrow$	This arrow defines a partial function.
$my\_function[x/y]$	The value y is set at x for $my\_function$ such that $x \in dom(my\_function)$ and $y \in range(my\_function)$
$my\_function[\![x]\!]$	The $[\![...]\!]$ specifies a parameter in $my\_function$ signature. Here x is a parameter of $my\_function$.
$concat(name1, name2)$	This function concatenates $name1$ and $name2$ and returns $name1, name2$

Sets

$\mathbb{I}d$	Set of all possible identifiers.
$\mathbb{C}ontextId \subseteq \mathbb{I}d$	Set of all possible context identifiers.
$\mathbb{M}ethodId \subseteq \mathbb{I}d$	Set of all possible method identifiers.
$\mathbb{C}lassId \subseteq \mathbb{I}d$	Set of all identifiers of defined classes.
$\mathbb{C}ontext$	Set of all possible contexts.
$\mathbb{F}eature \subset \mathbb{C}ontext$	Set of all possible features.
$\mathbb{M}Body$	Set of all possible method bodies.
$\mathbb{M}Arg$	Set of all possible method arguments.
$\mathbb{O}bject$	Set of all objects.
$\mathbb{B}oolean \subset \mathbb{O}bject$	$\{True, False\}$
$\mathbb{R}elationship$	$\{Implication, Suggestion, Requirement\}$

Values

$Void \in \mathbb{O}bject$	=	The void object.
$DefaultFeature \in \mathbb{F}eature$	=	The default feature that represents the base application.

Functions

$$\begin{array}{lcl} defined\_context & : & \mathbb{C}ontextId^{1..*} \\ & \nrightarrow & \mathbb{C}ontext \times \mathbb{C}ontext \end{array}$$

The function maps all the defined $\mathbb{C}ontextId$ to a $\mathbb{C}ontext$. The first $\mathbb{C}ontext$ of the Cartesian product is the context specified by $\mathbb{C}ontextId$ and the second is the surrounding context (parent).

$$\begin{array}{lcl} defined\_combined\_context & : & \mathbb{C}ontextId^{1..*} \\ & \nrightarrow & \mathbb{C}ontext \end{array}$$

The function maps all the defined $\mathbb{C}ontextId$ that designate a combined context to a $\mathbb{C}ontext$.

$$\begin{array}{lcl} active\_context & : & \mathbb{C}ontextId \\ & \nrightarrow & \mathbb{C}ontext \end{array}$$

The function maps all the $\mathbb{C}ontextId$ that designate an activated context to a $\mathbb{C}ontext$.

$$\begin{array}{lcl} defined\_method & : & \mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{B}oolean \\ & \nrightarrow & \mathbb{M}Body \end{array}$$

The function returns the $\mathbb{M}Body$ for $\mathbb{M}ethodId$ that is defined in $\mathbb{C}lassId$. The $\mathbb{B}oolean$ specifies if it is an instance method or not. The function represents the implementation that is currently used by a method.

$$\begin{array}{lcl} adaptation & : & \mathbb{C}ontext \times \mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{B}oolean \\ & \nrightarrow & \mathbb{M}Body \end{array}$$

The function returns the $\mathbb{M}Body$ for $\mathbb{M}ethodId$ that is defined in $\mathbb{C}lassId$ by $\mathbb{C}ontext$. The $\mathbb{B}oolean$ specifies if it is an instance method or a class method.

7.4.2 Core Mechanisms

The functions in the following table will be explained one by one in detail below.

$$\begin{array}{lcl} context & : & \mathbb{C}ontextId \\ & \nrightarrow & \mathbb{C}ontext_{\setminus \mathbb{F}eature} \\ feature & : & \mathbb{C}ontextId \\ & \nrightarrow & \mathbb{F}eature \\ adapt & : & \mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{C}ontext \times \mathbb{M}Body \times \mathbb{B}oolean \\ & \nrightarrow & Void \\ conflict\_policy & : & \mathbb{C}ontext \times \mathbb{C}ontext \\ & \rightarrow & \{-1, 1\} \end{array}$$

$next\_adaptation$	:	$\mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{M}Body \times \mathbb{B}oolean$
	$\nrightarrow$	$\mathbb{M}Body$
$eval$	:	$\mathbb{O}bject \times \mathbb{M}Body \times \mathbb{M}Arg^*$
	$\rightarrow$	$\mathbb{O}bject$
$proceed$	:	$\mathbb{O}bject \times \mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{M}Body \times \mathbb{M}Arg^*$
	$\nrightarrow$	$\mathbb{O}bject$
$first\_adaptation$	:	$\mathbb{C}ontext \times \mathbb{C}lassId \times \mathbb{M}ethodId \times \mathbb{B}oolean$
	$\nrightarrow$	$\mathbb{M}Body$
$activate\_context$	:	$\mathbb{C}ontextId$
	$\nrightarrow$	$Void$
$deactivate\_context$	:	$\mathbb{C}ontextId$
	$\nrightarrow$	$Void$

context$[\![identifier]\!]$

This function retrieves a context by its name or creates it if needed. It is always done in the scope of a parent context (at least the default feature) that will be used to find the closest parent feature in order to store defined relationships.

$parent\_context$ is the surrounding context.
If $identifier \in dom(defined\_context)$
  $c, p = defined\_context(identifier)$
Else
  $sc \in \mathbb{C}ontext_{\setminus \mathbb{F}eature}$ such that $sc, parent\_context \notin range(defined\_context)$
  $defined\_context' = defined\_context[identifier/sc,\ DefaultFeature]$
  If $parent\_context \neq DefaultFeature$
    $cc \in \mathbb{C}ontext_{\setminus \mathbb{F}eature}$ such that $cc, parent\_context \notin range(defined\_context)$
    $parents =$ All parents from $parent\_context$ to DefaultFeature included.
    $defined\_context' = defined\_context[concat(parents, identifier)/cc,\ parent\_context]$
    $defined\_combined\_context' = defined\_combined\_context[concat(parents, identifier)/cc]$
    $c = cc$
  Else
    $c = sc$
Return c

The function first checks, if the context is already defined. If true, the context can simply be returned. Otherwise, the function needs to create a new context. The creation mechanism starts by creating the simple context (sc) as the default feature for parent. Then, if the actual parent differs from the default feature, a combined context cc with sc and $parent\_context$ is created. Finally, c is returned.

feature$[\![identifier]\!]$

This function basically operates like `context` but is applied to a feature.

---

parent_context is the surrounding context.
If $identifier \in dom(defined\_context)$
  $f, p = defined\_context(identifier)$
Else
  $sf \in \mathbb{F}eature$ such that $sf, parent\_context \notin range(defined\_context)$
  $defined\_context' = defined\_context[identifier/sf,\ DefaultFeature]$
  If $parent\_context \neq DefaultFeature$
    $cf \in \mathbb{F}eature$ such that $cf, parent\_context \notin range(defined\_context)$
    $parents =$ All the parents from *parent_context* to DefaultFeature included.
    $defined\_context' = defined\_context[concat(parents, identifier)/cf,\ parent\_context]$
    $defined\_combined\_context' = defined\_combined\_context[concat(parents, identifier)/cf]$
    $f = cf$
  Else
    $f = sf$
Return f

---

adapt$[\![context\_id,\ class\_id,\ method\_id,\ method\_body,\ instance\_method]\!]$

This function creates and stores a new adaptation for a method (instance or class) in a class for a context.

$adaptation' =$
  $adaptation[context\_id,\ class\_id,\ method\_id,\ instance\_method/method\_body]$

---

conflict_policy$[\![context1,\ context2]\!]$

This function is a total ordering function that allows to compare the precedence of two contexts. Its genericness allows the framework users to define their own policy depending on custom needs.

Return 1 if *context*1 has precedence on *context*2
Return -1 if *context*2 has precedence on *context*1

---

next_adaptation$[\![class\_id,\ method\_id,\ method\_body,\ instance\_method]\!]$

Returns the next body after *method_body* for the method *method_id* in the class *class_id*. The next body is found using the precedence order defined by $conflict\_policy$.

eval$[\![instance,\ method\_body,\ method\_args^*]\!]$

Returns the evaluation of *method_body* with $method\_args^*$ in *instance*. The evaluation of the method is executed in the hosting OO language. The method execution semantics of the language is not modified.

proceed$[\![instance,\ class\_id,\ method\_id,\ method\_body,\ method\_args^*]\!]$

This function has to be called in the body of an adaptation. It uses the `next_adaptation` function to retrieve the adaptation that has to be evaluated.

$next = next\_adaptation(class\_id,\ method\_id,\ method\_body,\ method\_args^*)$
Return $eval(instance, next)$

---

first_adaptation$[\![class\_id,\ method\_id,\ instance\_method]\!]$

Returns the first adaptation for *method_id* in class *class_id* according to the precedence order defined by $conflict\_policy$ between all the contexts in *active_context*.

activate_context$[\![context\_id]\!]$

This function will deploy all the adaptations that are returned by the $first\_adaptation$ function for all methods adapted by the context. It will also activate the combined contexts that the context is part of and for which all the other sub-contexts are already active.

$active\_context' = active\_context[context\_id/defined\_context(context\_id)]$
$c, p = defined\_context(context\_id)$
$\forall cid \in \mathbb{C}lassId,\ mid \in \mathbb{M}ethodId,\ im \in \mathbb{B}oolean$
  such that $c, cid, mid, im \in dom(adaptations)$:
    $new\_method\_body = first\_adaptation(cid,\ mid,\ im)$
    $defined\_method' = defined\_method[cid,\ mid,\ im/new\_method\_body]$
$\forall cc$ such that $c \in range(defined\_combined\_context)$
  If $\forall cid \in defined\_combined\_context(cc)$ such that $cid \in dom(active\_context)$
    $activate\_context(cc)$

---

deactivate_context$[\![context\_id]\!]$

This function causes the redeployment of all the methods that were adapted by this context. It will also deactivate all the combined contexts of which the context is a part.

$active\_context' = active\_context[context\_id/NULL]$
$c, p = defined\_context(context\_id)$
$\forall cid \in \mathbb{C}lassId,\ mid \in \mathbb{M}ethodId,\ im \in \mathbb{B}oolean$
such that $c, cid, mid, im \in dom(adaptation)$:
$new\_method\_body = first\_adaptation(cid,\ mid,\ im)$
$defined\_method' = defined\_method[cid,\ mid,\ im/new\_method\_body]$
$\forall cc$ such that $c \in range(defined\_combined\_context)$
$deactivate\_context(cc)$

7.5 Architecture

This section will present the architectural structure of the entire framework. We will start with a general overview of the framework. Then we will define packages, modules, classes and the role they play in the framework.

7.5.1 Global Structure

Figure 7.13 defines the architecture of the whole framework and dependencies between classes. The framework uses the `Phenomenal::` name space (omitted for clarity reasons) in order to avoid the redefinition of unwanted methods when used with other gems in an application. The framework is subdivided into multiple *packages* that are represented by directories in the file structure.

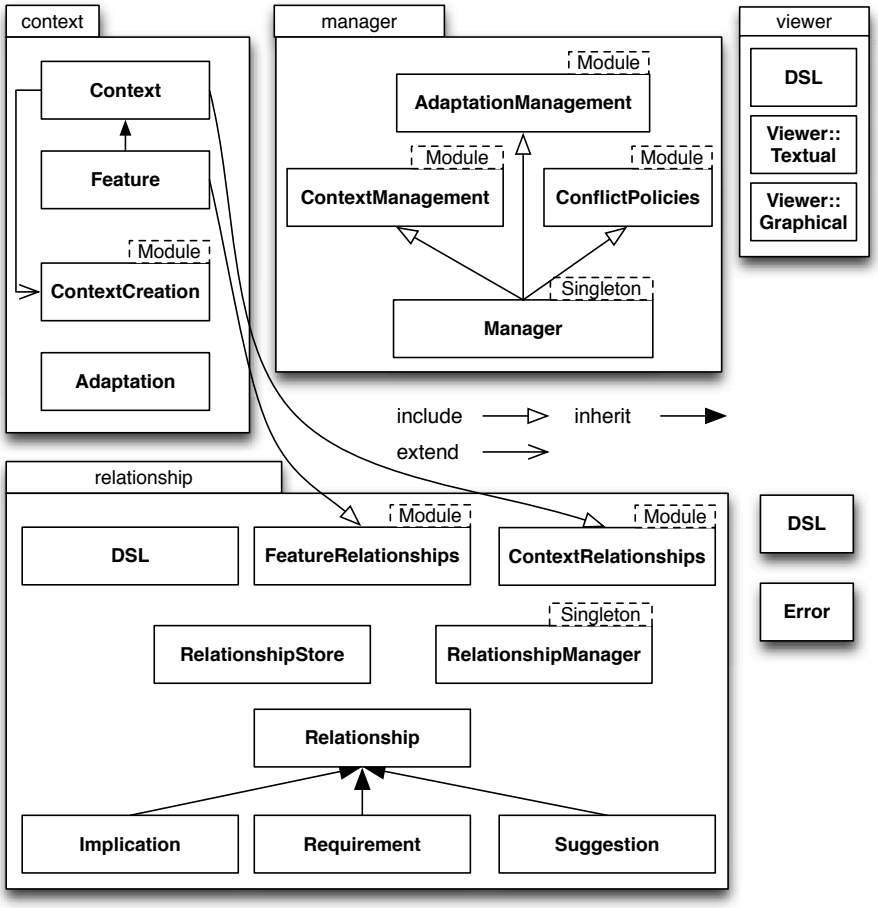


Figure 7.13 – Architecture Model.

The *context* package contains the building blocks for the framework; the *manager* pack-

age handles the interactions between the building blocks; and the *relationship* package contains all the logic related to relationships. The *context* and *feature* classes use modules from the *relationship* package to extend their behaviour with the relationships logic.

7.5.2 Context Package

The context package in Figure 7.14 is composed of building blocks for the framework. Indeed, the classes and modules inside reify the concepts of context, feature and adaptation. Furthermore, the package also contains all the logic for context reopening and creation.

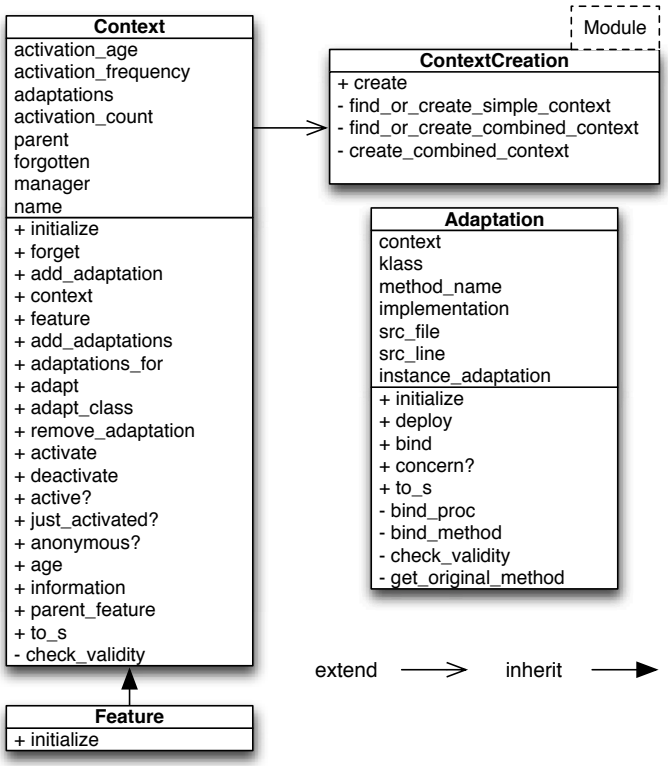


Figure 7.14 – Context package.

Context

Models a first-class context and implements the basic behaviour to activate/deactivate the context and add/remove adaptations independently of the other contexts. The creation of contexts is handled by `ContextCreation`. In addition, the relationships logic for contexts is included through the `ContextRelationships` module from the relationships package illustrated in Section 7.5.4.

ContextCreation

Implements all the logic required to detect the kind of context requested, either simple or combined. If the requested context already exists, it is simply returned. Otherwise, a new context is created and returned. In fact, this module implements the concept of open-context explained in Section 7.2.1.

Feature

Models a first class feature which inherits from `Context` because, as explained in Section 7.2.4, a feature *is a* context. The additional relationship logic is included through the module `FeatureRelationships` from the relationships package in Section 7.5.4.

Adaptation

Models a first-class adaptation and the logic for its deployment. In addition, the class contains all the logic needed to bind a *method* or a *block* (closure) to a specific class or instance. Therefore, this logic plays an important role in the proceed mechanism explained in Section 7.2.1.

7.5.3 Manager Package

The manager package in Figure 7.15 contains a class and several modules that bind the building blocks. Indeed, the package handles the interactions between adaptations or contexts and has to keep the application consistent. It also contains the management of conflict policies and built-in conflict policies.

ConflictPolicies

Implements two basic conflict resolution policies used to make a distinction between the contexts and build total order between them. Those policies can be selected by the user and are used by the `AdaptationManagement` module. Furthermore, it also implements a hook method used to create and change the conflict policy. The *age_conflict_policy*, which orders adaptations according to their contexts age (number of activations of contexts since the last activation of a context), is selected by default.

Manager

Implements the management of contexts and adaptations through the modules `ContextManagement`, `AdaptationManagement` and `ConflictPolicies`.

AdaptationManagement

Handles registration and deployment of adaptations. Using the information about contexts, it is able to determine which adaptations relevant to those contexts have to be deployed. Furthermore, the module also plays a role in the *proceed* mechanism explained

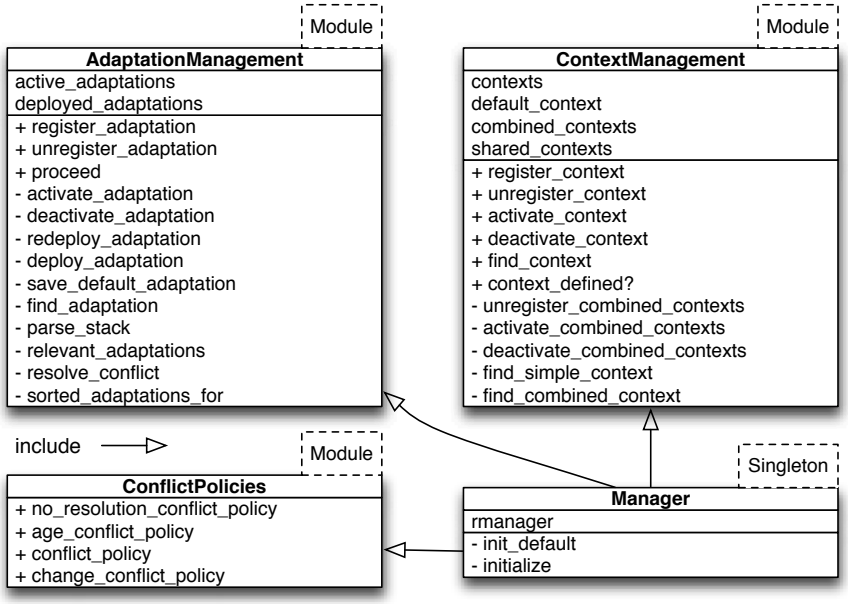


Figure 7.15 – Manager package.

in Section 7.2.2 to determine the next adaptation to be called.

ContextManagement

This modules handles registration and (de)activation of contexts.

7.5.4 Relationship Package

The package in Figure 7.16 contains all the classes and modules needed to add relationships between contexts. It contains the different relationships as well as their management for features and contexts.

RelationshipManager

Avoids duplication of relationships in the `RelationshipStore`. Furthermore, it handles relationships (de)activations for the features and ensures that all the relationships are satisfied to keep the system consistent.

RelationhipStore

Implements a fast data structure to store relationships allowing to retrieve them either by their source or by their target.

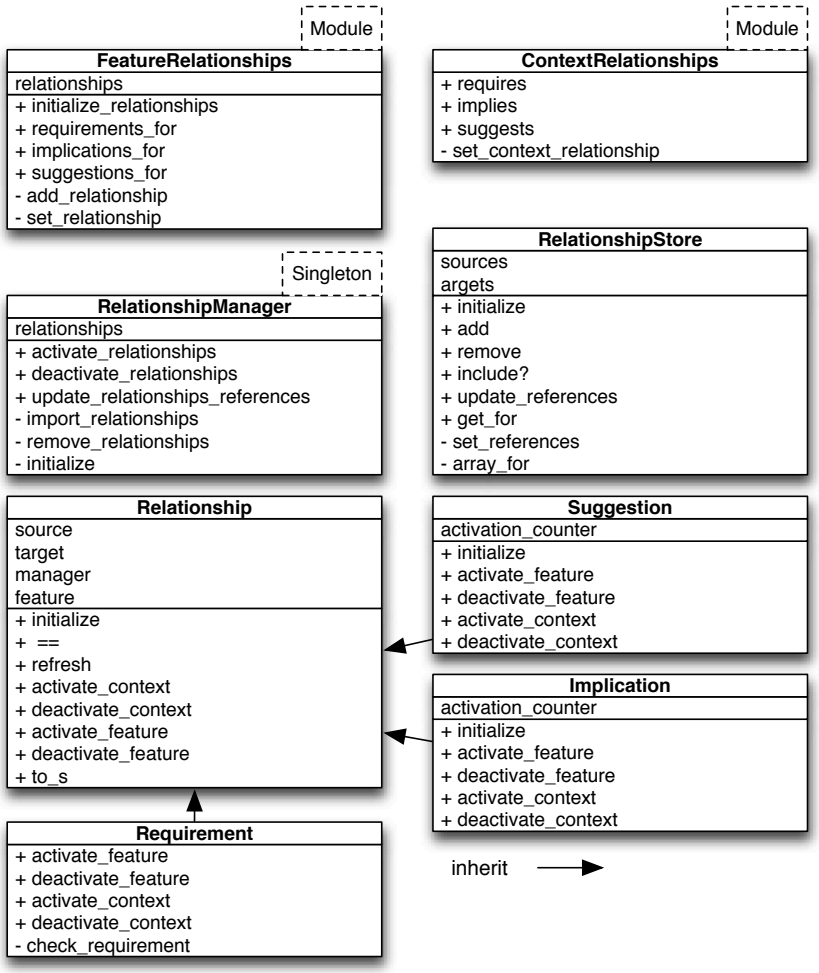


Figure 7.16 – Relationships package.

FeatureRelationships

Implements the logic needed by the relationships defined at feature level.

ContextRelationships

Implements the logic needed by the relationships defined at context level.

Relationship

Models an abstract relationship that contains all the basic behaviours for a relationship. It also defines template methods that must be redefined by the concrete relationships.

Implication

Implements the concrete relationship of *implication*. It inherits from `Relationship` and implements the template methods of this class for the *implication* relationship.

Requirement

Implements the concrete relationship of *requirement*. It inherits from `Relationship` and implements the template methods of this class for the *requirement* relationship.

Suggestion

Implements the concrete relationship of *suggestion*. It inherits from `Relationship` and implements the template methods of this class for the *suggestion* relationship.

7.5.5 Viewer Package

The package in Figure 7.17 contains classes that implement a textual and graphical representation of the application contexts, features and relationships. Those representations are used as a debugging tool for the system.

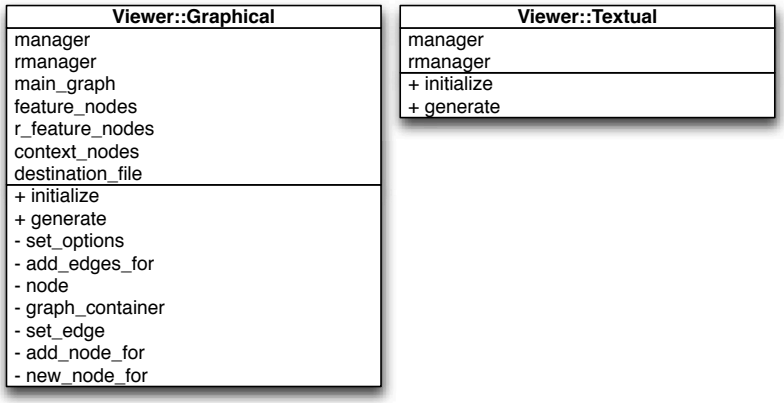


Figure 7.17 – Viewer package.

Viewer::Textual

Implements a textual view of the application's defined features, contexts and relationships. The purpose of this view is mainly to debug the application.

Viewer::Graphical

Same utility as `Viewer::Textual` but in a graphical way. In addition, the active contexts, features and relationships are colour-coded, and the relationships are represented according to their definitions in Section 2.3.5.

7.5.6 Miscellaneous

Dsl

Implements the definition of all the keywords that are provided by the domain-specific language of the framework. The keywords are prefixed by *phen_* in order to avoid names clashes with other gems. This module is included in the Ruby Kernel module at the loading of the framework. Since Kernel is inherited by all objects, the keywords are all available everywhere in the application as if they were actual keywords.

Error

Defines a standard exception for the framework.

7.6 Under the Hood: Core Mechanisms

The aims of this section is to help understand how the entire framework operates, and how the different interaction mechanisms of the building blocks in Section 7.5 are managed. The content is mainly high-level, but also includes implementation notes that can be easily skipped at the reader's convenience. All the concepts described in Section 7.2 will now be explained from an implementation point of view.

7.6.1 Conflict Resolution Policy

A conflict occurs when multiple contexts adapt the same method and are active at the same time. In this case, the system needs to choose the appropriate adaptation to be deployed into this method.

The conflict policy mechanism is used by the activation mechanism explained in Section 7.6.5 to choose a possible adaptation for a method among multiple candidates for deployment in case of conflict. Namely, choose the appropriate adaptation among all adaptations of active contexts for this method. It is also used by the proceed mechanism in Section 7.6.10 to select the adaptation that has to be called by a `proceed` call.

The comparison of the adaptations is carried out by the method `conflict_policy` in the `ConflictPolicies` module, which takes two contexts in conflict for a method as parameters and returns -1 (when the first context has precedence on the second) or 1 (otherwise). In addition, as the framework needs to work out of the box, some built-in ready to use conflict policies are provided.

Policy Definition

To set a new conflict policy for the framework, one only needs to pass an implementation that returns -1 or 1 as a block to the `phen_change_conflict_policy` as illustrated in Figure 7.18.

```
1 phen_change_conflict_policy do |context1,context2|
2  if #context1 has precedence on context2
3   -1
4  elsif  #context2 has precedence on context1
5   1
6  else
7   Raise an error
8  end
9 end
```

Figure 7.18 – Example of setting a new conflict policy by calling the `phen_change_conflict_policy` keyword.

As illustrated in Figure 7.19, the `phen_change_conflict_policy` keyword uses the meta-programming capabilities of Ruby. It will define an instance method using Ruby's `class_eval` and `define_method` that will redefine the original method `conflict_policy`. After the call to `phen_change_conflict_policy`, the framework will use this new conflict policy.

```
1 def change_conflict_policy (&block)
2  self.class.class_eval{define_method(:conflict_policy,&block)}
3 end
```

Figure 7.19 – Implementation of the `phen_change_conflict_policy` keyword.

Provided Policies

Two conflict policies are provided by default: the *no resolution* and the *age resolution*.

The *no resolution policy* gives priority to non-default contexts over the default one. Indeed, if neither context is default, an error is raised. From an implementation point of view, it returns 1 if the first context is the default feature and -1 inversely. Otherwise, if neither is default, an error is raised.

The *age resolution policy* in Figure 7.20 is the default policy of the framework. The *age* of a context is the number of context activations since the last activation of the former context. The policy compares the two contexts and returns -1 if the *age* of the first one is smaller than that of the second, 1 otherwise. Furthermore, the comparison never returns 0, because our implementation prevents having two contexts with the same age.

```
1 def age_conflict_policy(context1, context2)
2  context1.age <=> context2.age
3 end
```

Figure 7.20 – Age resolution conflict policy.

The age conflict policy is defined as: "One can associate time stamps to context values to keep track of the order in which the context evolves. A possible timestamp strategy is to give preference to the most recent context information." [DVC+07]. In our case, the timestamps are based on the activation age as explained above.

Such a policy is interesting because it relies on the natural intuition that contexts activated recently should have precedence on the ones that were activated previously. When a context has been activated recently, its defined behaviour is likely to suit the current situation better than the older one.

7.6.2 Context Definition

The snippet in Figure 7.21 represents an example of context declaration for the context `:Presenter` and the declaration of an adaptation for the method `notify` of the class `TodoList`.

```
1 context :Presenter do
2   adaptations_for TodoList
3   adapt :notify do
4     ... # Behaviour of the adaptation
5   end
6 end
```

Figure 7.21 – Example context definition.

To declare contexts or features we define two keywords, respectively `context` and `feature`. Those keywords are available everywhere in the application because they are added to the Ruby `Kernel` module which is automatically inherited by all objects.

The `context` keyword can be called by passing a name as parameter, for example "Away" or "Away,Network". "Away" represents a simple context and "Away,Network" a combined context. Once the kind of context has been determined, we first have to make sure that it does not already exist before creating it. If it exists, we must use the existing one, if not, we create a new one. Thanks to this language construct, we can reopen a context already declared to add adaptations or relationships everywhere, like open classes in standard Ruby. The next step is the self declaration of the context to the manager. Finally, the block passed to the method `context` is evaluated in the defined context in order to define the adaptations and the relationships.

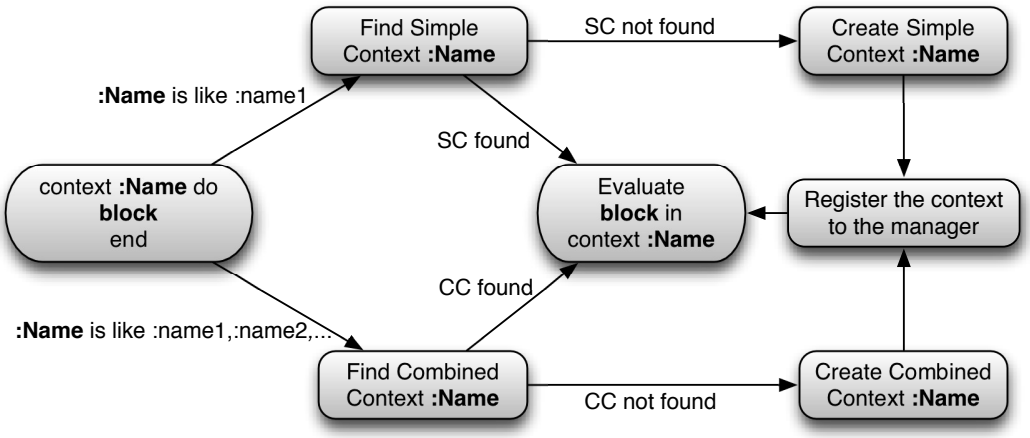


Figure 7.22 – Context declaration process.

7.6.3 Adaptation Definition

The block (lines 2-5 in Figure 7.21) is evaluated in the given context. The two methods `adaptations_for` and `adapt` are two methods defined in the class `Context` and they are the only ones needed to define adaptations.

The method `adaptations_for(Class)` just sets an instance variable inside the evaluating context. As a consequence, there is no need anymore to specify for which class the following adaptations are defined. This method behaves like the `public` and `private` keywords of Ruby: the same class is used by all the following adaptations defined by `adapt` and `adapt_class` until the next call of `adaptations_for`.

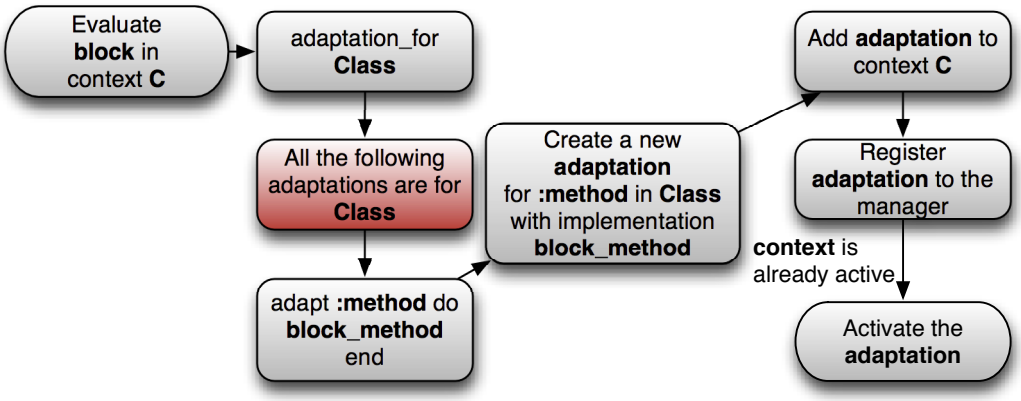


Figure 7.23 – Adaptation declaration process.

The red box in Figure 7.23 is the step that prevents our framework being thread-safe. Otherwise, if multiple threads open the same context at the same time, we cannot determine which class has to be adapted.

In the next step, the method `adapt` triggers the creation of a new adaptation for the method name and block passed in parameter. This freshly created adaptation is added to the context and registered with the manager.

Furthermore, as we are able to reopen contexts and add adaptations everywhere, the context can be active when adaptations are added. In this case, the manager triggers automatically the activation of the adaptation.

7.6.4 Context Activation

The activation of a context leads to some changes in its internal state. The first one is the `activation_age`, which is assigned to the class variable `total_activations` incremented at each activation of any context. The next one is the `activation_count` that is incremented at each activation and decremented at each deactivation. Thanks

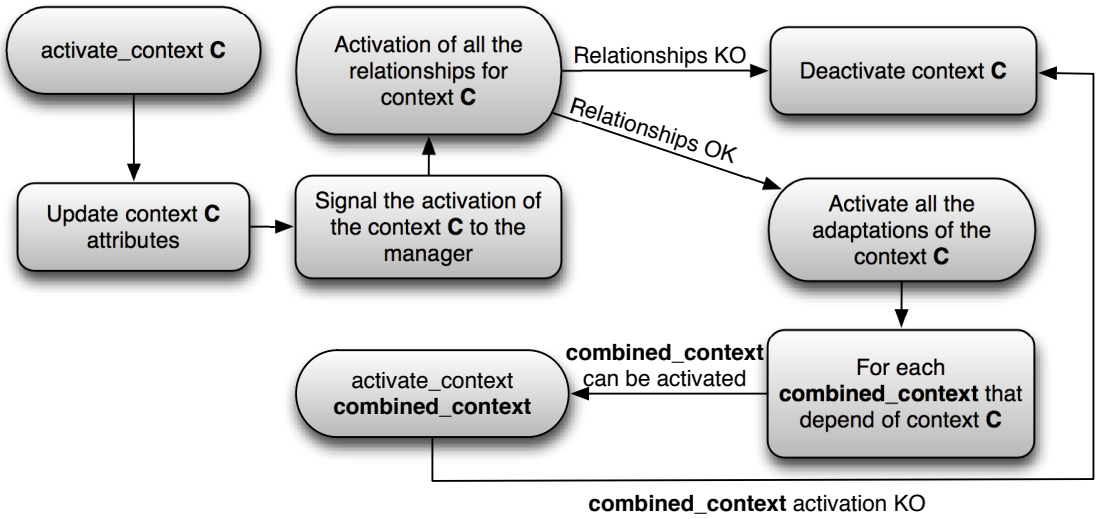


Figure 7.24 – Context activation process.

to this `activation_count` one can detect when a context really needs to be deactivated (`activation_count == 0`). The `activation_age` is used to compute the age of a context that is used by the conflict policy.

Subsequently, the context is registered with the manager. The latter triggers the activation of the the relationships that concern the context. This step is mandatory. If it fails, the entire activation process fails and a rollback is done to clean up the system.

Once the relationships are validated, the manager activates all the adaptations of the context.

If the freshly activated context is a composite of a combined context, and if all the composites are active, the manager tries to activate this combined context. In case of success, the activation process is done. If not, the composite context that has triggered the activation of the combined context is deactivated.

7.6.5 Adaptation Activation

In this section, we will explain *active adaptations* and *deployed adaptations* sets. Both sets play an important role in adaptation activation. The former contains all the adaptations that are actually active in the system, that is all the adaptations of all the active contexts. The latter contains all the active adaptations that are already deployed.

The activation of an adaptation means that it is now candidate to an effective *deployment*. The activated adaptation is added to the *active adaptations* set and because of its presence in this set, it is taken into account when resolving conflicts.

As shown in Figure 7.25, each time an adaptation is activated, a *redeployment* is triggered. This step consists in finding the adaptation that will be effectively *deployed* for the method. To achieve this operation, the conflict policy selects the adaptation that must be deployed for the class and method specified by the original adaptation.

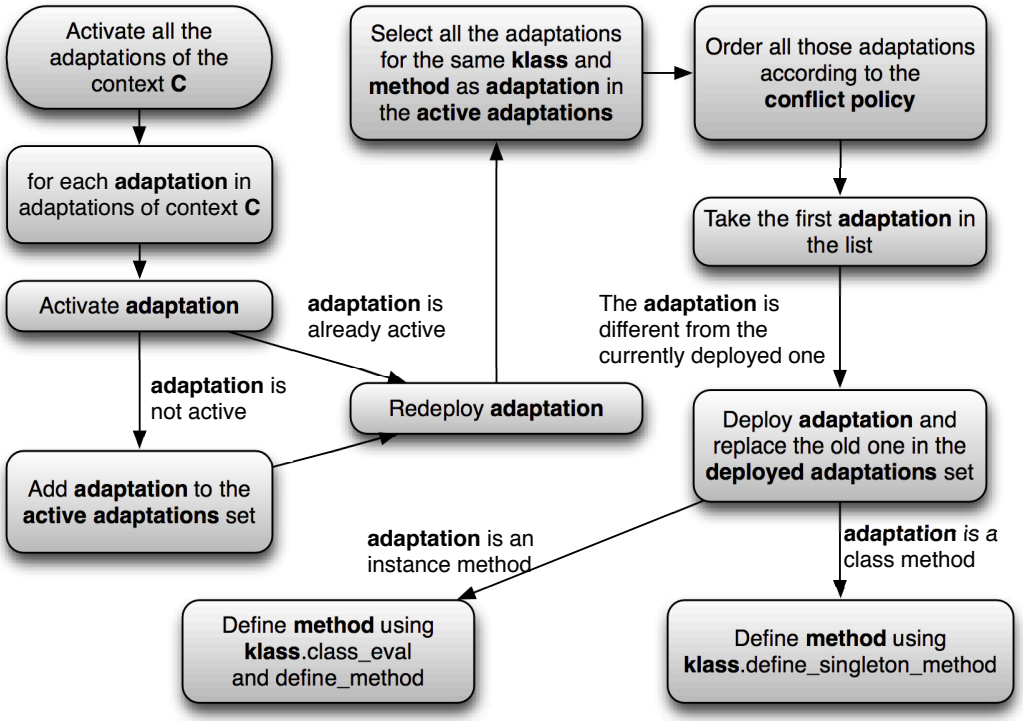


Figure 7.25 – Adaptation activation mechanism.

Finally, the adaptation selected by the conflict policy is now effectively *deployed* thanks to the meta-programming capabilities of Ruby. The mechanism used depends on the method that needs to be adapted, either instance or class. The illustration of this method is shown in Figure 7.26.

```
1 def deploy
2   method_name = self.method_name
3   implementation = self.implementation
4   if instance_adaptation?
5     klass.class_eval { define_method(method_name, implementation) }
6   else
7     klass.define_singleton_method(method_name,implementation)
8   end
9 end
```

Figure 7.26 – `deploy` method in `Adapatation`.

7.6.6 Context Deactivation

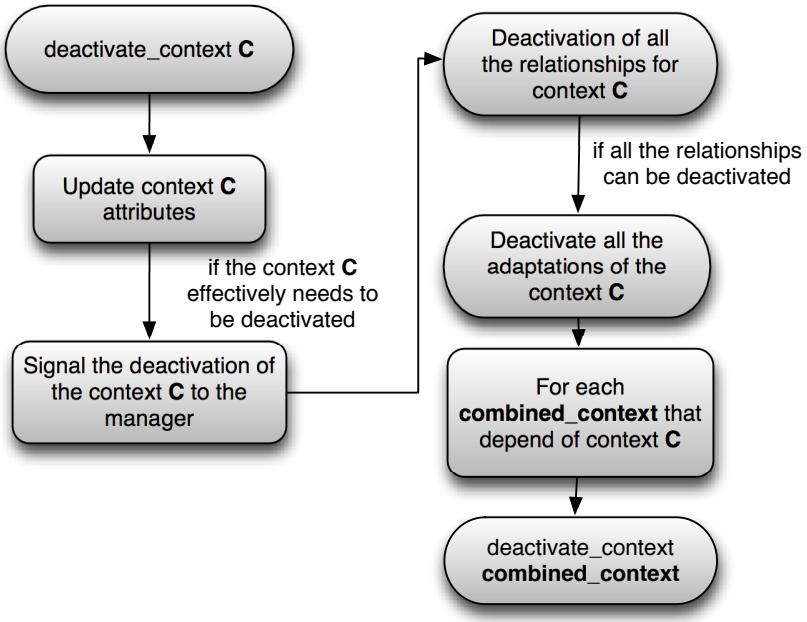


Figure 7.27 – Context deactivation process.

The deactivation process of a context in Figure 7.27 is more simple than the activation. The only point we must be careful about is when the context needs to be effectively deactivated. At each deactivation, the `activation_count` of the context is decremented. Only when the `activation_count` is equal to zero and was greater than 0 before the deactivation, must the context be effectively deactivated.

In addition, because of the relationships, the deactivation can either trigger deactivation of other contexts or be forbidden. The deactivation of other contexts is triggered by relationships like *suggestion* or *implication*. On the other hand, if a context is required by another context, the deactivation of the former is forbidden because of the *requirement* relationship.

7.6.7 Adaptation Deactivation

The Figure 7.28 illustrates the deactivation process of an adaptation. The process differs from the activation in Figure 7.25 up to the "Redeploy adaptation" step, but is the same afterwards.

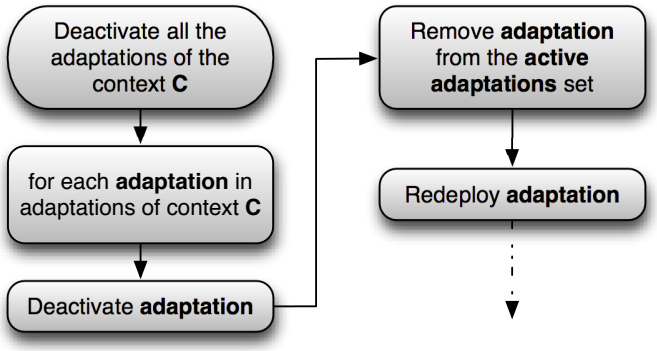


Figure 7.28 – Adaptation deactivation process.

7.6.8 Relationship Definition

There are two ways of defining relationships depending on where the definition is made, either in a context or a feature. In the first case, that is defining a relationship in a context, the relationship is stored in the parent feature, while in the other case, the relationship is stored in the feature itself. The concept of parent feature is defined in Section 7.2.4.

```
1 feature EnvironmentSense do
2   requirements_for :Desktop, :on=>:OperatingSystemSense
3 end
```

Figure 7.29 – Relationship declaration in a feature.

Figure 7.29 illustrates a relationship declaration in a feature. The keywords available to define the relationships are: `suggestions_for`, `requirements_for` and `implications_for`.

In addition, the developer may want to declare all the relationships in a single file. It is possible thanks to the latter keywords. Indeed, as each keyword has a source and target(s), the declaration within a feature is not mandatory and can be moved to a separate file. In this case, all those relationships will be stored in the default feature and are always active.

```
1 context :Android do
2  implies :Mobile
3 end
```

Figure 7.30 – Relationship declaration in a context.

Figure 7.30 illustrates a relationship declaration at context level. The keywords available

to define the relationships at context level are: `suggests`, `requires` and `implies`.

Each context keeps a pointer to the closest parent at declaration time. Thanks to this pointer, the parent feature can be found by running through all the parents until the parent is a feature. When the context has no parent, in this case the parent pointer is then set on the default feature.

7.6.9 Relationship Activation

The relationship activation process of a feature is different from that of a context. Indeed, a feature contains all the relationships of its sub-contexts. When a feature is activated, all those relationships are added to the active relationships of the application.

All the relationship objects of the framework inherit from the `Relationship` class that implements the common behaviour for all relationships. Each relationship has to redefine four methods: `(de)activate_context` and `(de)activate_feature`.

Indeed, as explained in Section 7.2.4, the features also store the relationships of their sub-contexts. As a result we distinguish the activation of features from that of contexts. In the implementation, this distinction is represented by the methods `(de)activate_context` and `(de)activate_feature`. `(de)activate_feature` is used to actually (de)activate the relationship when its containing feature is (de)activated, while `(de)activate_context` is a callback method triggered each time the source or the target context of the relationship is (de)activated.

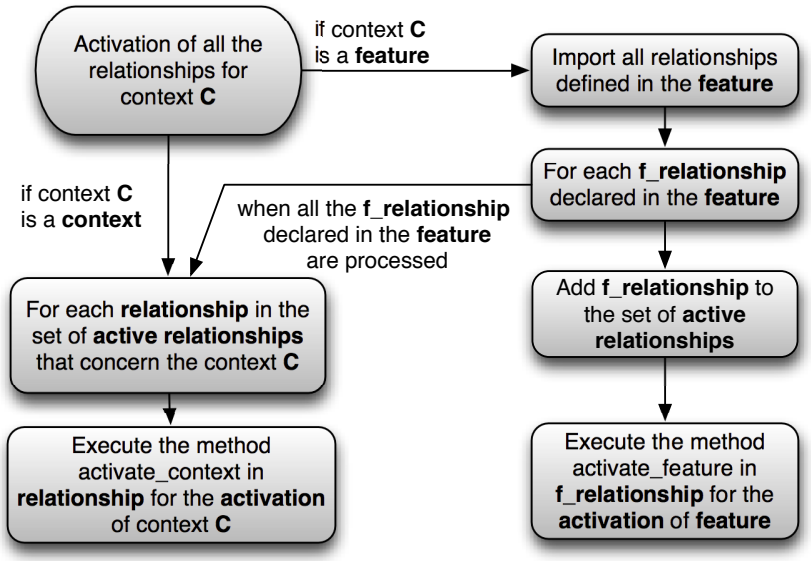


Figure 7.31 – Relationship activation.

As illustrated in Figure 7.31, there is a common behaviour for feature and context: the

execution of `activate_context` for each relationship that concerns the context. In addition, for the feature, the method `activate_feature` is executed for all the relationships declared in the feature.

Relationships deactivation

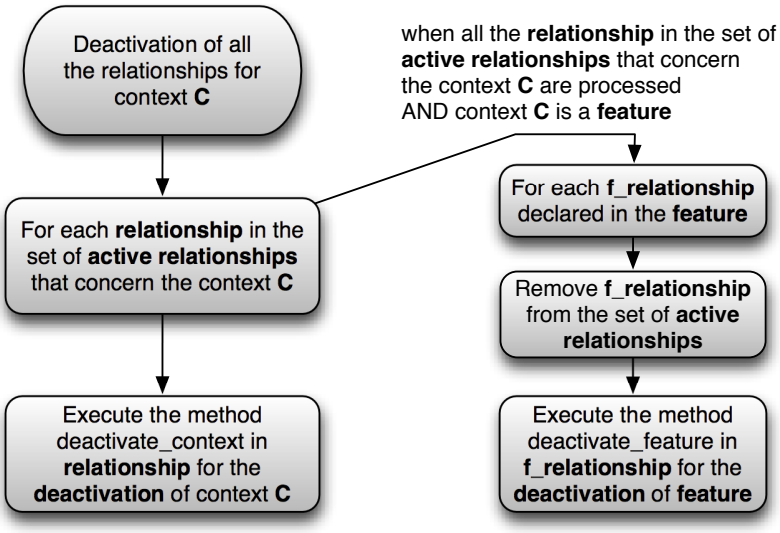


Figure 7.32 – Relationship deactivation.

The relationships deactivation in Figure 7.32 is almost the same as the activation in Figure 7.31.

The first distinction is that the methods `deactivate_context` and `deactivate_feature` are now called in place of, respectively, `activate_context` and `activate_feature`.

The second is that in the case of features, the relationships that are defined in a feature are now removed from the system.

7.6.10 Proceed

All the steps needed for the *proceed mechanism* are explained in the order in which they appear, and are implemented in module `AdaptationManagement`.

Find the Calling Adaptation

Finding the *calling adaptation* (method, class and context name) is very difficult because of the limited *calling stack* reification provided by Ruby in Figure 7.33. To achieve this task, we use a "hack" in order to retrieve it from the file and the line number of the *calling adaptation*.

We extract the file name and the line number of the call from the *calling stack* by using

```
1 example_basic.rb:10:in `block (2 levels) in <main>'
2 example_basic.rb:22:in `<main>'
```

Figure 7.33 – Example of calling stack.

a regular expression. The first line of the *calling stack* contains the line and file where `proceed` is called. Once we have the file name and the line number from the *calling stack*, we are able to retrieve the *calling adaptation*.

Implementation

First, we find among all the *active adaptations*, those that have the same file name as the file name retrieved from the *calling stack* of the `proceed` call.

Then, those adaptations are all sorted in descending order according to their declaration line. As a result, the adaptation that is defined last in the file is now at the top of the sorted list.

Finally, the *calling adaptation* is the first to have its declaration line smaller or equal to the `proceed` call line.

Finding the Adaptation to Call

Once we have the *calling adaptation*, we have to find which adaptation comes just before, according to the conflict policy.

All the *active adaptations* for the method and class of the *calling adaptation* are sorted according to the conflict policy. After that, we find the position of the *calling adaptation* among those and the *adaptation to call* is the first one with less precedence than the *calling adaptation*.

Execution of the Adaptation to Call

Once the adaptation to call is found, we have to execute it. Two criteria affect the execution of the adaptation implementation. The adaptations can adapt either instance or class method. Secondly, depending on how the method is defined, the implementation can be of a different type. Those criteria make it more complex to evaluate the adaptation within a specific class or instance.

Implementation

The implementation can be a `Method`, `UnboundMethod` or a `Proc`. In addition, an adaptation can be applied either to an instance or to a class. The different behaviours are shown in table 7.1.

`UnboundMethod` is an instance method not already linked to a specific instance. Such

methods need to be bound to an instance using `bind(instance)` before they are called.

`Method` is a class method always bound to its origin class.

`Proc` is a block that can be bound to local variables; once bound, the code can be executed everywhere and can still access those variables.

	`target` is the Instance	`target` is the Class
`impl` is a Method or UnboundMethod	`impl.bind(target)` `_.call(*args,&block)`	`impl.call(*args,&block)`
`impl` is a Proc	`target.instance_exec(*args,&impl)`	

Table 7.1 – Binding behaviours.

`target.instance_exec(*args,&impl)` evaluates the block `&impl` in the context of `target` (instance or class) by setting `self` to `target` with arguments `*args`.

`impl.bind(target)` binds the `impl` to a specific `target` instance and returns a `Method` object that can be called.

`method.call(*args,&block)` calls a `Method` object with arguments `*args` and a block `&block`.

7.7 Programming Language Requirements

In this section, we summarize the language constructs needed to implement the concepts of our framework.

First, the COP paradigm is an extension of the Object-Oriented Programming (OOP) paradigm. The behaviour adaptation is performed by modifying methods in classes. Furthermore, to implement the building blocks contexts and features as first-class entities, such paradigm is required.

Secondly, the language must have the following meta-programming capabilities:

- To change instance and class method implementations at run-time in order to deploy the adaptations, Section 7.6.5.
- To execute the adaptation bodies in a specific instance without replacing the deployed method, Section 7.6.10.
- To retrieve a reification of the calling stack which is required to determine the adaptation that performed a `proceed` call, Section 7.6.10.
- To add new methods in built-in language classes which is used to define the DSL as language keywords, Section 7.5.6.

Finally, other constructs are also useful, for example, closure (blocks in Ruby) to store the adaptation implementations, and efficient data structures like hash tables.

7.8 Limitations

Although the implementation is fairly complete and stable, the framework still has some limitations:

Firstly, at the moment, we do not have structural adaptations. The ability to add and remove classes and methods would be really handy when COP and FOP paradigms are merged.

Secondly, adaptations are system wide. This means that it is not possible to activate a context for a specific object instance or a specific thread. Currently, thread support is very limited in the default Matz's Ruby Interpreter (MRI) but in JRuby or Rubinius real multi-threading is possible. Thus, scoped context activation would be very useful in RoR web application because it would allow to serve HyperText Transfer Protocol (HTTP) requests with threads instead of processes and would improve the scalability and portability.

Thirdly, due to limitation of Ruby the proceed mechanism relies on a "hack", meaning that it is not very robust. It does not handle the (unlikely) case in which two adaptations are defined on the same line. Furthermore, the sort needed to match the calling line with its adaptation is not very efficient, as detailed in Chapter 11.

Fourthly, due to the `adaptations_for` implementation, the framework is not thread-safe. However, as explained above, this is not a real problem for the moment because of the poor support of multi-threading in Ruby.

Finally, at this stage, it is not possible to access a class variable within an adaptation. This is due to the scoping of Ruby for `Proc` objects. While `self` is correctly bound to the proper class, `@@` variables are found in the `Object` class. Since class variables are not heavily used in Ruby, this should not be a problem, otherwise, a workaround for the framework user would be to define accessors methods for the class variable and use it instead of the variable itself in the adaptations.

Chapter 8

Phenomenal Rails

Contents

No one can be a rock star without a great scene.

David Heinemeier Hansson

8.1 Introduction

The Phenomenal Gem extends the Ruby programming language with Context-Oriented Programming (COP) and Context as a Feature (CaaF) concepts. We will now go further and integrate them in the Ruby on Rails (RoR) framework, presented in Chapter 5, through the Phenomenal Rails Gem. Nowadays, more and more applications are web applications.

While standard web applications are very popular, the recent development of cloud computing started the era of Software as a Service (SaaS) applications. Contrary to usual web applications, they use a *multi-tenant architecture*. This means that, instead of running a separate dedicated server for each tenant (customers), all tenants share a (virtual) server. This can be achieved in different ways, but most SaaS prefer application-level multi-tenancy where the tenants use the same application instance. This particular architecture has an obvious economical advantage because hardware resources are much better shared and economy of scales start working. In addition to the hardware advantage, the greatest benefit of this approach is that there is only one code base to maintain[CC06].

Thus, the major challenge of SaaS is to maintain this code base well-structured while keeping as much tenant customization as possible. Also, behaviour adaptations fit exactly COP capabilities [TCW+12] and we think it will be the killer application of the Phenomenal Rails Gem presented in this chapter.

8.2 Motivation

A TODO-list on the desktop is good, but a TODO-list online available everywhere even on smartphones and with transparent update for the user is much better. The Phenomenal Gem allowed us to develop the former version but for a RoR application we needed something more to ease the developers' life.

All RoR applications have the same specific structure and special needs that can be handled within the framework. Firstly, web applications are composed of many views which are not Ruby methods and so not adaptable by the Phenomenal Gem. Secondly, Each HyperText Transfer Protocol (HTTP) request is independent from the other and may be served from different server processes. Thus, user contexts have to be maintained in a new way.

The new problems we address here on top of the concepts presented in Chapter 7 lead to the Feature as a Service (FaaS) concept. While a SaaS aims to provide an application on user need basis (e.g. through a monthly fee), we think that we can go further now that we have COP in hand and provide features on the user need basis, leading one day to a new economical model, who knows?

8.3 Concepts

8.3.1 File Structure Integration

As introduced in Chapter 5, the principles of Convention over Configuration (CoC) and Don't Repeat Yourself (DRY) are heavily used in RoR. The first consequence is that this leads to a strict file structure architecture, as illustrated in Figure 8.1. The *app/* root folder contains one folder per Model-View-Controller (MVC) component.

Figure 8.1 – Ruby on Rails file structure.

In order to integrate our features and contexts in this file structure, we had to find a way immediately accepted by RoR developers. After many attempts we ended with the solution illustrated in Figure 8.2. Beside the usual *app/*, we add a *app_phenomenal/* folder, which contains one sub-folder per feature or context. These sub-folders are named using the snake case of the context or feature name that it contains. The feature folder in turn, contains folders for optional sub-features or sub-contexts in addition to the usual MVC and a file describing the context that has the same name as the parent folder. These conventions allow the framework to load features and contexts into the application automatically. Furthermore, it provides better modularity. Indeed, instead of having all your models, controllers and views in the *app/* folder, they are now grouped by features.

8.3.2 Features Activation Conditions

Indeed, in the case of single user desktop applications, a context is (de)activated when a change in the situation occurs. However, in case of web applications, each HTTP request is independent from the other and the same client may be served by a different process during the same session, therefore Phenomenal Rails deactivates all contexts before each request and activates the one needed on a per request basis. This is why Phenomenal Rails Gem adds a new keyword to the Domain Specific Language (DSL) of features definition: `activation_condition`.

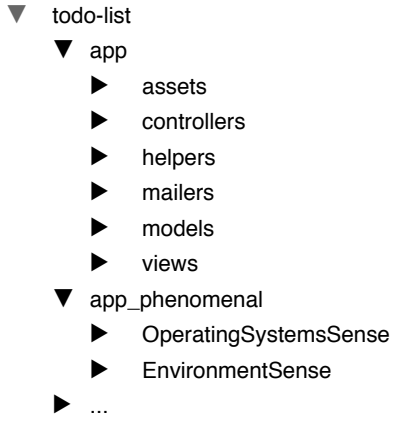


Figure 8.2 – Ruby on Rails file structure with Phenomenal.

Figure 8.3 shows an example of usage for this keyword. Suppose we want to adapt the way TODO-lists are handled to the type of user Operating System (OS) (very useful for mobile OS). An `activation_condition` is a block that will be executed before the request hits the controller of the RoR application.

This block has access to the session and the request details, enabling user specific behaviour in a very handy way. While COP does not provide a silver bullet, there is still a long conditional statement. The clear advantage is that now, this statement is not duplicated any more in multiple places in the code. When this code has been executed, all the behaviour changes needed for a specific OS are deployed.

We put this keyword in `Phenomenal::Feature` and not in `Phenomenal::Context` because a feature is responsible for adding a functionality. In our opinion, it is the responsibility of features to activate their sub-contexts since they are also responsible for the relationships of the latter.

If we look at the general COP architecture depicted in Figure 8.4, we can see that `activation_condition` and thus `Phenomenal::Feature` implements the *Context Discovery* component. This gives us a complete COP implementation for web applications.

8.3.3 Persistent Context

As explained above, contexts have to be activated on a request basis in web applications. The Phenomenal Rails Gem extends the Phenomenal Gem DSL by adding the *is_persistent* keyword as illustrated in Figure 8.5. Putting this keyword in a context (or feature) definition means that this context should remain active for every HTTP request.

When a feature or context is needed for the users this keyword avoids the unneeded activation/deactivation at each request, which improves performance.

```
feature :OperatingSystemsSense do
 activation_condition do |env|
  user_agent = env["HTTP_USER_AGENT"]
  if user_agent[/(Android)/]
   activate_context(:Android)
  elsif user_agent[/(Linux)/]
   activate_context(:Linux)
  elsif user_agent[/(Windows)/]
   activate_context(:Windows)
  elsif user_agent[/(Mac)/]
   activate_context(:Macos)
  end
 end
end
```

Figure 8.3 – Feature activation condition.

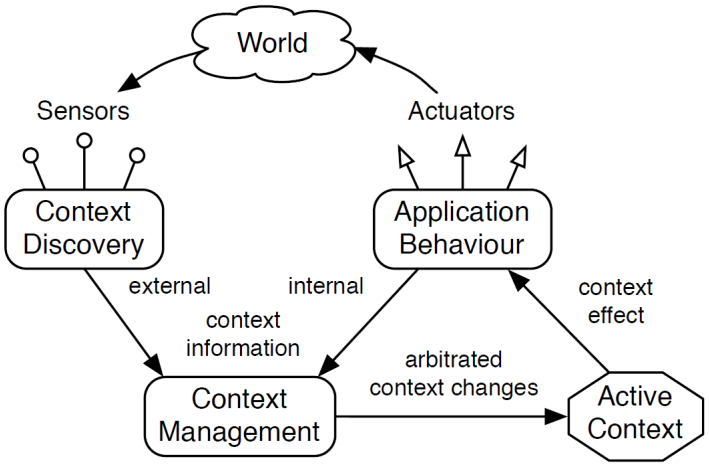


Figure 8.4 – Context aware system architecture.

```
feature :Base do
 is_persistent
end
```

Figure 8.5 – Persistent Context definition.

8.3.4 Views Adaptation

While the Phenomenal Gem is able to adapt any method in any Ruby class which is fine for *Model* and *Controller* components of the RoR MVC, a large part of web applications code resides in the *Views*. Indeed, when web application behaviours are adapted, *Views*

must also be adapted.

As a result, the main addition made by the Phenomenal Rails Gem is to provide a handy way to adapt views in RoR applications. These views are defined in various templating languages such as Embedded Ruby (ERB), HTML Abstraction Markup Language (HAML), etc. Moreover, in addition to adapting other file than Ruby files, we had to find a way to be independent of the language used in order to remain compatible with all RoR applications.

We first considered using *Helpers*, which in RoR are mixins of methods available in all views during rendering. They are normally used to extract Ruby logic from the views. Thus, one solution was to put calls to *Helpers* wherever an adaptation was needed. This way, we could use our Phenomenal Gem without any additions. However this approach had several drawbacks. Firstly, the templating language cannot be used in the helpers body, and plain HyperText Markup Language (HTML) would have to be written in strings. Secondly, this meant that we would have to change the base application structure to be able to adapt it.

Since the first solution was not applicable to real applications, we searched further into the RoR architecture. As described in Section 5.2.2, RoR views are organized in templates (in short, a page) that are themselves recursively based on partials called with the `render` method. This organization is independent of the templating system and well written applications use fine-grained partials to avoid code duplication between templates. The Phenomenal Rails Gem hooks to the RoR architecture to provide a way of overriding any template and partial in any rails application depending of the context.

We saw in Section 8.3.1 that each context or feature folder can have a *views/* entry. Therefore, when a view (template or partial) is present in an active context or feature, this file will be used instead of the one in the *app/views/* folder from the base application. It is possible that several contexts adapt the same view. In this case, the same conflict policy as the one used for method adaptations is used to select the view with the highest priority.

Thanks to this mechanism, views adaptations become really easy. One only need to put a view file in the appropriate context or feature folder and that's it. The view is adapts automatically without having to change the base application code if it is well structured. A small refactoring may still be useful in order to get more fine-grained partials.

An optional parameter added to the `render` method allows to specify a specific context or feature for a partial view. Using this keyword, the partial view is displayed only if the context or feature is active, nothing happens otherwise.

The current limitation of this system is that it does not have a *proceed* mechanism, thus you cannot call behaviour of less priority contexts within a view adaptation.

8.4 Under the Hood: Core Mechanisms

The aims of this section is to help understand how the Phenomenal Rails Gem integrates the Phenomenal Gem framework into RoR. The content is a bit lower level than in Section 7.6 because it mainly concerns the actual integration of the same concepts in the RoR framework.

8.4.1 Engine

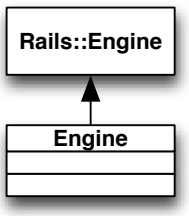


Figure 8.6 – Ruby on Rails Engine.

The Phenomenal Rails Gem is implemented as a RoR Engine[1], meaning that, in addition to be loaded alongside with the RoR application that uses it, it has access to the application configuration, it can add middlewares, routes, controllers, etc.

The `Engine` hooks to the hosting application at start-up and executes the following actions:

- Adds the models,controllers and helpers of features folders to the autoloaded paths of RoR using the `before_configuration` hook[2]. Then uses the `to_prepare` hook to load the root context folder content after all the initializers are run but before eager loading and the middleware stack is built. Finally, adds a callback to the `ActionDispatch::Callbacks.before` hook in order to deactivate all non-persistent contexts before each request, and reloads all contexts files in development mode. The file loading is explained in Section 8.4.2.

- Adds the custom middleware to the application middleware stack, giving an entry point to every request coming in and out of the application, the activation condition is explained in Section 8.4.3.

- Uses the `after_initialize` hook to load our customized view resolver as first resolver. The view adaptation mechanism is explained in Section 8.4.5.

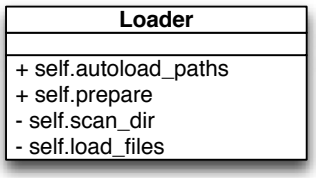


Figure 8.7 – File loader.

8.4.2 File Structure Integration

The loader module, in Figure 8.7, is a set of callbacks methods used by the engine to integrate files from the context root folder into the hosting application.

Firstly, `Loader.autoload_paths` scans the context sub-folders to retrieve controllers, helpers and models folders. Theses folders are added to the auto loaded path of RoR meaning that these files are either reloaded for each request in development mode, or loaded only once in production mode.

Secondly, `Loader.prepare` is called before each HTTP request to handles the context and features files. It deactivates all active contexts to ensure a clean system is left for the new request. Furthermore, in development mode, all context files are reloaded, to avoid restarting the server each time the developer makes a change in the context files. This mimics the usual RoR behaviour for normal classes.

8.4.3 Features Activation Conditions

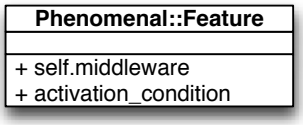


Figure 8.8 – Feature.

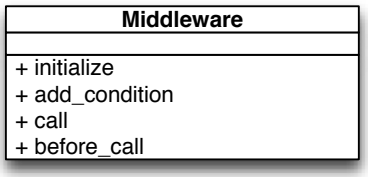


Figure 8.9 – Middleware.

[1] `http://edgeapi.rubyonrails.org/classes/Rails/Engine.html`
[2] `http://guides.rubyonrails.org/configuring.html#initialization-events`

Using the same technique as for contexts extensions, the Phenomenal Rails Gem adds the `activation_condition` instance method to `Phenomenal::Feature`, Figure 8.8. This method takes a block (closure) as an argument that is stored in the `Middleware` of Figure 8.9.

These blocks are then executed for each request with the request environment as parameter. Their execution takes place before the request hits the RoR application and only for active features.

The processing of a request is described in Figure 8.10

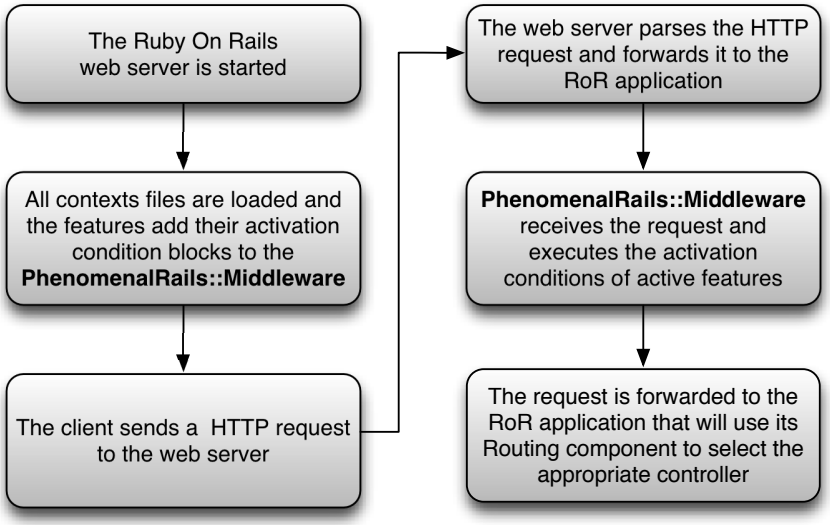


Figure 8.10 – HTTP requests handling.

8.4.4 Context Extensions

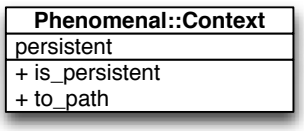


Figure 8.11 – Context.

Thanks to the Ruby open classes [TFH09], it is very easy to open the `Phenomenal::Context` class in the Phenomenal Rails Gem and add new methods, as illustrated in Figure 8.11.

The `is_persistent` instance method in Figure 8.12, is the first one added. A call to this method activates the context and sets the new persistent attribute on true such that `Loader` knows that it does not have to deactivate it between requests.

```
class Phenomenal::Context
 attr_accessor :persistent
 #DSL inside context definition
 def is_persistent
  self.persistent  = true
  activate
 end
end
```

Figure 8.12 – Adding persistence to contexts.

The `to_path` instance method is also added using the same technique. This method uses the context name to find the matching folder in *app_phenomenal/*. It is used by the view adaptations mechanism described in Section 8.4.5.

8.4.5 Views Adaptation

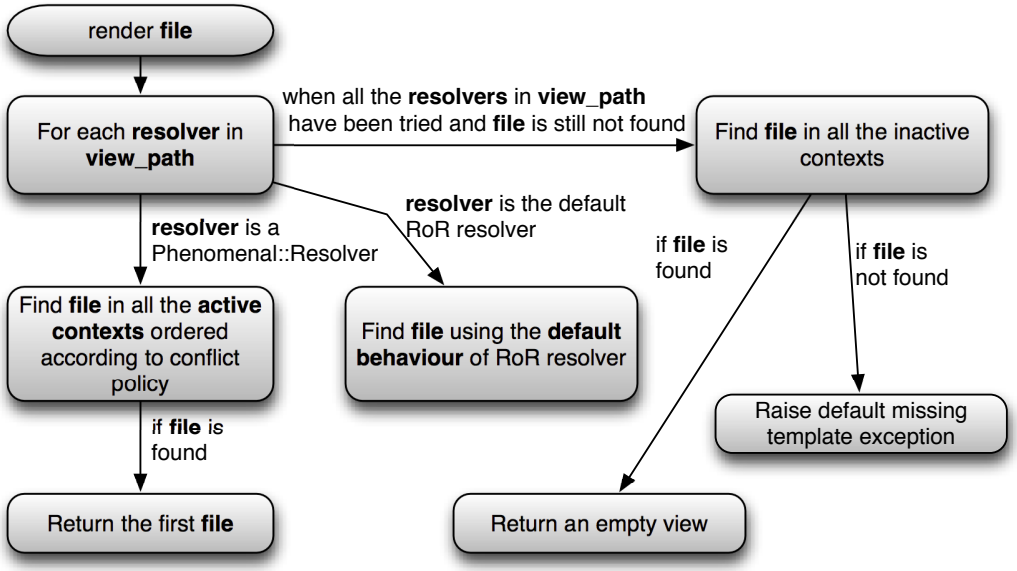


Figure 8.13 – View rendering.

As described in Section 8.3.4 the Phenomenal Rails Gem allows to adapt any template or partial simply by putting a file of the same name in the *views/* folder of the appropriate feature or context.

This is achieved by adding the custom resolver of Figure 8.14. In order to implement this component, we inspired ourselves from the SQLResolver of J. Valim[Val11]. We had to deal with caching and were confronted with other problems not covered by J. Valim, and since the RoR framework implementation is much less documented than the

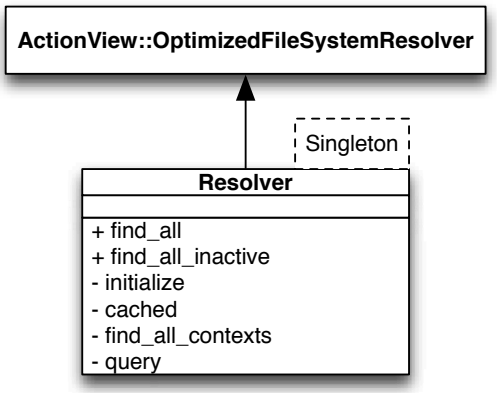


Figure 8.14 – Views Resolver.

framework usage, we also dug heavily in the RoR source code [3].

The view adaptation process is depicted in Figure 8.13. What we do is to subclass the RoR resolver of Section 5.2.2 whose task is to find a view, in order to create a new one that would be able to choose between different versions of a same view implemented in different contexts or features.

This involves overriding not only the `find_all` method to choose the appropriate folder among all the active contexts using the conflict policy, but also the caching mechanism of the parent class. In fact, RoR view caching is foreseen to store only one compiled view per name since there cannot be multiple views with the same name. Thus we add a per context view cache.

Our custom resolver is added to the resolver list (`ActionController::Base.view_path`) at application startup. This means that it executes before the RoR resolver but does not replace it. If a view is not found in any feature, the standard resolver will search in the base application.

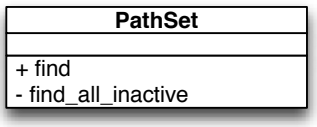


Figure 8.15 – Path Set.

Another problem that we faced was that we wanted the application to raise errors if a view was not defined. However, nothing has to be raised if a view exists only in an inactive context or feature.

[3] `https://github.com/rails/rails`

This was achieved by opening the PathSet class of RoR in Figure 8.15. We also added the `find_all_inactive` method to `Resolver`. Together, they allow to display nothing if a view exists only in an inactive feature or context but still raise an error if the view does not exists, thus avoiding unnoticed typos.

8.5 Limitations

The first limitation comes from the Phenomenal Gem because context activations are not scoped to a specific thread, meaning that our gem cannot be used in a multi-threaded implementation of Ruby, like JRuby, even though these servers are much more scalable. (The activation of a context for one client would modify the behaviour of the applications for other concurrent clients.)

Because we do not currently have a *proceed* mechanism for the views, this could lead to some code duplication in adaptations since we cannot reuse the original behaviour. Implementing this is very challenging because a way is required to efficiently access the Document Object Model (DOM) of the HTML tree.

Currently, all contexts are deactivated before each request. It would be more efficient to deactivate only the ones not needed by the new requests, but to achieve this, it would be mandatory to activate all the contexts only in one place in an atomic action. Otherwise, it would be impossible to know which difference of contexts exists between two requests.

The framework allows to override and define classes like controllers and models into features, but there is no control over the order in which the files are loaded, thus problems will arise if multiple features override the same method. We need true structural adaptations to do this in a clean and secure way.

Chapter 9

Development Approaches

Contents

Give me a lever long enough and a fulcrum on which to place it, and I shall move the world.

Archimedes

9.1 Introduction

For the development of the Phenomenal Gem we have tried to always use the best practices in terms of architecture, testing, etc. This section will present our development process, which was enhanced throughout the academic year, and the support tools we used.

9.2 Pair Programming

As there is always more in two heads than in one, we applied the eXtreme Programming (XP) pair programming principle for the entire framework development, at least until

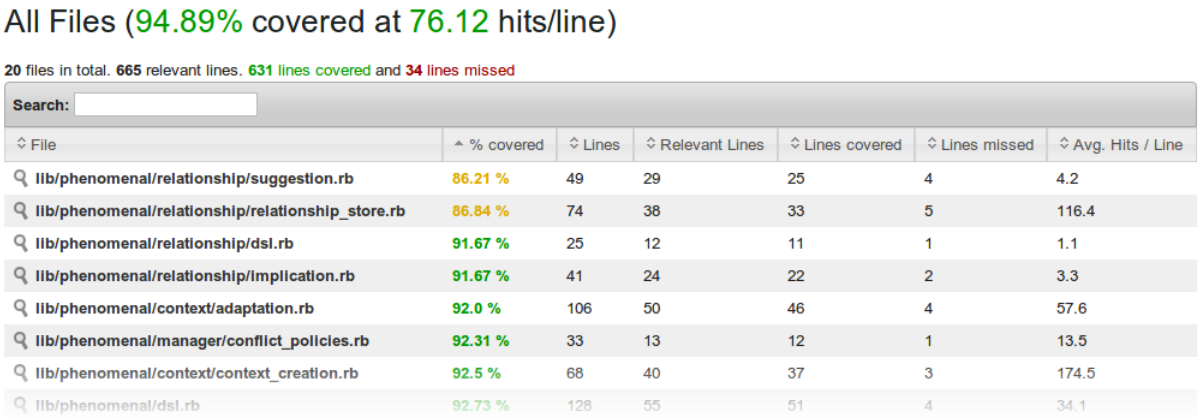
the last stages where all the functionalities were fixed. It may take more time at first but it was important to achieve better code quality so as to reduce global development time.

When real pair programming is not possible for some reason, it is crucial to remain in touch and structure ideas. Campfire[1], which is a team collaboration tool with real time chat, proved most useful in this respect.

9.3 Test-Driven Development

The Ruby community has a great philosophy of Test-Driven Development (TDD). We have always tried to write behavioural tests with the RSpec[2] library before writing actual code. This led to an extended test set that became very useful when we had to refactor the Gem. Writing failing tests, writing code, checking that all tests pass is definitely a must to avoid bugs in a software.

This TDD approach led to a set of 158 tests. We ran the SimpleCov[3] coverage tool on our test suite, and got a result of 94.89%, as presented in Figure 9.1.


All Files (94.89% covered at 76.12 hits/line)

20 files in total. 665 relevant lines. 631 lines covered and 34 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
lib/phenomenal/relationship/suggestion.rb	86.21 %	49	29	25	4	4.2
lib/phenomenal/relationship/relationship_store.rb	86.84 %	74	38	33	5	116.4
lib/phenomenal/relationship/dsl.rb	91.67 %	25	12	11	1	1.1
lib/phenomenal/relationship/implication.rb	91.67 %	41	24	22	2	3.3
lib/phenomenal/context/adaptation.rb	92.0 %	106	50	46	4	57.6
lib/phenomenal/manager/conflict_policies.rb	92.31 %	33	13	12	1	13.5
lib/phenomenal/context/context_creation.rb	92.5 %	68	40	37	3	174.5
lib/phenomenal/dsl.rb	92.73 %	128	55	51	4	34.1

Figure 9.1 – SimpleCov output insight.

We also use Travis-CI[4], a continuous integration server. This service sends an e-mail with the result of the test suite each time a new commit is pushed on the Git repository. That way we never forget to run the tests and are sure that a commit does not break anything.

[1] http://campfirenow.com/
[2] http://rspec.info/
[3] https://github.com/colszowka/simplecov
[4] http://travis-ci.org/#!/phenomenal/phenomenal

9.4 Bad Smells Analysis

In addition to having code exempt from bugs we also aimed to have an as clean as possible code. We use the Code Climate[5] tool to monitor our code quality. This web tool integrates directly into the Git repository and provides code metrics each time a new commit is pushed.

As shown in Figure 9.2 we have only A grades and one B grade. These grades are a measure of code complexity and duplication.

Rating	Class Name	Complexity	M	C/M	Duplication
A	Kernel	2	1	2.2	0
B	Phenomenal::Adaptation	165	10	16.5	0
A	Phenomenal::AdaptationManagement	151	14	10.8	0
A	Phenomenal::ConflictPolicies	30	4	7.4	0
A	Phenomenal::Context	160	21	7.6	0
A	Phenomenal::ContextCreation	64	5	12.8	0

Figure 9.2 – CodeClimate results insight.

9.5 Open Source

Hoping that the produced software would still be used after this thesis, we open sourced our project on GitHub[6]. This approach is a great motivation for us to apply the best practice developed above. The steps we followed to add a new functionality are listed below:

1. Create a branch with the name of the functionality.
2. Create tests and code for this functionality.
3. Initiate a Pull Request on GitHub, and solve the problems notified by other developers through the commit comments.
4. Merge the branch in master, once the functionality is completely implemented and all the tests pass.
5. Iterate for the next functionality.

[5] `https://codeclimate.com/`
[6] `https://github.com/organizations/phenomenal`

9.6 Industrial Minded

The Phenomenal Gem has two main development priorities: to provide a complete Context-Oriented Programming (COP) system and be as developer-friendly as possible. The second point is very important because if we want to bring COP out of the research labs, we have to provide something very clean and easy to use. This is why developing the Domain Specific Language (DSL) associated with the framework is time-consuming, because we always have to trade off functionality against Ruby and Ruby on Rails conventions.

Part III

Validation

Chapter 10

Benubo

Contents

The best frameworks are in my opinion extracted, not envisioned. And the best way to extract is first to actually do.

David Heinemeier Hansson

10.1 Introduction

In this chapter we will explain how to use the Phenomenal Rails Gem through an industrial case-study. Starting from the application developed by our industrial partner, we will analyse its needs in terms of contexts and features. Next, we will illustrate the refactoring process and how the application moved from a Software as a Service (SaaS) to a Feature as a Service (FaaS). In this chapter some general refactoring tips will also be given.

10.2 Motivation

Belighted founded in 2008 at Louvain-La-Neuve by Nicolas Jacobeus specializes in Ruby on Rails (RoR) technologies. Since then, Belighted has developed its capabilities around three main domains: web applications, web site development and RoR expertise. Although still young, Belighted is very ambitious and aims to become a major player in Europe for business web application development with RoR technologies.

Thanks to its office located in Louvain-la-Neuve and its very enthusiastic staff, Belighted collaborates closely with the Université catholique de Louvain (UCL). Indeed, Belighted offered us the opportunity to use as validation application their brand new web application Benubo[1]. They also provided us feedback about our Phenomenal Gem throughout its development. Moreover, their experience in the industry and their feedback helped us to develop our framework in order to make it developer-friendly.

Benubo is a SaaS Enterprise Resource Planning (ERP) built to manage Small and Medium Enterprise (SME) activities in a single tool. This tool embeds several functionalities represented in Figure 10.1:

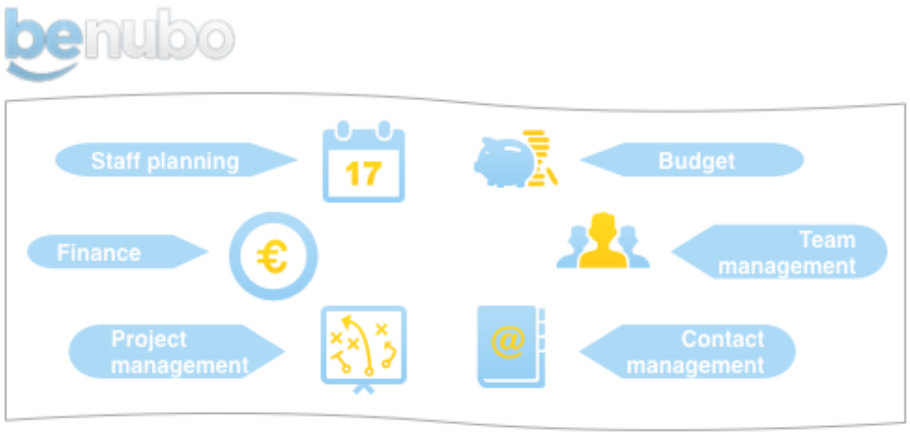


Figure 10.1 – Benubo functionalities.

[1]http://benubo.com/ , the application code cannot be shown entirely because it belongs to Belighted.

- *Staff planning* : allocates the resources on projects.
- *Project management* : follows the progress of the projects.
- *Invoicing* : sends invoices and acknowledge payments.
- *Team management* : monitors team's activity.
- *Contact management* : manages customers, suppliers and partners.
- *Budget* : checks profitability and sales progress.

10.3 Analysis

10.3.1 Foreword

Benubo is built as a monolithic application, which means that the functionalities are not modularized. In fact, RoR enforces the use of a single Model-View-Controller (MVC) pattern, and thus does not provide a straightforward way to modularize those functionalities. This choice is fine for small applications but becomes increasingly poor as they grow.

Therefore, with such a design, functionalities are much harder to maintain because there exist no boundaries among them. To address this problem we identify the different functionalities in Section 10.3.2. Once these are identified, they can then be modularized easily in features with the Phenomenal Rails Gem as explain in Section 10.4.

The Phenomenal Rails Gem refactoring then allows to tackle a common challenge of today's SaaS which is the run-time user customization, explained in Section 10.5. One solution to this challenge is the use of scattered conditional statements across the *views*, *models* and *controllers* of the application. Another one is to maintain one code base per user type, for example per company, using a Software Product Line (SPL). These two solutions, however, make it difficult to maintain the application.

10.3.2 Requirements

Feature Analysis

In order to identify the features we refine the application through generalization (see Section 3.3.2). This ends with the division of the application into a base application and several optional features, illustrated in Figure 10.2. The base application is able to run by itself and contains only the functionalities common to all users of the application.

Optional features are independent and can be activated according to customer needs or their financial capabilities. Furthermore, thanks to the dynamic adaptation approach of the Phenomenal Rails Gem, these conditional activations can be performed at run-time for each user.

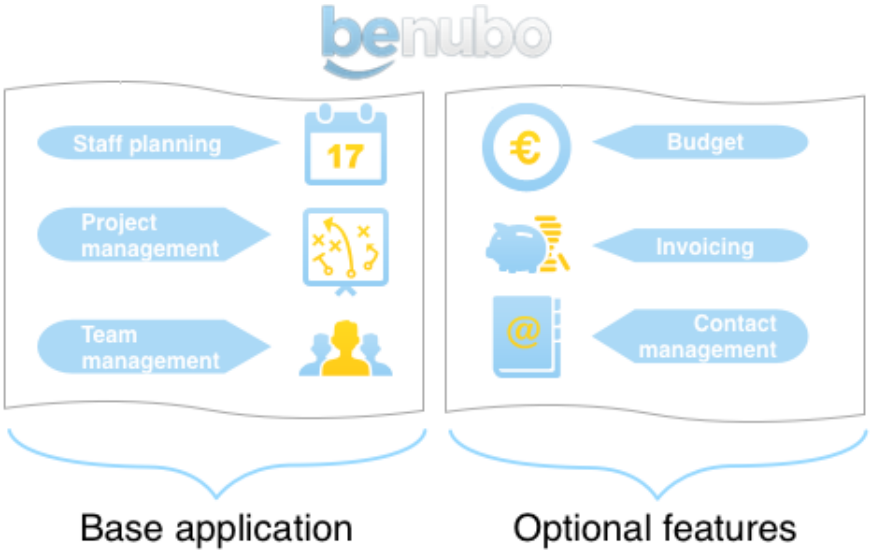


Figure 10.2 – Benubo with extracted Invoice, Budget and Contact features.

> **Guideline**
>
> Identify all the features from the application using the generalization mechanism. Wich functionalities can be removed from the application in order to make the base application less specific?

Feature Interaction Analysis

We have to be careful about possible interactions among features. Those kind of interactions are not an issue for the initial application because all the features are always active. However, as all the features have to be able to run either independently or in synergy, interactions need to be managed.

An illustration of this is the interaction between the *invoice feature* and the *contact feature*. Basically, the *invoice feature* uses the *contact feature* to retrieve the contact to which the invoice is addressed. However, as the former has to work without the latter, the contact selection must be replaced by a basic independent behaviour. In this case, the basic behaviour is a simple text field used to enter manually the complete address of the contact.

> **Guideline**
>
> Identify all the feature interactions from the application. Which functionalities need another one to do its job? What are the emerging behaviours resulting from the combination of features? What features are mutually exclusive?

Context Analysis

While the features aim to reify functionalities in applications, contexts on their side aim to adapt the behaviour of these features without adding new functionalities.

An example of Benubo' need for contexts is to have a context for each country. Indeed, the *invoice feature* has several behaviours that have to be modified depending on the country. For example, for the computation of the Value Added Tax (VAT), all the country contexts are combined with the *invoice feature*. A counter example is the language of the application. Since the main goal of Context-Oriented Programming (COP) is behavioural adaptation, the internationalization provided by RoR is a better solution.

Another example is the *trial context* that adapts features in order to limit their capabilities until users buy the application. Each feature can be combined with this *trial context* to define what its limited capabilities are.

> **Guideline**
>
> Identify all the contexts from the application. What functionalities need some changes of behaviour because of change in the environment such as location, network availability, etc ?

10.4 Refactoring

In this section, we present the actual refactoring of Benubo on the basis of the analysis made in Section 10.3. As mentioned in Section 10.3.1 the RoR architecture is based on a single MVC for the entire application. Thanks to the Phenomenal Rails Gem, each feature is organized in its own MVC.

Concretely, this refactoring implies moving from the file structure depicted in Figure 10.3 to the one in Figure 10.4. For clarity reasons, some folders, not relevant to the implementation, are hidden from theses figures.

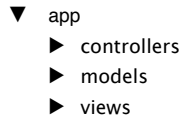


Figure 10.3 – Benubo file structure.

As a result, all the features are now in the *app_phenomenal/* folder and are represented by a folder with the feature's name. In addition, since the main goal of this section is to modularize the original application and keep maintain its behaviour, all features are persistent, meaning that they are always active for all users.

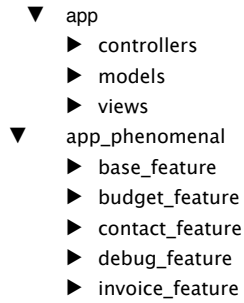


Figure 10.4 – Refactored Benubo file structure.

> **Guideline**
>
> A good convention to adopt is to define features or contexts into a Ruby file with the same name as the containing folder.

10.4.1 Budget Feature

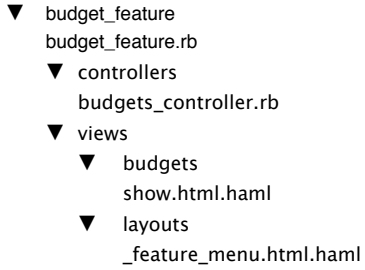


Figure 10.5 – *budget feature* files.

```
1 feature :BudgetFeature do
2  is_persistent
3 end
```

Figure 10.6 – `budget_feature.rb`

The *budget feature* folder is shown in Figure 10.5. The main file `budget_feature.rb` shown in Figure 10.6 defines the persistent feature in the system.

The *controllers* and *views* folders contain the original files moved from the base application MVC.

In addition to the original files, `_feature_menu.html.haml` shown in Figure 10.7, is a new partial view that contains the budget menu entry. As a result, the entire code concerning the feature is only in its own folder. The menu view of the application is

composed of the base menu and the features menu as shown in Figures 10.8 and 10.9. In Figure 10.9 the parameter `:feature` is used to enforce rendering of the file of the specified feature.

```
1 %li{:class => is_selected('budgets')}
2  =link_to t('menu.budget'), company_budget_path(current_user.company)
```

Figure 10.7 – `_feature_menu.html.haml` in *budget feature*.

```
1 =render "layouts/menu_logged_base"
2 =render "layouts/features_menu"
```

Figure 10.8 – `_menu_logged.html.haml` in the base application.

```
1 =render :partial=>"layouts/feature_menu",:feature=>:ContactFeature
2 =render :partial=>"layouts/feature_menu",:feature=>:BudgetFeature
3 =render :partial=>"layouts/feature_menu",:feature=>:InvoiceFeature
```

Figure 10.9 – `_features_menu.html.haml` in the base application.

10.4.2 Contact Feature

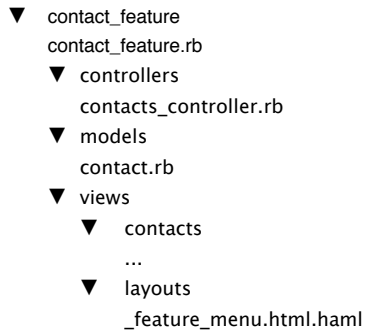


Figure 10.10 – *contact feature* files.

The *contact feature* folder is shown in Figure 10.10. The main file `contact_feature.rb` shown in Figure 10.11 defines the persistent feature in the system.

The *controllers*, *models* and *views* folders contain the original files moved from the base application MVC.

In addition to the original files, the purpose of `_feature_menu.html.haml` shown in Figure 10.12 is the same as the one defined for *budget feature*.

```
feature :ContactFeature do
 is_persistent
end
```

Figure 10.11 – `contact_feature.rb`

```
%li{:class => is_selected('contacts')}
 =link_to t('menu.contacts'), company_contacts_path(current_user.company)
```

Figure 10.12 – `_feature_menu.html.haml` in *contact feature*.

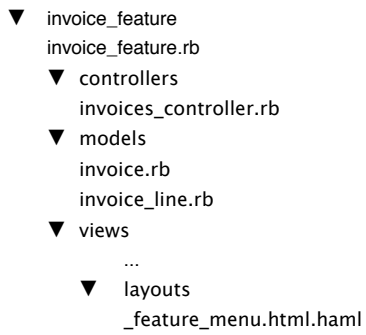


Figure 10.13 – *invoice feature* files.

10.4.3 Invoice Feature

The *invoice feature* folder is shown in Figure 10.13. The main file `invoice_feature.rb` shown in Figure 10.14 defines the persistent feature in the system.

The *controllers*, *models* and *views* folders contain the original files moved from the base application MVC.

In addition to the original files, the purpose of `_feature_menu.html.haml` shown in Figure 10.15 is the same than the one defined for *invoice feature*.

```
feature :InvoiceFeature do
 is_persistent
end
```

Figure 10.14 – `invoice_feature.rb`

```
%li{:class => is_selected('invoices')}
 = link_to t('menu.invoices'), company_invoices_path(current_user.company)
```

Figure 10.15 – `_feature_menu.html.haml` in *invoice feature*.

Guideline

A requirement to facilitate the usage of the framework is to avoid having big and monolithic views. Thanks to the partial views of RoR, separating views into several partial views is really easy. Once the applications views are well separated, using our *views adaptation* mechanism explained in Section 8.3.4 is very efficient.

10.5 Feature as a Service

After the refactoring of Section 10.4 we are now ready to exploit the full potential of our Phenomenal Rails Gem. On the basis of the traditional SaaS, we add in this section the capability to activate features per user at run-time, which is what we call FaaS.

Thanks to this dynamic adaptation of behaviour, we add interesting behaviour like the *trial context* and the *debug feature*. The former is combined with the features to limit their capabilities until the user has paid for the application. The latter aims to help the developer by adding debugging information to the application.

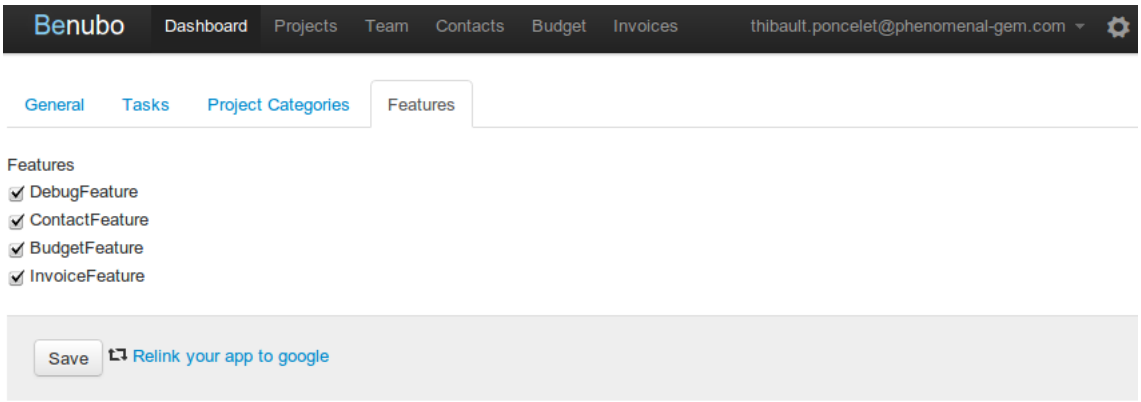


Figure 10.16 – Features selection with all features activated.

10.5.1 Base Feature

This persistent feature embeds all the logic needed to activate the features depending on the user. Its file structure is depicted in Figure 10.18.

The file `base_feature.rb` shown in Figure 10.19 contains the `activation_condition` block that adds behaviour to the middleware in order to retrieve the active features for the user's company and activate them. Furthermore, it also adapts two methods of `CompaniesController` to be able to save the selected features for each company. To keep the example simple, active features for each company are stored in the database using Comma-Separated Values (CSV).

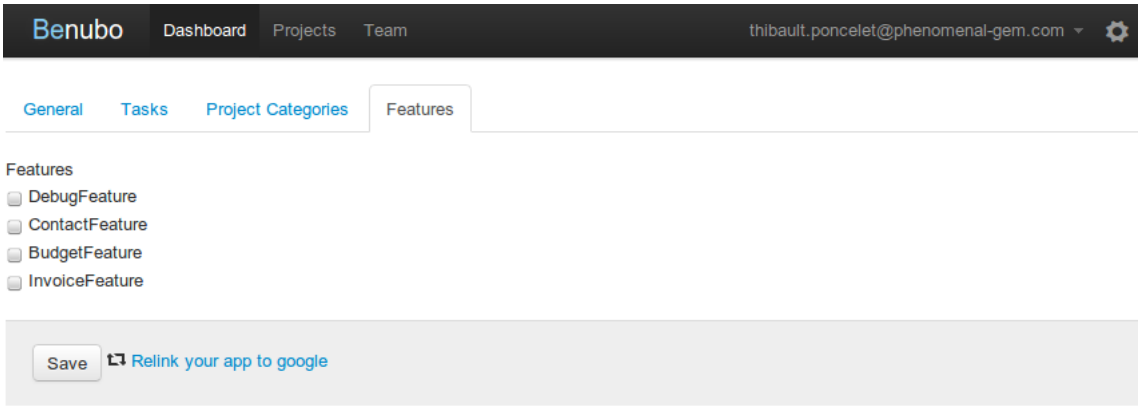


Figure 10.17 – Features selection with all features deactivated. All the features are hidden in the menu.

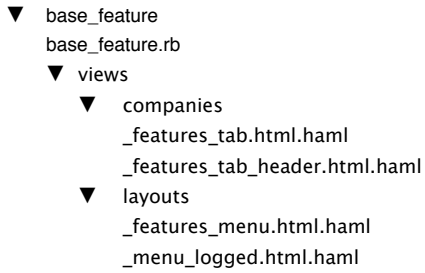


Figure 10.18 – *base feature* files.

This feature also adapts the menu view to avoid having code concerning features in the base application. As shown in Figure 10.20, the base application does not contain references to the features any more. Thanks to the view adaptation mechanism, the file `menu_logged.html.haml` shown in Figure 10.21 of the *base feature* is rendered in place of the one of the base application. The file `_features_menu.html.haml` from Figure 10.9 is now moved from the base application to the *base features* views.

Now that the *base feature* activates dynamically the other features, the `is_persistent` line is removed from *contact feature*, *invoice feature* and *budget feature*.

10.5.2 Invoice and Contact Features Interactions

Using combined features, we can add specific behaviour when both *invoice feature* and *contact feature* are active at the same time. The file structure used to achieve this is shown in Figure 10.22.

When *invoice feature* alone is active, the invoice contact has to be manually entered each time through two text form fields, name and address, as shown in Figure 10.23. Once

```
feature :BaseFeature do
 is_persistent
 activation_condition do |env|
  if env['HTTP_ACCEPT'].to_s.include?("text/html")
   user_id = env["rack.session"]["warden.user.user.key"].try(:at,1).try(:at,0)
   if user_id
    company=User.find(user_id).company
    company.features.try(:split,";").try(:each) do |f|
     activate_context f.to_sym
    end
   end
  end
 end
 adaptations_for CompaniesController
  adapt :edit do
  proceed
  @features = @company.features.try(:split,";")||""
 end
 adapt :update do
  params[:company][:features]=params[:company][:features].try(:join,";")
  proceed
 end
end
```

Figure 10.19 – `base_feature.rb`

```
=render "layouts/menu_logged_base"
```

Figure 10.20 – `_menu_logged.html.haml` in the base application.

```
=render "layouts/menu_logged_base"
=render "layouts/features_menu"
```

Figure 10.21 – `_menu_logged.html.haml` in *base feature*.

contact feature is also active, we use it to specify the invoice contact through a single select form field, as shown in Figure 10.24. The latter file overrides the former when both features are active.

In order to validate form entries, RoR uses validations on models. The method `validates_contact` of Figure 10.25 illustrates such validation. Since the combined feature does not use the same fields, both validation and the method used to represent the contact have to be adapted. Figure 10.26 shows theses adaptations for the `Invoice` model.

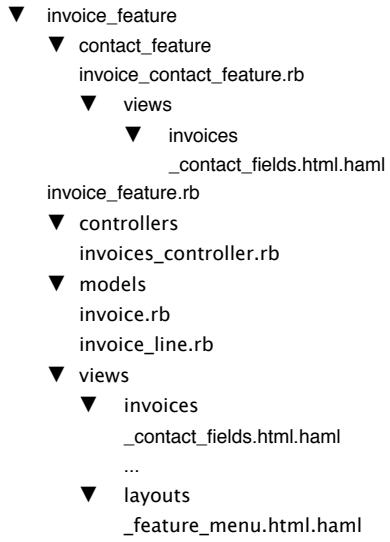


Figure 10.22 – *[invoice feature, contact features]* files.

```
1 =f.text_field :contact_name
2 =f.text_area :contact_address , :rows=>3
```

Figure 10.23 – `_contact_fields.html.haml` in *invoice feature*.

```
1 =f.select :contact_id,
2  options_from_collection_for_select(@invoice.company.contacts,
3   :id,
4   :name,
5   @invoice.contact_id),
6  :prompt=>true
```

Figure 10.24 – `_contact_fields.html.haml` in *[invoice feature, contact feature]*.

Guideline

Structure the *phenomenal_app* folder in a consistent way. First, each context has its own folder of the same name. Secondly, in this folder, a file with the contexts name defines this context in the system. Thirdly, features files are organized like in RoR, inside *models*, *views* and *controllers* folders. Finally, in case of context combination, the context used by another one is defined in a sub-folder of the latter.

10.5.3 Trial Context

As an example for the *trial context*, Figure 10.28 shows how the capabilities of the *contact feature* can be easily limited. Therefore, the `ContactController` is adapted to avoid

```ruby
class Invoice < ActiveRecord::Base
 ...
 def validates_contact
  errors.add(:contact_name, t(errors.messages.blank)) if contact_name.blank?
  errors.add(:contact_address, t(errors.messages.blank)) if contact_address.blank?
 end
 def contact_representation
  "#{contact_name} <br /> #{contact_address}"
 end
end
```

Figure 10.25 – `invoice.rb` model in *invoice feature*.

```ruby
feature(:InvoiceFeature,:ContactFeature) do
 adaptations_for Invoice
 adapt :validates_contact do
  errors.add(:contact, t(errors.messages.empty)) if contact.nil?
 end
 adapt :contact_representation do
  if contact.nil?
   # Call the original behaviour when the invoice
   # was first created without the contact feature
   proceed
  else
   "#{contact.name} <br /> #{contact.formatted_address} <br /> #{contact.email}"
  end
 end
end
```

Figure 10.26 – `invoice_contact_feature.rb`

addition and edition of contacts. The file structure used for this context is shown in Figure 10.27

10.5.4 Debug Feature

Because it is useful to know which contexts are active at a specific time. We introduce the *debug feature*. When this feature is active, the graphical view illustrated in Section 7.3.1 is generated and displayed at the bottom of the application.

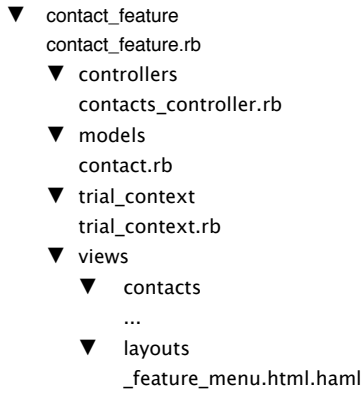


Figure 10.27 – *contact feature* files with *trial context*.

```
feature :ContactFeature do
 context :TrialContext do
  trial_behaviour = Proc.new do
   flash[:error] = "This action is not avalaible in trial mode"
   redirect_to company_contacts_path(current_user.company)
  end

  adaptations_for ContactsController
  adapt :new,     &trial_behaviour
  adapt :edit,    &trial_behaviour
  adapt :create,  &trial_behaviour
  adapt :update,  &trial_behaviour
 end
end
```

Figure 10.28 – `invoice_contact_feature.rb`

Figure 10.29 – *debug feature* files.

10.6 Feedback from Belighted

The aim of Benubo is to provide small companies with a complete toolset to manage all their internal business processes online. We are convinced that not all companies will want to use every feature of Benubo, and therefore the possibility to select desired features on a per-client basis is essential to us.

We had the opportunity to test the Phenomenal Gem, and discuss it with its creators, while developing the prototype version of Benubo. We found it to be an interesting, innovative approach to the aforementioned problem, which is typical of SaaS tools. We are used to service-oriented architectures, which help decompose an application into multiple services, each responsible for a feature. However, the COP approach promoted by the Phenomenal Gem has the advantage of allowing us to keep the application in a single code base and run-time environment, while still being able to separate logically the various functionalities and activate them on demand.

Some validation is still required, as the solution is rather young and we have yet to test its maintainability in a real-world production environment. However, we are convinced of its potential, and would be interested in seeing the Phenomenal Gem promoted in open-source communities.

Nicolas Jacobeus

10.7 Conclusion

As shown in Sections 10.4 and 10.5, the refactoring of an application with the Phenomenal Rails Gem is very straightforward. Once the refactoring is done, the implementation of the application is better organized and can easily be turned into a FaaS application.

One reason why the refactoring is so easy is because we keep all the RoR basic. The MVC architecture is also present in feature folders. The extension of the lazy loading and of the rendering mechanism explained in Chapter 8, eases the modularization without requiring any change to the base application code, whereas before the refactoring, it was difficult to see where feature-specific files were located. With the new feature file structure in *app_phenomenal*, mapping between features and their files is now obvious.

The main contribution of the Phenomenal Rails Gem to Benubo is its ability to activate features dynamically depending on the user. This means that a single application instance is able to serve many users with different needs, and still remain very easy to maintain thanks to the improved modularization.

Chapter 11

Benchmarks

Contents

There are lies, damned lies, and statistics.

Benjamin Disraeli

11.1 Introduction

In this chapter, we will assess the cost of using the framework in terms of performance. We will compare the efficiency of a simple application implemented in three ways. The first uses a Context-Oriented Programming (COP) approach through Phenomenal Gem, the second is based on conditional statements and the third uses the Strategy Pattern. Chapter 10 compared Benubo in terms of modularity with and without the Phenomenal Rails Gem. We will reuse it to compare the performance of the original, the refactored and the Feature as a Service (FaaS) implementations.

11.2 Phenomenal

Since COP provides a new way of adapting software behaviour, two common alternatives are used to compare our framework performance. The straightforward way of achieving

behaviour adaptations in current programming languages is with hard-coded conditional statements. However, these statements induce tangled and scattered code which is very hard to change afterwards. A common Object-Oriented (OO) pattern to avoid this, is by using the Strategy Pattern [Gam95] which allows to change an algorithm at run-time. The problem is that it creates an infrastructural burden and the adaptations points have to be foreseen. The question is, whether the cost in terms of performance is lower than the code quality gain.

```
1  class Foo
2   def meth(mode)
3    if mode==0
4     0
5    elsif mode==1
6     1
7    ...
8    elsif mode==K
9     K
10   end
11  end
12 end
```

Figure 11.1 – Conditional statements implementation.

In order to answer this, we use the same benchmark as in Subjective-C [GCM+11], by measuring the difference in execution time between equivalent applications operating in several different operation modes. The latter represents different behaviours of the application which are simply a different integer returned for each mode.

In addition to the hard-coded conditional statements (Figure 11.1) and the Phenomenal Gem (Figure 11.2) version, we added one that uses the Strategy pattern (Figure 11.3). With the Phenomenal Gem approach, operation modes are represented by different contexts adapting the same method. The second version uses one method that contains one big conditional statement (with a parameter) and finally, the Strategy Pattern version uses one Strategy class per operation mode. The test method `meth` simply returns an integer, whose cost is negligible, meaning that the main cost is the evaluation of the conditions, the activations of the contexts or the additional method call for the Strategy Pattern.

The applications have $K+1$ mode of operations (the default one plus K adaptations). For sufficiently large values of K, the cost of testing the branches becomes considerable. With Phenomenal Gem there is no additional cost once the adapted method is deployed and with the Strategy Pattern a small cost occurs at strategy switching, as well as another small cost for each call due to the added level of indirection.

To measure the difference between the three versions, we call the `meth` method M times

```
class Foo
 def meth
  0
 end
end
context :1 do
 adaptations_for :Foo
 adapt :meth do
  1
 end
end
...
context :K do
 adaptations_for :Foo
 adapt :meth do
  K
 end
end
```

Figure 11.2 – Phenomenal implementation.

```
class Foo
 attr_accessor :strategy
 def meth
  strategy.recieve
 end
end
class Strategy1
 def recieve
  1
 end
end
...
class StrategyK
 def recieve
  K
 end
end
```

Figure 11.3 – Strategy pattern implementation.

for every change of operation mode. The graphical result of the comparison between the three approaches is shown in Figure 11.4, for K=50 and for 50 changes of operation

mode.[1] In the case illustrated in Figure 11.4, all the versions grow linearly with the number of method call per operation modes but the contexts and strategies have a much less slope than the conditional statements version. After 250 method calls per operation mode, the COP approach is more efficient than the conditional one.

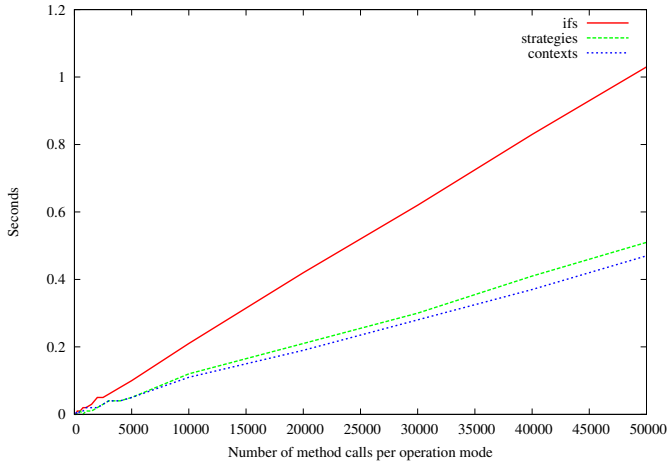


Figure 11.4 – Evolution of execution time with respect to the number of method calls per operation mode (M). With a fixed K=50 and 50 changes of operation modes.

This result is very interesting because it means that, in addition to providing a cleaner code, Phenomenal Gem is more efficient than the old conditional statements solution and while providing adaptations capabilities that have not to be foreseen, it achieves performance similar to the Strategy Pattern for a large number of operation modes.

Figure 11.5 illustrate how the execution time evolves while increasing the number of operation modes (K), for a fixed number of method call per operation mode (M=2000) and a number of changes of operation modes per K values of 500. It can be seen that the Phenomenal Gem and Strategy approaches remain stable while the solution with conditional statements grows linearly.

After these comparison tests, we conducted some more specific benchmarks of the Phenomenal Gem. Firstly, Figure 11.6 shows the context activation time with respect to the number of adaptations that it contains. It can be seen that this grows exponentially because each adaptation activation uses a sort for the conflict resolution that is more than linear.

Finally, Figure 11.7 shows the execution time of a method call that uses the `proceed` method. The main cost is due to the binding of adaptations to the current instance. In fact, another test showed us that the `BasicObject#instance_exec` Ruby method used to bind adaptations implementations dynamically is about 3 times as slow as a normal

[1]The tests are run on an Intel Core 2 Duo T9400 @ 2.53Ghz computer with Ubuntu 11.04 on the Matz's Ruby Interpreter (MRI) version 1.9.2p180.

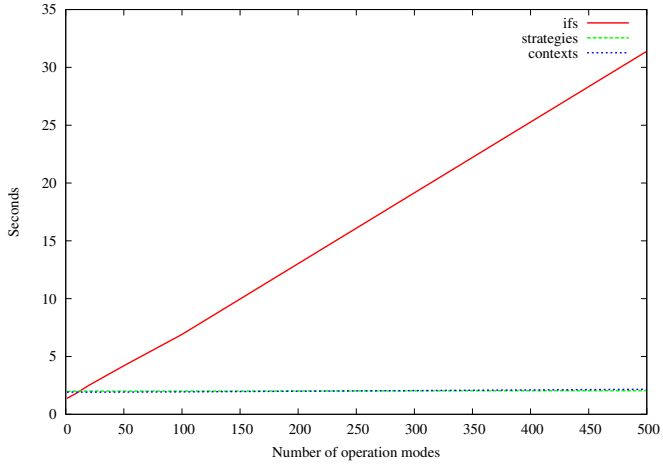


Figure 11.5 – Evolution of execution time with respect to the number of operations modes (K). With a fixed M=2000 and 500 changes of operations modes.

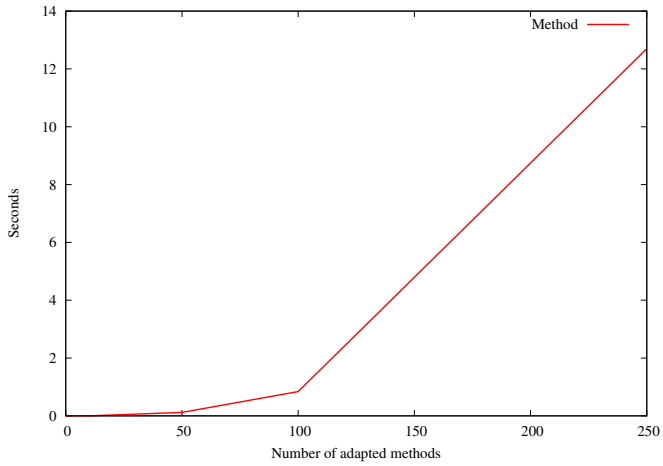


Figure 11.6 – Evolution of one context activation time with respect to the number of adapted methods it contains.

method call. Furthermore, the combination of `UnboundMethod#bind` and `Method#call` used to restore temporarily the default behaviour during the `proceed` call is about 4 times as slow as the normal method call. While a deployed adaptation call does not cost more than a normal call, the use of `proceed` avoids code duplication at a linear cost with respect to the number of adaptations active for the same method.

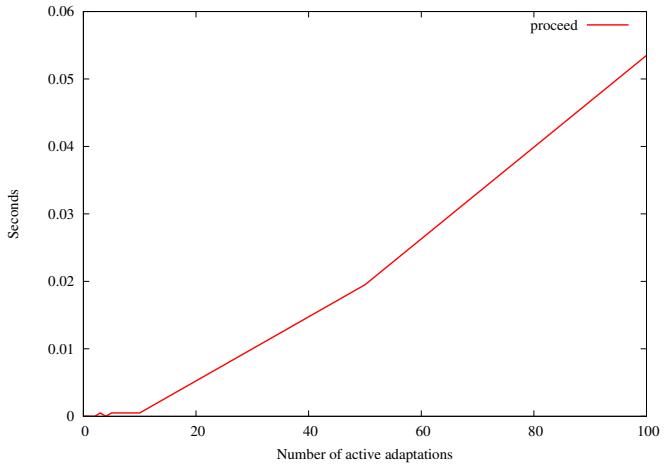


Figure 11.7 – Evolution of execution time of a method call with respect to the number of adaptations active for the same method. All adaptations using a `proceed` call.

11.3 Phenomenal Rails

The benchmarks of Section 11.2 give us an idea of some of Phenomenal Gem's performance. In order to further assess the Phenomenal Rails Gem capabilities, we used the Benubo application presented in Chapter 10. We compare the number of responses per second achieved by the Ruby on Rails (RoR) server while increasing the number of requests per second. This comparison was conducted on three different versions, which are the original versions provided by Belighted, the refactored one from Section 10.4 and finally, the FaaS version from Section 10.5. Figure 11.8 shows the results of the experiment[2].

From the graph in Figure 11.8 we can conclude that, although the Phenomenal Rails Gem leads to a small performance penalty, this is not significant for medium size web applications and provides new capabilities (FaaS) in addition to enhancing code structure. Furthermore, this performance penalty would hardly be noticed by the end-user compared to the enhanced customisation he will get.

The difference between the FaaS and Refactoring curves in Figure 11.8 comes from the fact that in the former, contexts are not persistent any more and thus, must be activated for each request. Furthermore, the difference between the Refactoring and the Original curves is due to the view adaptation mechanism detailed in Section 8.4.5.

[2]Conducted on an Intel Core 2 Duo T9400 @ 2.53Ghz server with Ubuntu 11.04 on the MRI version 1.9.2p180 and with RoR version 3.2.1. We used a single instance Mongrel (`http://rubydoc.info/gems/mongrel/1.1.5/frames`) web server version 1.2.0.pre2 in production mode with the deamon option (-d). The clients were simulated with the Httperf (`http://www.hpl.hp.com/research/linux/httperf/`) tool on an Ubuntu 11.04 computer with an Intel Pentium 4 @ 2.8 Ghz processor. The client and server were interconnected through a 100 Mb/s Local Area Network (LAN).

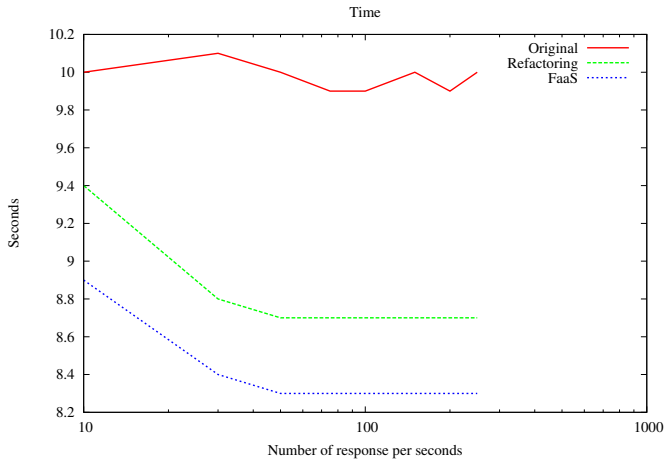


Figure 11.8 – Evolution of the number of responses per second for the three implementations of Benubo on the home page.

11.4 Conclusion

The Phenomenal Gem is most useful when the number of contexts switches are relatively small with respect to the number of calls of adapted methods. For that matter, usual applications are likely to fit this rule, since such context switches are normally linked to external events.

When using this framework, one should try to avoid too fine-grained contexts that often have to be (de)activated. In addition, since the `proceed` mechanism is linear, it should be used whenever possible, to avoid code duplication.

In case of the Phenomenal Rails Gem is only used for modularization, the application developer must trade off performance cost with code quality enhancement. From our point of view, if Phenomenal Rails Gem is used to build FaaS applications, the user-experience gain from the dynamic behaviour adaptation will be greater than the loss in performance.

We have learned thanks to Benubo that the contexts of real world applications need a limited number of adaptations. The penalty incurred by the exponential activation time of contexts shown in Figure 11.6 is thus limited. As a result, the framework is sufficiently efficient for standard RoR applications.

Part IV

Conclusion

Chapter 12

Future Work

Contents

And no, we don't know where it will lead. We just know there's something much bigger than any of us here.

Steve Jobs

12.1 Introduction

As nothing is perfect, there is always room for improvement, especially in computer sciences, and our framework is no exception to the rule. In this chapter, we will develop all the points from the more high-level to the lowest implementation issues that are subject to improvement. The following points stew from our own reflection on the topic, but we hope that other will come from the community of users, once we have put our implementation on the Web.

12.2 Phenomenal

12.2.1 Structural Adaptations

We have only developed behavioural adaptation for our framework. But as we dealt with adding and removing features at run-time, it will be useful to have the capability to also add/remove methods and classes at run-time. We have already started to think about how to add this capability but, for want of time, we have not implemented this functionality.

12.2.2 Relationships Set

Until now, we have implemented three relationships: requirement, suggestion and implication. However, other relationships can be very useful. Thanks to the fact that we always thought about extensibility, adding relationships to the framework is very easy.

12.2.3 Thread Specific Context

When we started to develop our framework, we did not intent to focus on multi-threading applications, because the Matz's Ruby Interpreter (MRI) does not deal very well with them. However, since other Ruby interpreters available which operate very well with real multi-threading, we have changed our mind. Indeed, it will be interesting to have contexts that are not global but specific to each thread.

12.2.4 Proceed Improvement

The proceed mechanism is implemented through a "hack" because of the poor reification of the calling stack in Ruby. Improvements should be made on this mechanism. Concretely, to improve the implementation, we need to find a way to remove the actual "hack". Furthermore, as shown in Chapter 11, the proceed mechanism is about four times as slow as a simple method call. Thus, in addition to removing the "hack", some implementation optimizations may be possible.

12.3 Phenomenal Rails

12.3.1 Activation Optimization

The requests to the web servers are handled in such a way that we need to deactivate all the contexts and then activate the right ones for each request. Another approach could be to deactivate only the contexts that are not needed any more. But as the (de)activation of all the contexts is not an atomic action, it is impossible to distinguish the difference of contexts between requests.

12.3.2 Proceed for Views

As the proceed mechanism is very useful for methods, we think that this could also be the case for views in a Ruby on Rails (RoR) application. The main issue is that the views can use different templating systems like Embedded Ruby (ERB) or HTML Abstraction Markup Language (HAML). Because of this, such a mechanism would probably have to work on the produced Document Object Model (DOM), but we have not investigated these possibilities yet.

Chapter 13

Conclusion

Contents

Adapt or perish, now as ever, is Nature's inexorable imperative.

H. G. Wells

The main problem addressed in this thesis was the lack of existing support for dynamically adaptable applications in the Ruby programming language as well as web applications with Ruby on Rails (RoR). Developing such applications was possible but could very quickly have led to poor design and messy code. For that purpose, we introduced the Phenomenal Gem, a Ruby based Context-Oriented Programming (COP) framework extended with some notions of Feature-Oriented Programming (FOP). This framework allows developers to handle contexts and features as first-class entities. As a result, they can handle behaviour adaptations in a straightforward and modular way and are put in the right frame of mind to build highly dynamic applications.

13.1 Contributions

While the actual implementation of the **Phenomenal Gem** and **Phenomenal Rails Gem** (web integration) are the main contributions of this thesis, their development has led to several new notions or extension of existing ones.

The notion of **Context as a Feature (CaaF)** brings the feature concept from the FOP

into the COP paradigm. This means that a feature *is* a context and refines it by adding the responsibilities of handling relationships. This notion provides a better structure of dynamically adaptable applications than if we only had contexts.

The **visualiser tool** is a first step to a debugging eco-system for COP development. To our knowledge, it is the first automated run-time representation of COP applications.

The **activation condition** notion that extends CaaF in the Phenomenal Rails Gem is a structured way of handling *Context Discovery* which is often omitted in other implementations. Furthermore, in addition to the relationships management, features become responsible for the activation of their sub-contexts.

The **validation case-study** on Benubo proved that the presented concepts and their implementation are actually useful in a real application. In addition to this, this case-study has led to the notion of **Feature as a Service (FaaS)**. Half-way between implementation guidelines and marketing needs, FaaS pushes the idea of Software as a Service (SaaS) further and allows to sell sub-components of an application. Thanks to the CaaF and the integration work done by the Phenomenal Rails Gem, FaaS can be achieved in a straightforward way.

The **Domain Specific Language (DSL)** defined with the framework provides a clear syntax for handling the notions reified by the framework.

The chapters listed below highlight their contribution to this thesis:

- Chapter 2 introduces the COP paradigm and helps to understand its basics.
- Chapter 3 introduces the relatively similar FOP paradigm and with Chapter 2 gives an overview of the theoretical foundation of the thesis.
- Chapter 4 presents the Ruby programming language and the reasons why we choose it for the implementation.
- Chapter 5 presents the same reasons and advantages for the RoR web framework used later for our case-study.
- Chapter 6 presents Subjective-C and contextR, two other COP implementations as well as rbFeature, a FOP implementation in Ruby. These frameworks help us to build ours.
- Chapter 7 introduces the core part of the thesis. In addition to presenting the developed concepts for the Phenomenal Gem and formal semantics for them, it also presents the architecture and core mechanisms of the framework.
- Chapter 8 presents the concepts of the Phenomenal Rails Gem and the core mechanisms.

- Chapter 9 presents the approaches used for the software development of the thesis.
- Chapter 10 validates the concepts and implementations by refactoring a SaaS Enterprise Resource Planning (ERP).
- Chapter 11 assesses the performances achieved by the implementation and proves that, while easy adaptability comes at a cost, this cost is not overwhelming.

13.2 General Conclusion

More than one year ago, we started to work on a first COP framework in Ruby as a project for the Programming Paradigms course of Professor Kim Mens. We already knew that it would become our Master thesis subject and we started to think about the Benubo case-study with Belighted at the same period.

This long term work represented an major challenge from the beginning. It ended with a new Ruby COP framework able to adapt the behaviour of any application with only one command. While this enhanced adaptability has a cost, the benchmarks and validation case-study showed that, at least for a reasonable number of adaptations, the performance cost is not noticeable by the end-user.

Along with the framework implementations, several COP concepts have been revisited and a first step was taken towards the merging of the COP and FOP paradigms. This led to the notion of CaaF which sees a feature as being a context. From the application of this notion emerged the new FaaS approach for developing highly customizable web applications. The refactoring of the RoR Benubo application and the feedback from Belighted finally gave us the opportunity to direct our research close to industrial needs. We really have good hope that our end-product will leave the research labs and we plan to continue to maintain and promote it in the years to come.

Appendix A

Application Programming Interface

This appendix introduces the DSL available to the framework user. The full RubyDoc is available on `http://rubydoc.info/gems/phenomenal/frames`.

A.1 Phenomenal Gem Version 1.2.2

A.1.1 Contexts Management

`phen_context(context[,contexts...],&block)` → `context`
`context(context[,contexts...],&block)` → `context`

Defines a new (combined) context.

`phen_feature(context[,contexts...],&block)` → `feature`
`feature(context[,contexts...],&block)` → `feature`

Defines a new (combined) feature.

`adaptations_for(klass)` → `klass`

Must be called in the body of a context or feature, defines the class for which the following adaptation defined with `adapt` and `adapt_class` belong.

`adapt(method,&block)` → `nil`

Must be called in the body of a context or feature, defines an adaptation for the instance method in the class specified by the previous call to `adaptations_for`.

`adapt_class(method,&block)` → `nil`

Must be called in the body of a context or feature, defines an adaptation for the class method in the class specified by the previous call to `adaptations_for`.

`phen_forget_context(context)` → `nil`

Removes a context from the system.

`phen_activate_context(context[,contexts...])` → `context or feature`
`activate_context(context[,contexts...])` → `context or feature`

Activates the context or feature.

`phen_deactivate_context(context[,contexts...])` → `context or feature`
`deactivate_context(context[,contexts...])` → `context or feature`

Deactivates the context or feature.

`phen_context_active?(context)` → `true or false`

Returns true when the context is currently active.

`phen_context_information(context)` → `hash`

Returns details about the context:

- *name*: the name of the context.
- *adaptations*: an array of the context adaptations.
- *active*: true if the context is active, false otherwise.
- *age*: the activation age of the context.
- *activation_count*: the number of activations of the context since the last actual deactivation.
- *type*: either `Phenomenal::Context` or `Phenomenal::Feature`.

`phen_default_feature` → `feature`

Returns the default feature of the system. This feature represents the base application.

`phen_defined_contexts` → `array`

Returns an array containing all the contexts and features defined in the system.

A.1.2 Adaptations Management

`phen_add_adaptation(context,klass, method_name, &body)` → `adaptation`

Creates and returns a new adaptation for the instance method in klass into context.

`phen_add_class_adaptation(context,klass, method_name, &body)` → `adaptation`

Creates and returns a new adaptation for the class method in klass into context.

`phen_remove_adaptation(context,klass, method_name)` → `adaptation`

Deletes and returns the adaptation for the instance method in klass from context.

`phen_remove_class_adaptation(context,klass, method_name)` → `adaptation`

Deletes and returns the adaptation for the class method in klass from context.

`phen_proceed([args...],&block)` → `object`
`proceed([args...],&block)` → `object`

Must be called into the body of an adaptation, returns the evaluation of the next method with less precedence according to the conflict policy. The arguments are passed to this next method.

`phen_change_conflict_policy(&block)` → `nil`

Sets a new conflict policy for the system. The block takes two contexts in argument and has to return -1 if first context has precedence on the second, 1 in the opposite case and raise a `Phenomenal::Error` if the conflict is not resolvable.

A.1.3 Relationships Management

`phen_requirements_for(source,:on=>[targets])` → `nil`
`requirements_for(source,:on=>[targets])` → `nil`

Define a requirement relationship for source on targets. When called in the body of a feature, this relationship is stored into it. Otherwise, stored in the default feature.

`phen_implications_for(source,:on=>[targets])` → `nil`
`implications_for(source,:on=>[targets])` → `nil`

Define a implication relationship for source on targets. When called in the body of a feature, this relationship is stored into it. Otherwise, stored in the default feature.

`phen_suggestions_for(source,:on=>[targets])` → `nil`
`suggestions_for(source,:on=>[targets])` → `nil`

Define a suggestion relationship for source on targets. When called in the body of a feature, this relationship is stored into it. Otherwise, stored in the default feature.

`requires(context,[contexts,..])` → `nil`

Must be called in the body of a context, defines a requirement relationship in its closest parent feature on the contexts in parameters.

`implies(context,[contexts,..])` → `nil`

Must be called in the body of a context, defines a implication relationship in its closest parent feature on the contexts in parameters.

`suggests(context,[contexts,..])` → `nil`

Must be called in the body of a context, defines a suggestion relationship in its closest parent feature on the contexts in parameters.

A.1.4 Debugging

`phen_textual_view` → `string`

Returns a textual representation of the current state of the system.

`phen_graphical_view(file)` → `nil`

Generates a graphical representation of the current state of the system.

A.2 Phenomenal Rails Gem Version 1.2.3

`is_persistent` → `context`

Must be called in a feature or context, states that the contexts is always active in the system.

`activation_condition(&block)` → `nil`

Must be called in a feature, specifies the logic to activate contexts before each HyperText Transfer Protocol (HTTP) request. The block defining logic has access to the Rack environment through the parameter `env`. This activation condition is only applied when the surrounding feature is active.

Appendix B

Graphical View Code

The following partial code is used to generate the graphical view of Section 7.3.1.

```
context :Trial do
 adaptations_for TodoList
 adapt :get_tasks do |date,time|
  ...
 end
end
feature :EnvironmentSense do
 context :Presenter do
  adaptations_for TodoList
  adapt :notify do
   ...
  end
 end
 context :Away do
  adaptations_for TodoList
  adapt :notify do
   ...
  end
 end
 context :Mobile do
  requires :OperatingSystemSense
 end
 context :Desktop do
  requires :OperatingSystemSense
 end
end

feature :OperatingSystemSense do
 context :Android do
  implies :Mobile
 end
```

```
32  context :IOs do
33   implies :Mobile
34  end
35  context :Linux do
36   implies :Desktop
37  end
38  context :Windows do
39   implies :Desktop
40  end
41  context :MacOs do
42   implies :Desktop
43  end
44 end
45
46 context :Away, :Network do
47  adaptations_for TodoList
48  adapt :notify do
49   ...
50  end
51 end
52
53 context :Network
54
55 phen_graphical_view
```

Bibliography

[AHH+09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

[Ape08] Sven Apel. Die rolle von features und aspekten in der softwareentwicklung (the role of features and aspects in software development). *it - Information Technology*, 50(2):128–130, 2008.

[Bla09] D.A. Black. *The well-grounded Rubyist*. Manning Pubs Co Series. Manning, 2009.

[CC06] F. Chong and G. Carraro. Architecture strategies for catching the long tail. `http://msdn.microsoft.com/en-us/library/aa479069.aspx`, April 2006.

[CGD11] Nicolás Cardozo, Sebastian Günther, and Theo D'Hondt. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *International Conference On Software Engineering Advances (ICSEA'11)*, pages 130 – 135. IARIA, 2011.

[DVC+07] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In Boicho Kokinov, Daniel Richardson, Thomas Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, volume 4635 of *Lecture Notes in Computer Science*, pages 178–191. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74255-5_14.

[Fer10] O. Fernandez. *The Rails 3 Way*. Addison-Wesley Professional Ruby Series. Addison-Wesley, 2010.

[Gam95] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[GCM+11] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c: Bringing context to mobile platform programming. In *Proceedings of the Third international conference on Software Language Engineering (SLE 2010)*, number 6563 in Lecture Notes in Computer Science, page 246265, Berlin, Heidelberg, 2011. Springer Verlag.

[GF11] Sebastian Günther and Marco Fischer. Supporting program variant generation and feature files in rbfeatures. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[Hei07] M. Heim. *Exploring Indiana Highways: Trip Trivia.* Travel Organization Network Exchange, 2007.

[HT99] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Pearson Education, 1999.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[PR01] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, September 2001.

[Sch08] Gregor Schmidt. Contextr & contextwiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, April 2008.

[Ste11] D. Stewart. An interview with the creator of ruby. `http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`, 2011.

[TCW+12] Eddy Truyen, Nicolás Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, and Theo DHondt and. Context-oriented programming for customizable saas applications. In *Symposium on Applied Computing*, SAC'12. ACM press, 2012.

[TFH09] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide.* Facets of Ruby. Pragmatic Bookshelf, 2009.

[Val11] J. Valim. *Crafting Rails Applications: Expert Practices for Everyday Rails Development.* Pragmatic Programmers. Pragmatic Bookshelf, 2011.

[Zav03] Pamela Zave. Programming methodology. chapter An experiment in feature engineering, pages 353–377. Springer-Verlag New York, Inc., New York, NY, USA, 2003.