UCLouvain

École polytechnique de Louvain

FBCOP vs COP: A Comparison of Their Maintainability and Reusability in Practice

Author: **Guillaume JADIN**
Supervisors: **Axel LEGAY, Benoît DUHOUX**
Readers: **Axel LEGAY, Benoît DUHOUX, Kim MENS**
Academic year 2023–2024
Master [120] in Computer Science

Abstract

École Polytechnique de Louvain

Master [120] in Computer Science

FBCOP vs COP: A Comparison of Their Maintainability and Reusability in Practice

by Guillaume Jadin

This thesis presents a comparative study of the maintainability and reusability between Feature-Based Context-Oriented Programming (FBCOP) and Context-Oriented Programming (COP). Through a unique application implemented in both paradigms, we examine the strengths and weaknesses of each approach, using tailored metrics and a questionnaire based on the Cognitive Dimension Framework (CDF) that gathers user experiences. Our analysis reveals that while FBCOP offers improved maintainability and reusability in certain aspects, such as feature implementation, it is also less maintainable and reusable than COP due to its project structure and architecture. We propose several solutions to address these inconsistencies, including a unified project structure and architecture, interactive tutorials, improved documentation, and enhanced error handling. Our discoveries globally provide a nuanced comparison between FBCOP and COP which reflects the complexities of the reality.

Acknowledgements

My sincerest gratitude goes to Benoît Duhoux for his unwavering support, guidance, and patience throughout the past year. Your ability to balance my moments of joy and fear with a well-timed joke was a precious gift, helping to ease the tension during a personally challenging time.

Deepest appreciation is extended to Axel Legay, whose omniscient presence and behind-the-scenes efforts were instrumental in shaping this master thesis. Thank you as well for teaching me how to write scientific papers and for guiding me in delivering an effective thesis presentation.

Heartfelt thanks to my friend Béatrice Lambert, who is also working on her master thesis at the same time and within the same paradigm as mine. You have been a wonderful confidante, allowing us to share our thoughts, fears, and discoveries with one another. I also appreciate your help in reviewing and clarifying many of the ideas in my thesis.

Gratitude is also due to my beloved Tania, who has always been there for me in my daily life, providing support as I shared my doubts, stress, and progress on this thesis.

I want to express my gratitude to my brother Alexandre and all my friends, including Maxime, Mathieu, and others. Thank you for staying in touch during this solitary period and for always being there to encourage the bear to emerge from its cave and join in on some parties.

A heartfelt thank you to my two grandmothers Nelly and Rose-Marie, who have always been there to share stories from their past and to inspire me with the hope of one day becoming even a fraction of the remarkable individuals they are today.

I am grateful for my dad and mom, who have always been there to listen to my concerns and help me navigate the challenges of daily life, never hesitating to support me whenever I needed them, despite the difficulties their faced.

Thanks also to my friend Hadrien for providing me with the best drink of my life, which also gave me the worst headache I've ever experienced.

Finally, my appreciation goes to the developers behind Reverso, who have been assisting me in writing English documents since I was 14. Additionally, thanks to the developers behind DuckDuckGo AI Chat for helping me improve my previously inadequate English, teaching me how to write better sentences, and allowing me to use generative AI in an open-source and anonymous manner.

Contents

Part I

Prologue

Chapter 1

Introduction

Within the realm of context-oriented programming paradigms, a plethora of studies can be found in literature that assesses a qualitative and quantitative comparison by evaluating their performance, discussing of their different development strategies, and analysing their characteristics, with the unique objective of answering to the following interrogation: *Which paradigm is better ?*

Although answers to this question are always nuanced, new context-oriented frameworks and paradigms frequently emerge to address inconsistencies and errors in previous approaches. This is the case of the feature-based context-oriented programming paradigm, which was conceived with the goal of differentiating itself from existing paradigms by combining context-oriented programming and feature modelling. This approach aims to provide developers with a new way of programming smart applications by aligning the process of thinking with the process of implementing by using features instead of classes. Alongside this novelty, its creators argue that their approach offers improved maintainability and reusability for developers of smart systems, surpassing the benefits of traditional context-oriented programming. Although they discuss this affirmation theoretically, they do not provide any practical evidence to validate their statement.

Therefore, this master thesis aims to address this unresolved claim by conducting a comparative study of FBCOP and COP through a unique application implemented in both paradigms (Chapter 2). The study will examine the strengths and weaknesses of each approach, focusing on their impact on maintainability, reusability but also readability, and understandability. Taking inspiration on existing literature, we will present multiple observations and interpretations (Chapters 5 and 6) from two distinct perspectives (Chapter 4) and then provide, as always, a nuanced conclusion (Chapter 9) that accurately reflects the complexities of the reality.

To support our conclusion, we will also propose several solutions (Chapter 7) that address the limitations of the feature-based context-oriented programming paradigm, allowing it to overcome its weaknesses and become more maintainable and reusable than COP.

Chapter 2

Running Example: Smart City Guide

This chapter presents the Smart City Guide application, our running example implemented within both COP and FBCOP frameworks, serving as a guiding thread during our analysis. Build upon the work of Cardozo and Mens [CM22], this personalized navigation solution is designed to enhance the tourism experience by offering users useful information about nearby landmarks (points of interest). Furthermore, the application can be personalized depending on individual users' needs by adapting to the user's age and language and considering user impairments to ease its utilisation. It also can dynamically adjust its behavior to match device connectivity modes, as well as the time of day.

In the following sections, we will provide a detailed description of the application's functionalities by presenting its context-feature model, and further illustrate these functionalities through scenario examples.
The presented FBCOP/RubyCOP and COP/ContextPy3 applications are respectively available in the Bitbucket repository at `https://bitbucket.org/benoitduhoux/rubycop_jadin/src/smart-city-guide/apps/smart_city_guide/` and in our GitLab repository at `https://gitlab.com/gujadin-uclouvain/master-thesis/apps/contextpy3-app/`.

2.1 Detailed Description

As discussed later in Section 3.2 and illustrated in Figure 2.1, the context-feature model provides a comprehensive representation of the application's capabilities. This section will delve into the details of these capabilities by explaining how they operate.

To optimize data usage, the application dynamically adjusts the type of data downloaded based on the user device connectivity. By default, it operates in *Limited Cellular* mode, which conserves mobile data deactivating the *Picture* feature. However, users can opt for *Normal Cellular* mode or switch to a *Wifi* connection to access content with images.

Our system can also adapt itself based on whether it is *Day* or *Night*. When night falls, the application display will switch to a *Darker Mode*, and the pictures of the landmarks will depict them at night by utilizing the *Alternative Picture* feature. This feature depends on a *Download Strategy* feature, which allows the selection of a strategy based on the activated context. The *Initial Picture* strategy will be chosen if it is daytime.

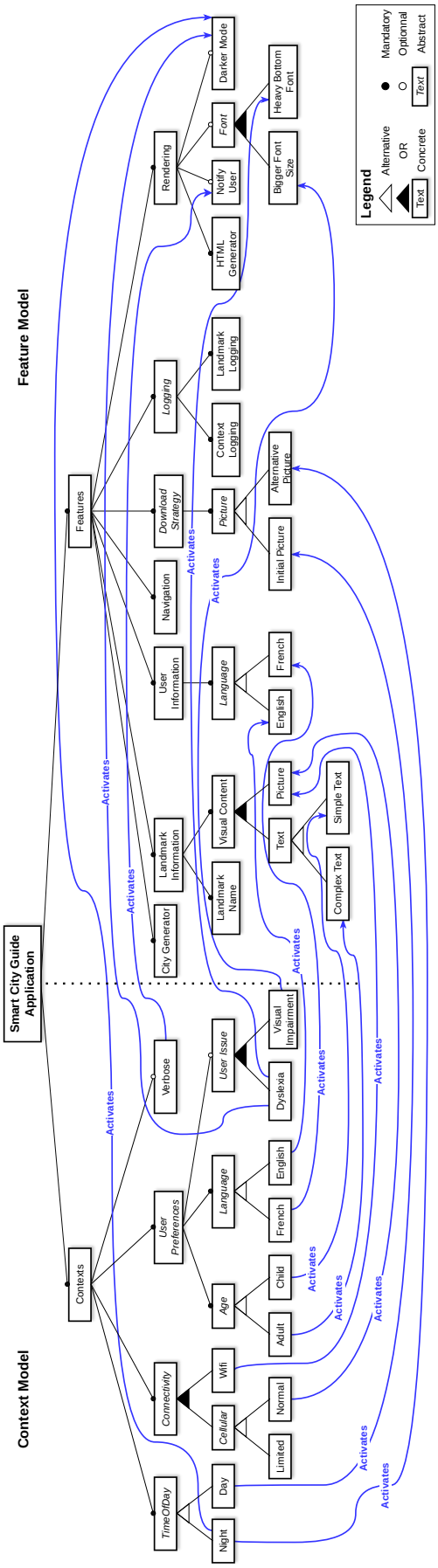


FIGURE 2.1: Context-feature model of our *Smart City Guide Application*.

In addition, if the user is considered as a *Child*, simpler and more relatable descriptions of the landmarks are displayed to facilitate easier comprehension. Otherwise, the user is considered as an *Adult* and the description will be more complex and presented in a more formal language. Depending on the user's native language, these descriptions can be displayed in *English* or *French*, automatically switching between them according to the user's preference.

The application also considers user impairments to enhance the user experience. For users with *Dyslexia*, a *Darker Mode* and a *Heavy Bottom Font* are applied to the text. A *Bigger Font Size* is available to assist users with *Visual Impairments*.

To ensure the user to remain informed on any changes, the application can be configured to *Verbose* mode, which enables three types of notifications:

1. Green notifications indicate the activation of a behavior (*Context Logging*).
2. Red notifications signal the deactivation of a behavior (*Context Logging*).
3. Blue notifications alert the user to the discovery of a new nearest landmark (*Landmark Logging*).

These discoveries are facilitated by the *Navigation* feature which calculates the distances between the user's device and landmarks coordinates to identify the nearest.

Ultimately, the data initially collected by the *City Generator* feature is converted by the *HTML Generator* feature into HTML content to display the information to the user in a clear and easily accessible format.

2.2 Scenario Execution

This section will illustrate how the application adapts and selects features by transitioning between different contexts using scenario executions to demonstrate these changes. HTML web page snapshots will be provided to showcase the dynamic rendering of the application, highlighting how the user interface changes in response to these adaptations. Before proceeding with our scenarios, we assume that the user has a device that can display HTML pages and has an internet connection.

We consider a user who wants to explore Louvain-la-Neuve and is currently located in the Grand Place. The user requests a fresh HTML page generated by our application from the server on their device. The user also provides their location as part of the request. In response, the server initializes a new instance of the application by setting up the default and required behaviors. The application then uses the user's location to identify the city they want to visit, determines the nearest landmark, and generates the corresponding HTML page output depicted in Figure 2.2, which is sent back to the user. At this point, the user has full access to the application's functionality.

In our scenario, the user is a child and sets their age on their device, triggering a new request to the server to adapt the application for a child user. In response, the application activates the *Child* context and deactivates the *Adult* context, resulting in a modification of the landmark descriptions. An updated HTML page is then sent back to the user and automatically reloaded on the child's device. The resulting new rendering is illustrated in Figure 2.3. Notably, we observe that the notifications warn

the user that the application's behavior has been modified.

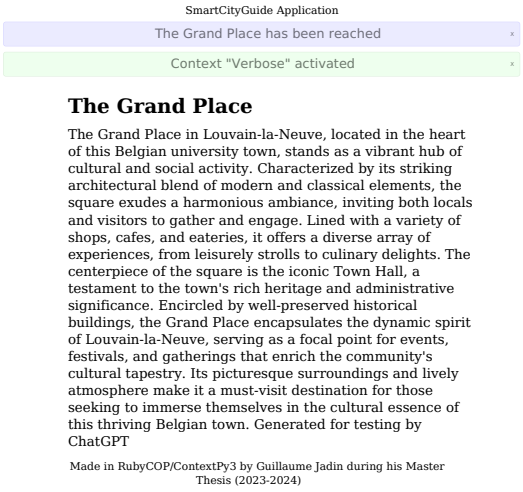


FIGURE 2.2: Screenshot of the HTML page output for an *Adult*.

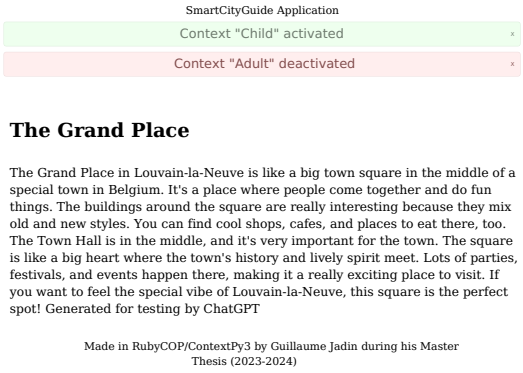


FIGURE 2.3: Screenshot of the HTML page output for a *Child*.

As previously mentioned, the application default connectivity mode is *Limited Cellular*, which reduces data consumption by not downloading pictures from the server. If the child wants to view pictures of their current landmark, they can switch to *Normal Cellular* mode. At the same time, the child has moved near the Hergé Museum. These two adaptations are depicted in Figure 2.4.

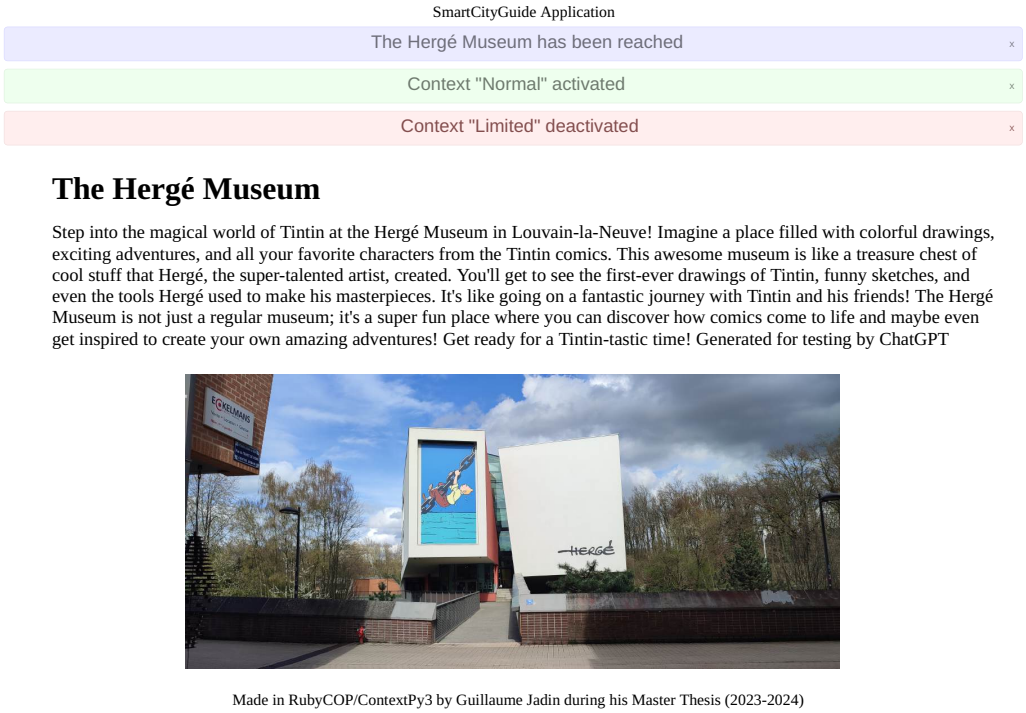


FIGURE 2.4: Screenshot of the HTML page output with *Normal Cellular* connectivity.

After exploring Louvain-la-Neuve for a while during the day, the child returns to the Grand Place as *Night* falls. At this point, their screen switches to a dark theme, as depicted in Figure 2.5, and the picture of the Grand Place changes to the nighttime variant.

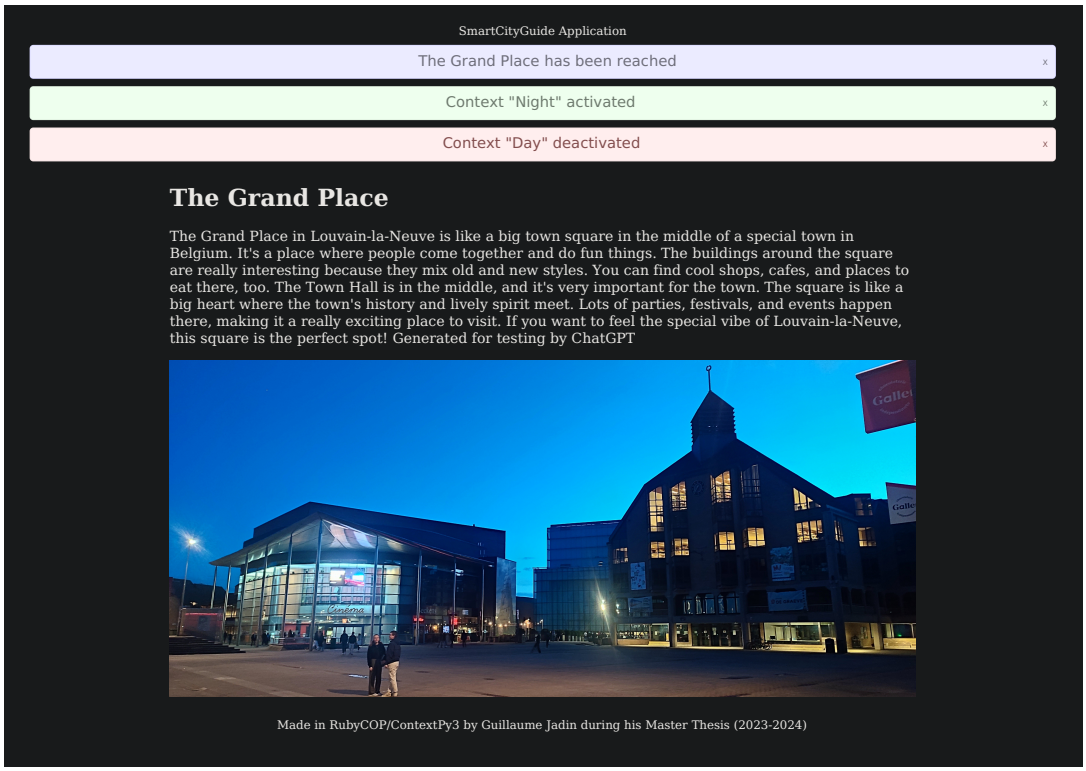


FIGURE 2.5: Screenshot of the HTML page output at nighttime.

Chapter 3

Background Material

Before delving into our comparison analysis within the realm of context-oriented paradigms, it is essential to establish fundamental concepts and provide precise definitions.

3.1 Context-oriented programming paradigm

This section introduces the fundamental principles of Context-Oriented Programming (COP) and demonstrates its implementation using the ContextPy3 framework, one of the many frameworks that implement COP.

3.1.1 Foundation

The Context-Oriented Programming (COP) paradigm was developed by Hirschfeld, Costanza and Nierstrasz [HCN08]. They describe it as a technique that *"provides mechanisms to dynamically adapt behavior in reaction to changes in context, even after system deployment at runtime"*. When contexts change, these behavioral adaptations are triggered by layers activation and deactivation. Based on this description, we need to explain some important concepts.

Definition 3.1 (*Behavioral Adaptation*)

Behavioral adaptations are defined as *"variations which consist of new, modified or removed behavior"* [HCN08] within a system.

Definition 3.2 (*Context*)

Contexts refer to any kind of information that a system can use to affect its behaviour in different situations [HCN08].

Information explained in Definition 3.2 may include various factors define in our running example such as *user age*, *device connectivity*, or *disabilities*. They all can influence the system's behavior.

Definition 3.3 (*Layer*)

Layers are used to group related behavioral adaptations that depend on the context in which they occur [HCN08].

In practice, we define the `Wifi` and `Geopositioning` layers in Listing 3.1 to adapt the class method `html_landmark_info/2`[1] in Listing 3.2. This is divided into multiple partial methods, each representing a behavior.

```
from contextpy3 import (
    Layer, proceed, active_layer, active_layers, inactive_layer,
    inactive_layers, after, around, before, base
)
wifi_layer              = Layer("Wifi")
geopositioning_layer    = Layer("Geopositioning")
```

Listing 3.1: Code snippet of the `app_layers.py` file representing the `Wifi` and `Geopositioning` layers declaration in ContextPy3.

```
from src.app_layers import *
import os, math

class Device(object):
    # ...
    @base                               # Base Method
    def html_landmark_info(self, landmark):
        from src.body.html import HTML
        # HTML().enclose("p", "test") => "<p>test</p>"
        return "".join((
            HTML().enclose("h1", landmark.get_name()),
            HTML().enclose("p", landmark.get_text())
        ))

    @around(wifi_layer)                 # Layered Method
    def html_landmark_info(self, landmark):
        from src.body.html import HTML
        return "".join((
            proceed(landmark),
            HTML().img(
                landmark.get_picture(),
                landmark.get_name()
            )
        ))

    @around(geopositioning_layer) # Layered Method
    def html_landmark_info(self, landmark):
        from src.body.html import HTML
        # HTML().img("./test.png", "test_img")
        #   => "<img src=\"./test.png\" alt=\"test_img\">"
        return "".join((
            proceed(landmark),
            HTML().enclose("p", "Here are the landmarks near
    your current location: {}".format(self.
    get_nearest_landmarks_with_distances()))
        ))
    # ...
```

Listing 3.2: Code snippet of `Device` class representing the concept of behavior adaptation using layers in ContextPy3.

[1] In this thesis, we note that methods are represented as `method_name/X`, where `X` indicates the number of parameters for the method `method_name`.

Definition 3.4 (*Partial Method*)

Partial methods represent behavioral adaptations of a specific method, divided into multiple methods with the same signatures [App+11].

We can categorize partial methods into two types: base methods and layered methods.

Definition 3.5 (*Base Method*)

Also referred as *plain method* in literature, a base method is the default behavior of a partial method that *"are not affected by the presence of layers"* [SGP12].

Definition 3.6 (*Layered Method*)

Layered methods are partial methods grouped into different layers which can dynamically replace or refine the base method through the activation or deactivation of their respective layers [App+09; App+11].

In practice, the dynamic replacement and refinement of a base method is facilitated by the `proceed()` technique, enabling the framework to dynamically alter the application's control flow.

Our code example shown in Listing 3.2 illustrates an application feature that generates an HTML content containing information about the landmark visited by the user. It also helps in understanding the usage of the `proceed()` function. This concept is further demonstrated through the schemas provided in Figures 3.1, 3.2, and 3.3. The partial method initially called in these three scenarios is identified by the cyan rounded note labeled with `METHOD CALL`. It is important to note that this method always conclude the method call and is denoted by the blue rounded note labeled with `METHOD RETURN`.

Figure 3.1 illustrates the initial state of the system where all layers are deactivated. We observe that that the base method is the only method taken into account during runtime when the `html_landmark_info/2` method is called. Essentially, it represents the partial method with the `@base` decorator, as depicted in Listing 3.2.

When the user is connected to a Wifi network, the system will activate the `Wifi` layer at runtime, as illustrated in Figure 3.2. In this scenario, the `Wifi` layered method is the initial partial method that will be called. As shown in Listing 3.2, this layered method includes a `proceed()` function call (line 19), which invokes the base method. Since the base method does not contain a `proceed()` function, it returns a value (a string in our case) that will be the result of the `proceed()` call within the `Wifi` layered method.

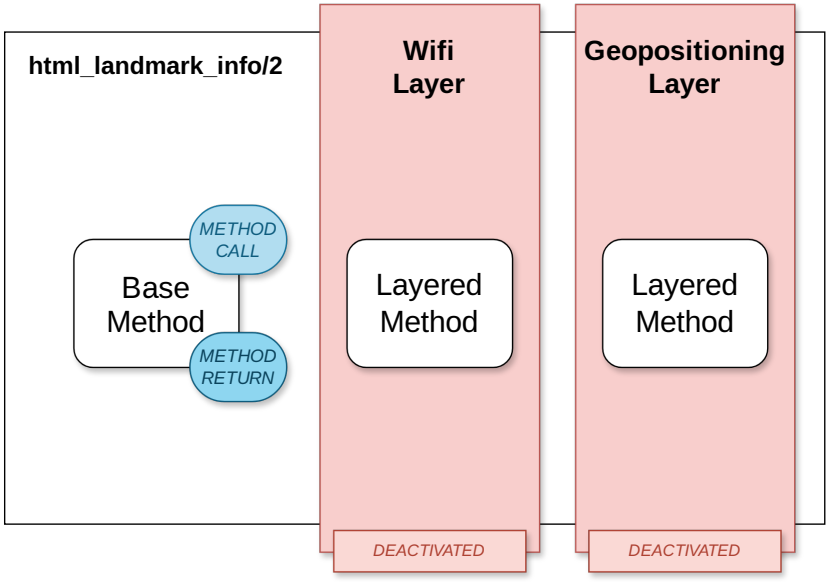


FIGURE 3.1: Diagram illustrating the activation sequence of layers when all are deactivated.

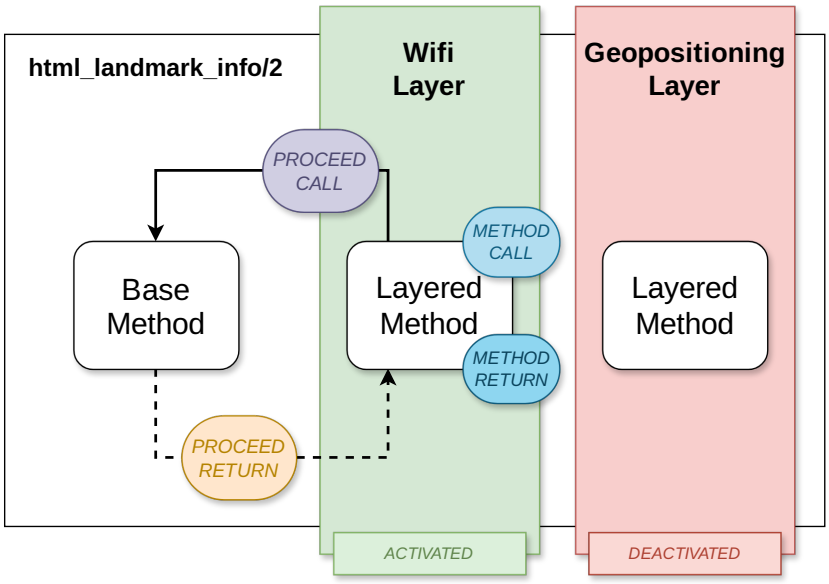


FIGURE 3.2: Diagram illustrating the activation sequence of layers demonstrating the `proceed()` technique after activating the `Wifi` layer.

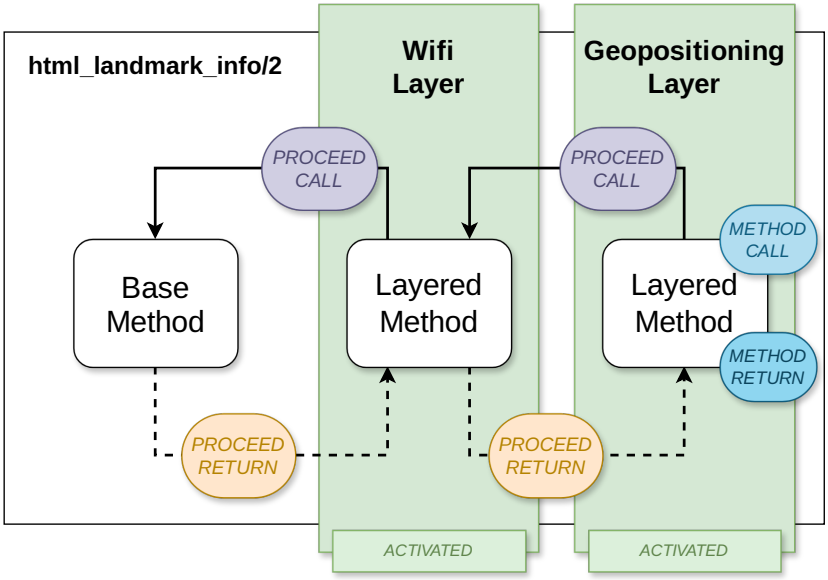


FIGURE 3.3: Diagram illustrating the activation sequence of layers demonstrating the `proceed()` technique after activating the `Wifi` layer followed by the `Geopositioning` layer.

In our third and final scenario, we illustrate the situation where the user activates the geopositioning feature on their device. This action triggers the activation of the `Geopositioning` layer at runtime, and its layered method becomes the initial partial method that will be called, as depicted in Figure 3.3. Similar to Figure 3.2, this layered method includes a `proceed()` function call (line 32) in Listing 3.2, which invokes the `Wifi` layered method. Upon its execution as described earlier, it returns a value that will be the outcome of the `proceed()` call.

3.1.2 Implementation: ContextPy3

ContextPy3 [Chr18] is a Python framework developed by Jonas Chromik based on the context-oriented framework for Python 2, ContextPy [PS08], originally created by Christian Schubert and Michael Perscheid.

We will utilize this framework compared to many others during our analysis for several reasons. Firstly, Python is widely used and known for its simple syntax and widespread utilization. Secondly, we were looking for a framework that shared a similar degree of syntactical sugar to Ruby, yet was distinct from it, allowing us to differentiate between framework-specific and language-specific characteristics.

Listing 3.1 illustrates the layer declaration approach employed in our ContextPy3 application. Although we opted to centralize all layer declarations into a single file to enhance readability, this is not a requirement imposed by the framework.

Listing 3.2 provides an example of partial method definitions in a class-implementation style. As shown, base methods and layered methods are annotated with decorators

`@base` and `@around(layer_name)`[2] respectively.

Finally, Listing 3.3 presents the execution code for our application. Notably, developers are required to explicitly activate layers that are always or by default activated. Furthermore, we opted to execute our application in the form of scenarios, where contexts are modified using the `activate(layer_names)` and `deactivate(layer_names)` methods. Then, the content is dynamically adapted and generated via the `device.render()` method.

```
from src.app_layers import *
from src.body.city import City
from src.body.user import User
from src.body.logger import Logger
from time import sleep

if __name__ == '__main__':
    # Keep track on which context has been (de)activated
    app = Logger()

    # Activate Verbose mode
    app.activate(verbose_layer)

    # Default activated layers at the initialization of the app
    app.activate(root_layer, cellular_limited_layer, day_layer,
    adult_layer, english_layer) # Use "global_activate_layer(
    layer_name)"

    change_layer_time = 5

    # Choose the city used by the application (JSON in "cities"
    folder)
    cities = City.generate()

    # Initialize the user and its device
    user = User("admin", cities["lln"])
    device = user.device
    device.update_position(50.66943, 4.61158)

    # Display User Information
    user.display()
    user.display_position()

    # ------------------------------------------------------ #
    print("'Child User'")
    app.activate(child_layer)
    app.deactivate(adult_layer)
    #device.display()        # Print in the console
    device.render()          # Adapt the HTML page dynamically
    sleep(change_layer_time) # Change Context in 5 seconds
    #...
```

Listing 3.3: Code snippet of the `app_runner.py` file illustrating the execution process of our application.

[2]Additional decorators, such as `@before(layer_name1)` and `@after(layer_name1)`, can also be used to execute partial methods before and after the partial method labeled with `@around(layer_name1)`.

3.2 Feature-based context-oriented programming paradigm

This section presents the core principles and philosophies of Feature-based Context-oriented Programming (FBCOP) and illustrates its implementation through its framework RubyCOP.

3.2.1 Foundation

The Feature-based context-oriented programming (FBCOP) paradigm is an approach introduced by Benoît Duhoux. His Ph.D. thesis [Duh22] aims to combine the principle of context activation/deactivation and dynamic method adaptability in runtime of COP with the principle of feature modelling [Kan+90] and by extension, the context modelling [Des+07] principle as well.

The feature modelling approach involves subdividing the entire program into multiple small features, each linked to contexts. Thus, when a context (Definition 3.2) is activated or deactivated, its associated features are also activated or deactivated accordingly.

Definition 3.7 (*Feature*)

A feature can be defined as a precise implementation of a system that is understandable by a end-user and used in a specific context.

Compared to COP, a feature is a behavioral adaptation (Definition 3.1) of a FBCOP application triggered by a context (de)activation.

If we consider a class named *Render*, it may be unclear without reading all the methods or without a documentation or good comments to understand what it is doing. However, if we subdivide this class into two features: *Generate Cities* and *Generate HTML*, the purpose of each feature becomes evident just by their names.

Because the context modelling notation is build on the same theoretical foundations of feature modelling notation [Des+07; Duh22], the creators of FBCOP drew inspiration from Hartmann and Trew's *multiple-product-line-feature model* theory [HT08] and proposed a unified approach *"to design both contexts and features with feature modelling to model a context model and a feature model, respectively"* [Duh22]. This approach is complemented by a mapping that specifies which context(s) trigger(s) which feature(s). Collectively, these two models and the mapping form the Context-Feature model, an example of which is illustrated in Figure 2.1.

Both context model and feature model consist of nodes, which represent contexts and features, respectively. In the feature model, nodes can be either abstract or concrete, with the main distinction being that a concrete node has a corresponding implementation.

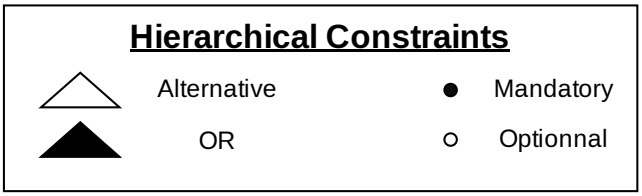


FIGURE 3.4: Hierarchical constraints of a context-feature model in FBCOP.

In both models, nodes can be related to their child nodes through four specific hierarchical constraints [Duh22], which are visually represented in Figures 2.1 and 3.4. These constraints are formally defined as follows:

- MANDATORY constraint: A child node is always activated when its parent is activated.
- OPTIONAL constraint: A child node can be activated if its parent is activated.
- OR constraint: At least one child node must be activated when the parent is activated.
- ALTERNATIVE (XOR) constraint: Exactly one child node can be activated when the parent is activated. All the others are excluded.

The FBCOP system architecture, illustrated in Figure 3.5, combines the principle of context-oriented software architecture [MCD16] with Hartmann and Trew's *multiple-product-line-feature model* theory. This architecture control flow explains the procedure to dynamically execute features by importing them into relevant classes [Duh22]. Before explaining the steps involved in achieving this solution, we will first define some terms.

Definition 3.8 (*Rollback*)

When a system attempts to transition from a valid configuration `C1` to a new configuration `C2`, but `C2` fails to meet the required constraints, the system **rolls back** to its previous state, restoring configuration `C1` [Duh22].

The control flow is triggered when a change occurs, made either by user interaction or sensors [Duh22]. Upon detection of such a change, the control flow proceeds through four distinct steps:

1. CONTEXT ACTIVATION: This step attempts to activate or deactivate contexts while ensuring that the constraints keep respecting the context model. If the constraints are not met, a rollback is performed.
2. FEATURE SELECTION: In this step, features related to activated contexts are selected, while features related to deactivated contexts are unselected.
3. FEATURE ACTIVATION: This step attempts to activate or deactivate features while ensuring that the constraints keep respecting the feature model. If the constraints are not met, a rollback is performed.

4. Feature Execution: During this final step, the behavior of classes is dynamically adapted at runtime by incorporating the definitions of activated features and removing the definitions of deactivated features.

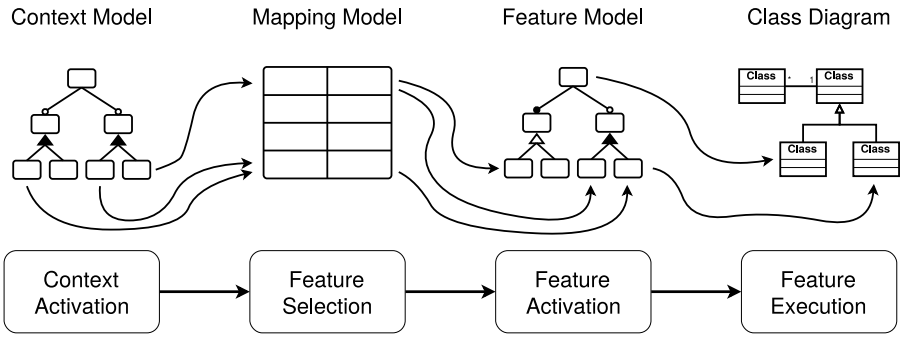


Figure 3.5: *"Overview of the FBCOP system architecture"* [DMD19; Duh22]

3.2.2 Implementation: RubyCOP

In this subsection, we talk about RubyCOP, the framework that implements the FBCOP paradigm in Ruby [Duh22; DMD19; Duh17].

During our analysis, we will employ two concepts that will be extensively used: the project structure and the project architecture. We will now introduce definitions related to these concepts.

Definition 3.9 (*Component*)

In the context of this thesis, we define a component as a specific element within a programming project with its own functionality, hierarchy in relation to other components, and proper interactions with them.

In practice, we enclose in the term *component* the concepts of *classes*, *methods*, *modules*, *features*, *contexts*, ...

Definition 3.10 (*Project Architecture*)

The project architecture represents the design and organization of the overall components and their relations.

In the context of the FBCOP framework, the project architecture is the implementation of the context-feature model built by declaring contexts, features, and their mappings.

Its hierarchical layout enables the framework and developers to understand the logic behind the project components.

Definition 3.11 (*Project Structure*)

The project structure refers to the organization and arrangement of files and folders within a project, determined by either the framework or project developers if no rules are specified by the framework.

Its hierarchical layout, when well done, helps developers manage and navigate easily through the project codebase.

The RubyCOP framework has a defined project structure that adheres to specific rules which serve to clearly separate the concepts of context, feature, and mapping models; as well as component declaration and definition. The following definitions will provide further clarification on these two last concepts.

Definition 3.12 (*Declaration*)

A declaration is an explicit statement that introduces a new component to the framework by specifying its name and any relevant relations. It serves as the first announcement of the component existence and characteristics, without providing its implementation details.

Definition 3.13 (*Definition*)

A definition is a detailed specification that implements a previously declared component by describing its behaviors and functionalities.

In RubyCOP, base methods and their related layered methods are organized hierarchically and separated in multiple features where each partial method is implemented within a specific feature. However, to ease our comparison, we will refer to them as base method (feature) and layered methods (features) all along this thesis.

The project structure (illustrated in Appendix A.2) includes a set of files and folders where each of them corresponds to concepts previously defined:

- `/contexts/context_model_declaration.rb`: This file contains the declarations of contexts within the current system. An example of such a declaration is provided in Listing 3.5.
- `/features/feature_model_declaration.rb`: This file contains the declarations of features within the current system. An example of such a declaration is provided in Listing 3.4.
- `/features/feature_definitions`: This folder contains the definitions of features within the current system. An example of this folder structure from our application can be found in Appendix A.2.
- `/mapping/mapping_model_declaration.rb`: This file contains the mapping between contexts and features within the current system. An example of such a mapping is provided in Listing 3.6.
- `/skeleton`: This folder contains the application execution file and all class definitions used in the project. In accordance with the principles of feature modelling, RubyCOP aims to keep these files as minimal and empty as possible to promote the use of feature definitions instead.

Finally, we will briefly introduce a RubyCOP tool that will be referenced in our analysis. Developed by Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens, the **context-feature model visualiser** tool is designed to offer a comprehensive visual representation of the context model, feature model, context-feature mapping, and defined code classes by allowing developers to monitor system behavior adaptation, observe the control flow of the FBCOP system architecture in real-time and gain an overall understanding of the project architecture [Duh+19a; Duh+19b].

```
class AppFeatureModelDeclaration < FeatureModelDeclaration
  include Singleton

  def initialize()
    super()
    _define_features_about_landmark()
    @root_feature.add_relation :Mandatory, [
     @landmark_information ]
  end

  private
  #============================================================#
  def _define_features_about_landmark()
    feature :@landmark_information, "LandmarkInformation", [:
     Landmark, :Device]
    feature :@landmark_name,    "LandmarkName", [:Landmark, :
     Device]
    _define_features_about_landmark_content_visual()
    @landmark_information.add_relation :Mandatory, [
      @landmark_name, @visual_content
    ]
  end
    #----------------------------------------------------------#
    def _define_features_about_landmark_content_visual()
      feature :@visual_content, "VisualContent", [:Landmark]
      _define_features_about_landmark_content_visual_text()
      feature :@picture,  "Picture", [:Device]
      @visual_content.add_relation :Or, [
        @text, @picture
      ]
    end
      #--------------------------------------------------#
      def _define_features_about_landmark_content_visual_text()
        feature :@text, "Text", [:Landmark, :Device]
        feature :@complex_text, "ComplexText", [:Landmark]
        feature :@simple_text,  "SimpleText", [:Landmark]
        @text.add_relation :Alternative, [
          @complex_text, @simple_text
        ]
      end
      #--------------------------------------------------#
    #----------------------------------------------------------#
  # ...
  #============================================================#
end
```

Listing 3.4: Code snippet of the feature model declaration of our application.

```
class AppContextModelDeclaration < ContextModelDeclaration
  include Singleton

  def initialize()
    super()
    _define_connectivity_contexts()
    @root_context.relation :Mandatory, [ @connectivity ]
  end

  private
  #============================================================#
  def _define_connectivity_contexts()
    abstract_context :@connectivity,  "Connectivity"
    context :@wifi,           "Wifi"
    _define_connectivity_cellular_type_contexts()
    @connectivity.relation :Or, [@wifi, @cellular]
    @connectivity.default @cellular
  end
    #-----------------------------------------------------#
    def _define_connectivity_cellular_type_contexts()
      abstract_context :@cellular,    "Cellular"
      context :@limited_cellular,     "Limited"
      context :@normal_cellular,      "Normal"
      @cellular.relation :Alternative, [
        @limited_cellular, @normal_cellular
      ]
      @cellular.default @limited_cellular
    end
    #-----------------------------------------------------#
  # ...
  #============================================================#
end
```

Listing 3.5: Code snippet of the context model declaration of our application.

```
class AppMappingModelDeclaration < MappingModelDeclaration

  include Singleton

  def initialize()
    contexts = AppContextModelDeclaration.instance
    features = AppFeatureModelDeclaration.instance
    @mapping = {
      # Connectivity
      [ contexts.wifi() ]             => [ features.picture() ],
      [ contexts.normal_cellular() ] => [ features.picture() ],
      # ...
  end
end
```

Listing 3.6: Code snippet of the mapping model declaration of our application.

Part II

FBCOP vs COP

Chapter 4

Approaches

4.1 Problem Statement

In the preceding part, we provided comprehensive information to facilitate a thorough understanding of our subject matter. In this new part, we delve into the core subject of this master's thesis, beginning with an exploration of the motivations behind its creation.

As discussed in our Background Material, we explain the fundamental principles behind FBCOP and COP as well as their own purposes. Since FBCOP is built upon COP, both paradigms offer programmers the ability to develop context-oriented systems, although with different approaches. The creators of FBCOP argue that its utilization enables developers to enhance the maintainability and reusability of their systems more effectively than COP. They assume that *"a clear separation of the contexts and features leads to a more maintainable and reusable approach"* [Duh22]. Nevertheless, they discuss this statement theoretically without providing empirical proof to support it.

Therefore, the goal of this thesis is to answer to the question *"Is FBCOP more maintainable and reusable than COP"*. This has been accomplished through the development of two identical context-oriented applications in both FBCOP and COP frameworks, as discussed in Chapter 2.

To provide a more detailed explanation of the thesis's objectives, we have divided our problematic into four research questions, each focusing on a distinct aspect. These questions will be addressed in the upcoming chapters, guiding our analysis.

RQ1 *How can we measure and evaluate the maintainability and reusability of FBCOP and COP?*

RQ2 *Which issues can we encounter when designing and programming FBCOP/COP applications?*

RQ3 *What impacts do the exclusive features of FBCOP and COP have on their maintainability and reusability?*

RQ4 *Which modifications can be implemented in FBCOP to enhance its maintainability and reusability?*

4.2 Techniques and Solutions

In the following chapters, we will discuss two distinct approaches which we think will help to answer our statements:

The first approach consists at adapting existing metrics from the literature to create new ones, which will be applied to our two applications developed with each framework. This analysis aims to provide a precise and objective understanding of the strengths and weaknesses of both frameworks.

The details of this metric approach are presented in Chapter 5. In this chapter, we will also provide our interpretation of the gathered outcomes, drawing connections between the results obtained from our applications and the frameworks and paradigms in which they are implemented. The findings from this chapter will serve as a foundation for our second approach, which requires a distinct methodology that diverges from the one employed here.

Indeed, our second technique aims to analyse our application in a more subjective way by asking to participants to answer to a questionnaire based on the Cognitive Dimension Framework (CDF) [GP96].

We will conduct a deep analysis of their responses to gain a precise understanding of into their thought processes and opinions regarding the two applications we provided. By combining this analysis with the conclusions drawn from our first approach, we will provide our interpretation and ultimately present a comprehensive conclusion that integrates both objective observation and subjective perspectives on both applications.

Finally, we will propose a range of potential solutions that we believe can address the various drawbacks and limitations of FBCOP.

Chapter 5

Metrics

Comparisons between frameworks have been extensively explored in existing literature, primarily focusing on assessing the performance of various features implemented in different COP frameworks [App+09]. However, since FBCOP represents a novel approach within the COP paradigm, and our focus on comparing user experiences, we need to use new metrics tailored to analyze these specific concepts.
In this chapter, we will first enumerate the various metrics we will employ to conduct our analysis and explain the reasons for selecting them. Following that, we will showcase the gathered results and provide our insights on their implications.

5.1 Description

To provide results based on objective observations, we must identify suitable metrics for COP frameworks. Unfortunately, metrics tailored specifically for this purpose are not widely available in existing literature. Hence, we had to adapt metrics from paradigms closely related to COP which will be the procedural paradigm and the object-oriented paradigm but also from general engineering principles.

5.1.1 Files & Folders Metric

The first metric we employ is a simple count of the number of files and folders within the project structure of both FBCOP and COP applications. This is the first step to observe the complexity of understanding, reading, maintaining and reusing a project when its structure increases [RH02].

5.1.2 Line Of Code (LOC) Metric

Our second metric counts the number of lines of code in each file [LH93], distinguishing between comments [RH02] and actual code. Usually, a file with more LOC have a tendency to be less readable and maintainable.

5.1.3 Partition Metric

The Partition principle, or the "divide and conquer" strategy, consists on splitting a big and complex task into small and easier subtasks which can be solved independently. In the scope of project programming, this principle allows developers to work on more reduce tasks which enhance the overall project maintainability and reusability [Pec12b; PY07].

COP and FBCOP have respectively a class-oriented and a feature-oriented approach of project implementation. Therefore, a class in COP implements one or multiple features and a feature in FBCOP completes the implementation of one or multiple

classes. Our next metric named *Average Number of Class-Feature Implemented* (ANCFI) aims to understand which paradigm are easily maintainable and reusable by counting the average number of features implemented in each class for COP and counting the average number of classes used to implement each feature in FBCOP. The paradigm with the smallest value have a more partitioned project architecture.

5.1.4 Cohesion Metric

We derive our last metric from one OOP metric which help measuring the cohesion between specific components (Definition 3.9) from our both frameworks.

Definition 5.1 (*Cohesion*)

The cohesion of a component refers to *"the degree of connectivity among the elements of this component"* to achieve a specific purpose. [ES95]

Generally, a high cohesion of a component means its elements focus together on the same objective, while a component with a low cohesion have elements which satisfies multiples unrelated tasks.

This cohesion principle is an important software quality criteria by being an easy way to determine if an applicant code may be maintainable and reusable or not.

This metric is known as *Lack of Cohesion of Methods* (LCOM) [LH93; RH02] which assesses the cohesion of a class by examining the interaction between local methods and local variables instantiated within the same class. According to the literature, a class with higher cohesion is generally easier to maintain.

Thanks to this metric, we wanted to objectively assess the cohesion of both COP and FBCOP. Therefore, we build a new metrics called *Lack of Cohesion of Partial Methods* (LCOPM) which evaluates the cohesion of all the partial methods from one specific class. This is done by analysing in which feature each base methods are implemented compared to their layered methods.

ContextPy3 require developers to implement all layered methods along their base method which force a better cohesion. In RubyCOP, the cohesion is not forced, as each layered method can be defined across multiple features beyond their base method, which requires a new methodology as depicted in Figure 5.1. This RubyCOP methodology is based on the project architecture, as depicted in the feature model in Figure 2.1. This approach is supported by the tree-like representations of project architecture and structure, which we assume share a similar layout.

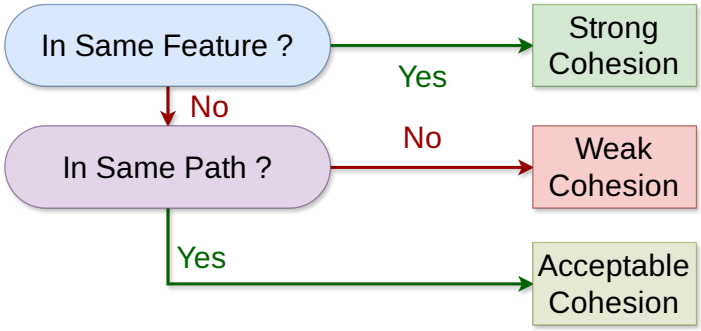


FIGURE 5.1: Diagram illustrating the analysis process of our LCOPM metric for FBCOP.

Initially, partial methods have a **strong cohesion** if the base method and all of its layered methods are implemented in the same feature. Our methodology complete this rule with two others if features are different:

1. Partial methods have an **acceptable cohesion** if the base method feature is an ancestor[1] of all of its layered methods features on the project architecture. Example depicted in Figure 5.3.

2. Partial methods have an **weak cohesion** if the base method feature is NOT an ancestor of at least one of its layered methods features on the project architecture. Example depicted in Figure 5.2.

5.2 Analysis & Interpretation

In this section, we will use the metrics previously defined to present the results obtained from our two running examples. We will analyze these results and provide an objective interpretation of the data by linking them to the theoretical insights derived from our metrics. The data was collected using Python scripts, which are available in our GitLab repository at `https://gitlab.com/gujadin-uclouvain/master-thesis/analysis-tools`.

5.2.1 Project Structure

Our first analysis focuses on the project structures of COP and FBCOP, as measured by the metric introduced in Subsection 5.1.1. Tables 5.1 and 5.2 provide an overview of the project structures for both applications[2].

Upon inspection, we notice that FBCOP project structure comprises a significantly larger number of files and folders compared to COP, even for a relatively small project. This suggests that as project size increases, the complexity of reading, understanding, reusing, and maintaining a FBCOP project tends to increase more quickly than a COP project.

[1] Parent or further ancestor of a node.

[2] Graphical representations of the project structures for COP and FBCOP can be found in Appendices A.1 and A.2, respectively.

File	Code Lines	Comment Lines	Total Lines
/app_runner.py	86	19	105
/app_layers.py	17	19	36
/body/choice.py	18	0	18
/body/city.py	43	4	47
/body/device.py	48	2	50
/body/html.py	49	0	49
/body/landmark.py	30	0	30
/body/logger.py	27	0	27
/body/server.py	131	4	135
/body/user.py	38	1	39
/tools/singleton.py	17	0	17
/	504	49	553

TABLE 5.1: Project structure & LOC Summary of our COP application.

5.2.2 Lines Of Code

Another aspect we can examine is the LOC for all project files. The data gathered from COP and FBCOP, are respectively presented in Tables 5.1 and 5.2 as well.

An analysis of these tables reveals that the overall FBCOP project contains approximately twice as many LOC as the COP project. Notably, the feature definition in FBCOP has a similar number of LOC to the entire COP project. This suggests that the `declaration` (context, feature, mapping), `skeleton`, `configuration`, and `run` files in FBCOP are comparable in size to the complete feature definition.

Moreover, despite having fewer files, each file in a COP project tends to have a higher number of LOC compared to the feature definition files in a FBCOP project. This implies that individual files in COP projects are likely to be more complex and challenging to read and understand due to their larger size.

5.2.3 Class & Feature Partitioning

By relying on the ANCFI metric, which is detailed in Subsection 5.1.3, we can now conduct a more in-depth analysis of our two frameworks. The data required for this metric are presented in Tables 5.3 and 5.4.

These tables offer valuable insights that support our assertion when using the LOC metric. Notably, our analysis reveals that COP developers typically implement around 4 features per class, whereas FBCOP programmers tend to complete one or two classes when implementing a feature. This suggests that FBCOP is more effective in partitioning a project, which becomes increasingly important as the project size grows.

File	Code Lines	Comment Lines	Total Lines
/main.rb	5	7	12
/options_interpreter.rb	13	6	19
/contexts/context_model_declaration.rb	86	15	101
/features/feature_model_declaration.rb	114	28	142
/features/feature_definitions/...	553	141	694
/mapping/mapping_model_declaration.rb	79	12	91
/skeleton/smart_city_guide.rb	125	26	151
/skeleton/choice.rb	3	7	10
/skeleton/city.rb	2	6	8
/skeleton/device.rb	2	6	8
/skeleton/html.rb	46	5	51
/skeleton/landmark.rb	2	6	8
/skeleton/logger.rb	3	6	9
/skeleton/server.rb	3	7	10
/skeleton/user.rb	2	6	8
/	1038	284	1322

File	Code Lines	Comment Lines	Total Lines
.../cities/generate.rb	57	10	67
.../download/picture/init_picture.rb	9	6	15
.../download/picture/alt_picture.rb	9	6	15
.../landmark/landmark_information.rb	27	7	34
.../landmark/landmark_name.rb	25	6	31
.../landmark/content/picture.rb	10	6	16
.../landmark/content/visual_content.rb	17	6	23
.../landmark/content/text/complex_text.rb	10	6	16
.../landmark/content/text/simple_text.rb	10	6	16
.../landmark/content/text/text.rb	24	6	30
.../logs/context_logging.rb	35	7	42
.../logs/landmark_logging.rb	25	7	32
.../navigation/navigation.rb	41	7	48
.../user/user_info.rb	63	6	69
.../user/language/en.rb	9	6	15
.../user/language/fr.rb	9	6	15
.../views/render.rb	23	6	29
.../views/color/darker_mode.rb	15	6	21
.../views/font/dyslexia_font.rb	19	7	26
.../views/font/big_font_size.rb	14	6	20
.../views/generator/html_generator.rb	66	6	72
.../views/notification/notify_user.rb	36	6	42
/features/feature_definitions/...	553	141	694

Table 5.2: Project structure & LOC Summary of our FBCOP application.

Class Name	# Features
Choice	4
City	1
Device	6
Landmark	6
Logger	2
Server	6
User	1
AVERAGE	**3.7**

Table 5.3: Number of features implemented in each class in COP.

Feature Name	# Classes
AlternativePicture	1
BiggerFontSize	1
CityGenerator	1
ComplexText	1
ContextLogging	1
DarkerMode	1
English	1
French	1
HeavyBottomFont	1
HtmlGenerator	1
InitialPicture	1
LandmarkInformation	2
LandmarkLogging	1
LandmarkName	2
Navigation	1
NotifyUser	1
Picture	1
Rendering	2
SimpleText	1
Text	2
UserInformation	1
VisualContent	1
AVERAGE	**1.2**

Table 5.4: Number of classes implementing each feature in FBCOP.

5.2.4 Cohesive Partial Methods

The final metric we will employ is the cohesion metric, as defined in Subsection 5.1.4, which compare the cohesion of the partial methods implemented in our two applications. We initially apply this metric to the ContextPy3 implementation. As previously mentioned, ContextPy3 forces a strong cohesion of its partial methods due to its class-based implementation. Consequently, our COP application is likely to have equal or better cohesion compared to RubyCOP. Our task now is to quantify the difference in overall cohesion between the two applications.

In the RubyCOP application, we can identify all three levels of cohesion defined in the preceding section. For instance, the method `generate_html/1` exemplifies strong cohesion, as its partial methods are all implemented within the *HTML Generator* feature as depicted in Listing 5.4.

An example of weak cohesion can be observed in Figure 5.2, specifically with the `generate_css/1` method. We observe that its base method is defined within the *HTML Generator* feature (Listing 5.4), while its layered methods are implemented in separate features, namely *Darker Mode*, *Bigger Font Size* and *Heavy Bottom Font* (Listing 5.5). This non-ancestral relation between these features makes it harder for developers to locate these partial methods, making maintenance more challenging.

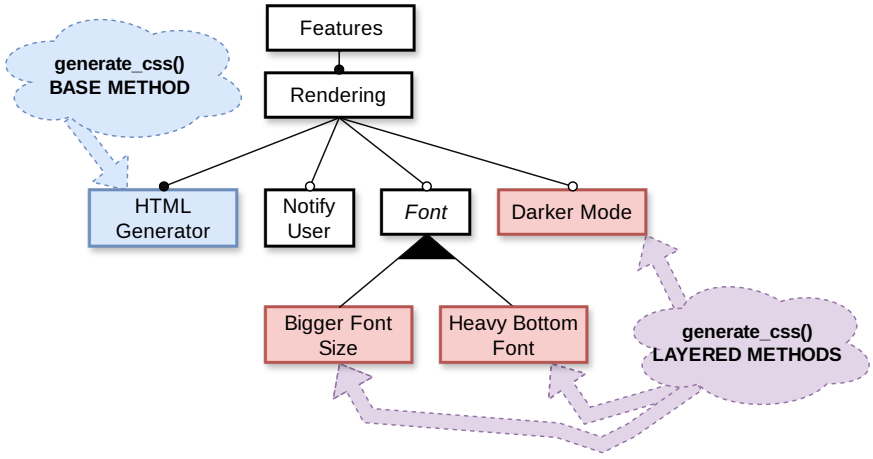


FIGURE 5.2: Diagram illustrating project architecture where the base method is not an ancestor of its layered methods.

Although we expect previously the project architecture and structure to share a similar layout, a comparison of the *Rendering* abstract feature in Figure 5.2 and the `views/` folder in Table 5.2 reveals notable differences, likely due to the lack of guidelines in designing the feature definition structure.

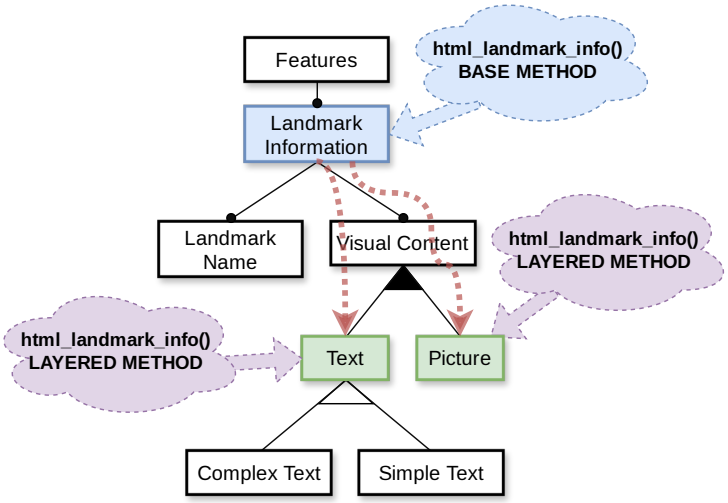


FIGURE 5.3: Diagram illustrating project architecture where the base method is an ancestor of its layered methods.

Furthermore, our RubyCOP application also provides acceptable cohesion, as depicted in Figure 5.3. This figure illustrates the logical organization of the base method preceding the layered methods, which mirrors the natural development process of features. Specifically, the base method represents the default behavior, which is subsequently refined or replaced by additional behaviors introduced by the layered methods.

Lastly, and contrary to our previous example, it is interesting to note that the architecture depicted by the *Landmark Information* feature in Figure 5.3 and the

structure represented by the `landmark/` folder in Table 5.2 exhibit a nearly exact layout. This observation leads us to conclude that, even though RubyCOP does not require partial methods to be implemented within the same feature, a well-designed project structure can effectively compensate the cohesion problem.

5.2.5 Conclusion: Rethinking FBCOP projects

Our previous analysis has led to an intriguing observation. As project size increases, FBCOP projects exhibit varying levels of maintainability and reusability compared to ContextPy3, depending on the metrics used. Notably, while the overall RubyCOP project is less maintainable and reusable than the ContextPy3 project, its class and feature definitions are more maintainable and reusable. This insight has led us to develop a new approach to characterizing and representing FBCOP projects, which we will utilize in our further analysis and solutions. We propose dividing FBCOP projects into two distinct parts:

1. The **architecture part** represents the overall understanding of a FBCOP project. It can be observed by developers by examining the project structure to understand the organization of folders and files, and by inspecting the project architecture to understand the declaration of contexts and features.

2. The **feature part** focuses on individual feature definitions.

The overall comparison between these two parts of a FBCOP project reveals that the architecture part tends to be more challenging to manage in RubyCOP, as it requires additional files and lines of code, which can negatively impact the overall maintainability and reusability of the FBCOP framework.

On the other hand, the feature part is more manageable than ContextPy3 because it can be partitioned more effectively into multiple files and features, resulting in a lower number of lines of code. Our final metric allowed us to conduct a more detailed analysis of this part by examining the cohesion gap between the implementation styles of the two frameworks. This analysis revealed that, in RubyCOP, cohesion is influenced by the initial design of the architecture part.

```
module HtmlGenerator
  module GenerateHTMLHead
        can_adapt :Server

    def generate_html(device) # BASE METHOD
      html = "<!DOCTYPE html>\n"
      head = HTML.enclose("head",
        read_template_file("head.html"), true, true)
      return html + head
    end
    end

  module GenerateCSS
        can_adapt :Server

    def generate_css(device) # BASE METHOD
      return read_template_file("style.css")
    end

    def generate_html(device) # LAYERED METHOD
      html = proceed(device)
      style = HTML.enclose("style",
        self.generate_css(device), true, true)
      return HTML.insert_between(html, "head", style)
    end
    end

  module GenerateJS
        can_adapt :Server

    def generate_html(device) # LAYERED METHOD
      html = proceed(device)
      script = HTML.enclose("script",
        read_template_file("script.js"), true, true)
      return HTML.insert_between(html, "head", script)
    end
    end

  module GenerateHTMLBody
        can_adapt :Server

    def generate_body(device)
      body = read_template_file("body.html")
      HTML.insert_between(body, "main", device.output())
    end

    def generate_html(device) # LAYERED METHOD
      html = proceed(device)
      body = HTML.enclose("body",
        generate_body(device), true, true)
      return html + body
    end
    end
end
```

FIGURE 5.4: Code snippet illustrating the `generate_html/1` and `generate_css/1` method within the *HTML Generator* feature.

```
module DarkerMode
  module DarkBackgroundMode
    can_adapt :Server
    def generate_css(device) # LAYERED METHOD
      content = "html * {color: #e8e6e3;background-color: #181
   a1b;}\n"
      default = proceed(device)
      return default + content
    end
  end
end
```

```
module BiggerFontSize
  module IncreaseFontSize
    can_adapt :Server
    def generate_css(device) # LAYERED METHOD
      content = "\n main, p {font-size: 140%;}\n"
      default = proceed(device)
      return default + content
    end
  end
end
```

```
module HeavyBottomFont
  module ChangeFont
    can_adapt :Server
    def generate_css(device) # LAYERED METHOD
      content = "@font-face {font-family: 'OpenDyslexic';
  src: local(''),
    url('./static/fonts/OpenDyslexic.otf') format('opentype');
}
html * {font-family: 'OpenDyslexic';}\n"
      default = proceed(device)
      return default + content
    end
  end
end
```

FIGURE 5.5: Code snippet illustrating the `generate_css/1` layered methods within the *Darker Mode*, *Bigger Font Size* and *Heavy Bottom Font* features.

5.3 Conclusion

The analysis of the project structures, lines of code, class versus feature partitioning, and cohesion of the partial methods based on our ContextPy3 and RubyCOP applications has led to some interesting insights.

We first observe that FBCOP project structure is more complex and has a larger number of files and folders compared to ContextPy3 for small projects and conclude that when project size increases, the complexity of reading, understanding, reusing, and maintaining a FBCOP project tends to increase more quickly than a ContextPy3 project.

Our second metric reveals that the overall FBCOP project contains approximately twice as many LOC as the COP project and notice that the feature definition in FBCOP has a similar number of LOC to the entire COP project. Building on our previous conclusion, we refine our understanding by declaring that the increased complexity in a RubyCOP project, as the project size grows, is due to the presence of additional files and folders (such as declaration, skeleton, configuration, and run files) that represent what we called the **architecture part** of the FBCOP project.

Then, we analyse the partitioning of both frameworks and find that COP developers typically implement around 4 features per class, whereas FBCOP programmers tend to complete one or two classes when implementing a feature. We refine our conclusion by explaining that FBCOP, due to its implementation in features, is better partitioned, which becomes increasingly important as the project size grows. This represent what we called the **feature part** of the FBCOP project.

Our final metric provides a more detailed analysis of the cohesion within the FBCOP feature part. We observe that a ContextPy3 project exhibits equal or better cohesion compared to RubyCOP. Our methodology for analyzing cohesion in FBCOP projects reveals that the framework can exhibit strong, weak, or acceptable cohesion between its partial methods. While some partial methods display strong cohesion, others have weak cohesion due to non-ancestral relations between features where they are implemented. A well-designed architecture part can mitigate this issue by influencing developers to implement partial methods with stronger or more acceptable cohesion.

Chapter 6

User Experience

The metrics results presented in the previous chapter provide an objective comparison of our two frameworks. However, to gain a more comprehensive understanding of how these frameworks are used in practice, it is essential to expand our analysis with feedback from developers, who, as the primary users of these frameworks, can provide valuable insights needed to reach our comparative goal.

In this chapter, we will first provide a more detailed explanation of the methodology used to collect these user feedbacks. Then, we will synthesize the responses from our interviewees and interpret them together with our previous findings. Finally, we will present our final interpretation of the comparison between our two frameworks.

6.1 Methodology

To achieve our comparative objective, we based our analysis on the Cognitive Dimensions Framework (CDF) [GP96]. We utilized the cognitive dimension questionnaire [BG07] as a foundation to develop our own questionnaire, which is described in Appendix B and available at `https://gitlab.com/gujadin-uclouvain/master-thesis/survey`.

The purpose of this questionnaire is to collect diverse perspectives from users[1] of both frameworks. Specifically, we have solicited feedback from feature-based context-oriented developers, asking them to share their personal experiences and opinions on examining our both framework applications.

Due to the limited number of active developers in the FBCOP community willing to participate in our study, we were able to conduct interviews with only three individuals who have different experiences with both frameworks. Specifically, our participants consisted of one with extensive experience in FBCOP but limited exposure to COP (#1), a beginner in both FBCOP and COP (#2) and a third who is an expert with significant experience in both FBCOP and COP (#3). The survey completed by our three participants is accessible at `https://gitlab.com/gujadin-uclouvain/master-thesis/survey/-/tree/main/out?ref_type=heads`.

6.2 Analysis & Interpretation

Following the explanation of our methodology, we will analyze the questionnaire results and providing an interpretation of them based on our previous results. Our analysis will focus on five key aspects: the time required to perform various tasks

[1] Users of the framework are developers.

and their perceptions about how project structure, project architecture, component declaration, and component definition are designed.

6.2.1 Utilization time

In our questionnaire, we requested the interviewed users to estimate the time required to complete five specific actions using both frameworks by indicating the percentage of time each action would take compared to the others. The previously talked actions were the following:

1. *Searching* for *information* within a pre-existing example

   ⇒ We asked participants to estimate the time they spend searching for information in existing examples or other sources in order to understand general or specific tasks.

2. *Translating* substantial amounts of *information* from some *other source* into the framework.

   ⇒ We asked participants to estimate the time they spend adapting information from external sources, such as other programming languages (Python, Ruby, ...) or resources (Stack Overflow, other projects, ...), in order to use it within their own project framework.

3. *Adding* small bits of *information* to a description that you have previously created.

   ⇒ We asked participants to estimate the time they spend implementing a new feature or structure within their own project framework.

4. *Reorganising* and *restructuring descriptions* that you have previously created.

   ⇒ We asked participants to estimate the time they spend refactoring existing code or documentation. I can be modifying its behavior or presentation.

5. *Playing* around with *new concepts*, without being sure what will result from these.

   ⇒ We asked participants to estimate the time they spend experimenting and trying to understand new, unfamiliar concepts on their own, without consulting external resources.

The average of their responses are reported in Tables 6.1 and 6.2.

At first, we observe that *searching how things work in examples (1)* in RubyCOP and ContextPy3 require approximately the same amount of time. However, participants exhibit different preferences. For instance, participant #1 finds ContextPy3 to be the most time-consuming for this task, whereas participant #2 has the opposite experience and participant #3 reports an equal experience. This disparity suggests that searching in examples is not a task specific to our frameworks, but rather a reflection of individual learning habits. It is possible that participants #2 and #3 are accustomed to relying on examples when documentation is not provided. In contrast, participant #1 appears to prefer *exploring new concepts (5)* independently, rather than relying on examples.

	%/Participants			
Tasks	#1	#2	#3	**AVERAGE**
(1) Search Information in Example	5	25	15	**15.0**
(2) Translate Data from Other Sources	15	25	10	**16.7**
(3) Add Information	5	5	10	**6.7**
(4) Restructure Descriptions	25	5	40	**23.3**
(5) Explore New Concepts	50	40	25	**38.3**

Table 6.1: Table showing the percentage of time spent by each participant using RubyCOP.

	%/Participants			
Tasks	#1	#2	#3	**AVERAGE**
(1) Search Information in Example	15	20	15	**16.7**
(2) Translate Data from Other Sources	30	25	60	**38.3**
(3) Add Information	15	5	10	**10.0**
(4) Restructure Descriptions	15	10	10	**11.7**
(5) Explore New Concepts	25	40	5	**23.3**

Table 6.2: Table showing the percentage of time spent by each participant using ContextPy3.

The task of *adding new elements (3)* in both frameworks is also found to be similar in terms of time consumption. While interviewees #2 and #3 report similar answers for both frameworks, interviewee #1 perceives ContextPy3 as more time-consuming. This contrast is likely due to the interviewee #1's familiarity with RubyCOP. In contrast, interviewees #2 and #3, who have respectively a weak and strong level of proficiency in both frameworks, report similar time spent for both, suggesting that their familiarity with both frameworks influences their perception of the duration of the task.

Upon examining the results for the *translation of information from other sources (2)* task, we discovered that ContextPy3 is the most time-consuming on average and for 2 out of 3 participants. We can explain this result because RubyCOP introduces more new concepts, leading to a more rigid design compared to ContextPy3, which shares similarities with programming in Python and is thus easier to adapt. We also observed that these considerations influenced our approach to programming our case study, as we initially formulated our first insights in FBCOP and then adapted them for COP.

Additionally, the process of *searching (1)* and *translating (2) information from external sources* holds significant importance particularly because the developers we interviewed do not possess advanced proficiency in the Ruby programming language.

The task of *restructuring and reorganizing previously developed descriptions (4)* required significantly more time in RubyCOP compared to ContextPy3. This disparity is largely due to the responses from participants #1 and #3, who reported twice the difference between the two frameworks. Due to their familiarity with RubyCOP, they tend to invest considerable time in refining code sections recently made to maintain clarity and cleanliness of each distinct feature. This is likely because they are aware that FBCOP was specifically designed to facilitate the decomposition of tasks, classes,

and methods, as revealed by our results in Subsection 5.2.3.

The results from our last task provided interesting insights. When *exploring new concepts (5)*, all participants agreed that RubyCOP is more time-consuming. This is likely due to its different programming approach, which focuses on implementing features rather than classes, and requires the declaration of multiple components before defining them. As a result, developers often require additional time to understand and experiment with unfamiliar practices not commonly found in other programming languages or frameworks. This finding reinforces our previous analysis which highlighted the difficulties developers face when implementing components in the architecture part of RubyCOP.

6.2.2 Project structure

This section will explore the differences between project structure (Definition 3.11) of RubyCOP and ContextPy3, based on our previous analysis.

We observe that ContextPy3 does not require any specific project structure, thereby allowing developers having a greater flexibility in organizing their code as they see fit. However, this flexibility also makes it more challenging for them to determine where to begin a new project.

The case of RubyCOP is intriguing in a distinct way. First, as observed in Subsection 5.2.2, RubyCOP imposes more rules than ContextPy3, as it requires its own project structure, which helps developers distinguish between the concepts of context and feature, as well as declaration and definition, through a clear separation of files and folders. Our interviewees found this separation to be easily understandable and not too difficult to adhere to. Moreover, they did not express any concerns or frustrations about having to work within this project structure.

However, these rules also have some drawbacks, including the lack of clear structural guidelines for defining features, as partially observed in Subsection 5.2.4. According to participant #3, a well-organized feature definition structure can be highly beneficial for developers when programming features. It means that, conversely, a poorly designed structure can lead to unnecessary mental effort, which could be avoided.

6.2.3 Project architecture

This section precise our previous discussion regarding the project architecture (Definition 3.10) in ContextPy3 and RubyCOP. We focus on developers' overall understanding and ease of use of the context-feature model.

We notice that the principle of project architecture is primarily applied to RubyCOP, as developers must explicitly declare contexts, features, and their mappings to build it, and only then they can define their features. In contrast, ContextPy3 automatically generates the project architecture similarly to languages like Python or Ruby. This generation based on the definition of classes, which is often closely related to the project structure in this case.

According to our interviewees and Subsection 6.2.1, designing the context-feature model is the most time-consuming aspect of developing with FBCOP. Moreover, creating the entire context-feature model from scratch can be challenging, as it must be designed before implementation can begin. Indeed, interviewees reveal that modeling a new project architecture requires significant effort, particularly in identifying essential features to implement and relevant contexts that activate or deactivate them.

To overcome these challenges, FBCOP creators proposed a step-by-step approach, starting with designing, the context model, followed by the feature model, and finally the mapping model, which generates the context-feature model. Interviewees #1 and #3 adopted this approach and reported that it was effective for them. Alternatively, FBCOP creators proposed a visual tool presented in Subsection 3.2.2 to facilitate the representation and modeling of the context-feature model. Participants #1 and #3 explain that this tool is really helpful to understand the project architecture and without it, it is a complex mental effort to produce. However, participant #3 noted that this solution is not adequately documented within the framework, which makes it hard for developers to even be aware of the tool's existence. Furthermore, even if they do know about it, they may still face difficulties when trying to start using it.

6.2.4 Components declaration

This section reports on the challenges and ease of use experienced by us and the interviewees when declaring contexts in ContextPy3 and features in RubyCOP.

In the case of ContextPy3, we observe that the lack of a mandatory project structure initially made by the creators means that developers must manually declare all the different layers and can place them in any file. This can result in developers dispersing the declaration of these layers across multiple files, making it more challenging to understand the overall layer structure.

As previously mentioned, FBCOP requires developers to create a project architecture. In practice contexts and features are declared in specific files manually by developers. Consequently, the flaws detected in Subsection 6.2.3 take their origin from this declaration process.

In addition to managing more files, as seen in Subsection 5.2.1, some interviewees reported that the declaration process can lead to common mistakes. One common error is omitting to declare a new feature when it is first defined. This often results in developers trying to test their newly defined feature, only to find that it does not appear as existing in the framework because it was not included in the project architecture by the framework.

Another one identified by interviewees and we concern the way feature declarations are written. As shown in Listing 3.4, depicting one way to declare features in RubyCOP, interviewees commonly make mistakes such as forgetting to mention the classes implemented by a feature or the name of the feature declared, which must match the name of the feature defined. These errors are often misinterpreted by the framework, leading to incorrect error messages for developers.

Another concern is that as project size increases, declaration files are likely to grow in size as well, leading to the same issues discussed in Subsection 5.2.2. Although this

concern was not raised by most of our interviewees, participant #3 emphasized the importance of adding comments to format these files in a way that enhances their readability. Examples of such comment formatting can be found in Listings 3.5 and 3.4.

6.2.5 Components definition

This final section presents the perspectives of our interview participants and our own views on how features are defined/implemented in RubyCOP.

Unlike the preceding sections, the definition process in RubyCOP is characterized by a lack of specific implementation rules for features. Instead, features can be placed in folders and files with arbitrary names, with the only constraint being the class name used for declaration.

In comparison to ContextPy3, our interviewees reported that identifying feature and method duplication in RubyCOP is more complicated due to the separation of features into multiple files, which, as our analysis in Subsection 5.2.4 reveals, is a consequence of RubyCOP lower cohesion.

We also observed that differentiating partial methods in RubyCOP is more difficult than in ContextPy3. This is because base methods and layered methods are not labeled in RubyCOP, unlike in ContextPy3. Furthermore, it is challenging to directly observe the control flow of partial methods because they are represented in different features across different files, unlike in ContextPy3 where partial methods are often in the same class and file.

Additionally, as mentioned in the previous subsection, we observed that the error handling system has multiple flaws, which were particularly frustrating for our interviewees and us. While both ContextPy3 and RubyCOP share flaws related to the clarity of error messages, which often report error locations related to the framework without clearly explaining the problem in the application, RubyCOP has more of these flaws due to its feature-implementation style, which deviates from traditional Ruby. Consequently, this problem generates error messages based on classes dynamically composed, making it difficult for developers to understand the issues.

6.3 Conclusion

This chapter presents the diverse opinions of our interviewees, gathered through a questionnaire that invited them to test and analyze our developed applications in both frameworks.

Our analysis reveals that RubyCOP generally requires a longer utilization time compared to ContextPy3, primarily due to its distinctive feature-based approach, which involves additional steps such as declaring context and features.

We also found that ContextPy3 offers greater flexibility in terms of project structure, but this flexibility can make it more challenging for developers to determine where to start a new project. In contrast, RubyCOP imposes a project structure that helps developers distinguish between concepts, although it may lack clear guidelines for defining features.

Our examination of the project architecture of RubyCOP shows that it needs to explicitly declare its contexts, features, and their mappings, whereas ContextPy3 automatically generates the project architecture. We conclude that designing the context-feature model is the most time-consuming aspect of developing with RubyCOP.

Regarding component declaration, ContextPy3 does not need a specific project structure, unlike RubyCOP, which requires developers to declare contexts and features in specific files. However, the way these files are created can lead to errors.

Lastly, the components definition in RubyCOP lacks of specific rules for features, making it harder to identify features and increasing the possibility of existing method and feature duplication.

Chapter 7

FBCOP Improvements

7.1 Unified Architecture & Structure

Our analysis reveals that understanding a comprehensive FBCOP project is a challenging task, as it require a dual responsibility: designing a project structure that facilitates feature definition for developers, and declaring features in the `feature_model_declaration.rb` file to enable the framework to generate the project architecture and adapt application behavior accordingly. Moreover, when FBCOP builds the project architecture, it collects all feature definition files without considering the hierarchical project structure established by developers, instead creating a separate hierarchy based on the feature declaration file, which is also designed by developers.

In this section, we present a solution inspired by the routing mechanism of the SvelteKit web framework, where web pages are routed based on the project structure [HC22b]. Specifically, the creation of a `+page.svelte` file within a folder (e.g., `myproject`) automatically generates a corresponding web page for that route (e.g., `https://localhost:8080/myproject`).

To address our problem, we adapt this principle to enable automatic feature declaration through the creation of feature files definition. In our proposed solution, developers will establish the desired project architecture within the `features` folder by designing a hierarchical structure that adheres to the following rules:

1. A folder's name corresponds to the declaration name of the feature.
2. Each folder represents either a feature or an abstract feature.
3. If a folder represents a feature, it must contain a `_def.rb` file that defines it.
4. Sub-folders within a folder represent its sub-features.
5. Each folder can contain a `_rel.rb` file that specifies the relations between its sub-features. If this file is not provided or if some sub-features are not mentioned, the corresponding sub-features are automatically assigned as Mandatory relations.

Figure 7.2 depicts our proposed structure in comparison to the existing one shown in Figure 7.1. Listing 7.1 demonstrates the implementation of a `_rel.rb` file in our solution. Notably, `@current_feature` serves as a reference to the *Language* abstract feature, which corresponds to the current folder. It is worth noting that our solution does not alter the content of feature definitions, but rather requires that the filename be changed to `_def.rb`.

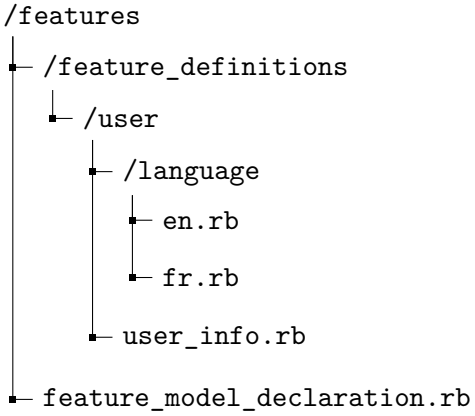


FIGURE 7.1: Diagram illustrating the existing FBCOP project structure for the *User Information* feature.

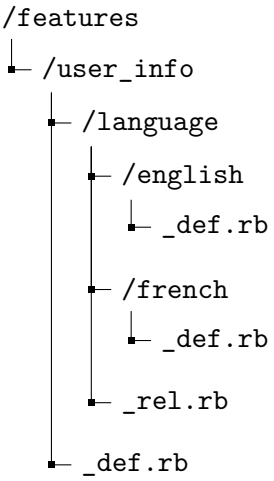


FIGURE 7.2: Diagram illustrating the proposed FBCOP project structure for the *User Information* feature.

```
# Import Needed Files

class CurrentFeatureModelRelation < FeatureModelRelation
    def initialize()
    super()

    @current_feature.add_relation :Alternative, [
            @english, @french # These are related to folder
    names existing in this file's folder
    ]
    end
end
```

LISTING 7.1: Proposed implementation of the `_rel.rb` file for the *Language* abstract feature.

Thanks to this proposal, developers can now design the project architecture concurrently with defining their features. Additionally, this approach addresses the lack of rules for feature declaration and definition. Moreover, our proposal can automatically resolve some cohesion issues, as observed in Subsection 5.2.4.

Although this proposal resolves several existing issues, it also introduces new challenges, such as an increased number of folders and files, a deeper project structure hierarchy, and an inconsistent methodology between declaring contexts and features.

7.2 Improved Editor

Given that FBCOP represents a programming paradigm distinct from existing languages, we propose the enhancement of the current source-code editor to assist

developers in creating FBCOP applications.

This editor aims to improve the documentation and ease the utilization of multiple tools such as the Context-Feature Model Visualizer [Duh+19b], which will enhance the understanding of project architecture creation. As developers add, remove, or modify components (contexts, features, ...), a dedicated editor for FBCOP featuring a built-in window that displays the visualizer alongside interactive buttons can provide real-time feedback on the changes, dynamic interaction with the tool and a more integrated and user-friendly experience. This can allows developers to directly observe how the framework will interpret the components declared to help avoiding errors. Moreover, the presence of buttons can also facilitate the automatic use of this tool when running and testing the application.

Finally, our solution can be combined with the previous one, enabling developers to manage the proposed project structure more effectively. A proper editor for FBCOP can simplify the project structure view of `_def.rb` and `_rel.rb` files. The project structure displayed in the editor can represents the project architecture, and selecting a folder (either a feature or an abstract feature) opens the corresponding `_def.rb` and `_rel.rb` files, if they exist.

7.3 Interactive Tutorial

Another solution to facilitate FBCOP programming is to provide tutorials that guide developers through the learning process. As analyzed in Subsection 6.2.1, exploring new concepts through existing examples is a time-consuming task for developers. Our proposed tutorials aim to solve this issue by offering interactive tutorials, similar to those provided by Svelte [HC22a], to explain FBCOP concepts in a more interactive and attractive manner.

This tutorial consists of a series of exercises that guide the developer through the steps required to build a FBCOP application using a simple, pre-existing example. The tutorial can be launched locally by developers and is composed of two main components: a web-based interface with exercises explanations and interactive capabilities, and a server that adapts the tutorial and the behavior of the example application based on the selected exercise. The exercises are designed to teach the following concepts through hands-on, do-it-yourself activities:

1. Introducing FBCOP and its underlying philosophy.
2. Exploring a classical project structure and linking each folder to explanations of key concepts, including features, contexts, declaration, and definition.
3. Presenting the context-feature model representation (project architecture) and its connection to previously explained concepts.
4. Declaring new contexts and explaining their impact on the context-feature model.
5. Declaring new features and explaining their impact on the context-feature model.
6. Mapping contexts to features (one-to-one, one-to-many, many-to-one and many-to-many) and explaining the resulting effects on the context-feature model.

7. Defining multiple features, declaring related classes and explaining the principle of partial methods and their usage.

8. Activating and deactivating contexts to demonstrate the behavioral adaptation process.

Eventually, we can combine our previous improvements with this one by incorporating tutorials that explain our new structure in a simple and intuitive way. Integrating our tutorial into an automatically generated environment, accessible with just one click, can significantly enhance the ease of use and make it more enjoyable to experiment with.

7.4 Error Handling

Our final improvement, considered essential by our interviewees, is the enhancement of the error handling system. As some participants noted in Subsection 6.2.5, the current error messages lack precision, making it difficult for developers to identify the location and nature of errors.

To improve error handling, we need to rework errors by first identifying the source of the problem, distinguishing between issues related to the feature part, context part, or mapping part; determining which context activation or deactivation triggered the error and which feature is affected. Next, we must determine whether the problem comes from the declaration or definition of the component. Finally, the system should generate a comprehensive and precise error message for developers by incorporate the gathered information, including the specific file and line of code, as well as a detailed error trace based on the feature structure, rather than the dynamically generated class structure.

Definition 7.1 (*Trace*)

"A trace is a finite sequence of observable actions" [Pec12a].

Additionally, it is possible to integrate these upgrades with our proposed editor, enabling it to warn developers about potential mistakes in real-time. This allows developers to catch and correct errors early on, eliminating the need to run and test the application to identify issues.

Part III

Epilogue

Chapter 8

Future Work

Before concluding this thesis, we will discuss potential improvements and future directions for enhancing our research.

Provide a quantitative analysis As outlined in our methodology of our user experiments in Section 6.1, our initial analysis was based on a limited sample of three participants. To conduct a more comprehensive quantitative analysis, it is essential to scale up our user experiments to include a larger and more diverse group of participants.

Ideally, this would involve recruiting individuals with varying levels of expertise in COP frameworks and COP in general, as well as those with no prior knowledge of COP or FBCOP. The latter group, in particular, would provide a unique perspective, as they would be encountering both paradigms for the first time, allowing us to collect insights from a completely unbiased viewpoint.

We also believe that our current analysis can be scaled up to accommodate a larger number of participants. By doing so, we will be able to validate our assumptions and findings through robust empirical evidence.

Validate our analysis for all COP frameworks Throughout this document, we have focused on comparing two specific frameworks: RubyCOP, the only existing implementation of FBCOP, and ContextPy3, a Python 3 COP framework. While this comparison provides valuable insights into the differences between these two frameworks, it does not necessarily generalize to the broader paradigms they represent. To establish a more complete analysis, it is essential to replicate our analysis across multiple COP frameworks, similar to the approach taken in previous performance comparisons of COP languages [App+09].

Prove our improvements feasibility In response to the challenges facing FBCOP identified in Chapters 5 and 6, we proposed several solutions for its framework in Chapter 7, highlighting their potential benefits. However, we did not provide evidence to demonstrate the practical feasibility of these solutions, nor did we discuss the potential technical difficulties that may arise during their implementation.

Use multiple running examples Chapter 2 represents our running example, guiding our analysis. However, by only considering this one example, we may not have considered alternative implementations or more complex applications to observe if our analysis can also work with them.

Analyse the FBCOP integrated UI FBCOP allows the possibility to developers to directly implement UI within feature definitions. To further enhance our analysis,

it would be interesting to compare FBCOP UI implementation with those from COP frameworks or from the native language if it doesn't exist.

Considering other FBCOP tools In this thesis we only focus on the Context-Feature Model Visualizer tool of FBCOP. Considering the other tools provided by FBCOP could enhance our interpretations and solutions by making them more related to the full range of FBCOP capabilities.

Chapter 9

Conclusion

This thesis seeks to investigate the maintainability and reusability of COP and FBCOP applications, with a specific focus on determining whether FBCOP is a more maintainable and reusable approach than COP. Our analysis covers both qualitative metrics and user experiences, providing an understanding through objective measurements and subjective insights.

We first observed that RubyCOP has a significantly larger number of files and folders compared to ContextPy3, which can be attributed to the requirement of FBCOP to declare the context-feature model which represent the project architecture. This context-feature model needs explicit declarations of contexts and features which we observed to be the most time-consuming task of implementing a RubyCOP application.

We also noted that RubyCOP has fewer lines of code than ContextPy3, which can be attributed to a more effective partitioning of feature definitions. In general, RubyCOP project structure is organized in a way that clearly separates the concepts of context, feature, component definition and component declaration. However, we observed that the structure of RubyCOP feature declaration files lacks of explicit rules, which can lead to increased errors as the project grows, particularly because features declaration and definition are split between multiple separate folders and files. Additionally, the absence of rules for defining features can result in poorly designed features and increase the possibility of feature and partial method duplication.

Our analysis of the cohesion of partial methods revealed the importance of the feature definition structure. Notably, we found that the cohesion of partial methods in ContextPy3 is either better or comparable to that in RubyCOP. To further investigate this observation, we developed a new methodology for RubyCOP to assess the cohesion between partial methods features. This approach enabled us to conclude that a well-designed project architecture is strongly correlated with acceptable or strong cohesion between partial methods features.

These observations highlight a critical nuance in the RubyCOP framework. Specifically, the absence of some important rules for the project structure have a negative impact on the design of the project architecture which increase the possibility of errors, code duplication, and time spent on implementation. This suggests a strong interdependence between the project structure and the project architecture. In practice, the project structure serves as a way for developers to organize their project's layout, while the project architecture serves both developers and the framework to organize the project behaviors. However, despite their shared goal of representing the overall project in terms of form and content, the project structure and architecture are

currently treated as separate entities, with distinct implementation and consideration.

These insights can lead our comparison to its final conclusion which reveals that *RubyCOP projects are less maintainable and reusable than ContextPy3 projects considering its structure and architecture, but more maintainable and reusable when considering the implementation of classes and features.*

Fortunately, we propose several solutions to address these inconsistencies by unifying the implementation of project architecture and design of project structure. Our solutions include interactive tutorials that guide developers in implementing features and designing context-feature models, thereby reducing the learning curve for new concepts. Additionally, we recommend improved documentation for the various tools, making them easier to use and learn. Furthermore, we suggest enhancements to the error handling system, which will help developers understand and avoid similar mistakes in the future, ultimately leading to more efficient and effective development.

Bibliography

[App+09] Malte Appeltauer et al. “A comparison of context-oriented programming languages”. In: *Proceedings of the 1st ACM International Workshop on Context-Oriented Programming*. COP ’09. Genova, Italy: Association for Computing Machinery, 2009. ISBN: 9781605585383. DOI: 10.1145/1562112.1562118. URL: https://doi.org/10.1145/1562112.1562118.

[App+11] Malte Appeltauer et al. “ContextJ: Context-oriented programming with Java”. In: *Information and Media Technologies* 6 (June 2011), pp. 399–419. DOI: 10.11185/imt.6.399.

[BG07] Alan Blackwell and Thomas Green. *A cognitive dimensions questionnaire*. =https://www.cl.cam.ac.uk/ afb21/CognitiveDimensions/CDquestionnaire.pdf. 2007.

[Chr18] Jonas Chromik. *ContextPy3 - ContextPy for Python3*. https://github.com/jchromik/contextpy3. Accessed: 06/12/2023. 2018.

[CM22] Nicolás Cardozo and Kim Mens. “Programming language implementations for context-oriented self-adaptive systems”. In: *Information and Software Technology* 143 (2022), p. 106789. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106789. URL: https://www.sciencedirect.com/science/article/pii/S0950584921002305.

[Des+07] Brecht Desmet et al. “Context-oriented domain analysis”. In: *Proceedings of the 6th International and Interdisciplinary Conference on Modeling and Using Context*. CONTEXT’07. Roskilde, Denmark: Springer-Verlag, 2007, 178–191. ISBN: 9783540742548.

[DMD19] Benoît Duhoux, Kim Mens, and Bruno Dumas. “Implementation of a Feature-Based Context-Oriented Programming Language”. In: *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming*. COP ’19. London, United Kingdom: Association for Computing Machinery, 2019, 9–16. ISBN: 9781450368636. DOI: 10.1145/3340671.3343357. URL: https://doi.org/10.1145/3340671.3343357.

[Duh17] Benoît Duhoux. *RubyCOP*. https://bitbucket.org/benoitduhoux/rubycop/. Accessed: 12/10/2023. 2017.

[Duh+19a] Benoît Duhoux et al. “A context and feature visualisation tool for a feature-based context-oriented programming language”. English. In: *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019)*. Ed. by Anne Etien. CEUR Workshop Proceedings. Funding Information: We are grateful to Jean Vanderdonckt for the many fruitful discussions on this topic. Publisher Copyright: Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).; 2019

Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE 2019 ; Conference date: 08-07-2019 Through 10-07-2019. CEUR-WS, July 2019.

[Duh+19b] Benoît Duhoux et al. "Dynamic visualisation of features and contexts for context-oriented programmers". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '19. Valencia, Spain: Association for Computing Machinery, 2019. ISBN: 9781450367455. DOI: 10.1145/3319499.3328240. URL: https://doi.org/10.1145/3319499.3328240.

[Duh22] Benoît Duhoux. "Feature-Based Context-Oriented Software Development". PhD thesis. Aug. 2022.

[ES95] Johann Eder and Michael Schrefl. "Coupling and Cohesion in Object-Oriented Systems". In: (May 1995).

[GP96] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.1996.0009. URL: https://www.sciencedirect.com/science/article/pii/S1045926X96900099.

[HC22a] Rich Harris and Svelte Contributors. *Svelte Tutorial*. https://learn.svelte.dev/tutorial. Accessed: 30/07/2024. 2022.

[HC22b] Rich Harris and Svelte Contributors. *SvelteKit Tutorial - Routing*. https://learn.svelte.dev/tutorial/pages. Accessed: 30/07/2024. 2022.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented Programming". In: *J. Object Technol.* 7 (2008), pp. 125–151. URL: https://api.semanticscholar.org/CorpusID:62418423.

[HT08] Herman Hartmann and Tim Trew. "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains". In: *2008 12th International Software Product Line Conference*. 2008, pp. 12–21. DOI: 10.1109/SPLC.2008.15.

[Kan+90] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Accessed: 2024-May-7. 1990. URL: https://insights.sei.cmu.edu/library/feature-oriented-domain-analysis-foda-feasibility-study/.

[LH93] Wei Li and Sallie Henry. "Object-oriented metrics that predict maintainability". In: *Journal of Systems and Software* 23.2 (1993). Object-Oriented Software, pp. 111–122. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(93)90077-B. URL: https://www.sciencedirect.com/science/article/pii/016412129390077B.

[MCD16] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. "A Context-Oriented Software Architecture". In: *Proceedings of the 8th ACM International Workshop on Context-Oriented Programming*. COP '16. Rome, Italy: Association for Computing Machinery, 2016, 7–12. ISBN: 9781450344401. DOI: 10.1145/2951965.2951971. URL: https://doi.org/10.1145/2951965.2951971.

[Pec12a] Charles Pecheur. *Concurrent Systems: Models and Analysis. Chapter 4: Semantics*. Oct. 2012.

[Pec12b] Charles Pecheur. *Software Quality Assurance. Chapter 1.5: Principles of Software Analysis*. Oct. 2012.

[PS08] Michael Perscheid and Christian Schubert. *ContextPy*. https://www.hpi.uni-potsdam.de/hirschfeld/cop/implementations/index.html. Accessed: 01/12/2023. 2008.

[PY07] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Jan. 2007. ISBN: 978-0-471-45593-6. URL: https://www.researchgate.net/publication/220694129_Software_Testing_and_Analysis_Process_Principles_and_Techniques.

[RH02] Linda H. Rosenberg and Lawrence E. Hyatt. "Software Quality Metrics for Object-Oriented Environments". In: 2002. URL: https://api.semanticscholar.org/CorpusID:59779685.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. "Context-oriented programming: A software engineering perspective". In: *Journal of Systems and Software* 85.8 (2012), pp. 1801–1817. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.03.024. URL: https://www.sciencedirect.com/science/article/pii/S016412121200074X.

Appendix A

More Figures

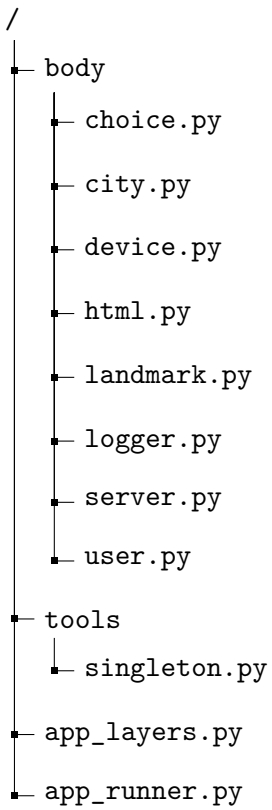


Figure A.1: Diagram illustrating our COP project structure.

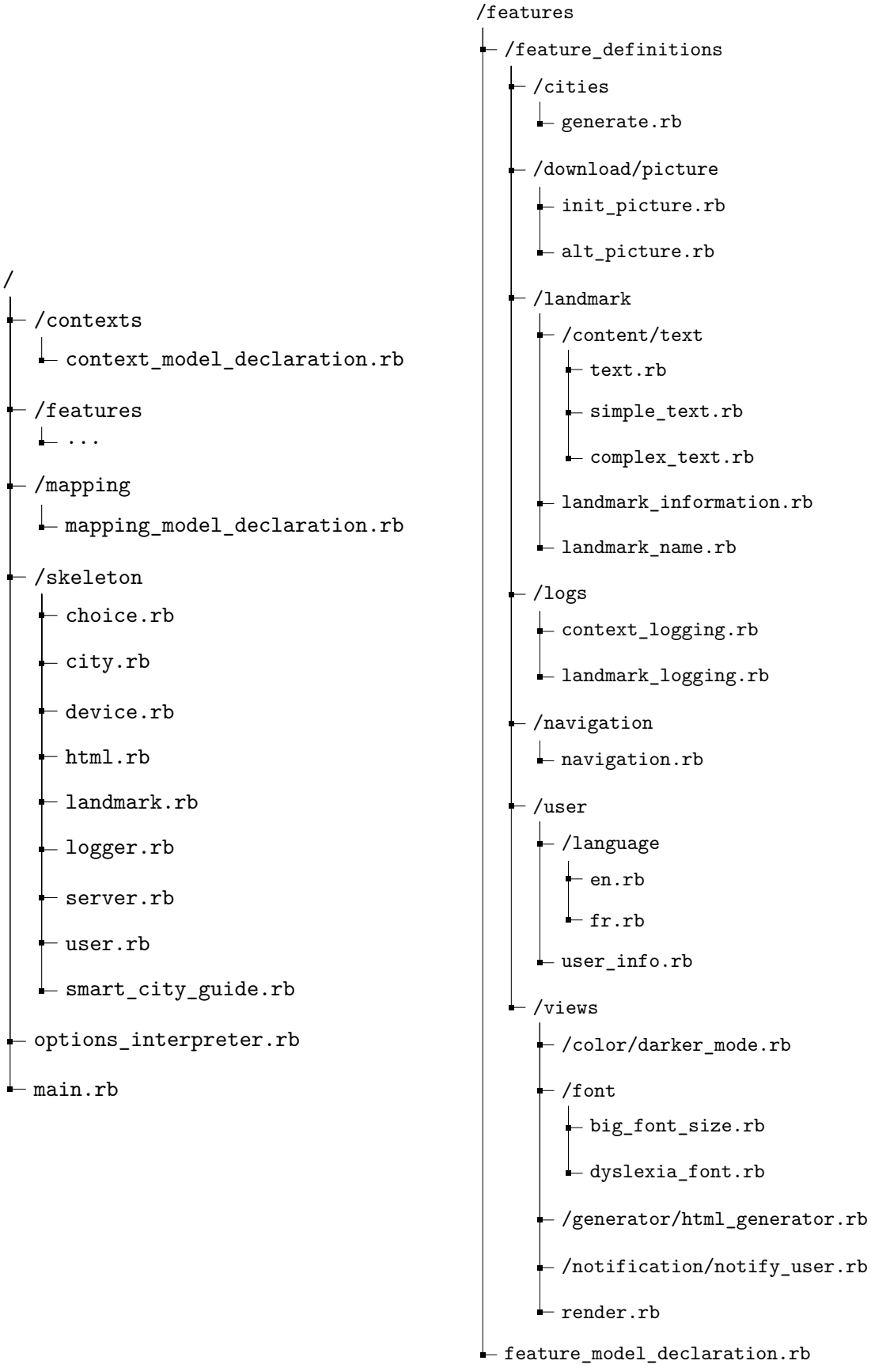


FIGURE A.2: Diagram illustrating our FBCOP project structure.

Appendix B

Template Survey

UCLouvain

École polytechnique de Louvain

Master Thesis

FBCOP vs COP

A Survey to validate the adaptability, evolvability and maintainability of (feature-based) context-oriented programming frameworks

---

Author:
Guillaume JADIN

Advisor:
Axel LEGAY

Assistant Advisor:
Benoît DUHOUX

Academic year:
SINF2MS/G
(2023 – 2024)

Reader's Personal Information:

First Name: ***Last Name:***

Email:

Contents

Part I
Readme

1 About this document

This document is greatly inspired by the work of Alan Blackwell and Thomas Green about Cognitive Dimensions Framework and more precisely a transcript of their document "A Cognitive Dimensions Questionnaire" written in February 2007. Some of the questions in this document are reused from the cited document, others are modified or completely new.

It aims to collect different points of view about the *Feature-Based Context-Oriented Programming* (FBCOP) and the *Context-Oriented Programming* (COP) frameworks utilization. More precisely, to understand what are the pros and cons of each of them in term of maintainability, evolvability and adaptability and compare them to finally draw conclusions about the strengths and weaknesses of each framework, ultimately proposing enhancements for FBCOP primarily, as well as COP.

With this file comes one archive files which contains the applications of each framework analysed in this document (both do exactly the same actions) and a video showing the applications render.

This document contains **forms**, please use these to answer questions.

2 About the applications

The analysis of these two frameworks will be conducted via an application named **Smart City Guide** which has been implemented in both of them. The first application is in *Ruby* and use the *FBCOP framework* named **RubyCOP**. The second one is in *Python 3* and use a *COP framework* named **ContextPy3**.

The application generates an HTML page for a user who visits a particular city giving information about the nearest landmark (point of interest) relative to the position of the user. Depending on the user parameters modification at runtime, the application will adapt its interface (the HTML page) and reload it dynamically when a modification (of context) is made. At this state, the application only run through a sequence of changing contexts and then stops.

The appendixes A (context model), B (classes list) and C (feature model) correspond to screenshots of the FBCOP tool which may help you better understand the overall structure of these applications.

3 About this analysis

To carry out this analysis you must follow the following instructions:

1. Watch the small video which explains how the application result works (no need to configure, install or run the applications)
2. Analyse the FBCOP application source code and answer to the questions at chapter II.
3. Analyse the COP application source code and answer to the questions at chapter III.
4. Send this document completed at guillaume.l.jadin@student.uclouvain.be.

Note that there are a lot of questions. So, it is possible to pass some of them if you don't have any idea on how or what to answer.

Part II
Feature-Based Context-Oriented Programming

1 Background information

What is the name of the FBCOP framework?	
In which language is implemented this framework?	

1.1 RubyCOP

How long have you been using this framework?	
What purpose do you use it for?	
Do you consider yourself proficient in its use?	
Have you used other FBCOP frameworks? (If so, please name them)	

1.2 Ruby

How long have you been using this language?	
What purpose do you use it for?	
Do you consider yourself proficient in its use?	

2 Utilization time of FBCOP frameworks

The following questions aim to understand the time needed to the user to perform some specific actions.

If you haven't ever programmed in RubyCOP, express your experience in another FBCOP framework (if such one exists).

If you haven't ever programmed in any FBCOP framework, estimate the time you may need to perform those actions by looking at the application source code.

When using FBCOP, what proportion of your time (as a rough percentage) do you spend :	
Searching for information within a pre-existing example	%
Translating substantial amounts of information from some other source into FBCOP	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new concepts, without being sure what will result from these	%
TOTAL	%

3 Personal Opinions of RubyCOP

In this section, we ask you to provide your personal opinion on the FBCOP framework RubyCOP utilization. With these questions, we aim to understand other points of view about the readability, evolvability of the code and how much effort you think you will put into it. You must answer with your own utilization of RubyCOP or/and the application example provided.

3.1 Is it easy to read/write a RubyCOP application?

How easy is it to see or find a specific content in the source code? Why?	
What kind of programming components (class, feature, context, method, ...) are more difficult to see or find? Why?	
How easy is it to compare or combine different programming components at the same time? Why? Example 1: *Combine two different features into one.* Example 2: *Compare three classes to observe which one is the most important.*	

When you need to make changes to your previous work, how easy is it to make changes? Why?	
Are there particular changes that, are more difficult or especially difficult to make? Which ones?	

When reading the source code, is it easy to tell what each feature, file, folder or class in the overall scheme are for? Why?	
Are there some features, methods or classes that are particularly difficult to understand? Which ones?	

3.2 Does it require a lot of effort?

Does RubyCOP let you express what you want reasonably briefly, or is it long-winded? Why?	
What sorts of programming components (class, feature, context, method, mapping, ...) take more space to describe?	

What kind of programming components require the most mental effort when you interact with a RubyCOP application?	
Do some programming components seem especially complex or difficult to work out in your head (e.g. when combining several components)? What are they?	

Do some kinds of mistake seem particularly common or easy to make? Which ones?	
Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?	

3.3 Does it have implementation rules to respect?

How easy is it to stop in the middle of features implementation, and check your work so far? Can you do this any time you like? If not, why not?	
Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?	
Can you try out partially-completed versions of an application? If not, why not?	

When you initialize a new FBCOP application, can you start implement it or does RubyCOP force you to think ahead and make certain decisions first?	
If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?	

3.4 Is it easy to debug?

Is RubyCOP clear in the errors it provides? If not, please give examples.	
Is it easier to debug code with print than with a debugger? (whatever such debugger exists or not) Why?	

3.5 Does the structure of a RubyCOP code influence its use?

Is it needed to add comments to clarify what the code is doing or does RubyCOP make it understandable as it's own? Why?	
Do you ever add specific formatted comments to clarify or emphasise the code structure? If yes, does it help the source code readability?	
Is the use of specific formatted comments mandatory or just highly recommended by the designer?	

Does RubyCOP provide any structure (file, folder, ...) already made to create new contexts and/or features more easily, clearly or succinctly? Do you think it is really helpful? Why?	
Does RubyCOP insist that you start by defining new contexts and/or features before you can do anything else? If you don't follow this rule, is it more difficult to program?	

Does the FBCOP framework impose a specific project structure? How is it defined?	
Does this structure help to better read and understand the application code? If so, explain why.	

3.6 Is it intuitive?

Do you find yourself programming in RubyCOP in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?	
Is it difficult to use RubyCOP the way the designer originally intended? Why?	

Can you think of obvious ways that the design of RubyCOP could be improved? What are they?	

Part III
Context-Oriented Programming

1 Background information

What is the name of the COP framework?	
In which language is implemented this framework?	

1.1 ContextPy3

How long have you been using this framework?	
What purpose do you use it for?	
Do you consider yourself proficient in its use?	
Have you used other COP frameworks? (If so, please name them)	

1.2 Python 3

How long have you been using this language?	
What purpose do you use it for?	
Do you consider yourself proficient in its use?	

2 Utilization time of COP frameworks

The following questions aim to understand the time needed to the user to perform some specific actions.

If you haven't ever programmed in ContextPy3, express your experience in another COP framework (if such one exists).

If you haven't ever programmed in any COP framework, estimate the time you may need to perform those actions by looking at the application source code.

When using COP, what proportion of your time (as a rough percentage) do you spend :	
Searching for information within a pre-existing example	%
Translating substantial amounts of information from some other source into COP	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new concepts, without being sure what will result from these	%
TOTAL	%

3 Personal Opinions of ContextPy3

In this section, we ask you to provide your personal opinion on the COP framework ContextPy3 utilization. With these questions, we aim to understand other points of view about the readability, evolvability of the code and how much effort you think you will put into it. You must answer with your own utilization of ContextPy3 or/and the application example provided.

3.1 Is it easy to read/write a ContextPy3 application?

How easy is it to see or find a specific content in the source code? Why?	
What kind of programming components (class, feature, context, method, ...) are more difficult to see or find? Why?	
How easy is it to compare or combine different programming components at the same time? Why? Example 1: *Combine two different features into one.* Example 2: *Compare three classes to observe which one is the most important.*	

When you need to make changes to your previous work, how easy is it to make changes? Why?	
Are there particular changes that, are more difficult or especially difficult to make? Which ones?	

When reading the source code, is it easy to tell what each feature, file, folder or class in the overall scheme are for? Why?	
Are there some features, methods or classes that are particularly difficult to understand? Which ones?	

3.2 Does it require a lot of effort?

Does ContextPy3 let you express what you want reasonably briefly, or is it long-winded? Why?	
What sorts of programming components (class, feature, context, method, mapping, ...) take more space to describe?	

What kind of programming components require the most mental effort when you interact with a ContextPy3 application?	
Do some programming components seem especially complex or difficult to work out in your head (e.g. when combining several components)? What are they?	

Do some kinds of mistake seem particularly common or easy to make? Which ones?	
Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?	

3.3 Does it have implementation rules to respect?

How easy is it to stop in the middle of features implementation, and check your work so far? Can you do this any time you like? If not, why not?	
Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?	
Can you try out partially-completed versions of an application? If not, why not?	

When you initialize a new COP application, can you start implement it or does ContextPy3 force you to think ahead and make certain decisions first?	
If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?	

3.4 Is it easy to debug?

Is ContextPy3 clear in the errors it provides? If not, please give examples.	
Is it easier to debug code with print than with a debugger? (whatever such debugger exists or not) Why?	

3.5 Does the structure of a ContextPy3 code influence its use?

Is it needed to add comments to clarify what the code is doing or does ContextPy3 make it understandable as it's own? Why?	
Do you ever add specific formatted comments to clarify or emphasise the code structure? If yes, does it help the source code readability?	
Is the use of specific formatted comments mandatory or just highly recommended by the designer?	

Does ContextPy3 provide any structure (file, folder, ...) already made to create new contexts and/or features more easily, clearly or succinctly? Do you think it is really helpful? Why?	
Does ContextPy3 insist that you start by defining new contexts and/or features before you can do anything else? If you don't follow this rule, is it more difficult to program?	

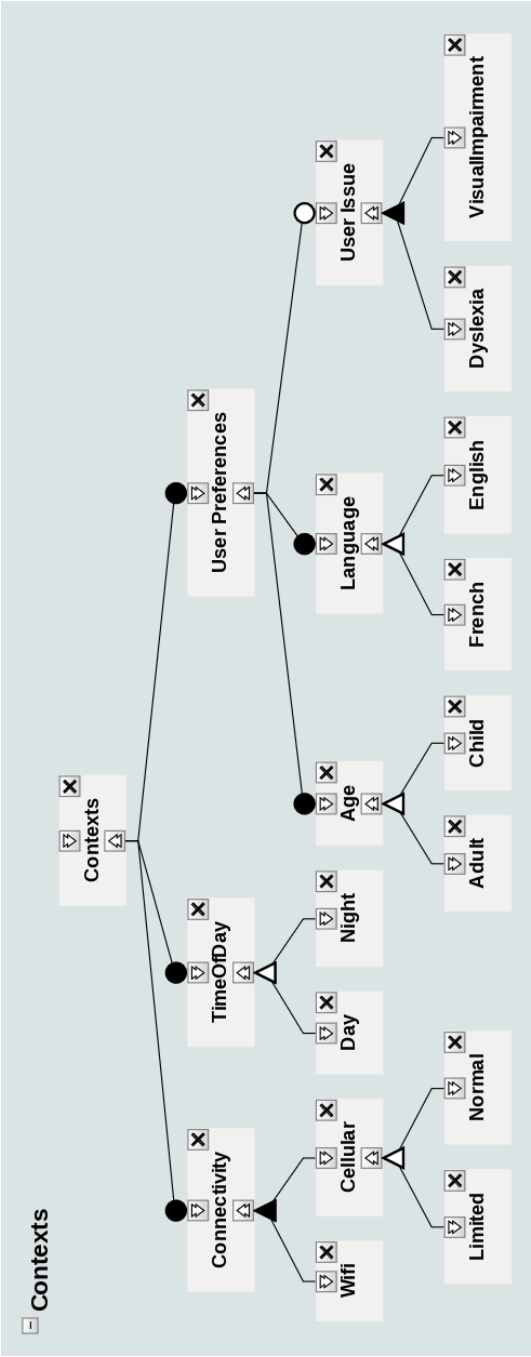
Does the COP framework impose a specific project structure? How is it defined?	
Does this structure help to better read and understand the application code? If so, explain why.	

3.6 Is it intuitive?

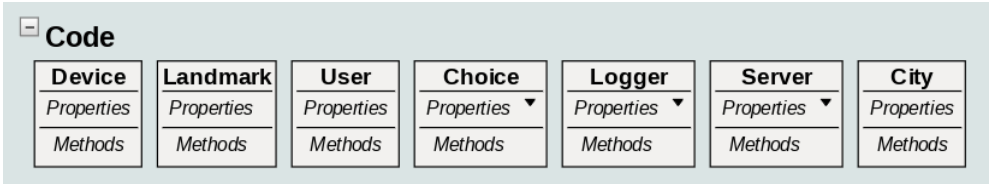
Do you find yourself programming in ContextPy3 in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?	
Is it difficult to use ContextPy3 the way the designer originally intended? Why?	

Can you think of obvious ways that the design of ContextPy3 could be improved? What are they?	

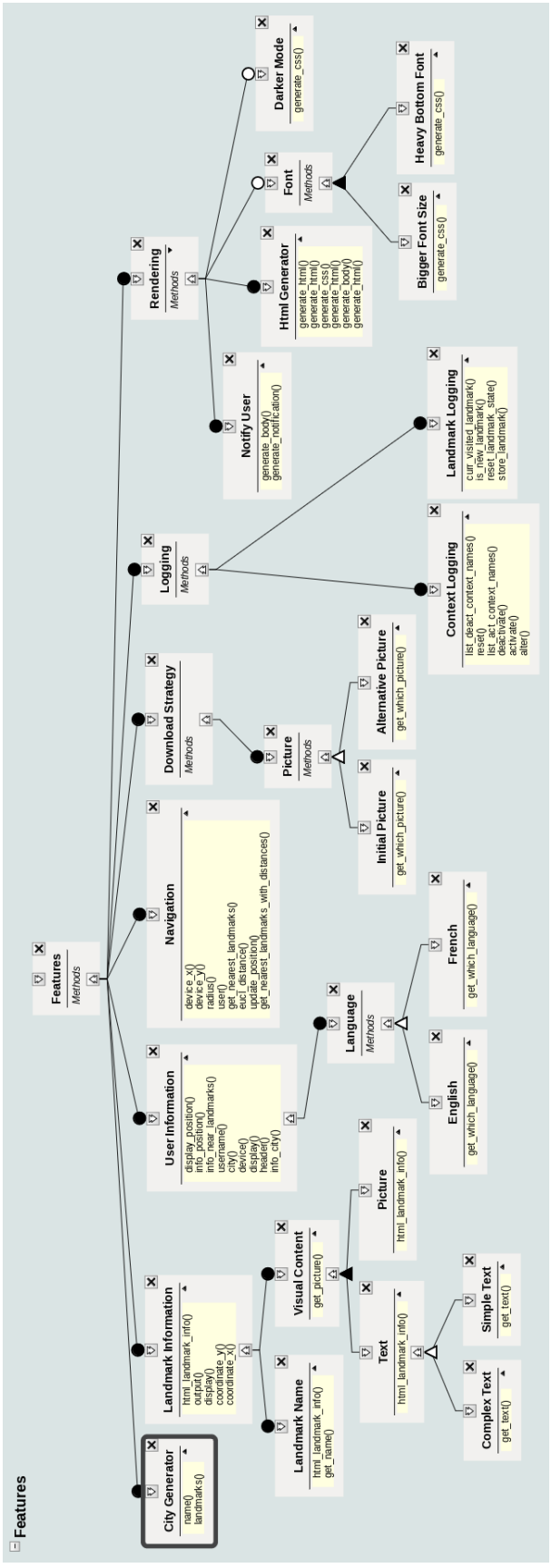
A Context model



B Table of implemented classes



C Feature model with implemented methods



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