UCLouvain

epl

École polytechnique de Louvain

Adaptive applications with multimodal user interfaces

Authors: **Erwan Delhove, Hoyien Tsang**
Supervisors: **Kim Mens, Bruno Dumas**
Readers: **Benoit Duhoux, Jean Vanderdonckt**
Academic year 2021–2022
Master [120] in Computer Science

Acknowledgments

We'd like to thank our supervisors, Prof. Kim Mens, Prof. Bruno Dumas for their input, ideas and guidance they have given us throughout this entire academic year.

We'd like to thank Benoît Duhoux, for his strong technical guidance and advises he gave us on the use of RubyCOP framework and without whom this work would not have been possible.

We'd like to thank Prof. Jean Vanderdonckt who accepted to be our reader and for the time he spent to read this master's thesis.

Finally, we'd like to thank our families and friends who supported us during all our studies.

Abstract

Context-oriented programming is a programming paradigm that allows an application to adapt its behaviour depending on the surrounding context rather than conventional conditions. This new mechanism allows developers to create a flexible adaptable application without having to be bottle-necked by the architecture.
But how powerful would it be to mix this paradigm to the multimodal user interface, as nowadays, the most popular applications are not even running on a computer but on a small devices. This is what has been explored throughout this academic year.

This master thesis shows that it is possible to integrate multimodalities in a FBCOP language. Our solution and implementation are based on the programming framework RubyCOP, written by B.Duhoux. Within this programming framework, we succeeded to integrate different new possible interactions such as vocal interactions and gesture recognition through a flexible architecture that would allow anyone who would like to add another modalities to do so smoothly.

First, the voice input interaction has been implemented using the google-speech API, the gesture recognition has been possible thanks to a Leap Motion, and the voice output has been achieved through the rubyflite library.
Then, after integrating the interactions within RubyCOP, we needed to challenge the solution via a study case following the FBCOP methodology. The main goal was to create this application using our solution such as it allows multimodal interaction depending on the current context.
We implemented a study case which is a smart meeting application where a user can create meetings, send messages or add participants. This application adapts its interaction mode, depending on the noise level around, user condition and the availability of input or output devices.
Eventually, this application has been tested using a test suite generated by an external tool written by P. Martou.

This work proves that it is possible to create an implementation of FBCOP that can change vary its interactions depending on the current environment, user's state or device's availability.

Contents

Chapter 1

Introduction

In this chapter, we introduce this thesis by first explaining its context. Then we describe the main problem to solve and the motivations behind it resolution. We then give an explanation on how we decided to approach this problem through existing technologies and tools. Thereafter, we describe the work division as its various contributions. Finally, we finished by a small road map to help the readers to understand more easily the global division of this thesis.

1.1 Context

Nowadays, there are many software that now supports a number of diverse possible interactions also called modality (e.g. we can interact with our phone using voices, or we can use gestures to interact with a Kinect or a leap motion, ...). Those interfaces are really interesting because they use interactions, which are way more human than what we used to work with (typically classical keyboard and mouse interface). For instance, when you ask a colleague to "close that window" nothing is more natural than just pointing at it just to be sure the interlocutor understands well which window you are talking about. The main topic of this master thesis, is the adaptation of such modalities where users could favour one interaction mode depending on their surrounding environment and needs. As an example, imagine yourself in a car driving at 200km/h on the highway. It would be really dangerous to interact with your mobile device via the classic means (touch and text) as you would be distracted from the road and increasing therefore the risk of accidents. It would be safer to interact with your app as if you were talking to someone sitting next to you rather than struggling to click and write text using your hands.

1.2 Problem and Motivations

The main problem of implementing such a multimodal application in a classical programming way is the complexity generated by the introduction of all these different ways of interacting with the application. Indeed, all theses modalities will be handled with a certain amount of condition which can lead to a more complex and larger code. Plus, as any developer should know, a more complex code becomes harder to read and to understand and thus will be harder to maintain or evolve. But imagine now doing the same application by introducing more constraints in order to have a dynamic application that can adapt itself to various constraints such as environment constraints(noise, humidity, ...), user constraints (preference, capacity, ...) or even system constraints (battery level, ...). Such application could become a nightmare to be improved and/or maintained. This is why we are looking to test the potency of creating such application in a more dedicated system which is FBCOP/"Feature Based Context-Oriented Programming".

1.3 Objectives and Approach

The main objective of this master thesis is to show it is possible to create an adaptive application which is able to interact in various ways depending of the current context in which it is used but also the ease of doing so in the given paradigm. For this matter, we decided first to find an interesting study case where the multimodal interaction brings an interesting value. Once this study case was found, we would use the Context-Oriented Programming methodology to model the study case and then implement a coherent scenario where diversified interactions are used. Most of the implementation will be done on top of the framework called RubyCOP, written by Duhoux, which already allows the developer to create basic adaptive applications that can be modified through a context simulator. Therefore, we can focus ourselves more on the introduction of the multi-modality part in such a framework.

The amount of handled interaction will naturally be determined by several factors: First the technological factor, as some interaction technologies are not compatible with the existing framework. Then, the study case limitation, because some interaction does not fit in some use cases and therefore it would not make any sense to spend more time handling it. And eventually, the material factor where it might not be possible to use certain interactions because of the lack of material to implement and test such modality.

In order to test the viability of our solution, we will implement the use case of a smart meeting application. In this meeting application we will be able to create

meetings, add one or more participant in that meeting, send messages in a chat room, and see the history of those meetings. This application will be able to adapt its interaction with the user depending on the surrounding environment and/or the user status.

1.4 Work division

A good point to emphasize about this master thesis is that it has been written by two collaborating students, Erwan Delhove and Hoyien Tsang. The entire work has thus been divided into two parts: Erwan for the front end part and Hoyien the back end part. Nevertheless, this division did not prevent both students from exchanging and collaborating through pair programming on different issues encountered on each side such as bugs, implementation difficulties or potential improvements. The individual contribution of each student is described in detail in the followings subsections.

1.4.1 Front end and Gesture handling

Erwan mainly worked on the development of the UI part of the application, because of its interest in designing clear and intuitive user interfaces to facilitate communication between the user and application. The other main part he worked on was the introduction of gestures as an input modality in the use case application (*Smart Meeting*). Due to the material limitations (only one Leap Motion available) and home working constraints, he was the only one able to work and ensure the gesture part implementation of the application.

1.4.2 Back end architecture and voice handling

Hoyien mainly worked on the architecture and the different choices regarding how to implement the different modalities, such as gesture or voice, he ensured to add those features in such a way that other kinds of modalities could be included in the future without technological limitations. The other main part alongside the back end architecture was the introduction of the voice modality in the use-case application (Smart Meeting). He also worked on the design of the context-feature model of the study-case at the very beginning on the project and made it evolve based on the different feedback given by the professors.

1.5 Contributions

Thanks to the work achieved in this thesis, several contributions in various fields were made. Theses various contributions are described a bit more in detail in the following subsections classified according to their importance.

1.5.1 Multi-modality in FBCOP

Our main contribution for this thesis is the addition and support of various modalities (voice, gestures and vocal feedback) in a recent FBCOP or "Feature Based Context Oriented Programming" framework called "RubyCOP". This addition should now allow future developers to support these different modalities in any of their FBCOP applications. Furthermore, our approach should allow the framework to evolve more easily in order to increase the number of supported modalities in the future (such as eye gaze detection, machine gesture feedback, ...) thanks to the technology-free nature of our architecture, this is explained in detail in the section 4.2.

On the multi-modality side, the work performed here helps to prove the possibility to create an application capable of having and using several modalities in a FBCOP way. The easiness of introducing new modality into the existing code as the dynamism of the modalities introduction influenced by different contexts such as user input, user preferences or surrounding environment will be demonstrated further in 6.

1.5.2 FBCOP framework improvement

Another contribution made possible by this thesis is mainly related to the improvement of the FBCOP framework called "RubyCOP". Indeed, since we had to create a large context-aware application in this framework with the help of an FBCOP methodology, we were thus able to test its overall performance, limitations and benefits as discussed later in section 7.3.

1.5.3 UI

In order to implement a proper use case, we had to create a multimodal interface using the framework's UI. This framework's UI is developed on top of FXRuby which itself is a UI library based on the FOX Toolkit, an open source UI library made by Van der Zijp in C++. However, the main issues regarding the FXRuby library are its age and the lack of documentation (there is only one book that shows some good use of it). Up until now, only a basic UI has been developed so far for case studies implementation in the FBCOP framework. However, in this

thesis, we managed to develop a case with a rich and complex UI showing many of the various possibilities offered by this library. The changes or additions that were made for the framework's UI part as the various difficulties we encountered when using it will be described further in 7.3.1.

1.5.4 Test Scenario Generator for FBCOP

Another contribution of this thesis is the usage and validation through our case study of an pretty recent test scenario generator for FBCOP applications [16]. We had the chance to use and test this generator for our own scenario. We have thereby seen its beneficial impact on test scenario generation but also discovered some issues regarding context switching in our usage pattern that will be described later in 6.4.

1.6 Roadmap

The rest of this thesis is structured as follows: The following chapter provides all the necessary background material about the FBCOP framework used throughout this work by explaining its key mechanisms, structures and tools with the help of some example of our application. The two next chapters describe the problems coming from the introduction of various modalities and the solutions that could be made to prevent them in an abstract level. Afterwards, we describe the solution that was implemented in the FBCOP framework to allow such multimodal integration by showing an actual piece of code to offer a better illustration. Subsequently, we provide a validation of our implementation through the study case of a smart meeting application. Subsequently, we discuss about the benefices and downsides about the usage of FBCOP approach for multimodal application, the quality of our implementation as well as the limit of the current framework. It is later followed by a chapter with all the potential future works that could achieved base on this work. Finally, we end this thesis with conclusion.

Chapter 2

Background Material

This chapter covers the different background materials needed to understand the work that has been done and which provide the basis for the rest of this thesis. More specifically, we describe the differences between the FBCOP and COP approach, followed by a detailed explanation of the RubyCOP framework, as well as the accompanying tools and methodology provided by that framework that allowed us to achieve the results of this thesis.

2.1 FBCOP vs COP

COP, which stands for "Context-Oriented Programming", represents a paradigm that defines a way to implement applications that can adapt dynamically to the current *context* where the system operate. Such an approach enables the definition of different application behaviours depending on various contextual elements such as the environment, user preferences or the system's state. FBCOP, which stands for "Feature Base Context-Oriented Programming", is a particular approach to COP where features and contexts are separated from each other into specific entities that are linked together using a mapping.

2.1.1 Contexts and Features

As explained before, an FBCOP application consists of *"Contexts"* and *"Features"*. The *"Contexts"* are used to describe the environment within which the application runs whereas the features describe all the functionalities it offers. In this approach, a *context "informs both recognition and mapping by providing a structured, unified view of the world in which the system operates"* [4] and a *feature* can be described as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"* [14]. In order to link these concepts mutually, a

mapping between the features and the contexts has to be defined. It is in the mapping where developers tell the application how it should react depending on the current environment. In other words, it is telling which feature is related to which context and thus when it should be activated.

2.1.2 Modelling

In FBCOP, at least two models are required to represent an application, one for the contexts and the other for the features. Additionally, it is possible but not mandatory to add a class diagram that will represent the classes in which the features will be added. Each model will express the division of each part as a tree structure and do so through relationships and dependencies. Context and feature model are represented via a tree data structure where a node can have dependencies or relationships between each other. In this model, a node is used to represent an *entity*. Theses nodes can either be *concrete* or *abstract* where a concrete node is a real entity implemented in the application and abstract node is an entity used to regroup multiple entities of a similar type. Constraints are a type of link between child entities and its parent to describe their interactions. Here are all the different existing constraints which can be used in a FBCOP model:

Alternative : relation that regroups a set of elements where **only one** entity has to be selected when the parent is.

Mandatory : relation used to emphasize the importance of a given entity and its **need** to be activated when the parent is.

Optional : relation that defines a entity that **can** be activated when the parent is.

Or : relation used to tell that **at least one** entity or more have to be selected in the given set of features when the parent is.

Dependencies on the other side are used to prevent incorrect/undesired states to exist by forcing/preventing the activation of given features. These dependencies can exist in the model as internal dependencies or between both model as external dependencies. They are usually expressed through simple propositional logic next to the model. In figure 2.1, one can observe an example of such a model, the style is heavily inspired by a previous work made by Hartman[12].

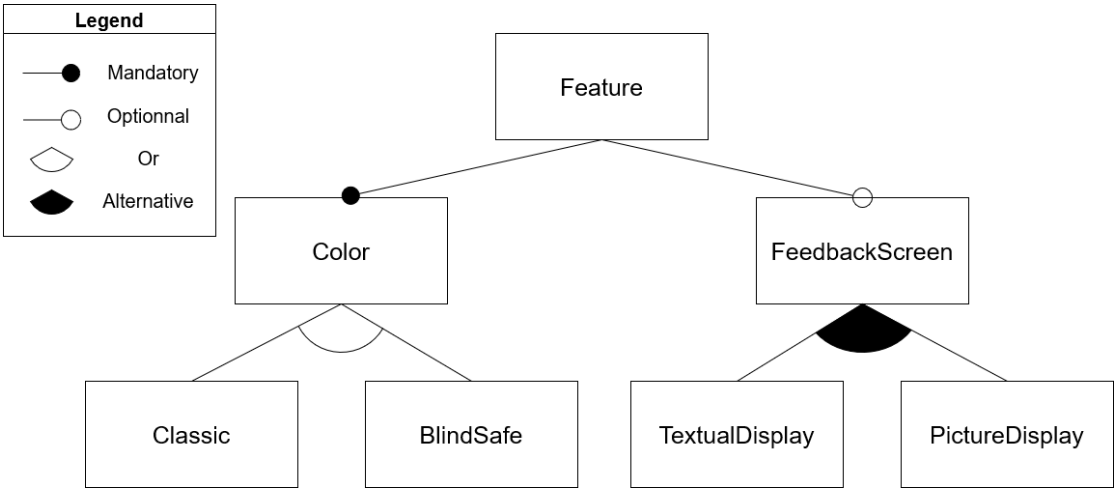

Figure 2.1: Example of feature model

2.2 RubyCOP

We wanted to prove that it was feasible to build a multimodal feature-based context-oriented application and that this modality integration fits very well in such a COP structure. For our approach, we decided to work using an existing FBCOP framework called RubyCOP developed by Duhoux [9]. In the following, we explain the structure as well as the different mechanism used to allow features to adapt to contexts.

2.2.1 Global Structure

The RubyCOP framework used for building FBCOP applications is structured as follow:

apps is the directory which contains the implementation of an FBCOP application. This is the location where a developer who wants to use the framework has to write its applications.

src contains all of RubyCOP framework's code such as the UI creation and the Feature Context handling.

tests contains all the existing unit testing used by Duhoux to test and verify the default mechanisms of the framework such as features (de)activations based on context change, model creation validity, ...

tools contains all the auxiliary tools of the framework to help a developer build such an FBCOP application. There are two visualisations tools and a context simulator. These tools will be described in more detail in section 2.2.5.

Gemfile contains all the ruby dependencies needed for the framework and tools.

2.2.2 Typical Application Structure

In this section, we are going to describe the typical structure followed by any RubyCOP application. Usually, they are divided as follow:

contexts is a folder that contains the context declaration file. An example of the content of such a file is shown on listing 1. Each context is usually defined in a private method to ease code clarity. Each context can be defined as an abstract or a concrete context depending on models need as seen in 2.1.2 followed by a variable name that will be used to link children and its name. Constraints can be defined through `context.relation` followed by the constraint type as seen in 2.1.2 and the list of concerned children in an array. The main application context is called `@root_context` and needs to be bound to other context in order to make them effective.

```
def initialize()
        super()
        _define_user_context()
        ...
        @root_context.relation :Mandatory, [@user, @environment,
        ↪  @connectivity, @outputs, @inputs]
        @root_context.relation :Optional, []
end

private

def _define_user_context()
        abstract_context :@user, "User"
        _define_audition_context()
        _define_vision_context()
        @user.relation :Mandatory, [@audition, @vision]
end
```

Listing 1: Example of context model declaration

features is a folder containing all files related to the features of the application. The structure inside the folder can be customized to the developer's needs

and will not impact the way that the files are imported as long as the feature model file contains the following line:

```
Dir[File.dirname(__FILE__) + "/feature_definitions/**/*.rb"].each {
↪  |file| require file }
```

Each feature is first described in a dedicated file before being linked to others in the feature model. An example of model definition can be seen in listing 24 in appendix A. In this example, one can see that the feature is actually declared as module. In this module, which acts as parent, there is another module called *UILayout*. This module UILayout is a sub-module that will be used to define the UI part of the general feature and thus contains all code related to the UI. One can also see the presence of new keywords such as `can_adapt`, `set_prologue` and `set_epilogue`. Theses keywords are mechanisms used by the framework that will be explained further in section 2.2.3.

mapping is a directory that contains all files related to the mapping (explained here 2.1.1). An example of mapping content is shown at listing 2 One can see the different elements that play an important role in its definition. First, we need the contexts and features model previously declared in the file described above. Then, we create a dictionary that will link a given set contexts to the corresponding set of features. By doing so, when all the given contexts will be activated, the corresponding behaviours will be triggered. For example here in figure 2, we can see that the context `daltonian` alone is linked to the feature `blindsafe`.

```
def initialize()
        contexts = AppContextModelDeclaration.instance
        features = AppFeatureModelDeclaration.instance
        @mapping = {
                # colors
                [contexts.daltonian()] => [features.blindsafe()],
                [contexts.perfect()] => [features.default_color()],
            ...
        }
end
```

Listing 2: Example of mapping declaration

skeleton contains all class files to which the mandatory and optional features will be added as well as the main class of the application(usually called *app_name.rb* where "app_name" is the name of the application).

main.rb is the file used to run the application. It can be run as following with our without options that will define the application mode:

```
ruby main.rb #-d with option
```

These different modes are defined in the `options\_interpreter.rb` which is explained below.

options_interpreter.rb is a file that defines two launch modes for the application (the *dev* and the *proto* mode). The *dev* mode is used to tell the application to send data, such as contexts or features activation, that are actually only used by features visualizer tools to display the application model. It can also be used as a test mode in order to only show given elements that should not be part of the final version such as logs or test widgets. On the other side, the *proto* mode is the default mode of the application and thus the one used by the final user.

2.2.3 Mechanisms

The RubyCOP framework is based on different mechanisms that allows an application to adapt its features depending on the current context. These mechanisms need to be understood by any developers that would like to use this framework in order to properly design FBCOP application.

can_adapt & feature model

In RubyCOP, each feature is divided into a set of modules that will be added to a designated class dynamically when the linked context gets activated. The `can_adapt` is a keyword used to tell to the compiler which is the designated class(es) that can be adapted by the given module. This keyword can be used multiple time for a same class in order to extend its definition dynamically depending on current activated contexts. As can be seen in listing 24 and 25 in Appendix A, both features expand the definition of the `SmartSession` class. In order to create the given adapting feature, you first need to register it in the feature model. The feature model is defined in `feature_model_declaration` file and will express all the relations between the existing features described previously in 2.1.2. It is essential to point out that an identical feature could adapt one or more class in the same module. However, a declared feature, which is a feature declared in the feature model, can only adapt one class at time meaning that you need to define two features in the model for a same module that updates two different classes (i.e. one for each class). For example, one can see in the listing 3 the `can_adapt`

is linked to `CreateMeetingView` and `MeetingView`. In order to instantiate the change on both class, two separate features have been created as one can see in listing 4. By doing so, each feature will have the same definition of the `update()` method with only one actual definition in the source code. The advantages and downsides of such reusability are explained in 7.3.2

```
  module View

    can_adapt :CreateMeetingView, :MeetingView

    ...

    def update(type, element)
      case type
      when :new_participant; add_participant(element);
      when :delete_participant; delete_participant(element);
      else
        proceed(type, element)
      end
    end

  end

end
```

Listing 3: View module in Participant list feature

```
  def _define_meeting_view()
    feature :@message, 'Message', [:MessageModel, :MessageView]
    feature :@create_participant_list, 'ParticipantList', [:CreateMeetingView]
    feature :@meeting_participant_list, 'ParticipantList', [:MeetingView]
  end
```

Listing 4: Declaration of Meeting feature in feature model

Remarks Each method needs to be defined at least once in order to be accessible from another feature but also activated before the feature that use it. In other word, if a feature "F2" uses a method defined in a feature "F1" then "F1" will always have to be activated before "F2" is used. If a feature is deactivated, then all the corresponding methods definitions will be removed. If there is only one definition for a method, it would not be accessible after the deactivation of the corresponding feature and may even lead to a runtime error. In order to access again a given definition, you need to reactivate the concerned feature where it is defined.

Prologues and Epilogues

They are two core principles to understand when making a RubyCOP application which are called *Prologues* and *Epilogues*. These two mechanisms play a significant role in the activation and deactivation of a feature. The *prologue* mechanism can be seen as a list of methods that will be called when the feature is activated. This list of methods will be called first and thus before the activation of the given feature. On the other side, the *epilogue* mechanism can be seen as a list of method that will be executed after the deactivation of the related feature. These two mechanisms are typically used to trigger direct application modification depending on the feature (de)activation. If we look at the example shown in listing 24 in Appendix A, we can see that there is one method for each mechanism. This means that when the *"Default Color"* will be activated then the `color_all` will be triggered whereas the `stock_color_palette` method will be called after the feature is deactivated.

Remarks It is important to emphasise the fact that an object **needs** to be instantiated in order to allow the *Prologues* and *Epilogues* to work correctly. If the object is not instantiated, nothing will be triggered at all. Also methods executed in such mechanisms usually do not have any argument required since they are called without any.

Proceed

This is one of the most important mechanisms in COP languages and in this framework. It can be compared to the `super` mechanism in OO language except that instead of calling the code of the parent, it will call the preceding implementation of the method when the feature was activated. So in other words, this mechanism allows a context related a feature to adapt its overall definition to the current need expressed by the various contexts activated. For example, one can see on line 17 in the feature's code display in listing 25 that there is already a method called `color_all`. If one looks at listing 24 on line 21, one can see another definition of this method. Without the *proceed* mechanism, all previous implementations will simply end up erased and the definition of `color_all` in *Default Color* will be the only one considered since this feature has been activated in last. With *proceed* however the previous definition of `color_all` in *Set Color Palette* will still be executed at line 24 in 24 where the *proceed()* function is called. Another interesting part of this mechanism is that, as compared to a `super`, the method needs to be call with the number of arguments the previous implementation required. In listing 25, the method `create_color_palette` has one optional argument. Thus as one can see in listing 24, the method can be called with one parameter whereas the `color_all` method can only be called without any.

Remark: the `proceed()` method can exclusively be used when an implementation of the given method existed before the activation of the feature. Otherwise, a call to this method will trigger a runtime error due to lack of a previous definition.

2.2.4 Existing architecture

The existing architecture of the RubyCOP framework is divided into 2 main parts:

- server: where all the communication goes through. Its primary task consists of getting messages from clients in a channel and broadcast them to every client that is also connected to that channel.
- client: represents all the clients that are connecting to the server in order to either send messages, or receive messages from other clients in a bidirectional way.

In practice, the server is being used by four clients: the context-aware app (which is a mix between the developer's code and the RubyCOP framework) and three external tools (the three boxes at the bottom on figure 2.2) made to help the developers to create and maintain such a complex application. They are explained in more detail in section 2.2.5. For example, the context-simulator tool communicates data through the server to alter the application in term of context changes. Conversely, the application sends internal information, including context-model data, to the different web based visualisation tools [7] [8].

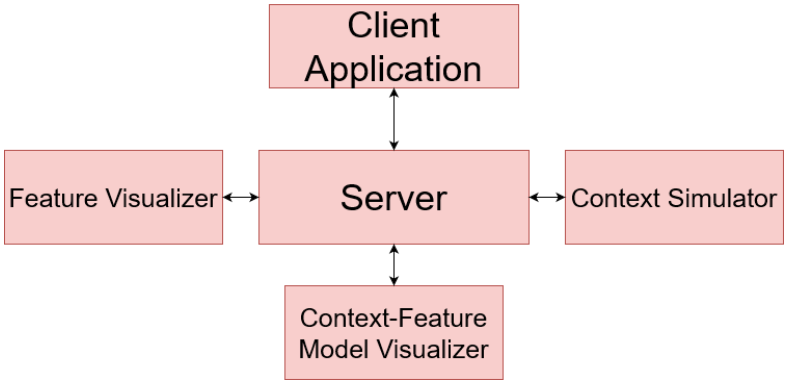

Figure 2.2: RubyCOP Architecture

2.2.5 Tools

In this section, we explain all the existing tools that come with the framework and that are really useful in the creation of complex context-oriented applications. Such

tools are integrated on an architecture that encourages language independence by using a client server approach with messages structured in a JSON format.

Visualizers

As FBCOP application can be hard to understand due to the considerable number of changes and adaptations, nothing is more practical than visualizers to obtain a simpler view of the actual application. There are in fact two types of visualizers: One specifically dedicated to observation of features and their adaptation and the other for the context activation and feature model of the application.

Feature visualizer: As said just above, this visualizer is principally dedicated to the observation of currently activated features. This visualizer help a developer to identify how a given class is currently adapted by which feature. It also helps to see the order of (de)activation of the features and can be used to comprehend why a particular method is erased or not modified. This visualizer also helps the developer to apprehend which feature are added/removed when a given context is (de)activated.

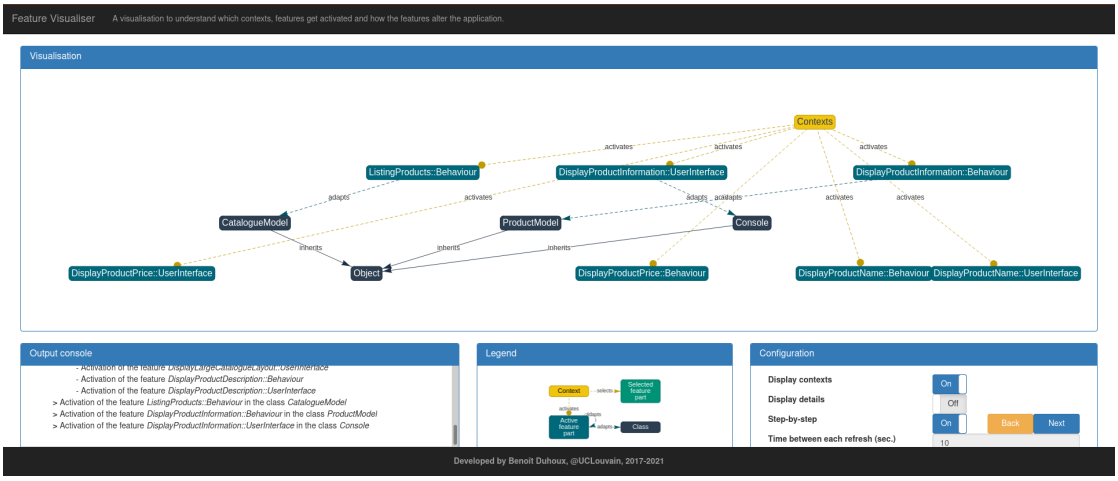

Figure 2.3: Feature visualizer

Context Feature model visualizer: The purpose of this visualizer is a bit different from the previous one. It allows the developer to obtain a better view of the logical structure of this application by showing it in three various models (context, feature and class diagram). This tool can be utilized to compare the current implementation of the application with the one describe in the model formulated beforehand. It can additionally be used to see which contexts and features are currently activated and what are the actual mapping between them.

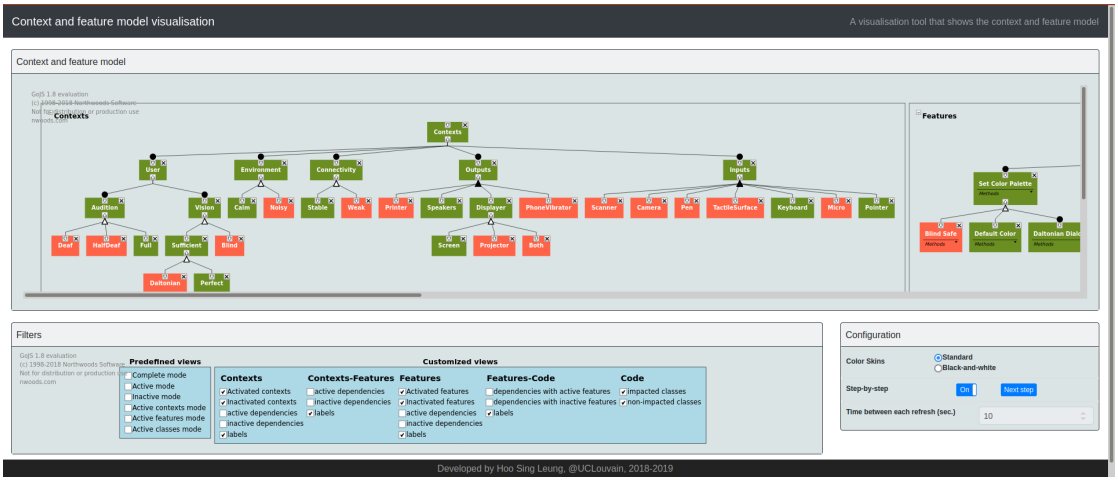

Figure 2.4: Context Feature model visualizer

Context Simulator

This tool is really convenient to simulate the (de)activation of a desired context to see its direct impact on the application but also to verify if such a change is allowed by the current mapping defined between the existing contexts and features. Some example of commands that it used can be observed in the following listing 5.

```
/:/ MICRO TEST /:/
activate: Micro
deactivate: Micro
activate: Calm; deactivate: Noisy
activate: Noisy; deactivate: Calm
```

Listing 5: Example of Context Simulator commands

2.3 FBCOP Methodology

In order to facilitate the creation of an FBCOP application, we based ourselves on a FBCOP methodology proposed by the RubyCOP framework's author. We decided to follow this methodology to reduce the difficulty behind the conception of such a complex application. It allows us to think more carefully about which features our application should have, to which context they will adapt and to how to represent their internal and external relations. This methodology, defined in an incremental software development way, is divided into four phases that will each be described more precisely in the following subsections.

2.3.1 Requirements

In this phase, the developers have to list all the contexts and features of the future application they desire to implement. They have to make a lexicon that gives a proper definition/description of all contexts and features listed before. Moreover, they have to give a proper *rationale* for each context and feature in the context-oriented system that motivates their presence in the application. Another essential point, they will focus on is the mapping description of the future application. They should thus be capable to ask themselves: "What context (de)activations trigger what feature (de)activations?" and "Why they do so?"

2.3.2 Analysis and Design

This phase consists of modelling the desired application to have a better view and understanding of it. Here the developers organise context and feature in a hierarchical model and define the dependencies.

The organisation, following the feature-based context-oriented programming methodology, can be seen as a cycle where we begin with one feature, describe one context that activates it, then see what other features can be activated using that context, ...

Once the models have been made, a mapping can then be defined. Another important thing when doing so is to provide a rationale as done for the contexts and features before but for each mapping relation.

2.3.3 Implementation

This phase focuses on the programming language that will be used to made the application come to life. Developers should ask themselves questions about development such as: "When are contexts activated?", "How are the features implemented?", "What class do these features adapt?" or "Do they replace or redefine specific behaviour?". They will also reflect on the type of class that have to be made in order to implement such application in the chosen language/framework.

Remarks: If the implementation is made using the RubyCOP framework, developers might have to consider looking more closely at section 2.2 to better understand how to use such a framework.

2.3.4 Testing

This phase is the last of an iteration. The goal reach here is to see if the final application that has been made exhibits the desired behaviour in all possible

situations. If they are not then, this is where the developers may think about doing another iteration of the whole methodology to fix the undesired behaviours. This also a phase used to think about the overall interest of the adaptations that have been achieved. "Are the dynamic adaptations of the system relevant, useful and acceptable?" A efficient way to verify that each context change trigger the desired behaviour is to use a test scenario generator like the one described in subsection 1.5.4.

2.4 Conclusion

FBCOP represents an approach to COP that relies on various important mechanisms such as *proceed*. This approach concentrate on defining feature on which news definition will be added dynamically depending on the current contexts. Each feature consists into a set of modules that can be attached to a corresponding default class used by the main in order to increase dynamically its content when related contexts are activated. Each application is structured on a similar pattern that separate contexts from features declarations.

As building such applications is a complex task due to the significant number of adaptations, there exists tools and methodology provided by the author to help any developer to either understand, interact or define more correctly and easily their FBCOP application.

Chapter 3

Problem Description

This chapter introduces in detail the problem that will be tackled in this thesis. We start by describing the main problem and its causes, then explain the various sub-problems to be taken in account before proposing a complete and correct solution to the main problem.

3.1 Integration problem

Currently, FBCOP languages are capable of adapting features depending on the current contexts. However, what would happen if we had to interact with the application using vocal command or gesture? For instance, imagine we develop a meeting application, where we could create meetings, write in a chat, ... In case someone is blind or does not have access to his keyboard, it would be convenient to be able to interact vocally to the application or using gesture. Using current FBCOP language, a developer would have to go through several steps in order to implement these particular behaviours in their application. Developers would have to implement a modality using a technology compatible with the language he is currently working on, then it would have to adapt all the existing features to make them work with the newly introduced modality. This way of implementing works, however, it is inefficient as the developers would have to go through the same step every time he wants to add this kind of behaviour in an FBCOP application. But instead of adding modalities within an FBCOP application, would it be possible to include those directly inside the FBCOP language itself? That is the principal problem we worked on throughout this thesis.

3.2 Technological problem

One problem we faced was how to overcome the technological limitation. Multimodality is most of the time based on low-level technology for behaviour detection such as gesture, pointing, or vocal interaction. Such technologies are habitually employing embedded devices rather than an ordinary computer. For instance, in order to speak to your application, you need a microphone, in order to capture your gesture you need a camera, etc. On the other hand, FBCOP is usually based on high-level languages that has limited the possibility of interaction with low-level devices.

Therefore, coupling both opposite worlds is challenging, as there may be incompatibility between the technologies, and we can end up being bottle-necked by either the modality or by the FBCOP language. Eventually, as a developer, assuming we can combine both without problem, the issue of integrating them in a seamless, reusable and flexible way still remains. We had to keep in mind, how to integrate those cleanly, so that their removal or modification does not affect the whole framework.

3.3 Modality Problem

Here we discuss about different problems faced when working with the various modality we used. Those problems are not specific to our work but for to each modality.

3.3.1 Voice

The first modality we decided to work with was the voice and here we describe the general issue that had to be dealt with.

Command Understanding

When adding voice features in an application in general, we have to make sure the system understand what we actually desire. We have to verify that, whenever our application hears a certain type of words or sentences, it responds with a correct command. For instance, if we say "Can you send a message to Erwan ?", the application should understand that it has to send a message to Erwan precisely.

Parasitic sound

Once we specified command on the voice recognition system, we still have to resolve the issue of how can we separate accidental noise, to actual voice interaction with

the application. For instance, if I say "See ya Erwan, I'll send you a message later", the voice recognition system might understand that I demand it to send a message for "You".

3.3.2 Gesture

Gestures recognition, the second modality added, introduced several problems and issues that we had to face in order to design the desired recognition system.

Gesture type

In case we are working with gestures, we first have to decide on what kind of gestures we want to work with. Many gesture types exist and depending on the gesture type we want to use, we might adjust the approach we take on the gesture management. For instance, the way we manage wide gestures such as whole body postures will be different than management of precise gesture such as hand signs. Such a choice will impact the technology to choose as well as our way to design the recognition system.

Gesture variety

Once the gesture type is clearly defined, we have to make the system understand how certain signs or gestures have to be interpreted and inform it that they correspond to a given action that will need to be performed by the application. For example, if we want to link a hand swipe gesture to a clear message action, we will first have to translate the gesture from a human perspective into a computer based representation. The representation made and verified, the system will then be able to recognise the desired gesture. Afterwards, we will just have to link the recognition to its dedicated action into the application to reach the desire result.

Gesture confusion

Another point that was important to deal with was the confusion between gestures. Indeed, if we want to be sure that a given gesture interpreted by the recognition library was the one attempted by the user, it was necessary to make the differentiation between those existing gestures relatively important. For example, if we want the system to be able to recognize a pointing gesture and hand swipe gesture, we will need to be sure that both representation doesn't overlap such that one of the gestures will never be performed. Also when designing such differentiation, we should pay attention to its level of abstraction. When the abstraction level is overly high, it may lead to unwanted recognition and leads to undesired behaviour (Theses bad recognition could be cause by the user itself when trying to reproduce

a desired gesture and accidentally perform another). When it's ridiculously low, it could prevent a potential correct recognition due to a small user mistake or recognition device limitation.

Parasitic gesture

As for any type of continuous user recognition system, without any prevention mechanism, there is a chance of transforming erroneous data that we could specify as "noise" into a concrete command. For example if the real user indirectly interacts with the recognition system, it could still perform a command without wanting it. As pressing inadvertently on the keyboard still triggered the concerned key interaction, any parasitic gesture interpreted by the recognition system will also triggered the linked action.

3.3.3 Vocal ouput

We define the vocal output as the ability of the system to answer to the user via a speaker with clear sentences. In order to add this capacity in our solution, we had to handle two main problems which are the voice mixing and the self-talking problem.

Voice mixing

Whenever a speaker receives many output at the same time, we have to ensure that sentences are being said one by one and not mixed together. Otherwise, the message sent by the machine would be incomprehensible and loose its main interest.

Microphone listening to speakers

This problematic is very specific to the combination of the voice recognition system and the vocal output system. We have to prevent the system to talk to himself. This problem can be illustrated by this modest scenario :

- User made voice command which did or not succeed
- A vocal output is sent in order to explain him the result of the given command
- Since the micro was still listening, the vocal feedback was considered as a new voice command attempt
- Since vocal output does not always express a correct command, then this "new" voice command typically fails
- Another vocal feedback was sent to the user, thus pursuing the loop

Chapter 4

Solution

This chapter explains the solution we provided to the problem described in the previous one. We will first describe the solution we proposed to the main problem and then detail the ones for the sub problems.

4.1 Adding interaction modalities in an FBCOP language

In order to add modalities functionality, we have to put connectors in an FBCOP language where the different modalities will be implemented. However, the type of those connectors can differ depending on the structure of the FBCOP language you are working with, it could be additional modules, directly written in the language, or it can be remote connectors where the FBCOP language connects to a modality module via the network. The choices made during the elaboration of our solution's architecture are based and tested on the FBCOP programming framework called "RubyCOP" written by Duhoux [9].

4.2 Solution Architecture

As stated previously in 4.1, we actually consider two choices that would allow us to integrate modalities in a FBCOP language. Adding a module directly written in the FBCOP language would make it work in standalone without having to care about being connected to a remote server, however we would be limited by the programming language in which the FBCOP language is written. For instance, RubyCOP is written in Ruby, if we decide to go with that solution, we will be limited by modalities for which Ruby has a specific library support.
On the other hand, remote connectors do get rid of that problem as everything is

exchanged via networks. It means, you would be able to write a modality module in a programming language that has more availability for it and then send the relevant information back to the FBCOP language. However, this solution is highly dependant on the network's health, in case your network is broken or unstable, your modality module will be unable to communicate with your FBCOP application.

4.2.1 Approach

Eventually, we decided to get with the second option, because it fits particularly well in the architecture of RubyCOP. In this case, we set modalities as external services similar to the example shown in 2.2, where they send information to FBCOP application via the Server.

In this particular study, we can also ignore all network problems, as everything is running locally. What we just have to do is to implement on the framework, a "facade" interface and then we can fully work on the external modality. You can see a small schema on figure 4.1 describing the architecture, where the red blocks represent the major parts of an FBCOP application and the green boxes represent the new services we implemented in this master thesis.

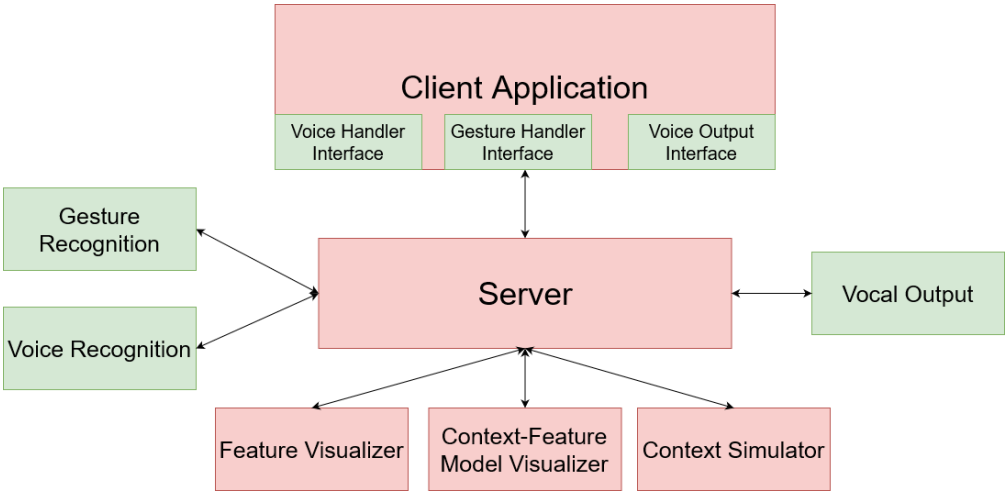

Figure 4.1: Solution Architecture

4.3 Modalities

Here we discuss the solution found and used for the different modality problem described in 3.3.

4.3.1 Voice

For the voice modality, we describe here how we make the system understand what we want, and how we also deal with the problem regarding the external noise.

Meaning frame

A naive solution to make the computer understand what we want, would be to link key sentences to a dedicated function that would execute the desired result. For instance, we could link the sentence "Call Erwan" to a command and when the computer will see that exact sentence, it will execute it. However, this naive solution could be altered by various details coming from the inherent complexity of human language such as word ordering, words omission/addition and more which will make this solution not viable. Therefore, we went for a solution proposed by Dumas, where we add meaning frame instead of exact sentence.

Following Dumas [10, p. 101], a meaning frame is defined as "artificial intelligence data structure which represents substructures of an idea" and in our solution regarding the voice interactions, we went for the definition described by Dumas where instead of using "ideas" we are using word sequences as frames and those frames are used as interaction operations [10, p. 101].
That means, depending on the combinations of trigger words understood within a sentence, the meaning frame will activate actions that correspond to that combination. For instance, if the computer has been programmed to trigger a phone call to Bruno when hearing the words "call" and "Bruno" in the sentence, it will effectively perform the action when listening to "Hey Siri, can you call my supervisor Bruno please?" as there will be the two trigger words which are "Bruno" and "call". This action is also going to work if we were to say "Hey Siri, Bruno call" as the word ordering does not matter in this system as long as the two keywords are present.

The figure 4.2 describes an example of an application of the meaning frame concept. We support a set of trigger words where each words can be linked to one or many possible actions. Each action on the right side of the schema will be triggered if and only if all the trigger words linked to it are recognized together in a sentence. For instance, if we say "Can you write some text there?", the system will see that the sentence contains the word "can", "you", "write", "a", "message" and "there". With these words, it will check in the possible actions list if any actions can be activated using the words in the sentence. In our example, it will see that the action "Write_that_there_action" can be activated because there is the word "write" and "there" and thus, the system will trigger the action.

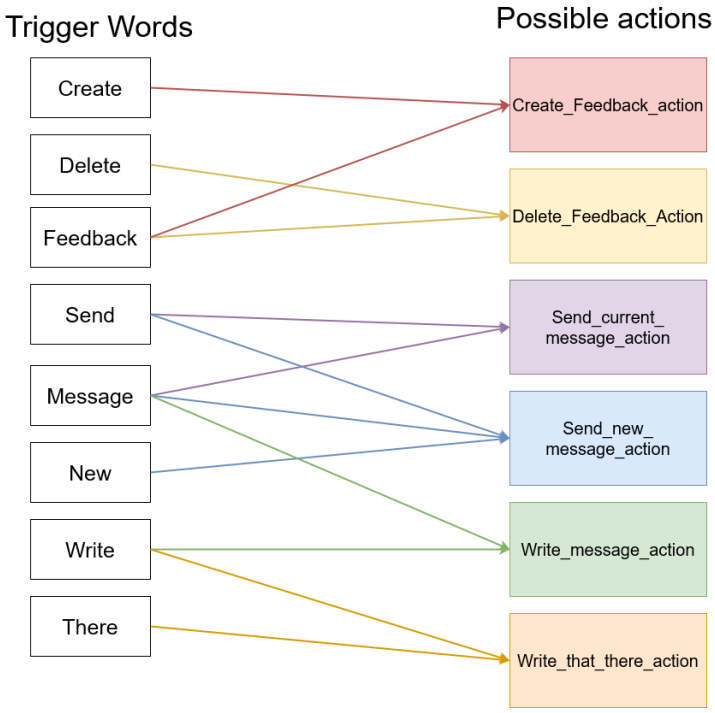

Figure 4.2: Meaning frame example

The downside of this approach, is that we cannot capture whether or not the given order is positive or negative, e.g. assume there is a command that is triggered via the keywords "try it", if we talk to our application and say "Don't try it", it will trigger the command even though we explicitly requested it to not perform that action.

A potential workaround to prevent such behaviour to happen could be to check if there is any negative words within the sentence and if there are, it would cancel every actions. But this workaround can become extremely complicated as there are many negative words to handle, plus, in case of double negations it will still cancel the action. Another weakness of this implementation is the system being able to handle only one command at a time. If we ask the system "create a feedback then delete the feedback", it will only execute the first command. Allowing the handling of multiple commands is currently impossible, as it would require to perform a deeper study on Natural Language Processing.

Two part meaning frame

However, during the validation phase, we noticed some messages were way too long to be said at once. For instance if we want to send a long message, by the time we even finish saying it, we either ran out of breath or the voice recognizer stopped

listening and just took a portion of the sentence you said. As a countermeasure to these types of command, we could introduce a two part meaning frame idea, where we say first the intent and then we give the details about our intent. Let's suppose you want to send a message. In the basic implementation, you would have to say "can send the following message: 'hello everybody, hope you are doing well'". While the two part meaning frame, you would simply have to say first "hey, care to send a message for me?", then the voice handler will ask: "what message do you want to send?", allocating you time to think about what you want to send.

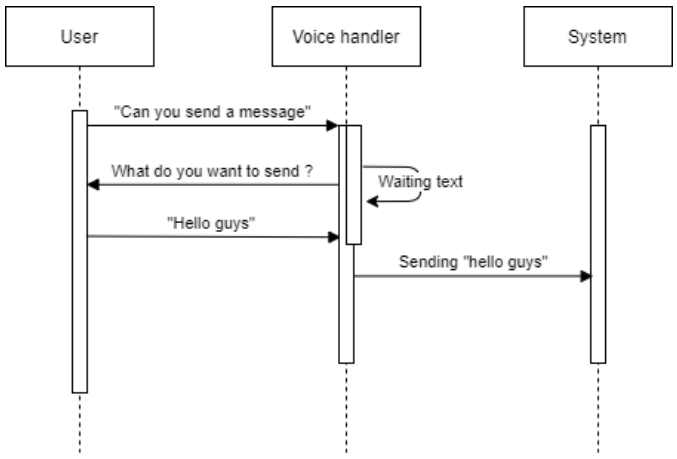

Figure 4.3: Workflow of a two-part meaning frame

Noise Filtering

In order to solve the issue caused by the noise around (as described previously in 3.3.1) we decided to use something commonly used by most voice recognition system : keywords activation. Big companies working on vocal interaction such as Google or Apple, are using keywords to make their system understand if what is following is an actual command ("Hey Siri", "Ok Google", "Alexa", ...) or not. On our side, the keyword has been based on how effortless it was for us to say it and how easy voice recognition system would understand. For instance, when we tried to trigger the voice command using the keywords "hey cop". Unfortunately, our voice recognition system kept misunderstanding the last word as "cup" rather than "cop". Thus, we tried different words combination to realise that the well known key phrase "Hey Siri" was easily detected by our recogniser.

4.3.2 Gesture

In this part, we describe more closely the gestures choices that were made, such as the type to be used, the variety to supported and how we decide to approach the

problems related to bad recognition explained before in 3.3.2.

Gesture size

As explained before, different gestures recognition technologies already exist, but before deciding which one to use, we had to determine which kind of gesture we wanted to work with. We decided that classical hand gestures were more fitting to our use case because of the natural way of this interaction with such a desktop application. Since the user will usually sit in front of its computer, it would not make sense to force him/her to stand up for each gesture interaction.

Gesture variety

Now that the type of gesture has been determined, we had to decide on the variety of gestures we want the application to recognise, and then assign them to dedicated and logical behaviour. A first gesture that came directly to our mind was the pointing gesture. As described in the intro at 1.1, nothing is more natural for a human who wants to describe something to directly point at it. Thereafter, the swipe gesture was chosen since it was an easy gesture to recognize and perform. This choice is not motivated by any scientific purpose at all but by personal feeling.

Gesture differentiation

A good way to differentiate a gesture from another is to make each gesture unique through several details. Theses details can be global such as hand orientation, velocity, direction but can also be more specific such as finger type, angle between two bones, space between finger, ... On our side, we decided to make the "pointing interaction" unique by having at least three fingers other than the index being bent since "swiping interaction" usually requires the whole hand to be taken into account. The details and the reason of this choice are explained later in the implementation section 5.2.2.

Parasitic Gesture filtering

In order to prevent parasitic gesture from being recognized, it's a good practice to put a key detection mechanism to be sure of the user's intention to interact with the application. However, considering the number of recognized and effective gestures was very low, we decided to not integrate such mechanism in the application. Furthermore, the chances of doing undesired interaction with the application are less frequent here due to the specificity of given gesture (e.g pointing with three finger bent) or their limited usage in the validation (e.g swipe to erase text) did not encourage us to give it a go. However if the number of handled gesture increases

it could be interesting to implement this filter as we will describe later in section 8.1.3.

4.3.3 Vocal output

For the vocal output, an explanation regarding the problem explained in 3.3.3 is provided here.

Concurrent voice

In case our system receives many output at the same time, we decided to use a shared resource. Whenever a sentence needs to be said, it will check if the shared resource is available, and if it is not, it will wait until that resource is available. Once it is available, it is going to take the resource for him and then it will utter the sentence and then release the resource.

Prevent self talking application

For the self talking problem, we already partially blocked it in 4.3.1 by ignoring surrounding noises via the keywords activation. However, if we are using the 2 parts meaning frame as explained previously in 4.3, we will still encounter a problem where the app does not listen to command now, but responds to itself. For instance, a typical scenario can go like this:

- User says : "Hey siri", "can you send a new message ?"
- System answers : "Yes, what kind of message do you want to send ?"
- Since the micro was still listening, the system thinks we want to send "Yes what kind of message do you want to send ?"
- System sends that message.

In order to prevent this scenario from happening, we decided to block a certain amount of predictable sentence, for instance the questions the system might ask. This trick did not always work as the microphone sometimes only partially listens to the feedback or even did not understand the same content that what was said. Therefore, the sentence heard, since it did not correspond to any of the predefined questions, was not blocked at all.

Chapter 5

Implementation

This chapter will detail everything regarding the implementation of our solution in the RubyCOP programming framework as well as the implementation of the different modalities previously described. We will first describe the different used technology for each modality, then explain how to implement those modality and finally explain how we managed to set everything together.

5.1 Used technology

In order to produce this multimodal application, numerous kinds of libraries should be installed. Typically we supported one library per type of modality. However, in order to find the most appropriate libraries, we had to install and test quite a few more. Here, we will only describe the ones we managed to get to work and describe their advantages/disadvantages then we will explained which ones were used in the application and why.

5.1.1 Communication protocol

As stated in the past, each modality will have to communicate to the FBCOP application via a server through the network. Therefore, we will need a protocol to would allow us to communicate via that a server. Fortunately, the RubyCOP programming framework on which we worked on, already comes with a tooling server capable of broadcasting messages to each service connected to it using Web-socket protocol. Web-socket is a protocol that allows a server and a client to communicate in a bi-directional way. In our case, This tooling server is used as a broadcast emitter to anyone connected to a same channel. For instance, if a voice service is connected to the channel A and an FBCOP application is connected to the same channel. Then, whenever the voice service delivers data in the channel A,

the server will dispatch the data on every other connected client, in our case, the FBCOP application.

5.1.2 Voice

Following the architecture explained in section 4.2, we decided to go for a python API called "speech recognition" which allowed us to swap easily among different available voice recognition libraries.

PocketSphinx

We first decided to try an old C library called "pocketsphinx" which allows to run the voice recognition locally without having to rely on any API. One of its advantages was that it was easy to implement and fast to integrate to the current framework. The main issue that comes with that library is that the model is untrained and had to go through some preparation before being usable. For instance, we had to reduce the dictionary, in order to make it understand basic sentences. Overall, the performances offered by "pocketsphinx" were unsatisfactory even with a reduced dictionary. We could have improved its detection capacity by doing a lot of training sessions but due to the lack of time and the multimodality focus of this thesis rather than voice recognition alone, we decide to try other API/libraries.

Houndify

Next available technology we evaluated was an API called "Houndify". Thanks to its complete API documentation and the simplicity of creating test samples, we have clearly seen its reasonable efficiency. However, the number of API calls per month were so limited even in the free mode that it was necessary to buy a monthly subscription in order to have a decent limit of calls for testing purposes. Therefore we decide to try yet another one instead of going further with it.

Google-Speech

Ultimately, we decided to go for the google-speech API, which also offers decent efficiency and a free limited usage of its API with the testing key. The only main downside of this technology was that it had a time restricted use of a maximum of one hour of audio per month. However, this was a non-problem for us since the API is only used for a study case rather than a full business application.

Final choice

When comparing them, side-by-side, Houndify and the google API seem to be relatively similar. The main difference between these two APIs is the number of calls allowed (Houndify) compared to a time usage limit (Google). However, we still decided to work with the google one in the end. Since our voice commands were usually a massive number of minor calls to the API, the Houndify API limit would have had more impact on us than the google one.
This why we decided to go with the google API and we can say it was a correct choice since, as predicted, we did not suffer from its limitation throughout the experimentation done with it.

5.1.3 Gesture

After having decided which kind of gesture we would go for, as described in section 4.3.2, we had to choose which technology to use and there were two different ones that allowed us to make it possible.

OpenCV

OpenCV is a low-level library that allows to capture and detect all kinds of gestures through your own camera. It is a powerful library that would perfectly fit in our use-case and would allow us to reuse the same library in case we decided to handle other type of gestures in the future (posture, positions, pointers,...). However, the required work to create basic examples and detect basic finger movements or position was way too difficult and it would have required most of our time on that topic alone, which is out of scope of this thesis. Therefore, instead of using it (even though it could be used as a future work), we decided to use a Leap Motion instead.

Leap Motion

Leap Motion is a small camera that is able to detect and capture precisely gestures and finger positions. Since a *Leap Motion* is proprietary technology it has only one kind of working library which is the "Leap Motion API". In order to properly interact with the *Leap Motion* equipment we had to use one of the existing SDKs. Since we already installed and used python in our project, we decided to go with the dedicated SDK. However, this python SDK (produced with SWIG, by the creator of the application) was made to work with python 2.7 rather than any python 3 which is a big problem considering that python 2.7 will no longer be maintained and that its last released version was on 2020. This limitation could be corrected as explained in section 8.2.1 since this SDK was generated using SWIG for python 2.7.

5.1.4 Vocal feedback

Eventually for the multimodal output part of the application i.e. vocal feedback, we only tried one library which is Ruby Flite. Ruby Flite is said to be *a small speech synthesis library for ruby using CMU Flite*[1]. The number of existing voices available in the library is relatively small but since this application would be used only as a "*Proof Of Concept*", this limitation was not really considered as a problem. Furthermore, this library does not need many requirements and can be installed quite easily. Hence, we decided to stick with it and not look for another one that might be more complicated and not really worth our time.

5.2 Modalities implementation

In this section we specify how we concretely implemented the modalities using the technologies mentioned above.

5.2.1 Voice recognition

As previously stated before, we decided to implement the voice recognition using the python API called "speech recognition". This API after hearing what we said, will deliver what we said to the microphone via REST API, and then the google-speech service will translate the sound into a machine readable string. Once, we receive the answer coming from the google-speech server, we send that string over the Web-socket to the tooling server, that will send that to the FBCOP app, if it is connected.

5.2.2 Gesture recognition

For the gestures, we use a Leap Motion to detect the hand movement. Thanks to its powerful detection capacity, we are apt to capture accurate gesture such as swipe or pointing sign. However, the main difficulty that came with the Leap Motion was the vast amount of data it was sending to our server. As it tries to capture gesture as precisely as possible, it used to gather a big amount of data regarding each position of each part of your hands. Also, another fundamental challenge that comes with the Leap Motion was the understanding of the documentation, as it describes the hands and gesture in a very technical way that was relatively unknown to us: (e.g. metacarpal, proximal, ...).

[1] as described on the library readme on https://github.com/kubo/ruby-flite

Swipe recognition

The first gesture implemented was the swipe gesture as there was already some predefined work in the Leap Motion library. In order to use the swipe gesture, we had to activate the dedicated option in the library. This can be done by performing the following command:

```
controller.enable_gesture(Leap.Gesture.TYPE_SWIPE)
```

The actual code representing the structure can be seen on the listing 6. The data send for this gesture is a dictionary containing two keys: a key called "type" which tells that the given data is a gesture and another called "gesture" that defines the type of gesture that has been performed. At this point only one type of gesture could be performed.

```
data = {"type": "gesture", "gesture": self.__get_gesture_type(gesture)}
```

Listing 6: Pointing gesture data representation

Pointing recognition

The pointing recognition process was the most complex process since it required to analyse and transmit more data. For example, we had to implement the pointing mechanism, which marks the difference between the two gestures and the click mechanism. The various mechanisms will be explained in the following paragraphs. Concerning the data structure used, it needed to be improved by adding extra keys to describe this gesture more properly. The pointing gesture was described through four elements which were each linked to a corresponding key in the dictionary. The actual code representing the structure can be seen on the listing 7. The "type" was employed to describe the gesture as a movement from a finger. The "pos_x" and "pos_y" where used to define the coordinates of the point in term of percentages between 0 and 1. The actual representation of the coordinates is a bit different than usual representation since the (0,0) point is situated in the top left corner rather than in the bottom making the calculation different for the y axis. The "clicked" key defines if the actual finger move performed by the user is a click. This mechanism is explained more in detail below.

```
data = {"type": "finger", "pos_x": x_leap, "pos_y": y_leap, "clicked?":
↪  self.__has_clicked(finger)}
```

Listing 7: Pointing gesture data representation

Finger Bending As explained before in 4.3.2, we make the pointing gesture unique by having three bent fingers (except the index since its pointing direction is used as data). The fundamental reason behind this number is twofold. First when we look at some classical pointing icon representation as on figure 5.1, we can see that they are usually represented by two unbent fingers, one being the index that serves as the pointer and the other being the thumb. Furthermore,

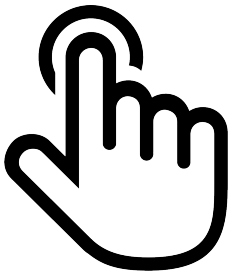

Figure 5.1: Example of cursor icon found on an internet website

when we attempted to implement the bending differentiation, we calculated the angle between metacarpal and proximal bone for each finger then check that the angle was bigger than a "control value" to be sure that this finger was indeed being bent by the user. The "control value" was obtained through trial and error way and could thus still be improved through further attempt. However, as one can see on this figure 5.2, there is no metacarpal bone in the thumb, meaning we had to take another bone into consideration to see if it was actually bent. We did not perceive any interest on calculating a fifth angle for the thumb as its position does not play an critical role in the gesture definition thus we stayed with only three fingers being bent.

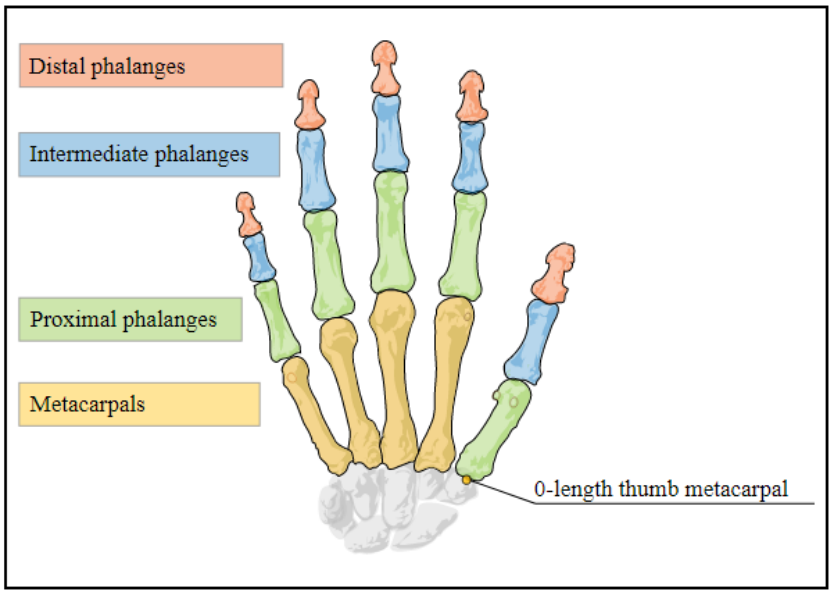

Figure 5.2: Hand picture from Leap motion API documentation

Clicking In order to be able to distinguish a click interaction from a simple pointing one, we based our self on the documentation given on the Leap Motion website. As one can see on figure 5.3, the Leap motion distinguishes the user finger interaction in three specific zones: *hovering* ($[+1, 0[$) , *neutral* as 0 and *touching* ($]0, -1]$). Based on theses data and trial and error, we were able to define that a click was performed by the user finger is in the touching zone and under -0.3 distance. A timer was also made for this click interaction to prevent any spam that could break the UI and thus lead to a bad user experience.

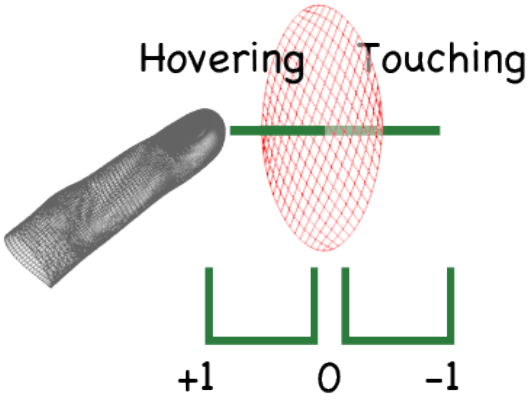

Figure 5.3: Touch Zone explanation from Leap Motion documentation

Data Amount

In order to avoid data overflow on the server side, we decided to reduce the amount of data sent by our dedicated python script. For example, only the required data were sent to the application. For pre-defined gestures such as swiping, we chose to send the movement only when it was completed since it was the completion of this movement that should trigger the corresponding behaviour. For the pointing gesture, as each new point sent will cause the pointer position to be updated, we set a time limit of a given number of milliseconds between each send. It is important to notify that timers have been chosen in a trial and error way and thus could still be improved as a future work.

5.2.3 Vocal output

In order to get a vocal output, as explained in 5.1.4, we relied on the RubyFlite library to generate voices using the function `speak()` on a string element. Most of the implementation time was spent on the plugging process where the modality had to be adjusted to work as a service following the architecture described in 4.2. Some time has also been spent on solving the issue regarding the mixing voices previously mentioned in 3.3.3. This issue was solved by making the service thread-safe using a semaphore[2].

```
def recv_message(message)
  Thread.new {
    @semaphore.synchronize {
      message.speak()
    }
  }
end
```

Listing 8: Synchronized vocal output generation

5.3 Adding modality connectors in RubyCOP

Here we explain the different steps we did to integrate the voice and the gesture handling in the framework. In order to do that, we first need to add the different communication channel in the server that will allow the modality and the handlers to interacts, then we have to create a communication client that will send or receive data from the server, and finally the classes that will handle the received data.

[2]https://www.geeksforgeeks.org/semaphores-and-its-types/

```ruby
def create_channels()
  channels = []
  opts = Slop.parse() {
    |option|
    option.on('-a', '--all', 'Create all the communication channels for the
    ↪  different tools') do
      channels << Channel.new(:VocalOutputCommunication, 4444)
      channels << Channel.new(:GestureRecognitionCommunication, 5555)
      channels << Channel.new(:VoiceRecognitionCommunication, 6666)
      channels << Channel.new(:ContextSimulatorCommunication, 7777)
      channels << Channel.new(:FeatureVisualiser, 8888)
    end
  }
  return channels
end
```

Listing 9: Channel creation in the tooling server

5.3.1 Adding channel in the tooling server

As explained in 4.1, we will implement connectors in the RubyCOP framework that will receive information coming from the remote modality services. In order to do that, we first have to add a channel in the tooling server which will be dedicated to the modality. This change is implemented in the file tools/server/server.rb In the code 9 we see all the channel created for the vocal output, gesture recognition and the voice recognition, but there is in addition the channel for the existing services before like context-feature model visualizer or context simulator.

5.3.2 Adding modality communication client

Once we included the corresponding channel in the tooling server, we have to add modality communication client in the framework. These clients will be used to connect to the server and receive data from it. Within the RubyCOP framework, we have to create a class that inherits from `ToolsCommunication` that let us override two important methods : recv_message(msg) and send_message(data). Those two methods allow the object to receive and deliver message to the server.

In the listing 10, we can see how we implemented the communication client for the voice recognition. We simply create a class called "VoiceRecognitionCommunication", and then make it inherit `ToolsCommunication`. In the class constructor we just have to call the parent constructor and precise on which channel we want to connect and then implement recv_message(msg) to receive data coming from the server. The call "VoiceManager.instance.trigger_command(tokens)" will be

explained later in the modalities handlers implementation section.

```
class VoiceRecognitionCommunication < ToolsCommunication
  def initialize
    super(6666)
  end

  def recv_message(msg)
    msg = msg.delete_prefix('"').delete_suffix('"')
    tokens = tokenize_message(msg)
    VoiceManager.instance.trigger_command(tokens)
  end
end
```

Listing 10: Adding voice communication example

5.3.3 Implement Voice modalities handlers

In this section we explain how we handle data sent by the voice modality and what we actually do and allow developers to develop in their application.

Adding Voice command

For the voices we implemented a voice manager which is a singleton class that is going to register the different meaning frame desired by an application. We represent meaning frames here as a series of key-values where the keys are the trigger words represented as an array of string and the value is the actions that needs to be activated whenever the trigger words are heard. For instance a valid command would look like listing 11 or listing 12.

In the listing 11 we tell to the system, whenever you hear "erase" and "message" within a same sentence, execute the method called "erase_voice".

```
VoiceManager.instance.add_command(["erase","message"], method(:erase_voice))
```

Listing 11: Adding command in the voice manager

```
VoiceManager.instance.add_command(["send","new","message"],
↪  method(:send_voice),"What do you want to say ?")
```

Listing 12: Adding two parts command in the voice manager

In the particular case of the listing 12, the third argument represents the question the system needs to ask the user in a two-parts meaning frame scenario as described in 4.3.1. After the user has said the desired sentence, the system will trigger the method "send_voice".

Once desired commands have been added in the system, we can begin to listen to voice input.

Meaning frame implementation

Whenever the RubyCOP application receives an input from the server. It will check if the sentence contains any any valid combinations listed in his map. In listing 13, we browse a map and check for every key, which are arrays of string, if all elements are present. If one of the keys is fully contained in the sentence, it will return the value, otherwise it will return nil.

```
def get_value_or_nil(tokenized_message, map)
  merged = tokenized_message.join(' ')
  map.each do |key, value|
    return value if contain_all_element_of(tokenized_message, key) || merged
    ↪  =~ key
  end
  nil
end
```

Listing 13: Checking if the sentence contains one of the combination listed in the map

In case the application recognizes a valid command, it will execute it by calling the action that is to be performed otherwise, it will throw an exception instead. In case the application recognizes a valid an action that represents a two-part meaning frame, it will first ask the question, and then assume the next input as the answer to that question.

Voice commands history

Alongside the voice manager, we also implemented a class called "VoiceComHistory" which is a singleton that is keeping track of the states of messages sent by the user. This class includes the observer design pattern which allows developers to register a view that can displays what is the command that has been said by the user, but can also see if the voice command was understood, and finally the feedback returned by the application. Whenever the voice history is updated, it notifies all of its observer via the method `notify(:type, *args)` which contains two arguments:

the types of the messages and the message itself. The voice communication history works with four types:

- correct_command : indicates the command has been understood and successfully applied
- bad_command : indicates the command was not understood
- question : indicates that the application is asking a question, meaning, the next input will be considered as the answer of that question
- error : indicates the command has been understood but an error occurred during the action itself

In the listing 14, we show a small example of a view that includes observer can implement the update method where all depending on the cases, we decide to display the command or the feedback that is returned by the voice manager.

```
def update(type, elem)
  case type
  when :bad_command then write_command(elem, false);
  when :correct_command then write_command(elem, true);
  when :error then write_feedback(elem, false);
  when :question then write_feedback(elem, true);
  else
    proceed(type, elem)
  end
end
```

Listing 14

5.3.4 Implement Gesture modalities handlers

In this section, we explain how we handle the data coming from the gesture modality service and what's the possibility in term of gesture that can be done by any FBCOP developer in their applications.

Adding a gesture command

To help user bind specifics commands to the desired gesture, we created a gesture manager class. This class, which is a singleton, can be called anywhere in the application in order to bind gestures recognition. The main role of this class is to allow developers to add/remove gestures handling when needed. This can be

done through two methods which are `add_gesture` and `remove_gesture`. A short example of theses two methods can be seen below

```
GestureManager.instance.add_gesture('name', behaviour, params_names)
GestureManager.instance.remove_gesture('name')
```

Where the `name` parameter represents the name of key used to find the gesture behaviour `behaviour` is either lambda method or a reference to a method and `params_names` is a list of all the parameters that the `behaviour` method needs to work. These parameters will be found in the message receive by the manager from the gesture recognition service and as to be set in the exact order to prevent any problems. Another important thing, is that the `name` provided is the same as the gesture type send by the gesture modality in order to be taken into account.

Data handling

The gesture manager, inspired by the voice one, works as followed:

- The data described in 5.2.2 is received by the manager.
- The manager verifies if it's a gesture or a finger action that has been performed.
- If it's a gesture then it looks in gesture dictionary with the gesture name and checks for potential parameters. If it's a finger, then it applies the predefined behaviour for pointing.
- If it discovered a linked method then it executes it with the parameters found in the data based on the parameter list.

Remark Each gesture behaviour combination is represented by pair `[key => value]` where the key is the gesture name and the behaviour is a reference to the method.

Simulating an application cursor

This is one significant problem that came when trying to integrate the pointing mechanism in an application. We wanted a more dedicated behaviour for this gesture in order to avoid undesired behaviour like the user leaving application or interacting with the system. A possible solution to prevent such scenarios from happening was to create a dedicated cursor i.e. an in app pointer cursor. However, this solution introduces various problems.

Linking Coordinates to an application One problem that directly came was about the usage of the user's pointing finger coordinates. How can we know on which component the pointer is and make it react as if it was a classical pointer?

A first possibility was about getting these coordinates then look back in the application through all component to see what the user was currently pointing. But such approach quickly becomes impossible due to the high cost of coordinates recalculation. Indeed, in the UI library used, the coordinates of a UI element vary according to the window in which it is contained. Thus to find a given component, the new coordinates of this element had to be recalculated for each new window or component entered. This means that if a text field was in n windows, we had to calculate the same coordinates n times in order to find the correct coordinates. A proper example of this problem can be seen on figure 5.4. If the user tried to interact with the text field in the window D then the system will first have to detected that the coordinates in window A are touching the window B. Then by re-adapting the given coordinates to this window to see that the window D is also touched. Then recalculating again the coordinates to see that the user was indeed touching the current text field.

Default cursor's events simulation Furthermore, even if the coordinates problem was handled, another one remained which is about the re-implementation of all the classical cursor's interactions (hover, left click, mouse leave, ...) with each app component. Such interactions could have been done through simulating as it can been seen on the UI library documentation. However, a propagation mechanism had also to be produced in order to completly implement the pointing cursor.

Conclusion As all these problems made this approach too complex and cumbersome, but also prevent us to reach the desired behaviour, that we changed for a simpler one which was using the actual cursor by mapping the coordinates to the actual screen size. This choice will still contain the user disruption with system described above but at least will save us too much work. This has been possible thanks to xdotool which is a set of commands in Linux that can be used to simulate mouse activity (move, left_click, ...) and others elements. The commands can be called in the Ruby environment like this:

```
system('xdotool', 'mousemove', app_x.to_s, app_y.to_s)
system('xdotool', 'click', LEFT_CLICK.to_s)
```

Where `app_x` = `leap_x` * `screen_width` and `app_y` = (1 - `leap_y`) * `screen_height` as the leap motion set the y coordinates to 0 for representing the up position and 1 for the bottom.

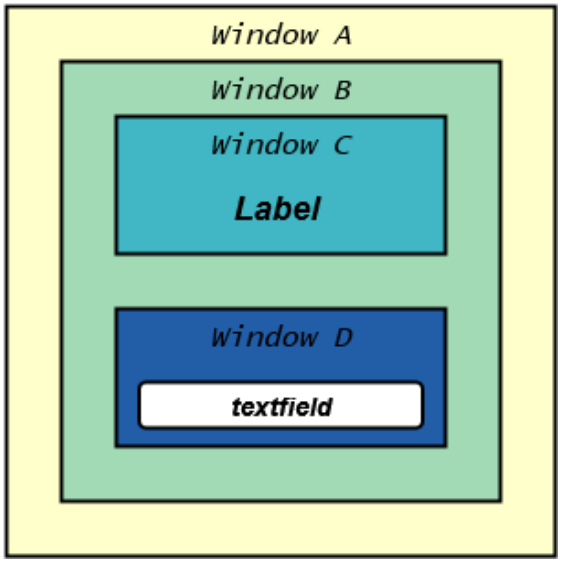

Figure 5.4: Multiple windows cursor's problem

Chapter 6

Validation

In this chapter, we will describe the creation of the smart meeting use-case employing the FBCOP methodology previously explained in 2.3. We will describe everything from the specification to the testing part using the RubyCOP version that contains the implementation detailed in the previous chapter.

6.1 Specification

In order to prove the feasibility of creating a context-aware multi-modal application, we initially had to define a use-case on which we could base our implementation. Many propositions were mixed in during the brainstorming phase, where many modalities were involved (tabletop[5], voice, gesture, proxemic [1], ...) and many ideas have been proposed. Eventually, two main ideas were particularly noticed. One which was about a teacher giving a class to various students who would have the capacity of interacting with a common app that would act differently depending on the context and the other being a meeting between many people around a tabletop. However, for the first case we would need to create a distributed application which is not the purpose of this thesis. As where in the second use-case, a tabletop will have been required and the only available was in Namur. After some reflection on this topic, we noticed that both of these use-cases involved bringing people together around a same application. Consequently, we generalized this concept into a smart meeting application, where people could interact with each other by sending messages, adding concerned participant in the meeting and more. The particularity of such an app was to adapt its content and interactions depending of the user capacities, available modalities and surrounding environment.

6.2 Modelling

This section details the context-feature model of our use-case as well as the lexicon which helps understanding it.

6.2.1 Model

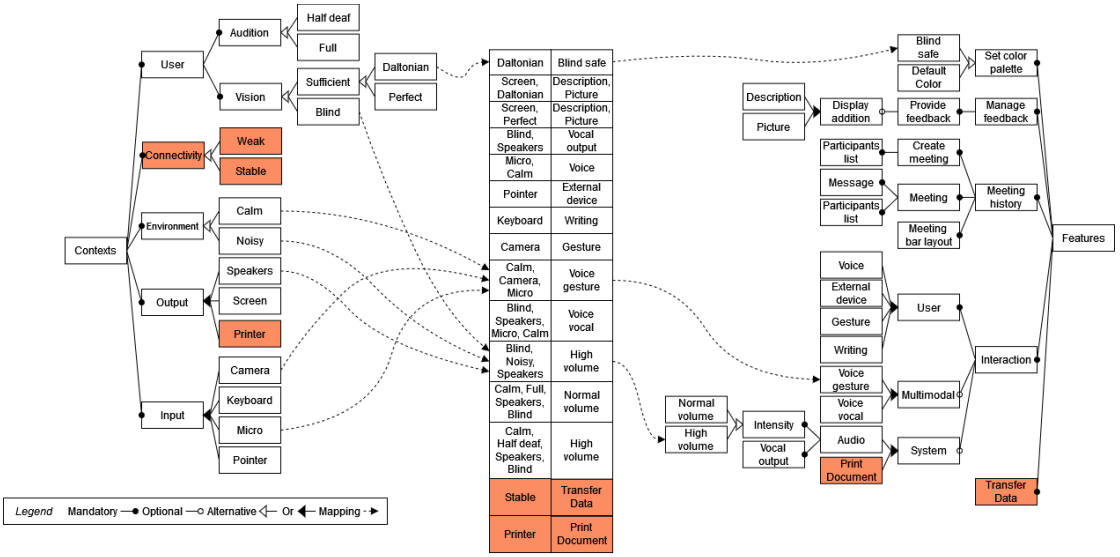

Figure 6.1: Mapping model of our use case

We can see in the model depicted in figure 6.1 the different contexts and features that define our application as well as the mapping between them described by the table in the middle. We colored in red, contexts and features that was first part of our model at the very beginning, but wasn't implemented in the next phase because it was either uninteresting or didn't make any sense to implement in our case.
It is also worth noticing, that we are considering the different modalities as features in our model.

6.2.2 Lexicon

Here, we give the lexicon of our feature-context model which is used to defines and describes each part of our models in other to facilitate its comprehension to anyone.

Features

Set Color palette Represents the different colours that can be applied to the application There exists two distinct modes for that feature :

- Blind safe : which is a coloration especially made for blind color people
- Default color : which is the basic default coloration chosen for the application

Manage Feedback Possibility for the user to manage all the feedbacks from the application (create or delete feedback). The feedback can be provided with additional displays such as an image/icon to quickly distinguish the feedback and a textual description to helps the user to better understand the feedback given.

Meeting history Is the ability for user to see the list of meetings in which they are. This feature is divided into three sub-features:

- Create meeting: a window that allows the user to create a new meeting. This feature allows to include the possibility to add new participants to that meeting.
- Meeting: is the pane use to display the meeting itself between the user and many participants. During this meeting, users can have access to that meeting messages but also see who is participating as well as adding new participant if necessary.
- Meeting bar layout: is a small view of the most important meeting data and actions that allows the user to join back a session or definitely leave it.

Interaction Describes all the different modalities between the users and the system. This feature is sub divided between 3 majors categories.

- User: which regroup all the input interaction coming directly from the user. For instance, this could be voice interaction, command via gesture or any external devices such as a laser pointer.
- System: which are the output interaction coming from the system. For instance, the system would be able to interact with us via the sound. This interaction has two features that is adjusting the volume of that output.
- Multimodal: describes the mix between different input and output interactions. For instance, there is a mix of the gesture and the voice input where the voice and gesture are mixed (e.g if we say "put that there" by pointing the desired location). But it also has the feature to mix voice input and vocal output, where a vocal command is answered vocally by the system.

Contexts

Here is the list of the different contexts taken in account:

User Describes the context regarding the user's state. We only considered two main states which are the user's vision and audition :

- Vision: For the vision, we considered the user could be either *Blind* or *Sufficient*, where *Blind* defines someones with serious vision problem or not vision at all and *Sufficient* describes someone that can see without major problem. However, the *Sufficient* context was divided into two types which are *Daltonian* and *Perfect* people. The *Daltonian* context defines someone that is affected by any kind of color blindness and *Perfect* describes someone with no color trouble. For ease of implementation, we assumed no real differentiation between monochromatism, dichromatism or abnormal trichromacy. [1]

- Audition: For the audition, we considered the user could be either half-deaf or full. Full means that the user can hear perfectly without having to increase the volume.

Environment Describes the environment in which the user is currently in. We actually only described the environmental noise level which can either be *Noisy* or *Calm*.

Output Describes which kind of output devices are currently connected to the computer. We only consider two main types of output devices that are the *Screen* and the *Speakers*.

Input Describes which kind of input devices are currently plug in to the computer. We consider four main types of input devices which are the *Keyboard*, *Micro*, *Pointer* and *Camera*. *Pointer* is considered here to be the cursor on screen that can be controlled with the computer mouse.

6.3 Implementation

In this section, we will mainly describe how we used our solution to implement features so that they allow multimodal interactions.

[1] One can go on livepositively.com for further information on the differentiation between theses different troubles

6.3.1 File structure

The application's file structure follows the typical FBCOP application structure explained before in section 2.2.2. Despite the quality of this proposed structure, we decided to add some elements for our purpose. Here is the list of added folders or files for improving our application structure:

pictures is a folder containing all the pictures used by the application, which were principally used to create the feedback system. Some of the images contained in it can be seen on figures 6.5, 6.6 and 6.7.

texts contains all textual descriptions of the application that were mainly use as feedback for the user. Examples of texts contained in this folder can be observed on figure 6.2. The goal of this folder was to facilitate the change of theses descriptions without modifying the class(es) using them.

ContextSimulation.txt is a help file that contains some predefined commands that can be used in the context modifier tool to simulated desired context (de)activation for the application. The example of command contained in this file can be seen on listing 5.

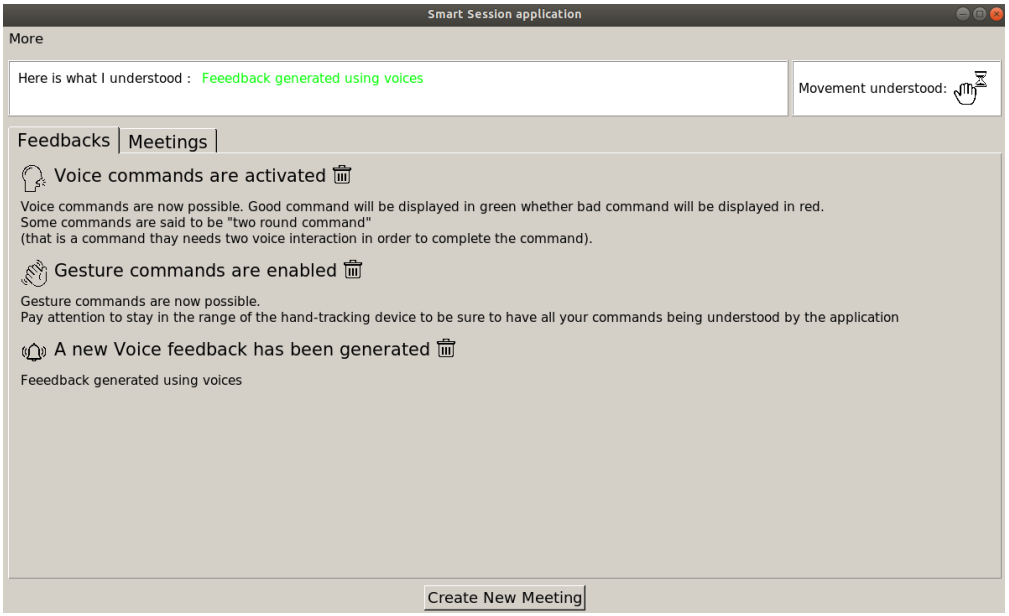

Figure 6.2: Example of Feedback

6.3.2 Voice feature implementation

Here we described how we added the voice feature in our application and how it interacts with other features.

UI

In the UI part, we created a View Module that is adapting the main application class. Within this UI, we simply add a white band at the top of the application screen that is showing what the system understood from the user, but also a message coming from the system that is answering from your request. The text containing what has been understood by the system, is colored in green if the system perceived what you said, otherwise it is colored in red as the feedback text provided. For instance in 6.3, the text "Add Benoit" is understood and the system answers by confirming that the user "Benoit" has been added in the meeting.

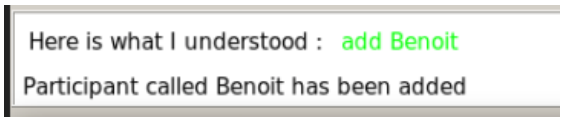

Figure 6.3: Example of a correct voice command

On the other side, we can see on figure 6.4 that the text is colored in red because there was an error during the execution of the command, and the system is making us know what happened via an appropriate feedback text.

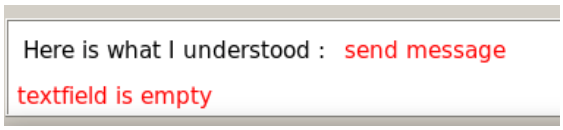

Figure 6.4: Example of an error report by the application via the feedback

Link with other features

In this application we added the features that is supporting voice command in modules that is adapting the desired view. In listing 15,is a simplified version of the meeting module which is integrating the voice interactions with only one command which is the message sending.

```
module MeetingVoice
    can_adapt :MeetingView

    set_prologue :init_voice
    set_epilogue :remove_voice_commands

    def init_voice()
      VoiceManager.instance.add_command(["send","new","message"],
      method(:send_voice),"What do you want to say ?")
    end

    def send_voice(msg)
      @meeting_model.send_message(MessageModel.new(msg))
      VoiceComHistory.instance.push_feedback("New Message directly sent")
      return true # no clean
    end

    def remove_voice_commands()
      VoiceManager.instance.remove_command(["send","new","message"])
    end
```

Listing 15: Simplified meeting voice module

The listing 15 initialize itself, by adding the command in the `VoiceManager` via the `add_command` method and when the feature is disabled, the command has to be removed from the `VoiceManager` via the method `remove_command`. The presence of the third argument containing the question "What do you want to say?" indicates that the command will be a two-parts meaning frame as described before in 4.3.1. The second argument is a reference to the class method `send_voice` that will be executed after receiving the user's answer to the question.

6.3.3 Gesture feature implementation

In this subsection, the introduction of gestures in our smart meeting application is detailed. Elements such as UI integration and gestures binding are described more precisely.

UI

In order to provide the user a proper feedback, we have designed a small gesture feedback window as a white stripe in the top right corner of the main window. This view was used to display the current gesture understood by the user through a dedicated icon. On the figures 6.5, 6.6 and 6.7, we can see the tree type of gestures that are currently detected and displayed by the gesture feedback.

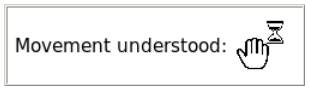

Figure 6.5: No gesture

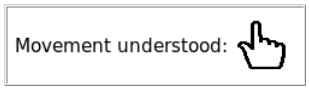

Figure 6.6: Pointing

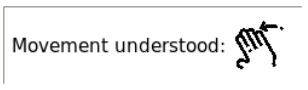

Figure 6.7: Swipe

Swipe

The swipe gesture can be performed through a simple hand swipe from any direction. By doing so the user is able to erase the content of the last text field he worked on. In order to add the gesture into the application, we had to perform the command previously describe in 5.3.4. We also had to find a way to link any writable field of the application to a focus mechanism so that we will be able to know which field need to be erased. This was made possible by having an attribute `current_field` in the GestureManager that is updated when a given filed is selected or used.

6.3.4 Multi modalities

Earlier on, we showed that we were capable of adding other modalities than the classic keyboard and mouse. However, those different modalities were strictly used individually. Here we prove that our solution additionally allows combinations of modalities and thus create real multimodal interactions.

Voice gesture

We implemented an multimodal interaction combining voice command to request an action and gesture for pointing the desired location. This combination is heavily inspired by the famous scenario of "put that there" describe by Bolt[2] but also from more recent attempt [22]. However, instead of drawing a precise form on the pointed location, we ask our application to write a given text on a pointed text field if there is any.

To implement this behaviour, we had to add a voice command for the words "write" and "there" which was bind to a method detecting the cursor position. Thanks to the FXRuby framework, we were able to easily detect which component was under the cursor by simply browsing the component tree from the top to bottom and applying the predefined function `underCursor()` which returns whether or not a UI component is currently under the mouse cursor. This works especially well, as the pointing gesture was taking control of the mouse cursor.

```ruby
def write_message(msg)
  children = @main_window.parent.children
  written = false
  until children.empty?
    child = children.shift()
    if (!child.instance_of?(Fox::FXTextField))
      children.concat(child.children)
    else
      if child.underCursor?
        child.text = msg
        GestureManager.instance.current_field = child
        written = true
      end
    end
  end
  if !written
    raise ArgumentError, "not textfield currently pointed at"
  end
end
```

Listing 16: Handling the "write that there" scenario"

An example of such interaction can be observed on figures 6.8 and 6.9 One can see on figure 6.8 that the cursor is on the chat's text field and that the voice command "Can you **write** something **there**" has been said. Then one can see the final result when the user says "Hello everybody" on figure 6.9. If the user moved their cursor away from the desired text field when pronouncing their message, the system will have provided them an appropriate feedback to inform them of the failure.

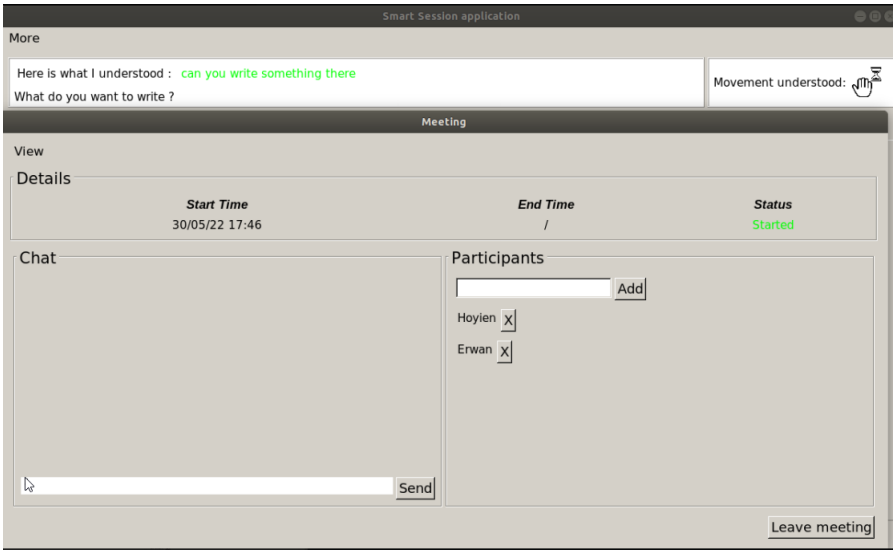

Figure 6.8: "Write that there" part 1

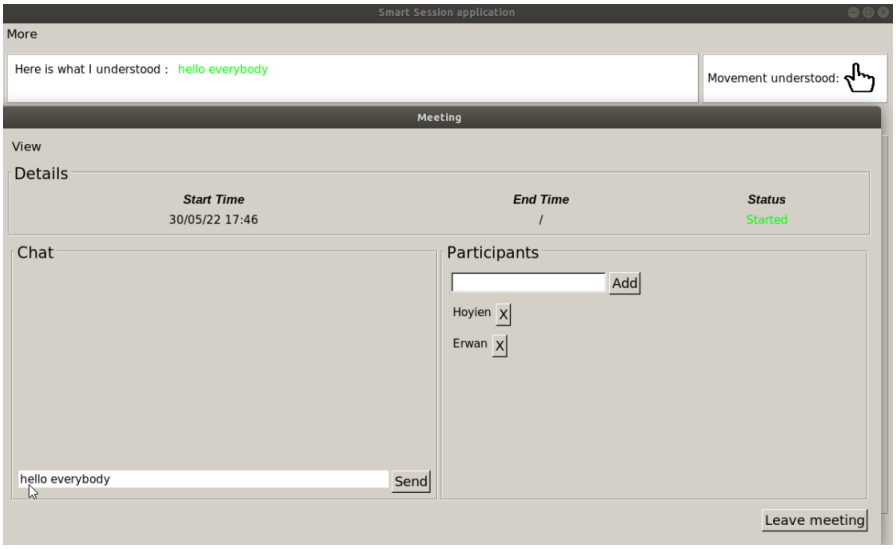

Figure 6.9: "Write that there" part 2

Voice vocal

As the first interest of implementing voice command was to allow blind user to interact with the application, we quickly noticed that, without any kind of vocal feedback from it, a blind user wouldn't be able to understand if the command he said was working. Thus, an additional feature that would mix the voice input and the vocal output was required. However, we encountered the problem previously in section 3.3.3 which was partially fixed with the introduction of command keyword

activation *Hey Siri*.

```
module VoiceVocal

  behaviour_adaptation()

  module Behaviour

    can_adapt :SmartSession

    def write_feedback(message, is_correct)
      proceed(message, is_correct)
      VocalFeedbackManager.instance.give_feedback(message)
    end

  end

end
```

Listing 17: Voice command vocal feedback

6.4 Testing

For testing our application, we used a scenario generator written by P.Martou [16] that generates random context changes and following those changes, we have to check the proper activation and deactivation of the features.

In the beginning, we assumed that only one user would use a given application instance. But what actually happened is that, the testing tool did not consider the permanent context which is mostly about the user's state (blind, deaf, ...) and still made a constant permutation between those state even though it is quite unrealistic. For instance, you may end up with a scenario with a deaf user who suddenly retrieves his hearing ability, or a perfectly normal user who happens to lose his ability to see right after. In the figure 6.10, we begin as blind and half-deaf. However, when we arrive at the 3^{th} state, notice how he regains his listening capacity (with the deactivation of the "HalfDeaf" context) to become Full. More surprising at the 8^{th} state, where our user is miraculously retrieving his ability to see even though blindness is currently permanent in reality.

To cope with this issue, we decided to tune the scenario such as there are many users using the same application instance instead of just focusing on one. As an example with the context changes described in the figure 6.10, we could say that a blind user is using it first before giving the device to someone who does not have any sight issue.

6.4.1 Test suite

This test suite has been generated by using the final model presented in 6.2 without the orange box.

	ADDED CONTEXTS	DELETED CONTEXTS
1	Pointer, Camera	
2	Keyboard	Micro, Camera
3	Full	Pointer, HalfDeaf
4	Micro, Screen	Speakers
5	Camera, Speakers	Keyboard
6	Pointer, Keyboard	
7	Noisy	Pointer, Micro, Calm
8	Perfect, Sufficient	Blind, Keyboard, Speakers
9	Daltonian	Perfect
10	Pointer, HalfDeaf, Micro, Keyboard, Calm	Noisy, Full
11	Noisy, Speakers	Camera, Screen, Calm
12	Perfect, Camera, Calm	Noisy, Pointer, Daltonian
13	Pointer, Screen, Full	HalfDeaf, Camera, Keyboard

Figure 6.10: Generated context switch for our application

However, the generated testing suite was focusing more on the context switching rather than the overall logic of the use case. Therefore, it was hard to create a decent scenario from it.

Chapter 7

Discussion

In this chapter, we describe the main strengths and weaknesses of using multimodal in an FBCOP environment. We then talk about the different advantages and inconveniences that our solution brings to the FBCOP programming framework. A personal feedback about the RubyCOP programming framework is provided afterwards. And eventually, we will discuss about the lesson learned throughout the implementation of the solution and the validation.

7.1 What does FBCOP bring to multimodal

Using an FBCOP approach to make a multimodal application brings some advantages and disadvantages. In this section, we are going to explain what they are and how they help or complicate the building process of such applications.

7.1.1 Application dynamism

A good advantage brought by the usage of FBCOP in designing multi-modalities is the ability to implement the numerous interactions independently of their context of use. Indeed, since all the features and contexts mapping is gathered in one file, we could then focus on how to integrate such modalities in our use case without thinking about when they should be activated or not. Such division allow to have a clearer and more readable code than in a classical programming approach where we would have to entangle them into a set of *if* constraint to define their contexts of use.

7.1.2 Difficulty

As any approach, there is some downside to be noted. Indeed, by introducing such an incredible flexibility in a framework, its overall complexity has increased rapidly. This increased complexity makes the creation of an application much harder using this approach than a classical one because we now have to think about the order in which the features are activated as there is a chance that some combination doesn't work when they should. For example, if we activate the *Camera* context before the *Micro*, we have to be sure the voice panel displays the command feedback will still appear on the left side of the screen in order to not confuse the user. Also if a given feature is needed before activating another one, we have to precise it in the feature model to avoid exceptions. A good way to test your application is to use a test scenario generator for FBCOP. By doing so, you will have a strong test scenario that will you to check for potential errors by combining in an efficient way most of the scenario that your application may go through without testing all of them.

7.2 Thoughts about our solution

In this section, we discuss about the quality of our approach in FBCOP in term of reusability, readability and scalability.

7.2.1 Architecture

In our approach, we decided to base the integration of the different modalities on the client server architecture described before in 4.2 that already existed. The advantage of implementing such integration on this existing architecture allow us to have a reusable approach, where the technology used behind can be interchanged or simulated without the RubyCOP application noticing it. Therefore, we could evolve the different modalities depending on the current trend and use state-of-the-art technologies regarding gesture or voices later on. For example, if we find a better voice recognition library in another language than python, we can easily change the dedicated code without altering the application. Also, we make room for many other modalities to be plugged to the framework later on without much difficulty. For instance, if you want to add a new modality, you just need to create a script that will perform the desired modality (such a gesture/voice detection, vocal feedback, ...) in the desired language. Then we will make it communicates via a Web-socket to the central server. On RubyCOP side, we will need to implement a singleton handler that will be used to receive input or send output depending on whether it is an input modality or an output modality. In the application, we could then easily decide on which data to react by adding or removing element as described

before in 5.3.3 and in 5.3.4.

Downsides

A big downside of this approach is the way we activate some of those modalities. As we did not succeed to add some modalities behaviour in a context-orient way due to UI bugs, we had to use a hack in order to allow desired behaviour to happen. These behaviours were mostly related to the `participant_list` feature which is shared between two different class (`Meeting` and `CreateMeeting`). Actually, when we tried to re-implement them in a dynamic way, the application encountered various unexpected bugs we were unable to fix. Those will be described further in 7.3.1. So we decided to preserve this hack in hope that it could be fixed in the future as one can see in section 8.1.3. The "hack" merely consists into including a boolean value in each modality handler that will tell if the given modality has to be performed or not. This boolean will then be bound to `prologue` and `epilogue` mechanism detailed in 2.2.3 to be respectively set to true or false. By doing so, we have been apt to statically defined some modality behaviour without having them working when the related context is deactivated. The principal reason why doing so is bad, as explained here in 4, is that it breaks the context-oriented paradigm as it removes its dynamism by adding untangled if conditions. We thus loose the advantages without freeing our self of the defaults of performing such paradigm.

7.3 Our experience with RubyCOP

In this section, we speak about our overall feeling about the usage of the RubyCOP framework when trying to add multimodal features and its validation in our study case.

7.3.1 Existing bugs

By implementing our study case in the framework, we were able to test its current version and found some bugs and/or unexpected behaviour that should be corrected as explained later in section 8.1.1. These different bugs are explained in detail into the two following subsections.

UI Bugs and limitations

We first discovered some inconsistencies/limitations when using FXRuby (the UI library used by RubyCOP). In most case, the bugs encountered could be prevented by modifying the way we implement features, but, some of these bugs seems to still happen periodically and could not be solved yet by any fix or workaround.

A classic example of these bugs is demonstrated by the implementation of the feature called *"meeting creation"* through voice command which always leads to a UI crash (due to a multi-threading error). In this case, due to these unexpected and uncomprehending bugs and also lack of workaround, this voice command could not have been integrated into the final application.

Another example of bug that have been encountered was when switching from *Blind* context to *Sufficient* in our application. This simple change can lead, sometimes, to incorrect behaviour such as a part of the textual feedback list which is not being updated as expected (extra text remaining or not appearing). Nevertheless, most of these bugs could be due to RAM space limitation and the use of this application on a virtual machine, which, usually, leads to loss of information, it is still important for us to enumerate them in the hope they could be fixed in the future. Through this use case we showed that despite the completeness and base solidity of this UI library, its actual use in an FBCOP application of bigger size on small machine could easily leads to frustrating and important bugs undoubtedly altering the global user experience. These bugs also confirmed us the importance to pay attention on the way of realizing such a complex RubyCOP UI on such type of environment with a limited RAM space. For example, a effective way to reduce UI impact on the machine was to prevent unnecessary creation/deletion of a component over and over again when one instance could be kept and updated in consequence.

RubyCOP Problems and Bugs

Other problems were related to the FBCOP framework on itself, preventing it from working correctly as it was intentionally designed. One of the problems found is related to multiple context activation – i.e., when a context is activated two or more time and thus trigger the same feature again and again.

There is also the simultaneous (de)activation of some context's combination that leads to a non proper (de)activation of a given feature. For example, we have a feature called *"Voice"* that could be only activated when the room is *Calm* and when you possess a working *Micro*. If both contexts (*Calm* and *Micro*) are suddenly deactivated at the same time, then the targeted feature (*Voice*) is still maintained when in fact none of the required constraints are met.

7.3.2 Reusability in RubyCOP

As any classical programming framework, RubyCOP offers the possibility to write reusable code In order to make such a thing possible, you first need to tell in the feature which class it can adapt through the keyword *can_adapt* as seen in 2.2.3. If done well, a same generic feature could potentially adapt multiple classes in a

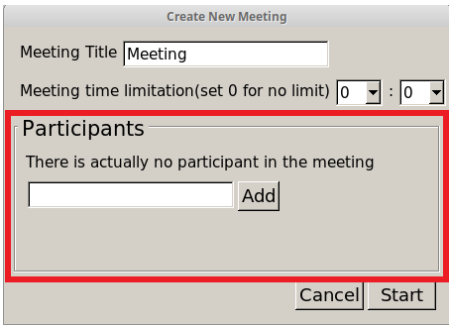

Figure 7.1: Create Meeting Pane

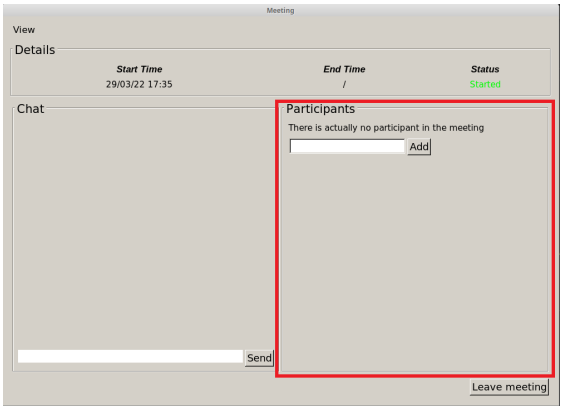

Figure 7.2: Classic Meeting pane view

similar way. One good example of this reusability possibility in our study case can be seen in the participants list feature shared between the *Create Meeting dialog* and the *Meeting dialog* (See figure 7.1 and figure 7.2). Since both dialogs need a same participant list option, the feature was only made once and told to adapt both classes.

Benefits and Downsides

The interest of supporting only one implementation for a similar feature used in two distinct parts of the application is that helps to reduce the number of introduced bugs for that feature. For instance, if this component was duplicated, when you encounter a bug for that feature, you will need to correct potentially two implementations when in the other case, only one file will need to change in order to see this modification. Also if the developer decided to re-implement the whole feature again, there is a chance that the new implemented feature doesn't work the same as the other, resulting in a potential bad experience for the end user that might assume that both components are identical in appearance and behaviours.

However, in our study case, this choice wasn't only beneficial, because alongside with UI bug, this choice forced us to make a hack in order to integrate the desired interactions for this feature as explained above in 7.2.1. We could be able to make such interactions by developing a duplication of this feature and adding each interaction as separate feature defined in `CreateMeeting` and `Meeting`. But because our goal was to write a code that is as clean as possible, and because these bugs were only related to the UI rather than the framework itself, we decided to keep such a reuse in our code.

7.4 Lessons learned

After having work with such a framework for almost a complete academical year, we've been experienced to distinguish various good and bad practices that any future FBCOP developer should pay attention to.

7.4.1 Bad & Good Practices

There are 5 different practices that one has to take in account in order to avoid undesired result and/or possible future problems.

1. **Naming any method** : When writing FBCOP code, you have to consider that any method declare in a feature will be shared with the adapted class. In other words, all methods defined are *public* and could be changed or updated by any new activated feature (*private* methods cannot be defined since they could not be used by the targeted class they adapt). To avoid name clash, you should not use generic method names but go for more specific one (unless it's a method you want to be improved by each new feature).

2. **Ordering method definition** : The declaration order of a method occupies a crucial role in FBCOP program. If you define a method in another file that where it's being used in another one, this could leads to an *undefined method* problem. As a matter of fact, if the defining feature is not activated when the using feature is then a runtime error will occur. It's therefore recommended to declare methods where they are used or, in case it's necessary, to force the activation of the required feature(s) before activating the others.

   For example in listing 24 we can see that the method *stock_color_palette()* is not declared because it's has been done in *parent* feature at listing 25. To avoid that this feature is activated without the required one, we have to tell the framework that they are related together as shown in listing 18.

```
def _define_set_color_palette_feature()
  feature :@set_color_palette, 'SetColorPalette', [:SmartSession]
  feature :@blindsafe, 'BlindSafe', [:SmartSession]
  feature :@default_color, 'DefaultColor', [:SmartSession]
  feature :@daltonian_dialog, 'DaltonianDialog', [:SmartSession]
  @set_color_palette.relation :Alternative, [@blindsafe, @default_color]
  @set_color_palette.relation :Mandatory, [@daltonian_dialog]
end
```

Listing 18: Example of feature ordering in the feature model

3. **Defining the same method** : When two diverse features define a same method that is undefined in the parent, it's a good practice to define this method as an empty method in the parent and to enhance it's definition in both child features by using the *proceed()* mechanism in order to preserve the content of each definition and thus avoid any final overwrite by any definition. An example of such practice can be identified on listings 19, 20 and 21

```
class MeetingView

  include IdGeneration

  def init_interactions()
  end

  def remove_interactions()
  end

end
```

Listing 19: Empty method in parent

```
  module MeetingGesture

    can_adapt :MeetingView

    set_prologue :init_gesture
    set_epilogue :remove_gesture

    ...

    def init_interactions()
      proceed()
      init_gesture()
    end

    def remove_interactions()
      proceed()
      remove_gesture()
    end

  end
```

Listing 20: Gesture adaptation of init_interaction

```
  module MeetingVoice

    can_adapt :MeetingView
    set_prologue :init_voice
    set_epilogue :remove_voice_commands
    ...
    def init_interactions()
      init_voice()
      proceed()
    end

    def remove_interactions()
      remove_voice_commands()
      proceed()
    end
  end
end
```

Listing 21: Voice adaptation of init_interaction

4. **Making Context feature** : One essential thing to pay attention to when making a feature is to remember the basics of such paradigm that concentrate on adding/removing features depending of (de)activated contexts. Thus when making a given feature, it's important to think about what is mandatory and what is not. Based on that some implementations parts should be made in dedicated optional features. You should not use this paradigm in a lazy way where a big feature will contain all functionalities. Then simply change the behaviour based on *epilogues* and *prologues* that modify a boolean variable to activate or not the desired part.

5. **Context-Oriented Observer** : an interesting element to rethink arises from the way to implement an observer in such kind of application. Here since all methods are shared between each feature, there is a significant chance to encounter a problem when realising an observer in the classical way. Indeed, method such as update could be overwritten by any activated feature or maybe skipped in case a given element is already handled. In order to avoid such issues, it's important to think carefully on how to design the *update* method. A efficient way of doing so is to define a default update method in the parent that will only throw exceptions telling the received message is not handled as it can be see on the listing 22 This method should also contain a parameter that defines what kind of update has been received in order to be properly dispatch in the adapting features implementation. Then each feature adapting this class should follow the same method signature and only

add a part dedicated to them. In listing 23, one can see how the adaption will improve the parent update version by adding an handling for a given type of message. It's important to always put at the end of each update a *proceed* call with the same arguments so that unhandled message here will be directed to the next adapting feature. **Pay attention to exclusively handle a given message once or the oldest implementation will never be reached!** When multiple features handle a same message type, only one feature should link the update method to it's dedicated handling method. Then the other feature should override this handling method through the proceed() mechanism as explained in section 2.2.3. If both feature are optional and may not be activated in same order, you could refer to the practice number 3.

```
def update(type, elem)
  puts "#{self.class.name}: Unknown/Unhandled type of message receive [#{type}]"
end
```

Listing 22: Default update method in SmartSession

```
def update(type, elem)
  case type
  when :new_movement then set_gesture_icon(elem)
  else
    proceed(type, elem)
  end
end
```

Listing 23: Adaptation of update method for gesture

Chapter 8

Future Works

8.1 Framework Improvements

The framework used to test the introduction of new modalities was, as a whole, correct. However as describe in the previous chapter, different problems already existed before we introduced our modalities handler. Problems that may have been increased by the introduction of theses new interactions managers. A future work that will need to be done is related to the overall quality of this framework. We described in next sections all parts that, we think, can be improved.

8.1.1 UI problems

As mentioned before in 7.3.1, a set of various UI bugs coming from *FXRuby* that prevented us from implementing some features or even forced us to perform a language *'hack'* as described in section 7.2.1. One potential future work could be to investigate all of these various UI bugs to see if they can be fixed or if they are specific to the UI library. In case the UI doesn't fit in this framework it might need to be changed.

8.1.2 RubyCOP improvement

Another thing that could be fixed are related to the current implementation of the RubyCOP programming framework on itself. As some of the bugs described before in 7.3.1 could become tedious for future FBCOP developers, one could try understand their cause and see how to correct them.

8.1.3 Multi-modalities

Something that will need probably to be done in a near future is an improvement of the current implementation of the multi-modalities on the framework side. As this thesis was a first experience for us in the creation of multimodal application in a context oriented framework and the fact we only limited ourselves to a small use case, some implementation or design choices might not have been done well and/or might be incomplete. Here is a list of potential elements that could be improved:

- the lack of filtering for the gesture part that could become a problem for applications with a lot of gestures.
- self talking application problem describe in section 3.3.3 that was only partially prevented in our current implementation.
- a better keyword activation for voice filtering instead of our easiness-based choice.
- fixing the "hack" performed to use the reusable feature `participant_list` without any problem.

A good way to test the limitation of the multimodal part will probably be to try various other use cases for more a diverse set of multi-modalities or one could also try to extend the actual use case with an increase numbers of interactions possibilities as described in section 8.2.

8.2 Improve use-case

Since were limited by the time, some of the desire behaviour could not be made and the chose use case has to be made shorter to ease our implementation. In the actual use-case, our application is more a Proof of Concept rather than an actual application that can be used. A potential future work would be to improve such a use case in order to reach the size of a real multimodal application. In the following subsections, we give some tracks of elements that could be changed or improved?

8.2.1 Improve modalities services

Modality services have been implemented here by developers with a basic knowledge of such type of interactions. These services, especially the gesture recognition, could be rework in order to capture more precisely the user interactions and thus ease the application usage.

Gesture

Concerning the gesture part, we were only able to use a python 2.7 library for the leap motion as they were no generated version for python 3. Since this library was generated through SWIG, a more actual version for python 3 could be generated by using an appropriate SWIG version. This change can be considered as a minor thing but as python 2.7 is outdated, it would be interesting to stop using it.

Voice

Voice command Regarding the voice interactions, we already explained how we achieved to make the system understand basics command using the meaning frame technique. However, it would be interesting to find ways to make the system understand a same request with synonym or different formulation that has the same meaning. If we keep using the meaning frame concept, especially the two_parts meaning frame explained in 4.3.1, we could extend that idea by implementing a Markov chain that would look into the current state of the voice handler and then make commands according to the state the handler is currently in. Following that method, we would be able to create a N-parts meaning frame.

Noise filtering In this thesis, we used the "Hey Siri" keywords for activating the voice command. However, this choice was made out of convenience, because our goal was to see the impact of filtering rather than the quality of the keyword itself (easy to pronounce, no conflict with real word, ...). An interesting future work could be to find another keyword that is easily detected without being misunderstood with classical words but also easy to use

8.2.2 Improve features set

A good way to transform such POC into a more concrete application is to add new features that were not introduced to lack of time such as meeting canvas, introduction of an online mode and real participants to make this a real multimodal FBCOP meeting application.

8.2.3 Potential evaluation

Another potential work could be to evaluate the quality of the multimodal context oriented user interface that has been generated using a set of methods as described in this HCI paper [15] This study would help to see the actual quality of the use interface made for this study case and actually see if the adaptation to contexts makes it harder to use.

Chapter 9

Conclusion

To conclude this thesis, we show that it is possible to enhance the Feature-Based Context Oriented Programming framework with multi-modalities features. Based on the RubyCOP framework, we provided a solution that allows to introduce any kind of new modalities without having to care about the language in which to implement them. This feat is achieved by connecting the different modalities implementation to a remote tooling server that will exchanges the data between the modalities and the FBCOP application.

We then validated our solution by implementing a study case of a smart meeting application that provides the possibility to the user to interact using additional modalities such as input voice command, input gesture, and a vocal output. The application adapted its interaction availability following the surrounding context that will define element such as the presence of adapted device or user capacities. The implementation of this study case was then tested based on test scenario generated via a FBCOP test suite generator.

With that validation, we were able to notice the advantages and downsides of using a FBCOP framework to realise multimodal applications which can be merely resumed into a better clarity and separation of code and an increased complexity coming from the vast possible set of context activation order. It additionally helped us to see the imperfection our approach in term of abstraction that could be fixed in a foreseeable future. We also discovered some interesting gaps about the UI and core logic in the RubyCOP programming framework that should be fixed to improve its potential use by future FBCOP developers. Finally, we also learnt various good and bad practices for implementing such solutions and prevent the introduction of accidental complexity due to bad comprehension or usage of such a paradigm.

Appendices

Appendix A

Code samples

```
module DefaultColor

  module UILayout

    can_adapt :SmartSession

    set_prologue :color_all
    set_epilogue :stock_color_palette

    def create_color_palette(new_color_palette = color_palette())
      proceed(new_color_palette)
    end

    def color_palette()
      color_palette = {
        ...
      }
      return color_palette
    end

    def color_all()
      create_color_palette()
      puts "Default color set"
      proceed()
    end

  end

end
```

Listing 24: Default color feature

```
module SetColorPalette

  user_interface_adaptation()

  module UILayout

    can_adapt :SmartSession

    def create_color_palette(new_color_palette=Hash.new)
      UI::getUIManager.color_palette = new_color_palette
    end

    def stock_color_palette()
      @old_palette = UI::getUIManager.color_palette
    end

    def color_all()
      if @old_palette
        color_all_children(@main_window.parent)
      end
    end

    def color_all_children(ui_frame)
      ...
    end

    def translate_color(ui_frame, method_name)
      ...
    end

  end

end
```

Listing 25: Set color Palette feature

Appendix B

Test suite

Test suite
CORE CONTEXTS : 'Audition', 'Screen', 'Context', 'Input', 'Sufficient', 'Environment', 'Output', 'Vision'
CORE FEATURES : 'Feature', 'Interaction', 'Description', 'SetColorPalette', 'Meeting', 'CreateMeeting', 'Picture', 'ProvideFeedback', 'MeetingHistory' 'Message', 'ParticipantsList', 'ManageFeedback', 'MeetingBarLayout', 'DisplayAddition'
BASE CONTEXTS : HalfDeaf, Blind, Micro, Vision, Audition, Environment, Input, Speakers, Output, Calm, User, Context
BASE FEATURES : Voice, ManageFeedback, VoiceVocal, Intensity, DefaultColor, VocalOutput, Feature, Multimodal, SetColorPalette, Audio, Meeting, ProvideFeedback, MeetingHistory, Interaction, UserCapacity, Message, System, CreateMeeting, HighVolume, MeetingBarLayout, ParticipantsList

	ADDED CONTEXTS	DELETED CONTEXTS	ADDED FEATURES	DELETED FEATURES
1	Pointer, Camera		VoiceGesture, Gesture, ExternalDevice	
2	Keyboard	Micro, Camera	Writing	Voice, VoiceVocal, Multimodal, VoiceGesture, Gesture
3	Full	Pointer, HalfDeaf	NormalVolume	ExternalDevice, HighVolume
4	Micro, Screen	Speakers	Voice	Intensity, NormalVolume, VocalOutput, Audio
5	Camera, Speakers	Keyboard	VoiceVocal, Intensity, NormalVolume, VocalOutput, Multimodal, Audio, VoiceGesture, Gesture	Writing
6	Pointer, Keyboard		Writing, ExternalDevice	
7	Noisy	Pointer, Micro, Calm	HighVolume	Voice, VoiceVocal, NormalVolume, Multimodal, VoiceGesture, ExternalDevice
8	Perfect, Sufficient	Blind, Keyboard, Speakers	Description, Picture, DisplayAddition	Intensity, VocalOutput, Writing, Audio, HighVolume
9	Daltonian	Perfect	BlindSafe	DefaultColor
10	Pointer, HalfDeaf, Micro, Keyboard, Calm	Noisy, Full	Voice, Writing, Multimodal, VoiceGesture, ExternalDevice	
11	Noisy, Speakers	Camera, Screen, Calm		Voice, Multimodal, VoiceGesture, Description, Picture, DisplayAddition, Gesture
12	Perfect, Camera, Calm	Noisy, Pointer, Daltonian	Voice, DefaultColor, Multimodal, VoiceGesture, Gesture	BlindSafe, ExternalDevice
13	Pointer, Screen, Full	HalfDeaf, Camera, Keyboard	Description, Picture, DisplayAddition, ExternalDevice	Writing, Multimodal, VoiceGesture, Gesture

Bibliography

[1] Till Ballendat, Nicolai Marquardt, and Saul Greenberg. Proxemic interaction: designing for a proximity and orientation-aware environment. In *ACM International Conference on Interactive Tabletops and Surfaces*, pages 121–130, 2010.

[2] Richard A Bolt. "put-that-there" voice and gesture at the graphics interface. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 262–270, 1980.

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

[4] Joëlle Coutaz, James L Crowley, Simon Dobson, and David Garlan. Context is key. *Communications of the ACM*, 48(3):49–53, 2005.

[5] Pierre Dillenbourg and Michael Evans. Interactive tabletops in education. *International Journal of Computer-Supported Collaborative Learning*, 6(4):491–514, 2011.

[6] Carlos Duarte, Simon Desart, David Costa, and Bruno Dumas. Designing multimodal mobile interaction for a text messaging application for visually impaired users. *Frontiers in ICT*, 4:26, 2017.

[7] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. Dynamic visualisation of features and contexts for context-oriented programmers. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2019.

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

[9] Benoît Duhoux, Kim Mens, and Bruno Dumas. Implementation of a feature-based context-oriented programming language. In *Proceedings of the Workshop on Context-oriented Programming*, pages 9–16, 2019.

[10] Bruno Dumas. *Frameworks, description languages and fusion engines for multimodal interactive systems*. PhD thesis, Université de Fribourg, 2010.

[11] Bruno Dumas, Denis Lalanne, and Sharon Oviatt. Multimodal interfaces: A survey of principles, models and frameworks. In *Human machine interaction*, pages 3–26. Springer, 2009.

[12] 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*, pages 12–21. IEEE, 2008.

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

[14] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[15] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. Evaluation strategies for hci toolkit research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2018.

[16] Pierre Martou, Kim Mens, Benoît Duhoux, and Axel Legay. Test scenario generation for context-oriented programs. *arXiv preprint arXiv:2109.11950*, 2021.

[17] Kim Mens, Rafael Capilla, Nicolás Cardozo, and Bruno Dumas. A taxonomy of context-aware software variability approaches. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 119–124, 2016.

[18] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. Modeling and managing context-aware systems' variability. *IEEE Software*, 34(06):58–63, 2017.

[19] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 7–12, 2016.

[20] Nesrine Mezhoudi and Jean Vanderdonckt. A user's feedback ontology for context-aware interaction. In *2015 2nd world symposium on web applications and networking (WSWAN)*, pages 1–7. IEEE, 2015.

[21] Vivian Genaro Motti and Jean Vanderdonckt. Context-aware adaptation of user interfaces. In *IFIP Conference on Human-Computer Interaction*, pages 700–701. Springer, 2011.

[22] Vik Parthiban, Pattie Maes, Quentin Sellier, Arthur Sluÿters, and Jean Vanderdonckt. Gestural-vocal coordinated interaction on large displays. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'22)*, 2022.

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

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