

FEATURE-BASED CONTEXT-ORIENTED SOFTWARE DEVELOPMENT

Benoît Duhoux

*Thesis submitted in partial fulfillment of the requirements for
the Degree of Doctor in Engineering Sciences and Technology*

August 2022

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Kim Mens (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Bruno Dumas (Advisor)	UNamur, Belgium
Pr. Charles Pecheur (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Jean Vanderdonckt (Secretary)	UCLouvain/ICTEAM, Belgium
Pr. Nicolás Cardozo	UniAndes, Colombia
Pr. Kris Luyten	UHasselt, Belgium

Feature-Based Context-Oriented Software Development

by Benoît Duhoux

© Benoît Duhoux 2022

ICTEAM

Université catholique de Louvain

Place Sainte-Barbe, 2

1348 Louvain-la-Neuve

Belgium

ABSTRACT

Context-oriented programming enables dynamic software evolution by supporting the creation of software systems that dynamically adapt their behaviour depending on the context of their surrounding environment. Upon sensing a new particular situation in the surrounding environment, which is reified as a context, the system activates this context and then continues by selecting and activating fine-grained features corresponding to that context. These features, representing functionalities specific to that context, are then installed in the system to refine its behaviour at runtime. Conceiving such systems is complex due to their high dynamicity and the combinatorial explosion of possible contexts and corresponding features that could be active.

To address this complexity, we propose a feature-based context-oriented software development approach to design and implement context-oriented applications. This approach unifies context-oriented programming, feature modelling and dynamic software product lines into a single paradigm. In this novel paradigm we separate clearly and explicitly contexts and features that we model in terms of a context model, a feature model and the mapping between them. We also design an architecture, implement a programming framework, and develop a supporting development methodology and two visualisation tools to help designers and programmers in their modelling, development and debugging tasks. Furthermore we also develop a user interface library in our approach to create applications with user interfaces that are adaptive.

To validate our feature-based context-oriented software development approach, we designed five case studies and implemented three of them. Then

we discussed the design qualities to evaluate our implementation of the programming framework. We also assessed the usability of the programming framework from our own perspective based on the cognitive dimensions of notations framework. Finally we also conducted four user studies with real users in which we asked them to play the role of designers and programmers to validate the understandability, usefulness and usability of our approach. The results we gathered from our participants are promising and provide us several paths to enhance our approach.

ACKNOWLEDGMENTS

Tout d'abord, j'aimerais te remercier Kim M. pour la chance que tu m'as donnée de faire ce doctorat. Tu m'as poussé en avant dans cette thèse avec ton pragmatisme afin que je puisse relever des défis qui me semblaient infranchissables. De plus, avec ton expérience, tu as pu m'encadrer d'une manière que je ne pouvais espérer meilleure. C'était un réel plaisir de travailler avec toi.

J'aimerais aussi te remercier Bruno D. pour l'intérêt que tu as porté à ma recherche. Tu m'as également appris énormément de ton domaine : l'interaction homme-machine. Nos discussions étaient toujours très intéressantes et enrichissantes, même si j'y ramenaient très souvent mon regard de génie logiciel. Merci aussi à toi pour ce stage de recherche que tu m'as offert. J'ai pu y découvrir un nouvel environnement, faire mieux connaissance avec ton équipe et d'autres chercheurs.

Ensuite, merci à vous, membres de mon jury, Nicolás C., Kris L., Jean V. et Charles P., pour vos feedbacks pertinents lors de ma défense privée qui m'ont permis d'améliorer mon manuscrit. Merci aussi à Nicolás C. et Jean V. pour tous ces moments partagés ensemble.

J'aimerais aussi remercier les personnes du département INGI pour cette ambiance incroyable de travail, ainsi que pour toutes ces activités que nous avons faites ensemble. Sans vous, cela n'aurait pas été pareil. J'aimerais aussi remercier mes deux équipes de recherches ainsi que mes co-bureaux pour toutes nos discussions. J'aimerais remercier en particulier, Guillaume M. et Xavier G. pour toutes nos discussions toujours intéressantes, Gorby K. et Mathieu J. pour ce nouveau projet commun qu'était le Club INFO, Axel L.

pour ton investissement et ton écoute, Quentin D. pour ton template de thèse, qui, il faut l'avouer, m'a simplifié la vie lors de l'écriture de ma thèse et Fabien D. pour ta décoration sur la porte de mon bureau en cette fin de thèse.

Mille mercis au personnel administratif et technique d'INGI et en particulier à Vanessa M. pour ton aide inestimable à toutes mes questions concernant l'administration et la logistique, Sophie R. pour ton aide sur mes questions comptables, Ludovic T. et Anthony G. pour votre réactivité à mes questions techniques.

J'aimerais aussi remercier mes étudiants qui ont participé à nos différentes études utilisateurs à travers les années, et sans qui nous n'aurions pu valider notre approche de cette manière. Merci aussi à mes mémorants qui m'ont apporté différents points de vue sur les extensions de ce travail. Un merci particulier à Pierre M., Hoo Sing L., Edwin D., Ho Yien T., Julien L. et Céline N. pour vos contributions dans cette thèse.

J'aimerais également remercier tous mes amis qui m'ont soutenu durant toute ma thèse, et qui m'ont permis de m'évader : Isabelle, Frédérique, Christel, Ju, Nicolas, Michel, Dimitri, Sabrina, Nathalie, Laurent, Sébastien, Michelangelo, Jérémy, Wathy, Sébastien, Lionel, Alexandre, et tant d'autres...

Un tout grand merci Papy pour ton soutien à travers toutes ces années.

Enfin, j'aimerais te remercier Parrain (Vincent). Tu as cru en moi et m'as soutenu, et sans cela, je n'aurais jamais réussi à évoluer de cette façon. Merci aussi à tes parents, Jacques et Brigitte, qui m'ont également supporté dans mes choix depuis ces longues années, ainsi que ta famille, Claude, Victoria, Jean-Marc, Isabelle, Lucas et Daliane.

MERCI!

CONTENTS

Abstract	i
Acknowledgments	iii
Table of Contents	iv
I Prologue	1
1 Introduction	3
1.1 Setting the scope	3
1.2 Research questions	5
1.3 Contributions	7
1.4 Supporting publications	10
1.5 Running example: a smart messaging system	13
1.6 Roadmap	14
2 State of the art	17
2.1 Modelling	18
2.2 Implementation	25
2.3 User interface adaptation	26
2.4 Supporting tools	30
2.5 Discussion	33

II	Approach and usage	35
3	Feature-based context-oriented programming paradigm	39
3.1	Principles and concepts	39
3.1.1	Feature	40
3.1.2	Feature model	41
3.1.3	Context	43
3.1.4	Context model	44
3.1.5	Context-feature mapping	46
3.1.6	Mapping model	47
3.2	System architecture	48
3.3	Development methodology	50
3.3.1	Requirements	50
3.3.2	Design	52
3.3.3	Implementation	52
3.3.4	Testing	53
3.3.5	Iterative methodology	53
3.4	Conclusion	53
4	Feature-based context-oriented programming framework	55
4.1	Defining the application classes	56
4.2	Declaring contexts and features	57
4.3	Defining features	58
4.4	Declaring the context model	61
4.5	Declaring the feature model	64
4.6	Mapping contexts to features	66
4.7	Managing the activation order	67
4.8	Activating contexts	70
4.9	Proceeding feature execution	71
4.10	Language versus framework	74
4.11	Conclusion	75
5	User interface adaptation library	77
5.1	Representation of user interfaces	78
5.2	Overview	79
5.3	API and usage of the UIA library	80
5.4	UIA library and features	87
5.5	Conclusion	90
6	Visualisation tools	91
6.1	CONTEXT AND FEATURE MODEL VISUALISER	91
6.1.1	Visualisation	92
6.1.2	Functionalities	92

6.2	FEATURE VISUALISER	98
6.2.1	Visualisation	98
6.2.2	Functionalities	101
6.3	Conclusion	105
III	Implementation	107
7	Entities	111
7.1	Context and feature entities	111
7.2	Context and feature model declarations	117
7.3	Context and feature model definitions	118
7.4	Mapping model declaration	121
7.5	Mapping model definition	122
7.6	Conclusion	123
8	Modelling	125
8.1	Structure of an entity model	125
8.2	Satisfiability algorithm	127
8.3	Conclusion	129
9	Architecture	131
9.1	Control flow	131
9.1.1	CONTEXT ACTIVATION	132
9.1.2	FEATURE SELECTION	133
9.1.3	FEATURE ACTIVATION	134
9.1.4	FEATURE EXECUTION	135
9.2	Dynamic adaptation	136
9.3	Proceed mechanism	141
9.4	Conclusion	146
10	Tool support	149
10.1	Overview of the communication	150
10.2	TOOLSUPPORT in the implementation architecture	151
10.2.1	CONTEXT AND FEATURE MODEL VISUALISER	153
10.2.2	FEATURE VISUALISER	153
10.2.3	CONTEXT SIMULATOR	154
10.3	Extensibility and evolution	154
10.4	Conclusion	155

IV	Validation	157
11	Validating FBCOP's expressiveness	161
11.1	Smart messaging system	162
11.2	Another smart messaging system	182
11.3	Smart risk information system	186
11.4	Smart meetings system	187
11.5	Smart city guide	193
11.6	Conclusion	196
12	Validating FBCOP's design	197
12.1	Design qualities	197
12.2	Cognitive dimensions framework	200
12.3	Conclusion	206
13	Validating the FBCOP approach with users	207
13.1	Setup for all user studies	208
13.2	Preliminary study of the FEATURE VISUALISER	208
13.3	Preliminary study of the CONTEXT AND FEATURE MODEL VI- SUALISER	215
13.4	First complete user study	220
13.5	Second complete user study	238
V	Epilogue	251
14	Future work	255
14.1	Improving FBCOP's expressiveness	255
14.2	Adding sensory layer and context definitions	257
14.3	Improving and adding tools	258
14.4	Performance and scalability issues	258
14.5	Other concerns	259
15	Conclusion	261

Part I

Prologue

CHAPTER

1

INTRODUCTION

1.1 Setting the scope

In the vision of pervasive and ubiquitous computing, end-users can perform any user action or task on any device at any moment of the day. For instance, Benoît can read and answer a mail on his smartphone while waiting for the bus to visit some friends. He can consult his agenda on his phone to fix a lunch with his godfather while he is walking on the street, or control his heart rate on his smartwatch while running. All these computing services allow end-users to be connected everywhere and at any time. However they could become even more powerful by taking into account any information the system can collect about the surrounding environment in which the device of the end-user runs. Assume someone is driving on the highway and receives a message. It could be really dangerous to read and answer it. But knowing he is driving, the service could adapt by reading the message out loud so that he can keep his focus on the road. The system could also propose some predefined messages to minimise the interaction the driver needs to have with his phone, or it could redirect the messages to the car dashboard. Another interesting example is when a researcher enters in a meeting room where she must present her work. To avoid being disturbed by notifications from her social networks, when her phone detects she enters the room, it can snooze automatically her notifications from Instagram and Twitter. Even if these examples are simple, we can immediately see the interest of such services that are able to adapt depending on the surrounding environment in which

they run.

Sensing the environment of running systems dynamically and adapting their behaviour at runtime depending on this sensed information allows the creation of context-aware systems. This information is described as contexts, representing either the user itself (the user's preferences, ...), external services (the weather, ambient noise), or internal sensors of the device on which they run (connectivity, battery level). Developing systems that dynamically adapt their behaviour depending on changing contexts can be more complicated with traditional programming paradigms such as the object-oriented programming paradigm due to the high-level of dynamicity of such systems. To address these issues the context-oriented programming paradigm emerged to build such context-aware systems. Context-oriented systems are able to adapt their features whenever a change is detected by either adding, removing, replacing or refining some of their functionalities. When a new particular situation is sensed, the system reifies this information as a context. Then it selects the features that will be activated or deactivated depending on the newly discovered context. Finally it installs or removes these features in the system to adapt its behaviour at runtime.

Designing context-oriented systems can be quite complex however, because developers have to think about what features such a system should have and to what contexts it should adapt their behaviour. The adapted components can range from user interface components to core logic or even data components. They should think carefully about which contexts trigger which features. When modelling such systems, this leads to an interesting challenge that is the exponential number of combinations we can have between the contexts, the features and how they are linked.

In this thesis we attempt to tackle the design and development issues to help designers and developers of the context-oriented systems in their respective tasks with a new paradigm: feature-based context-oriented programming (FBCOP, for short). This paradigm relies on the context-oriented programming paradigm [HCN08], the notion of feature modelling [Kan+90] and dynamic software product lines [HT08; Ach+09; COH14; Men+17]. The first issue this novel paradigm tackles concerns the foundations of context-oriented programming by proposing an explicit separation between contexts and features. Taking inspiration from Hartmann and Trew [HT08], this paradigm models them as a separate context model and feature model, including a mapping model between them. Then we propose a specific architecture and programming framework for FBCOP. We also provide a first prototype of an adaptive user interface library so that the FBCOP approach can be used to create more realistic applications for users. Having a programming paradigm without an accompanying development methodology may be useless. We therefore propose a supporting development methodology for the FBCOP

paradigm as a complete process to help designers and developers to create such context-oriented programs. Finally we implement two visualisation tools to help programmers of FBCOP systems in their development and debugging tasks.

In addition to the case studies we will present and implement throughout the thesis as proofs of concept of our approach, we will also discuss the design qualities of the implementation of the programming framework, assess FBCOP's usability based on the cognitive dimensions of notations framework, and perform user studies with potential designers and developers to assess the comprehensibility and usability of the FBCOP approach.

Next, we will introduce more precisely the research questions that will be addressed in this thesis.

1.2 Research questions

Context-oriented programming is not a novel programming paradigm. Much work has been done to build its underlying principles, its programming languages and so on. But many areas of this paradigm are not yet covered from either a technical or application point of view, such as the modelling, the integration of user interface adaptation, supporting tools to help designers and developers and more. To contribute to the evolution of this paradigm we explore the maintainability and reusability of systems that adapt their behaviour depending on the surrounding environment (*i.e.* contexts) in which they run. This leads us to work on the following main research question: “*how can we build context-oriented systems in a more maintainable and reusable way?*”. This question is split in the following questions that will guide our research.

RQ1 *How can we design the contexts and features of a context-oriented system to promote the maintainability and reusability of these entities?*

Contexts are a key notion [Cou+05] in a context-oriented system because they reify the particular situation of the surrounding environment in which the system runs. When changes are sensed in the environment, the system dynamically adapts its behaviour by adding or removing features (also called behavioural adaptations). Different techniques exist to design contexts and features. In the modelling area they can be modelled either with an explicit separation [HT08; Ach+09; COH14; Men+17] or together [Des+07]. In the implementation area they are often designed without a clear and explicit separation between both [Gon+11; Lin+11; SGP11]. We explore the potential of a single unified design bridging the areas of modelling and implementation.

RQ2 *How can we build an architecture or architectural pattern to support the designers and programmers in the conception of such context-oriented systems?*

Much work has been done to design different architectures for context-aware systems, *i.e.*, systems that can adapt their behaviour depending on the sensed contexts [BDR07; CGM09]. Nevertheless such architectures do not propose a clear separation of contexts and features to facilitate the maintainability and reusability of such notions.

RQ3 *How can we design a programming language or framework to help programmers develop more maintainable and reusable context-oriented programs?*

There are many context-oriented programming language implementations [CH05; AHR08; CD08; HCH08; GPS10; AKM11; Gon+11; SGP12b; SGP12a; PV12; Gon+13]. Most of these implement either a layer-based approach or do not explicitly implement contexts and features. Nevertheless we think an explicit and clear separation between these two notions increases the maintainability and reusability of them when developing and designing context-oriented systems.

RQ4 *How can we facilitate the development of adaptive user interfaces into our approach?*

Developing context-oriented systems without considering the user interface aspect is limiting if one wants to create realistic context-oriented applications. While the programming language community has mainly worked on the context-oriented programming paradigm focussing on the core logic, the human-computer interaction (HCI) community has mainly worked on how adaptive systems change dynamically their user interfaces (the presentation and interaction widgets). To address both the core logic and the user interface aspect, how can we uniformly provide support to develop adaptive user interfaces into our approach? Can we consider a user interface as another kind of ‘behaviour’ adaptation to simplify the architecture? How can we build dynamically such user interfaces? Do we need to use a bottom-up or a top-down approach to design and compose at runtime the user interface?

RQ5 *How can we guide designers and developers when conceiving context-oriented systems?*

Designing and developing context-oriented systems is not a simple task. Designers and developers should think about the features of the system, to what contexts it should adapt, what contexts trigger what features and how the features adapt the system. As they must think of all these

things and how they interact, how can we guide them with an appropriate development methodology to design and develop such systems?

RQ6 *How can we help context-oriented developers in their implementation and debugging tasks?*

Developing software systems with a new programming paradigm is not straightforward. Context-oriented programming languages are designed to help developers create systems that are able to adapt their behaviour depending on some particular situations (*i.e.* contexts). Given that the developers need to provide the contexts, the features, a mapping describing what contexts trigger what features, it becomes quite complex for them to keep a global overview of the system with the exponential number of combinations it may have. They must also think about which features adapt which application classes of the system. It is not so easy to visualise all these concepts and their interactions during their development tasks. Since context-oriented systems have a highly dynamic nature, visualising how such systems evolve at runtime remains another daunting task for the programmers. Therefore how can we support programmers while implementing and debugging?

Beyond these visualisation issues, other research can be explored to help developers in their development and debugging tasks. For example, can we build a dedicated testing methodology for context-oriented systems that goes beyond traditional testing approaches?

1.3 Contributions

To design context-oriented systems in a more maintainable and reusable way, we propose a feature-based context-oriented software development approach that reconciles context-oriented programming [HCN08; SGP12a; Car+14], feature modelling [Kan+90] and dynamic software product lines [HT08; Ach+09; COH14; Mur+14; CHD15; Men+17]. This solution leads us to propose the feature-based context-oriented programming paradigm. We define its underlying principles, its underlying architecture, an implementation of this programming framework on top of the Ruby programming language, a library to ease the development of adaptive user interfaces, a development methodology and two visualisation tools to help designers and developers in their tasks. In this section we list our main contributions that address the research questions raised in Section 1.2. For each contribution we precise which research question it tackles.

C1 *Explicit separation of contexts and features*

Contexts and features (also known as behavioural adaptations) are key notions in feature-based context-oriented programming. While contexts are characteristics of the surrounding environment in which a system runs [Abo+99], features can be defined as “*any prominent or distinctive user-visible aspect, quality, or characteristic of a software system*” [Kan+90]. Contexts and features are complementary notions that go hand in hand when building context-oriented systems that can adapt their behaviour (described in terms of features) dynamically whenever changes (reified as contexts) are detected in the surrounding environment. Notwithstanding their complementarity and differences, it has been observed that the feature modelling notation that serves to model features [Kan+90] can also be used to model contexts [Des+07]. Taking this observation and Hartmann and Trew’s *multiple-product-line-feature model* [HT08] as inspiration, we propose to design both contexts and features with feature modelling to model a context model and a feature model, respectively. To model how a set of contexts can trigger a set of features, we also define a mapping between contexts and features [HT08; COH14]. This contribution addresses research question **RQ1**.

C2 *Creation of a context-oriented architecture*

We propose an architecture [MCD16] for context-oriented software systems that reconciles the practicality of encoding variations at the code level in a context-oriented programming language, with the clarity provided by a high-level architecture for adaptive systems. The purpose of our architecture is to cover the different phases required to enable context-orientation in a software system, while making explicit which parts are provided as part of the adaptation framework, and which are to be coded by the application programmer. We concentrate our effort in particular on the following phases: *context activation*, *feature selection*, *feature activation* and the *deployment* of the features in the system. This contribution tackles research question **RQ2**.

C3 *Implementation of a programming language*

Based on our feature-based context-oriented architecture, we implement a FBCOP programming language [DMD19], which clearly separates the notion of contexts from the notion of features. As a case study for our architecture and to demonstrate its feasibility we develop it in the Ruby programming language to create a Ruby application framework. This contribution answers research question **RQ3**.

C4 *Integration of user interface adaptation in our programming language*

We also study how we can dynamically compose and adapt the user interfaces without making our architecture complex. Much work tackles user interface adaptation using model-driven engineering [Cal+03a; López+08] or other techniques [DCC08; Yig+19], but integrating these techniques seems to make our architecture more complex than needed. To avoid this we investigate another research path by suggesting an abstraction of a user interface library in our application framework that the programmer can use to create its user interfaces and let the system manage the user interface composition in the same way as how it dynamically composes the behaviour of core logic features. Such a proposal allows considering user interfaces widgets and adaptations as simple fine-grained features (like the core logic fine-grained features). However this requires developers to be aware of how the composition works and the way to define these adaptations becomes their responsibility. This contribution addresses research question **RQ4**.

In this dissertation, when we will use the terms “user interface adaptation library” or “adaptive user interfaces”, this thus means that we can build user interfaces that are adaptive since the user interface library is integrated into our FBCOP approach, which itself is adaptive.

C5 *Design of a development methodology*

Designing and developing a system is more straightforward with a dedicated methodology. In addition to proposing a new programming paradigm we therefore also propose a dedicated development methodology to help designers and developers better understand how we can use our feature-based context-oriented approach to create context-oriented systems. Without such a development methodology it could be more tedious for them to create such systems due to the complexity of designing and developing such systems. This contribution tackles research question **RQ5**.

C6 *Creation of visualisation tools*

It comes as no surprise that designing and implementing such context-oriented systems remains quite complex due to their highly dynamic nature and the exponential number of combinations we can have between the contexts and features they adapt [Mur+14]. To help developers achieve this complex task we will create two separate visualisation tools. The **CONTEXT AND FEATURE MODEL VISUALISER** [Duh+19b] provides a global overview of the system by exposing the system’s context and feature model as well as the mapping between them, but also which

features adapt which classes of the system. This visualisation enables developers to explore the active and inactive contexts and features of the system by using filters to customise what information (contexts, features, active, inactive) they like to see or hide in the visualisation. The **FEATURE VISUALISER** [DMD18], on the other hand, is conceived as an inspection tool to observe the dynamic behaviour of a context-oriented system. It displays dynamically which contexts trigger which features, how the features adapt the system (i.e., what classes of the system) and in which order the features are activated. This contribution answers research question **RQ6**.

C7 *Validation of our feature-based context-oriented approach with real context-oriented designers and developers*

Designing a new approach proposing a solution to the designers and developers of context-oriented systems must be validated by them to assess its usability and usefulness. For that we conducted four user studies with master-level students in computer science and engineering enrolled in software engineering courses. Depending on the validation they had to play the role of a context-oriented designer or a context-oriented programmer.

The complete source code of this thesis is available on this accompanying repository <https://bitbucket.org/benoitduhoux/rubycop/>.

1.4 Supporting publications

This dissertation is supported by the following primary publications.

[MCD16] In this paper “*A Context-Oriented Software Architecture*”, we propose a layered software architecture that reconciles the sensory input, context discovery and activation, and the selection, activation, and execution of feature variants in a single implementation framework, which can be customised by application programmers into actual context-aware applications. This paper addresses the research questions **RQ1** and **RQ2** with the contributions **C1** and **C2** since this work allows to better define and specify a clear separation of contexts and features and proposes a complete architecture needed to design context-oriented applications.

[DMD18] This paper “*Feature Visualiser: an Inspection Tool for Context-Oriented Programmers*” suggests a visualisation tool to help programmers to inspect in detail what contexts trigger what features, and how the feature parts adapt what application classes of the application. This

paper also motivates a novel case study: a risk information system. This paper tackles the research question **RQ6** with the contribution **C6**. We also conducted a preliminary user study to assess its usefulness and understandability with real programmers (contribution **C7**).

[DMD19] This paper “*Implementation of a Feature-Based Context-Oriented Programming Language*” shows the object-oriented architecture, design and implementation issues of such a feature-based context-oriented programming language, which we implemented on top of the *Ruby* programming language as an application framework for context-oriented programmers. We illustrate our language design with a small example of a feature-based context-oriented program written in this language. This paper answers the research questions **RQ1** and **RQ3** with the contributions **C1** and **C3**.

[Duh+19b] This paper “*Dynamic Visualisation of Features and Contexts for Context-Oriented Programmers*” presents another visualisation tool that is intricately related to the underlying architecture of a feature-based context-oriented programming language, and the context and feature models it uses. The visualisation confronts two hierarchical models (a context model and a feature model) and highlights the dependencies between them. This paper addresses the research questions **RQ6** with a contribution in **C6**. An initial user study of the visualisation tool is performed to assess its usefulness and usability (contribution **C7**).

[Duh+19a] This paper “*A Context and Feature Visualisation Tool for a Feature-Based Context-Oriented Programming Language*” is a longer version of our previous paper [Duh+19b] in which we explain in more detail our approach. This paper also tackles the research questions **RQ1** and **RQ2** with the contributions **C1** and **C2**.

In addition to these primary publications, we have also secondary publications that help us to better define some aspects in this work:

[MDC17] This paper entitled “*Managing the Context Interaction Problem: A Classification and Design Space of Conflict Resolution Techniques in Dynamically Adaptive Software Systems*” surveys a number of conflict resolution strategies, and proposes a design space in which to classify, compare, and explain the differences between them. Moreover this paper also addresses another case study: an home automation system. This work allows us to better specify contexts and features of context-oriented applications when designing such applications. For that it helps us to better define the contributions **C1** and **C2** with respect to the research questions **RQ1** and **RQ2**.

[CDD19] This paper “*Supporting Citizen Participation with Adaptive Public Displays: A Process Model Proposal*” suggests a process model destined to serve as a guide for designers of adaptive public displays. We also explore another running example: a voting system. This paper allows us to better identify the needs for designers when designing dynamic context-aware systems. As such it contributes to the research question **RQ5** and contribution **C5**.

Our FBCOP approach also serves as foundation and inspiration for other research paths, such as:

[Mar+21] This paper entitled “*Test Scenario Generation for Context-Oriented Programs*” proposes a methodology to automate the generation of test scenarios for developers of feature-based context-oriented programs. While this work addresses partially research question **RQ5** by extending the contribution **C5**, it also partly tackles research question **RQ6** by adding a new tool to help testers of such systems.

[Mar+22] This paper “*Generating Virtual Scenarios for Cyber Ranges from Feature-Based Context-Oriented Models: A Case Study*” relies on feature-based context-oriented modelling to generate relevant cyber range scenarios from an explicit user profile and exploits described in attack-defence trees. This paper demonstrates our modelling approach can be reused in another field to design variabilities in cyber ranges depending on the user profiles (*e.g.*, budget or users’ skills) and attacks or defences they want to train.

Finally we also supervised many master theses that tackled some parts or potential extensions of this dissertation. Duhoux [Duh16] implemented a first prototype of a context-oriented software architecture including adaptive user interfaces. In this architecture, contexts and features were already separated. To simulate such applications, we also developed a simulation tool to run different scenarios of such applications. Kühn [Küh17] reconciled context-oriented programming and feature modelling and proposed a first prototype of a conflict resolution strategy to solve some context-interaction problems. Leung [Leu19] implemented the **CONTEXT AND FEATURE MODEL VISUALISER** tool to get an overview of the context-feature model of FBCOP applications. Van den Bogaert [Van20] explored the usage of a SAT solver to verify the consistency of the (context and feature) models at runtime and detect model anomalies at design time. Martin [Mar21a] suggested a novel prototype of FBCOP to add flexibility and expressiveness in the mapping model in large and complex systems, a transaction system to activate and deactivate contexts to the installation and removal of features in the system behaviour, and

an adaptation mechanism based on pointers. Martou [Mar21b] studied how designers and programmers could test their context-feature model after designing them by generating tests scenarios inspired by a combinatorial interaction testing. Delhove and Tsang [DT22] explored how multimodality can be integrated in FBCOP applications. Finally, Iglesias Garcia [Igl22] extended the FBCOP architecture to include sensor detection and simulation and Mouligneaux [Mou22] integrated data into contexts to avoid to discretise the different contexts and studied how these variables can be used in the features.

1.5 Running example: a smart messaging system

Throughout this dissertation, we will use a case study that will serve as running example to exemplify the different notions in FBCOP. This running example will also be used to illustrate all the code snippets we will provide. This case study is a smart messaging system that allows users to communicate between them.

The messaging system's main functionality consists of sending or receiving messages to and from other users through chat.

Messages can be of different types. By default, we assume the messages are textual. However, messages can also be richer to include a picture, video, emoticon, position and more, or even a combination of these different types. Richer messages can be exchanged depending on the status of the Internet connection. In fact, when a Wi-Fi connection is sensed, users can send and receive any kind of messages. But when a cellular connection (e.g., 3G, 4G or 5G), only textual messages or emoticons can be sent and received. The (user) positions can be sent whatever the connection but only if the GPS is activated. Furthermore, when users drive, users should have only the opportunity to send predefined messages to minimise their interactions with their car dashboard. For that, a *Bluetooth* connection must be established between the car dashboard and the smartphone.

The Internet connection type also affects the main features of sending and receiving messages. In case users have no Internet connection, they cannot send and receive messages. Nevertheless users can always write messages that will be sent as soon as a new Internet connection is sensed.

Whenever users receive a new message, they are notified via a sound alarm or their device's vibration mode. What notification mode is used depends on the user availability and the ambient noise level. However the notification system can be muted when the ambient noise level is quiet, so as not to disturb the user, or when the user is occupied.

The device type (smartphone, car dashboard or desktop) adapts the layout of the information being displayed. For that, the master/detail pattern is a

well-known pattern to address screen size issues. When the screen is too small (like smartphones), the view displays either the list of user chats (*i.e.* the master part) or the selected users chat (*i.e.* the detailed view). But when screen sizes are larger, information are displayed side-by-side, *i.e.* the chats list can appear on the left side of the layout and the selected chat can be detailed in the right part of the layout.

Depending on the device, more or less information can also be explicitly shown [TC99]. For example, when users open a chat on their smartphone, users can only see the primary information such as the author and the content of each message. But when users have larger screens, more information can be displayed directly such as the sending date of the message. In this example, we consider the sending date as a secondary information. This means that such an information is still present but not directly visible on the device's screen if it is too small. In that case, users will have to interact with the system to see such an information.

The application can also depend on the user age and disabilities. For users having sight problems, texts are enlarged. Otherwise the texts are of normal size. Children cannot upload a profile picture to preserve their identity. In addition, filters are applied to censor inappropriate language in the children's chats.

1.6 Roadmap

In this section, we will outline the structure of this dissertation which is divided in five parts: PROLOGUE, APPROACH AND USAGE, IMPLEMENTATION, VALIDATION and EPILOGUE.

Prologue

This current chapter (Chapter 1) first set the scope of this thesis with a small introduction. We then presented the research questions we will tackle in this dissertation and described briefly its contributions that will answer to these questions. We also listed the various supporting publications that we published and that will be presented during this work. We also described the running example we will use throughout this dissertation: a smart messaging system.

Next, we will introduce the state of the art (Chapter 2) related to our work. We will present related research to design and implement dynamically adaptive software systems in the research fields of *modelling*, *implementation*, *user interface adaptation* and *supporting tools*. For each research field, we will position our proposal to its literature.

Approach and usage

After positioning ourselves, we will introduce the FBCOP programming paradigm (Chapter 3) in which we will define the underlying principles and concepts, the system architecture and its control flow and the supporting development methodology.

Then we will introduce the FBCOP programming framework (Chapter 4) that will illustrate how application programmers can implement context-oriented applications with our approach.

Next we will describe and exemplify the user interface adaptation library (Chapter 5) allowing programmers to create adaptive user interfaces in our approach.

Finally we will present two FBCOP visualisation tools (Chapter 6). In this chapter, we will describe the visualisation and functionalities of the `CONTEXT AND FEATURE MODEL VISUALISER` and the `FEATURE VISUALISER`.

Implementation

After introducing and exemplifying the FBCOP approach, we will explain how the FBCOP programming framework was implemented. We will first describe how the contexts and features are declared and defined, as well as the mapping (Chapter 7). We continue with how we integrated the modelling in our programming framework (Chapter 8). Then we will explain how the architecture is implemented (Chapter 9), *i.e.* its control flow, how the dynamic adaptation alters the system at runtime and how the *proceed* mechanism, a key notion in context-oriented programming [HCN08], is implemented and executed. Finally we will describe how we developed the tooling support (Chapter 10) to easily create new tools to help application programmers.

Validation

In this part, we will demonstrate first that the FBCOP approach is sufficiently expressive to create context-oriented applications (Chapter 11). We will show that with the help of five case studies: two variants of the *smart messaging system*, a *smart risk information system*, a *smart meetings system* and a *smart city guide*. These case studies are either designed and or implemented by us or by others. For each case study, we will illustrate its context-feature model, its implementation and/or its execution according to what was done by the authors.

Then we will assess FBCOP's design (Chapter 12). We will discuss first what design qualities (maintainability, extensibility, adaptability, readability and scalability) we have or not as a consequence of our design choices in

FBCOP. We will then evaluate FBCOP's usability based on the cognitive dimensions of notations framework.

Developing a complete approach that is not understandable, irrelevant or unsuitable implies that this approach will be never used. Thus we must also validate its usefulness and usability with real participants (Chapter 13). For that we conducted four user studies in which we asked our participants to play the role of designers and/or programmers depending on the study. The two first user studies were preliminaries studies, each concerns one of the visualisation tools which we developed. The two last user studies were more complex user studies since we assessed the usefulness and usability of the full FBCOP approach (*i.e.*, including modelling, programming framework, supporting development methodology and visualisation tools).

Epilogue

In this part, we will conclude this dissertation. We will first discuss the potential improvements we can still make to complete our FBCOP approach (Chapter 14) and finally conclude this dissertation (Chapter 15).

CHAPTER

2

STATE OF THE ART

Self-adaptive software systems are able to modify themselves automatically in response to events occurring during their execution [ST09]. Such events can be internal (*e.g.* failure) or external (*e.g.* user interaction) events of the software system. Such systems are often conceived in terms of an adaptation loop, called *MAPE-K*, which decomposes the adaptation process into *Monitor*, *Analyse*, *Plan* and *Execute* processes and which make use of a shared Knowledge-base [ST09]. From the sensory input, the *monitoring* process aims to gather sensed information and reify it in such a way that the *analysis* process is able to interpret it and detect if a change to the system is required. Once a change is needed, the *planning* process must decide what the changes are and how they will be applied by the *execution* process. Finally, these changes are executed by the system depending on what it is going on in its environment.

Context-aware systems can be regarded as a specific class of self-adaptive systems. Krupitzer et al. [Kru+15] confirmed this statement by showing how such self-adaptive systems can adapt themselves depending on their surrounding environment. Cardozo and Mens [CM22] also see context-oriented systems as a special kind of self-adaptive systems but relying on a different underlying implementation technique.

Now that we know that context-oriented applications are self-adaptive systems that adapt their behaviour depending on the surrounding environment, we will explore the literature on how we can conceive such context-

oriented applications. For that we will first review different modelling technologies to see how contexts and features can be modeled at design time (Section 2.1). Next we will look at how such systems can be implemented (Section 2.2). We will also discuss many approaches on how adaptive user interfaces can be implemented (Section 2.3). We will also look at the literature for supporting tools that help designers and programmers when conceiving context-oriented applications (Section 2.4). Finally we will conclude this chapter with a discussion to make more explicit the connections between our contributions and the state of the art (Section 2.5).

2.1 Modelling

Modelling software systems able to adapt their behaviour at specific times (*i.e.* at design-, compile-, configuration-, deployment-, or at run-time) has been largely explored. As our work is about context-aware systems and more specifically context-oriented systems, we focus in particular on approaches that emphasise the modelling of both contexts and features to design such systems. After quickly introducing our methodology to find papers that interest us in the research literature, we compare the different approaches found based on a set of questions that distinguish the key concepts underlying these approaches. Finally, we will put in perspective our own approach to this related work.

In addition to approaches that emphasise the notions of context and feature, other approaches might also be suitable to model highly dynamic applications that are able to dynamically adapt their behaviour depending on contexts, such as for example role-based modelling. In role-based modelling, an instance's behaviour is adapted by roles according to the current context in which that instance runs. In other words, this means that the behaviour of the instance can be different according to the role it plays for the particular situation in which the instance is. Kühn et al. [Küh+14] investigate different role-based modelling solutions to design such systems. Even if such a modelling is also appropriate to build context-aware systems, we will focus our research on the notion of context and feature since our lab has a valuable expertise in these concepts.

Methodology We applied the following methodology to find articles of interest. We looked for papers that contain at least the keywords “context-oriented” or “context-aware” and the keyword “dynamic software product line” in their title or abstract and that were published in the proceedings of *SPLC* (Software Product Line Conference), *VaMoS* (Variability Modeling of Software-intensive Systems) or *SEAMS* (Software Engineering for Adaptive and Self-Managing Systems). From these papers, we conducted some snow-

balling (discovering interesting missing papers from the references of the collected papers) to get a larger vision of the existing work in this field. Some additional references were provided by our network that envision this field in their manner to complete our literature study.

We analysed all these papers through several questions:

- “what formalism do they use to model contexts?”,
- “what formalism do they use to model features?”,
- “are contexts and features explicitly separated in the modelling?”,
- “how is the mapping from contexts to features formalised?”,
- “does the approach enable dynamic reconfiguration and/or restructuring of the models at runtime?”, and
- “do they discuss about a consistency checking mechanism when a re-configuration occurs?”.

The results of this analysis are discussed next and summarised in Table 2.1.

Modelling approaches A software product line describes a family of software systems that share a common set of features satisfying the specific needs of a particular domain. Products (software systems) of that family are produced by selecting and composing a set of features (configuration) corresponding to the needs of a particular client or variant of the system. Often feature modelling [Kan+90] is used to describe the possible features, together with their relations and constraints, of such a family. The features of a family describe the commonalities (features always present) and variabilities (features that may be present) of the different products/systems belonging to the family. Dynamic software product lines (DSPL) try to address the need of dynamic software adaptation to provide the flexibility required by more dynamic environments and users [Cap+14]. Their goal is that the product/software adaptations can be selected even at runtime [BLH10]. To model these systems feature modelling is often used. For self-adaptive systems to become truly context-oriented, an alternative to extending feature models with additional attributes that may contain contextual information, is to model explicitly the contexts, based on which the features need to be adapted dynamically. Some research on software product lines has proposed to use feature modelling not only to model the features of the systems in a product line, but also to model the possible contexts, together with their relations and constraints, on which (the selection of) these features depend [Ach+09; SLR13; COH14; Mur+14; CHD15; Men+17; Sou+17]. All these approaches are related directly or indirectly to the work of Hartmann and Trew [HT08] who

propose a *multiple-product-line-feature model* composed of two separate sub-models: a context variability model (representing the contexts and their intra-dependencies) and a traditional feature model. This allows them to model not only what the common and variable features are, but also how contexts affect what features should (not) become part of a product, by declaring explicit dependencies from the context model to the feature model. A main difference between Hartmann and Trew’s work and other approaches [Ach+09; SLR13; COH14; Mur+14; CHD15; Men+17; Sou+17] is that Hartmann and Trew’s proposal is dedicated to the configuration time and not for the run-time. Therefore Hartmann and Trew’s modelling allows to generate a product but this product cannot reconfigure its behaviour dynamically when the surrounding environment changes. Furthermore even if the *mapping* based on the “excludes” and “requires” dependencies used by Hartmann and Trew are mainly reused in many other approaches [SLR13; COH14; Mur+14; CHD15; Men+17; Sou+17], Acher et al. [Ach+09] suggested to use propositional logic to be more expressive in the *mapping* with event-condition-actions rules between the context and feature model.

In addition studying the interest of separating contexts and features in two different models, Capilla et al. [COH14] also proposed another strategy which aimed to model the context and non-context features as parts of a single unified feature model. In this alternative strategy, they use labelling to distinguish the context features in the feature model from the more traditional features. By comparing both approaches, they conclude that while the strategy with a clear separation of (context and feature) models increases the number of dependencies between the two models, is “*more reusable when several contexts are involved*”, while the strategy using a single unified feature model “*simplifies the model and reduces the number of dependencies*” between the features [COH14].

All these approaches discussed above use feature modelling to formalise both contexts and features. Other approaches rely on other techniques to model contexts while maintaining feature modelling for the features [JLS10; Mau+18]. For example, Jaroucheh et al. [JLS10] suggest to model contexts with ontologies because such modelling offers a high expressiveness to describe the contexts and propose interesting reasoning techniques on contexts [BDR07; Bet+10]. In brief, ontologies describe the contexts in a bottom-up fashion with context primitives composed of context entities, attributes, associations and rules [JLS10]. They also use the notion of “stereotypes” on the ontology elements to express the mapping between the contexts primitives and the context feature model [JLS10].

Mauro et al [Mau+18] propose to capture contextual information directly within the feature model. They claim that such an approach is “*less flexible, modular and expressive*” but addresses the maintainability issue of large soft-

ware product lines [Mau+16]. They also add the notion of validity formulas (propositional formulas defining some conditions) on each feature to ensure the feature is selectable in some cases [Mau+16].

In addition to approaches that rely fully or partially on feature modelling to model context-aware systems, other approaches exist that use other formalisms [Hal+06; Des+07; Ben+08a; Ben+08b; Mor+08; FWM08; Car+15]. Hallsteinsen et al. [Hal+06] suggest to model the contexts and features similarly with entity types since they consider *“a software system and its context as a system of interacting entities”*. To link the contexts and features, they use the concept of property annotations associated with ports to *“reason on how well a variant matches its context”*. Desmet et al. [Des+07] suggest to model context-aware systems through context-oriented domain analysis (CODA). CODA is syntactically inspired by feature-oriented domain analysis [Kan+90] but with a different semantics. Their approach models the system as an hierarchical structure where the features compose the top of the model and are refined by context-dependent adaptations on the bottom of the model. The context-dependent adaptations are dynamically selected by the system if the condition on the incoming edge is respected (*i.e.*, if the contexts are respected for this specific adaptation). Bencomo et al. [Ben+08a; Ben+08b] propose an approach that relies on orthogonal variability modelling for the features and state modelling for the contexts. The orthogonal variability model formalism allows to model only the variability points and variants of a system [PBV05]. In their approach, the system can evolve from one configuration to another with transitions, where the transitions and the policies associated (in the form of *“event-do-actions”*) represent the context variability and in which case the system must reconfigure itself. Morin et al. [Mor+08] explore the modelling of features with aspect-oriented modelling, where each aspect model is *“composed of a graft model (*i.e.*, what to weave), an interface model (*i.e.*, where to weave) and a composition protocol describing how the graft model must be weaved into the interface model”*. To simplify the formalism of the context model, they use a minimal representation of the different situations that occur in the environment. Finally to map what contexts must trigger what features, they use adaptation rules that link directly the values of the sensors to the dedicated variants. Fernandes et al. [FWM08; FWT11] propose to design separately the feature model and the context feature model to avoid a loss of understandability and readability. Both are modelled with an extended feature modelling formalism: UbiFEX [FWM08]. As they are split, they propose to use context rules to relate the contexts and features. Finally, Cardozo et al. [Car13; Car+15] suggested to formalise the contexts and their activation semantics with petri nets. In their solution, the behavioural adaptations (features) are directly attached to the contexts. Therefore they did not propose a modelling for the features.

Dynamic reconfiguration and consistency checking Except for the proposal of Hartmann and Trew [HT08], all approaches in Table 2.1 allow dynamic reconfiguration. This means that the system is able to adapt its behaviour dynamically when contexts change. However such operations can be unsafe and could lead to errors and inconsistencies if the system reaches an invalid configuration, if we do not check the new proposed configuration before deploying it. Even if many of the papers referenced did not discuss explicitly about what mechanisms they implemented, they mainly claim that they check the novel configuration before they deploy it. In case of invalid configurations, they do not reconfigure the system. For example, Fernandes et al [FWT11] use their own verification mechanism (UbiFEX-Simulation) to ensure all constraints are respected. Other approaches can also be used such as model checking [Sou+17], model checking and usage of a specific solver at runtime [SLR13], automated decisions [Mur+14], Petri nets [Car13; Car+15], constraint-programming solvers [Mau+16] or SMT (Satisfiability Modulo Theories) solvers [Mau+18].

Dynamic restructuring Going further, we can also adapt the structure of the models by adding, replacing or removing new contexts or features due to a new version of the system that was not designed previously. In the approaches we analysed, some of them allow such a dynamic restructuring [Hal+06; Ben+08a; SLR13; COH14; Mur+14; CHD15; Men+17; Mau+18]. However this evolution is a complex task [BQ15].

Positioning Our own approach is strongly inspired by the work of Hartmann and Trew [HT08]. Therefore we model the contexts and features with feature modelling in terms of a separate context and feature model. We agree with Capilla et al. [COH14] that a clear separation of the contexts and features leads to a more maintainable and reusable approach. However we propose a simpler context-feature mapping than some other approaches, such as for example Acher et al. [Ach+09], to reduce the complexity of our approach, even if this comes at the cost of losing a bit in expressiveness. In our mapping, we can only express mapping relations where each relation is a list of contexts triggering a list of features. With such an approach, we are able to design context-aware systems that reconfigure or adapt dynamically their behaviour in response to contextual changes in their surrounding environment. We also ensure consistency checking based on an algorithmic approach before adapting the behaviour by verifying that all the constraints of the models are satisfied. However our approach does not support dynamic restructuring of features or contexts that were not designed up front (though this could be explored as part of future work).

Table 2.1: Summary of different modelling approaches to define contexts and features in dynamically adaptive context-oriented systems. An hyphen “-” in a cell means that the question is not discussed for the paper.

	Context modelling formalism	Feature modelling formalism	Are contexts and features separated?	Mapping model formalism	Dynamic updates	Consistency checking
[Hal+06]	Entity type	Entity type	-	Property annotations	Reconfiguration and restructuring	-
[Des+07]	CODA	CODA	No	Ad-hoc	Reconfiguration	-
[Ben+08a] [Ben+08b]	State modelling	Orthogonal variability modelling	No	Event-do-actions	Reconfiguration and restructuring	-
[HT08]	Feature modelling	Feature modelling	Yes	“Requires” and “excludes” dependencies	-	-
[Mor+08]	-	Aspect-oriented modelling	No	Adaptation rules	Reconfiguration	-
[Ach+09]	Feature modelling	Feature modelling	Yes	Propositional logic	Reconfiguration	-
[JLS10]	Ontological modelling	Feature modelling	Yes	“Stereotypes”	Reconfiguration	-
[FWM08] [FWT11]	UbiFEX	UbiFEX	Yes	Context rules	Reconfiguration	Ad-hoc
[SLR13]	Feature modelling	Feature modelling	Yes	“Requires” and “excludes” dependencies	Reconfiguration and restructuring	Model checking and solver

continuation on the next page

Table 2.1: Summary of different modelling approaches to define contexts and features in dynamically adaptive context-oriented systems. An hyphen “-” in a cell means that the question is not discussed for the paper.

	Context modelling formalism	Feature modelling formalism	Are contexts and features separated?	Mapping model formalism	Dynamic updates	Consistency checking
[COH14] [CHD15] [Men+17]	Feature modelling	Feature modelling	Both	“Requires” and “excludes” dependencies + Labelling	Reconfiguration and restructuring	-
[Mur+14]	Feature modelling	Feature modelling	Both	“Requires” and “excludes” dependencies + Labelling	Reconfiguration and restructuring	Automated decision
[Car13] [Car+15]	Petri nets	-	No	-	Reconfiguration	Petri nets
[Mau+16]	-	Feature modelling	No	Validity formulas	Reconfiguration and restructuring	CP solver
[Mau+18]	-	Feature modelling	No	Validity formulas	Reconfiguration and restructuring	SMT solver
[Sou+17]	Feature modelling	Feature modelling	Yes	“Requires” and “excludes” dependencies	Reconfiguration	Model checking

2.2 Implementation

Some of the modelling papers we discussed previously also introduce which kind(s) of implementation approaches they used to implement context-aware systems. We can find different approaches such as component-based approaches [Hal+06; FWM08], model-driven approaches [Ben+08b], context-oriented programming languages [Car13; Car+15; Men+17], context-aware reconfiguration engines [Mau+18] or even combinations of component-based and reflection approaches [Ben+08a], aspect-oriented programming with model-driven approaches [Mor+08] or a mix between model-driven and service component approaches (that mix service-oriented and component-based approaches) [PBD09]. However, none of the papers that model contexts and features with feature modelling discuss about which kind of implementation mechanisms they use, except Mens et al. [Men+17] who explain they use a dedicated context-oriented programming language.

From this quick overview and different surveys [BDR07; Car13; Kru+15], we can already observe that many implementation solutions exist to implement context-aware systems. Among these solutions, some of them are based on IFs-statements or design patterns such as the decorator, strategy, visitor, chain of responsibility, or state pattern. Nonetheless we did not consider these solutions to implement context-oriented applications because some implementation techniques are less maintainable such as the IFs-statements, and those based on design patterns create too much boilerplate code due to the additional structure introduced by these design patterns [CM22]. Other solutions based on programming languages also exist to implement such systems, such as subject-oriented programming [HO93], role-oriented programming [Her07; Küh+14] or context-oriented programming [HCN08]. For subject-oriented programming and role-oriented programming, the notion of context is replaced by a notion of subject or the role played by an object. However as they have been already surveyed and because our lab has a significant expertise with context-oriented programming [GMC08; Gon+11; PV12; Gon+13], we mainly focus our analysis here on context-oriented programming. In addition, Cardozo and Mens [CM22] claim that context-oriented programming is a “*viable option*” for developing context-oriented systems. They also explain that using such a language allows to preserve a clear separation between the application and adaptation logic.

The context-oriented programming paradigm was explicitly conceived to facilitate the implementation of context-aware systems [KR03; HCN08]. This programming paradigm provides dedicated programming language abstractions to adapt the behaviour of a software system dynamically upon changing contexts. In context-oriented programming, contexts and behavioural adaptations (modelled as features in this dissertation) are first-class language en-

titles. The behavioural adaptations get (de)activated in the code whenever their corresponding contexts become (de)activated. Nowadays many different implementations of COP language prototypes exist [KR03; CH05; GMH07; LDN07; AHR08; CD08; GMC08; HCH08; GPS10; Was+10; App+11; AKM11; Gon+11; HIM11; KAM11; Lin+11; SGP11; SGP12b; SGP12a; PV12; Gon+13; SMH17]. To implement the behavioural adaptations, many approaches are based on a layer-based approach where each layer corresponds to a particular context and contains the dedicated adaptations for this specific context. Yet other approaches exist, many of which were implemented in our lab [GMH07; GMC08; Gon+11; PV12; Gon+13] to manage the behavioural adaptations. However, none of these approaches separate explicitly the contexts and features from an implementation perspective. Preliminary work to emphasise the behavioural adaptations as actual features in a context-oriented programming approach has been proposed by Poncelet and Vigneron [PV12] and Car-dozo et al. [Car+14] since contexts and features have similarities in their dedicated programming approaches [Car+11]. But none of them have an explicit separation of contexts and features.

Positioning In this dissertation, we take this idea yet a step further and explicitly separate the features representing the behavioural adaptations from the contexts that trigger them, even at the level of the implementation language.

2.3 User interface adaptation

Adapting dynamically the features of an application aims to better fit them to the end-users' requirements depending on the status of the surrounding environment in which they run. For a long time, the human-computer interaction (HCI) community has been investigating how user interfaces can be adapted so that their usability is still preserved [TC99]. In the mindset of this community, the first focus is on how users and tasks can be modelled [TC99; CCT01a; Cal+03b; Pat04; Van+05; Dem+07; DCC08; MVA08; Mar+17].

Thevenin and Coutaz [TC99] define adaptation with two different notions: adaptivity and adaptability. While adaptivity is the fact that an adaptation is triggered by the system, adaptability is the fact that the human triggers the adaptation. Since our FBCOP approach encompasses both kinds of adaptation, we will use the keyword "adaptation" and the adjective "adaptive" throughout this dissertation. These words capture both notions of Thevenin and Coutaz [TC99].

Several various design spaces for user interface adaptation have been proposed [TC99; Van+05; MV13]. For example, Thevenin and Coutaz's design space relies on four axes that model the *actor*, the *target for adaptation*, the

means and the *time of the adaptation* [TC99]. The actor represents who initiates the adaptation (*i.e.* either the user or the system). The target for adaptation stands for from what the adaptation is needed (*i.e.* the user, environment or platform). The means of adaptation represents which aspect of the behaviour is adapted (*e.g.* the system task model, the rendering techniques or the help subsystems). And the time denotes when the adaptation is made (*i.e.* either statically or dynamically). Vanderdonckt et al. [Van+05] define a design space for context-sensitive user interfaces based on seven axes that respond each to a particular question: *with respect to what?*, *what?*, *for what?*, *who?*, *how many?*, *when?* and *with what?*. These questions describe the entities for which an adaptation must be done (*e.g.* the user, the device, the physical environment, and so on), the aspects that will be adapted (*e.g.* the presentation, the dialog, and more), the steps considered for the adaptation (*i.e.* the initiative, the proposal, the decision or the execution), the entity that triggers the adaptation (*i.e.* the user, the system or both), the number of needed reconfigurations to reach the expected adaptation, the time of the adaptation (*i.e.* at design and/or run time), the kind of models used to perform the adaptation (*i.e.* a passive, active or shared model), respectively. Other design spaces [MV13; Bou+17] or conceptual reference frameworks [Abr+21] have been also explored to define either the user interface adaptation or intelligent user interface adaptation.

Since there exist so many design spaces with many different criteria, we will only select some of them that are relevant for us to better relate our user interface adaptation library to this large field. We mainly took inspiration from Vanderdonckt et al.'s design space [Van+05], but we use the perspective of the work developed in this thesis to define a subset of their design space based on our expertise of conceiving context-oriented systems. The criteria of relevance answer to the following questions:

- what is the context of the adaptation (*i.e.* user, environment or physical characteristics)?,
- what is the aspect (*e.g.* presentation, navigation, content, and so on) being adapted?,
- how are the contexts and adaptations modelled?,
- when does the adaptation occur?, and
- which mechanisms are used to adapt the user interfaces?.

Context of adaptation Thevenin and Coutaz [TC99] define the concept of plasticity in user interfaces as “*the capacity of a user interface to withstand variations of both the system physical characteristics and the environment*

while preserving usability”. Following this definition, plastic user interfaces adapt their user interfaces to the environment and platform, but do not to the user as context. Other approaches also target the platform and environment [CLC05b].

A lot of approaches are only platform-based [EVP00; Con+03; MPS03; Bal+04; CVC08; MVA08; PSS09; KS15]. Nevertheless Kurz et al. [KPG04] propose an adaptation solution depending on the platform but do attempt to include the user.

Motti and Vanderdonckt [MV13] also surveyed many frameworks that treat user interface adaptation and classified each of them by denoting which contextual dimensions are used. Among these surveyed solutions, only this of Dey et al. [DAS01] considers all kind of contexts (*i.e.* user, platform and environment). In addition to their survey, they also propose a conceptual framework for context-aware adaptation (*Triplet*) that treats all types of contexts. Other approaches also adapt the user interfaces to the user context, platform and environment [Cal+03b; Lim+05; DCC08; Lóp+08; Blo+11; Yig+20].

Aspects of adaptation Adaptation can be performed at many levels:

- presentation-level [EVP00; CLC05a; CLC05b; Cle+07; CVC08; MVA08; PVM09; Blo+11; Fis12; KS15],
- layout-level [Yig+19; Yig+20],
- style-level [Yig+19],
- navigation-level [Fis12; Yig+19; Yig+20],
- content-level [Mal+10; MVA08; Fis12; Yig+19],
- data-level [Cle+07],
- modality-level [Yig+20],
- interation-level [Blo+11],
- dialog-level [EVP00; CLC05a; CLC05b; Cle+07],
- widget-level [Cal+05]
- task-level [CLC05a; Cle+07; Yig+20] and
- service-level [Cle+07; Yig+20].

We can observe that most work focusses on the presentation-level, that most other levels are less discussed in literature.

Modelling of contexts and adaptations Many approaches model contexts with ontologies [Cal+03b; Cle+07; Sot+07; PVM09]. But other techniques for context modelling are also used in the HCI community such as profiles (*CC/PP*: composite capabilities/preferences profiles) [KPG04], properties [Blo+11], models [Lim+05], and ad-hoc solutions [Giu+19; Yig+20].

For the adaptations, this community often defines them through rules that are triggered when the adaptation needs to occur [CCT01a; CLC05b; Lim+05; Sot+07; Lóp+08; MV13; Yig+19]. Only a few approaches rely more on software engineering mechanisms to model the adaptation, such as through aspect-oriented modelling [Blo+11] or feature modelling [Mar+17].

Time of adaptation In the HCI community, the time at which the adaptation is done (*e.g.* design or run-time) is also important [Cal+03b; DCC08]. While the adaptive user interfaces and how these user interfaces evolve depending on the contexts are defined and foreseen at design time, the user interfaces are really adapted at runtime [Dem+07; Lóp+08; Yig+19].

Implementation mechanisms for adaptation A lot of approaches rely on the *CAMELEON Reference Framework* to adapt their interfaces dynamically depending on the contexts [CCT01a; Cal+03b; Lim+05; Van+05; MVA08; Mar+17]. This *CAMELEON Reference Framework* [Cal+03a] is a top-down approach and is built on a four-layered approach. First, the designers define the tasks and concepts of the system in order to design the different user interactions. These user tasks are then reified into abstract user interfaces, which are still independent of the concrete interaction modalities. After this transformation, these abstract user interfaces are reified in turn as concrete user interfaces to be interactor-dependent. Finally the concrete user interfaces are reified a last time into final user interfaces that are platform-specific. Many approaches following this framework use model-driven engineering as implementation mechanism [CCT01a; Cal+03b; Lim+05; MV13; Mar+17; Yig+20]. From their initial models (*i.e.* the user tasks), the different approaches transform and reify them with rules into transient models to finally generate the final code that is executable. In addition to relying on model-driven engineering, Martinez et al. [Mar+17] combine it with interactive genetic algorithms.

In addition to using model-driven engineering, Demeure et al. [Dem+07; DCC08] propose an architecture that combines model-driven engineering and interactors toolkits, and Sottet et al. [Sot+07] combine model-driven engineering with a service-oriented approach. Other approaches are also based on model-driven engineering [PS02; Con+03; CLC05a; PSS09]

However other implementation mechanisms also exist and are built on machine learning [Bou+17; Joh+19], adaptive layouts [Bou+17], adaptation

engines [Lóp+08], component-based approaches [Gab+11; KS15; Yig+19], generators [Luy+08; MVA08], architectural solutions [KPG04; Cle+07; DCC08; Blo+11], frameworks [Con+03; CLC05b], language solutions [CVC08], combinations of them [PVM09], or ad-hoc solutions [VLC03; Giu+19]. More approaches were also surveyed by Motti and Vanderdonckt [MV13].

Positioning In our approach, we propose a prototype of a user interface library integrated into FBCOP to develop adaptive user interfaces.

While all types of contexts are treated (with our FBCOP approach), in the case studies we have considered so far the adaptations are mainly dedicated to the presentation. But nothing prevents us from having other kinds of adaptations, such as those focusing on modalities and interactions for example.

In our approach, even if the user interface adaptation must be defined at design time, user interface adaptation is executed at runtime as with the previous approaches. Therefore we also have adaptive user interfaces. However, since we ask programmers to implement the user interface parts in the different features, our solution does not follow the “*focus on specification instead of coding*” statement [MPV11] that we can find with model-driven engineering solutions. In our approach, we opt for a different specification approach which translates closer to the coding. In addition, with our solution, we transfer the responsibility to the programmers. This increases the expressiveness to create user interface adaptations since we are not limited to the models and adaptation rules.

Furthermore, with our approach, we advocate that programmers can also ensure a good separation of concerns (between the user interfaces and core logic) [Con+03; God08] since the adaptations of the user interface are separated into the different feature parts.

2.4 Supporting tools

In our FBCOP software development approach, we propose a modelling and programming framework to support FBCOP designers and programmers when they conceive FBCOP applications. However others tools can also help them in their different tasks, such as a development methodology, visualisation and debugging tools or testing mechanisms.

Supporting tools are important to better understand the approach and demystify its complexity. As stated by Hermans [Her21], programmers must understand where to start when they must implement software systems. For that, they must understand the different interactions that exist between the different notions and components of the approach. By helping them with supporting tools, programmers can rely on different tools to understand and check whether what they have done is correct.

As we propose a development methodology, two visualisation tools, and a context simulator, we will briefly discuss for each of them the alternatives we found in the literature that could be interesting to relate to our approach or be inspired by it to create other visualisation tools.

Development methodology Rosenberger et al. [RGR18] introduce a process model that helps designers to analyse the contextual requirements and to define contextual functionalities. Three steps compose the process model: activation determination, process definition and context elicitation. In the initial step, designers must first determine the activities that must be contextualised in their application. They must then define how their application should act when context changes are sensed. Finally they have to determine the contexts of their applications.

Henricksen and Indulska [HI06] describe a generic software engineering process that is composed of the following steps: analysis, design, implementation/programming, infrastructure customisation, and testing. This process model seems traditional when conceiving applications but they extend some steps to include aspects coming from the creation of context-aware applications. One interesting phase the authors develop is the analysis. In this phase, designers must first define the features of their application. This initial step is common when designing applications. But here, the authors extend the analysis phase to include the contexts and the mapping on which context-aware applications rely on to adapt their behaviour depending on contexts. After defining the features, designers have to determine thus the contexts of their application. Finally designers must refine features to relate them to their corresponding contexts to define the mapping and continue to iterate till the requirements are met.

From an analysis of 11 in-depth interviews conducted with designers of context-aware systems, Bauer et al. [BNK14] also propose a process model in four steps. In the process model, designers must first frame a design space of contexts by determining what contexts are relevant for their applications. Then designers have to encode the features triggered by the contexts. Next they must unify and complete their solution by adding needed constraints to better design their application. Finally they have to evaluate if their solution meets initial requirements.

Visualisation tools There exist several visualisations or tools helping developers to manage the complexity of having contextual information, many fine-grained features or contextual adaptations. For example, some tools help to visualise models or interactions that features can have with the system's classes.

Bobek et al. [Bob+15] suggest a *ContextViewer* tool that displays contextual information to users to help them to better understanding context data.

Kästner et al. [Kas+09] propose the *FeatureIDE* tool to visualise feature models, which is an open-source visualisation framework integrated in the Eclipse development environment. For dealing with larger feature models, Mendonca et al. [MBC09] tackle the scalability issue with a *S.P.L.O.T.* web-based system that represents feature models in a much more compact tree-like structure. Urli et al. [Url+15] present a visual and interactive blueprint that enables software engineers to decompose a large feature model in many smaller ones while visualising the dependencies among them.

Illescas et al. [ILE16] put forward four different visualisations that focus on features and their interactions at source code level, and evaluate them with four case studies. Apel and Beyer [AB11] present a visual clustering tool that clusters program elements (like methods, fields and classes) based on the features they belong to, as a way to assess the cohesiveness of the features. Features whose elements form clusters are more cohesive than features whose elements are scattered across the layout.

Interaction tools Furthermore others tools offering an interaction either to design models or activate contexts to trigger adaptations exist.

Nieke et al. [NES17] created the *DarwinSPL* tool suite for integrating modelling in context-aware software product lines. This tool helps developers to model the three dimensions (spatial, contextual and temporal) of the variabilities of such approaches.

Boucher et al. [BPH12] proposed to derive user interfaces from feature models in order to let the users select the appropriate configuration of their systems while ensuring the configuration remains valid.

In the domain of context-oriented programming, Duhoux [Duh16] proposed tool support to simulate the execution of a context-oriented system with many context-specific adaptations.

Positioning Taking inspiration from the Henricksen and Indulska's process model [HI06], we simplify the process model by removing the infrastructure customisation. In fact, this step serves to include new facts or situations in their management layers and generate scripts that manipulate the databases. Since this is related to their architecture, we will remove it because we do not model our contexts with their modelling technique (*context modelling language*).

In addition we better consolidate the importance that features and contexts must be thinking together through all our process model since we think a strong relation exists between them.

In the same perspective of Apel and Kästner [AK09], we build a process model that traces the features during the different phases to more easily detect inconsistencies between the models through the different phases, but adding the notions of contexts and mapping (to get closer to our approach).

Furthermore we also described the different deliverables that must be released after each phase in order to prepare the next one.

Since we have a clear separation of contexts and features, no supporting visualisation tools in the presented papers cannot be directly compared because these tools only treat one of them or both but without an explicit separation of these both.

However all these tools and visualisations can be seen as complementary to the ones proposed in this dissertation and stress the importance of tooling support to manage complexity and increase understandability of designers and programmers.

2.5 Discussion

In this chapter we explored the literature on how to develop context-oriented applications through four perspectives: modelling, implementation, user interface adaptation and tooling support. For each of them, we presented their state of the art and how we positioned our work in relation to this related work. We will now relate our own contributions to this state of the art.

To conceive highly dynamic applications that respond to their surrounding environment, we propose a feature-based context-oriented software development approach.

In our approach we voluntarily separate contexts and features in designing and implementation to simplify the conceptions of context-oriented applications, and also to gain in maintainability, reusability, and understandability. This clear and explicit separation of both refers to the contribution **C1** that aims to define the underlying principles of the FBCOP approach. When analysing the state of the art for this separation, only the modelling topic addresses it while the others do not. This motivates us to strongly take inspiration from the *multiple-product-line-feature model* of Hartmann and Trew [HT08] in which the contexts and features are modelled into two separate feature models with a mapping between both describing what contexts trigger what features.

Contribution **C2** which proposes an architecture for such systems is novel in the sense that no architecture introduces this clear separation between contexts and features.

To implement such context-oriented systems, we also propose a programming framework (contribution **C3**) to assist programmers in their implementation tasks. Again, as contexts and features are clearly distinguished in the

implementation with a context and feature model based on feature modelling, we propose something new with respect to the context-oriented programming literature.

To create applications with user interfaces, we differ our approach to the literature. Instead of having a library or programming framework that treats user interface adaptation, we develop an user interface library (contribution **C4**) to help programmers to implement user interfaces. Since the user interfaces are implemented similarly to core logic components, they are adaptive by definition.

Finally our supporting development methodology (contribution **C5**) and visualisation tools (contribution **C6**), to help designers and programmers to conceive FBCOP applications, are inspired by the literature but emphasise the separation between contexts and features.

Therefore our FBCOP approach is a new software development approach that revolves around context-oriented programming [HCN08], feature modelling [Kan+90] and dynamic software product lines [HT08; Ach+09; COH14; Men+17], with a clear and explicit separation between contexts and features.

Part II

Approach and usage

The first part of this dissertation set the scope of our research. In the literature, we have observed that a lot of work has been done on how to design and implement dynamic adaptive software systems, in the areas of modelling, implementation, user interface adaptation and supporting tools. However, all of this work addresses only some of these fields of research or are dedicated to a specific concern (*i.e.*, the core logic or the user interfaces) of applications. Nevertheless a unified approach integrating all these areas and concerns would be more efficient for designers and programmers of such systems since only one technology might be used to conceive them. Therefore, this motivated us to propose a feature-based context-oriented software development approach (FBCOP for short) that integrates context-oriented programming [HCN08], feature modelling [Kan+90], dynamic software product lines [HT08; Ach+09; COH14; Men+17] in a unified approach to create and develop dynamic software systems that are able to adapt their behaviour depending on the contexts in which the systems run. In this approach, we also integrate user interface adaptation (UIA) through a library to build and compose dynamically the user interfaces at runtime. Finally we also develop a supporting development methodology and visualisation tools to help designers and programmers in their various tasks when conceiving such systems.

This second part thus aims to introduce our FBCOP approach through a designers and programmers perspective on how they can use it to create dynamically adaptive software systems. We will start by introducing the FBCOP programming paradigm (Chapter 3) in which we will explain its underlying principles and concepts, its system architecture through a control flow and a supporting development methodology. We will then describe how programmers can implement such applications with FBCOP (Chapter 4) and how they can use our UIA library to implement their user interfaces (Chapter 5). Finally we will present two visualisation tools (Chapter 6) we developed to help programmers in their development and debugging tasks to create FBCOP applications.

CHAPTER

3

FEATURE-BASED CONTEXT-ORIENTED PROGRAMMING PARADIGM

In this dissertation, we propose a feature-based context-oriented software development approach (also called FBCOP approach) that reconciles the notions of context-oriented programming [HCN08], feature modelling [Kan+90] and dynamic software product lines [HT08; Ach+09; COH14; CHD15; Men+17] and unifies them into a single software development approach.

In this chapter we will first describe the principles and concepts underlying the FBCOP programming paradigm, then outline the control flow of its system architecture, and explain the supporting development methodology we suggest when conceiving FBCOP applications.

3.1 Principles and concepts

In this section, we will define and exemplify each notion needed to better understand FBCOP. We will start by describing what is a feature and a feature model. We will then explain what is a context and a context model. Finally we will describe what is a context-feature mapping and the mapping model.

3.1.1 Feature

Kang et al. define a feature as “*any prominent or distinctive user-visible aspect, quality, or characteristic of a software system*” [Kan+90].

Whereas the above definition is quite generic and applies to any kind of software features or even non-software features (car production for example), in this dissertation we will adopt a more specific definition of features.

Definition *Features are implementation components, or parts thereof, that are visible and distinguishable to the end-user and which may be relevant depending on the particular user or usage context.*

In our definition of feature we can further identify some properties or design choices that can further clarify the kinds of features we are interested in: the concern they address, their granularity and their binding time.

Concern The features we are interested in may include more core logic components as well as user interface components, or parts thereof.¹ In other words, a feature can be dedicated to one or many core logic components and/or one or many user interface components. Therefore to ensure a good separation of concerns, a feature is separated in feature parts, one for each component. For example, in our smart messaging system, a feature is to send a message to other people. Such a feature is dedicated to both concerns (*i.e.*, core logic and user interface). Therefore this feature is composed by two feature parts. The first feature part is to add a ‘send’ button (user interface concern) and the second feature part is to send the message through the Internet network (core logic concern) when users click on the ‘send’ button.

Granularity Features can either be fine-grained or coarse-grained. Fine-grained features only interact (*i.e.*, adapt) with a few classes or methods. When features only adapt a method, we will also say that these features are variants for methods. An example of a fine-grained feature could be the feature to send or store the messages. In our smart messaging system, this feature only refines the behaviour of sending a message. At the opposite end of the spectrum, if features interact with many components in the system, they are considered as coarse-grained features since they adapt the behaviour of many system classes. For instance, a feature that enlarges all the texts for visually impaired people is coarser-grained as it needs to interact with several classes to change the size of all the UI objects.

¹Another concern, that is not explored in this dissertation but could be explored in future work, is a data concern focussing on what and how data is accessed and used by the system to implement its core functionalities.

In our FBCOP approach, we want to promote features as fine-grained as variants for methods or even parts thereof allowing to dynamically override the behaviour of a simple method by a more specific one. With fine-grained features, we promote the reusability of these features inside an application itself or between other projects. Nevertheless it is not always possible to implement only fine-grained features. This thus explains why we do not want to rule out coarse-grained features, even if we think they should be limited.

Binding time A feature can have multiple binding times [Kan+90], from a design binding time to a runtime binding time, through the compilation or even the configuration one, for example. In our FBCOP approach, the features are designed during the design time but are selected, activated and (un)deployed at runtime since the features are triggered when new changes appear in the surrounding environment.

3.1.2 Feature model

Kang et al. propose to model the features of a software system with a feature model [Kan+90]. Such a model is often used in software product lines to define a family of similar systems with many variations. This model captures the commonalities and variabilities of a system. Commonalities represent the features that need to be present in each instance of the system, while the variabilities are features that may be present in some instances of the system only.

Feature diagram One of the visual representations of feature models are the feature diagrams. A feature diagram is a tree-like structure that can be used to model different features of a system [Kan+90]. Figure 3.1 depicts an excerpt of a feature diagram representing the feature model of our messaging system. The nodes of such a diagram represent the features of the system and the edges represent hierarchical constraints between a feature and its parent feature such as: *mandatory*, *optional*, *or* and *alternative* constraints. A *mandatory* constraint requires that the child feature and its parent are always present in the system together. It is drawn as a black circle on the child in the diagram. In our smart messaging system we have for example considered that messages of type *Text* are always available whatever the current situation. This means that the end-user will always write a textual message. An *optional* constraint means that the child feature may be present if its parent is present in the system. This constraint is drawn as a white circle on the child. For example, the ability to *Receive* messages will not always be present in the system: if the system has no connection, it cannot receive messages. Another optional feature is for the system to provide *Predefined* messages that can for

example be selected by the user when driving a car, to avoid having to type messages while driving. *Alternative* constraints enforce that exactly one child feature is present in the system if the parent is. This constraint is drawn as a white triangle on the parent. In our example, we can only have one direct child feature of *Send*. When the device is connected, the *Send Message* feature will be chosen to send messages over Internet. Otherwise the *Store Message* feature is chosen that stores sent messages on the device while waiting for a new connection to be established. The *or* constraint, drawn as a black triangle on the parent, implies that at least one child feature is present in the system if the parent is. If *Rich* (representing a richer messaging mode than pure text) is present, we need to have either *Picture* or *Position* or both present in the system.

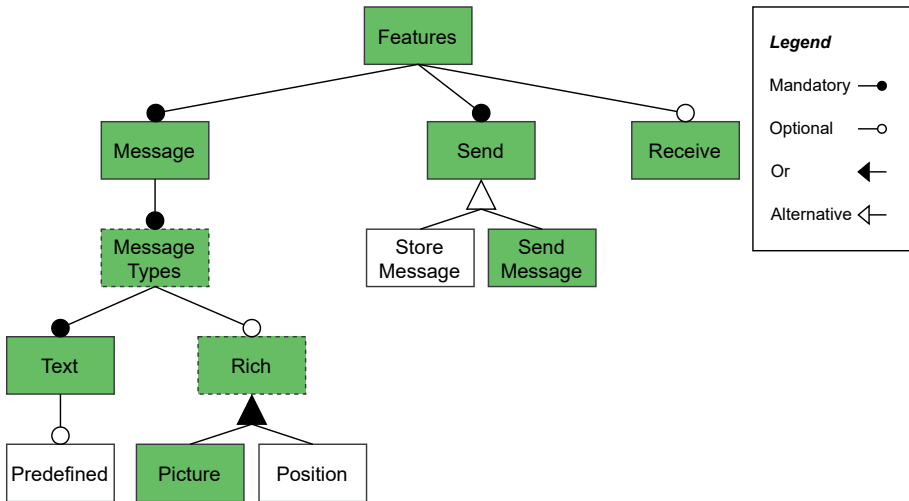


Figure 3.1: Feature diagram representing a simplified version of the feature model of a messaging system. Dashed boxes are abstract features and solid boxes are concrete features. Green boxes represent a valid configuration of the feature model.

In addition to hierarchical constraints, the feature diagram may also link nodes that are not in a parent-child relationship, through cross-tree constraints such as *exclusion* or *requirement* dependencies. The *exclusion* dependency between features represents a mutual exclusion relation where the features cannot be present in the system simultaneously. A *requirement* dependency means the source feature can be present in the system if and only if the target feature is already present in the system.

All these constraints are needed to determine what is a valid *configuration* of the feature diagram. A valid configuration corresponds to a selection of features that together satisfy the constraints imposed by the feature diagram.

Such a configuration in Figure 3.1 can be the selection of the features: *Features*, *Message*, *Messages Types*, *Text*, *Rich*, *Picture*, *Send*, *Send Message* and *Receive*. If the *Picture* feature is missing, this configuration is no longer valid since the *Rich* feature is active and implies that at least one sub-feature is active.

Finally we can also distinguish abstract features from concrete ones in a feature model. Whereas abstract features are mainly infrastructural and help programmers group related finer-grained functionalities but have no actual functionality associated to the abstract feature itself, concrete features contain code that may affect existing system behaviour when they get selected. In this dissertation we will draw the abstract features with a dashed box and the concrete features with a solid box. For instance, in Figure 3.1, an abstract feature is *Message Types*. This feature serves only to the designer to organise the different types of messages the messaging system accepts to facilitate his design. While *Store Message* is a concrete feature as it contains code to store the unsent messages.

More information on a generic formalism of feature diagrams can be found in the Schobbens et al.'s work [SHT06].

3.1.3 Context

Based on our definition of a feature, the relevance and need for certain features may depend on a particular user or context of use. This means that the behaviour of the system may vary depending on the information sensed in the surrounding environment in which it operates. This information can take the form of user preferences (a user's age, (dis)abilities), information from external services (weather conditions), or internal data about the device on which the system runs (remaining battery level or other sensor information) [Abo+99; Cou+05].

Many definitions of the term “*context*” have been given and surveyed by Abowd et al. [Abo+99]. They categorized different kinds of contexts based on the famous “W” questions (“When”, “What”, “Who” and “Where”) to explain “Why” the system needs to adapt its behaviour. According to these questions the contexts can be for example the time of the day (“When”), the entity with which the system interacts (“What”), the kind of end-user using the system (“Who”) or the location where the system runs (“Where”). Thevenin and Coutaz [TC99] also mention the sources of the contexts for which adaptations may be interesting can from the user, the (external) environment and the physical characteristics of the system (*i.e.* the internal environment of the running device). Coutaz et al. [Cou+05] state that “*Context is key*” in the development of context-aware systems: the context is not only a state but is part of a process where contexts serve as triggers to install or remove features that refine the behaviour of the system.

Since many similar definitions of context exist in the literature we will not list all of them in this dissertation but we will provide our own definition of this term in feature-based context-oriented programming adhering to Coutaz et al.'s observations [Cou+05].

Definition *Contexts reify any information sensed from the surrounding environment, through user interaction, or information received from sensors describing the external conditions and internal state of the device on which the system runs, to represent particular situations for which they trigger relevant features to adapt the behaviour of the software system.*

For example, contexts coming from the user could be the user's age, (dis)abilities or even their intentions via the user interaction. Those coming from external devices could be weather conditions, geolocalisation or any message that a sensor can send over the network; while the contexts of internal devices can be about the remaining battery level, memory usage of the device and much more.

Before explaining how these contexts are connected to the features to affect what and how features are chosen depending on the currently active contexts, we will explain how the different contexts can be structured into a combined context model that bears a lot of similarity with how features are structured together in a feature model.

3.1.4 Context model

As for the features it is interesting to design the contexts in a structured way. Desmet et al. [Des+07] suggest to model them through a kind of a feature model in their context-oriented domain analysis approach. This approach was suggested to better collect the requirements of context-aware systems and is strongly inspired by feature-oriented domain analysis [Kan+90], even if they are strongly different semantically. Hartmann and Trew also proposed to use a feature modelling notation to design the context variability model of their *multiple-product-line-feature model* in order to express the contexts that will affect what features will become part of a product [HT08]. In addition, other researches have shown interest in the idea to use feature modelling to model the contexts of context-aware systems [Ach+09; JLS10; COH14]. Following this consensus [Des+07; HT08; Ach+09; COH14; CHD15; Men+17] we also reuse feature modelling to design the contexts of our context-oriented systems. Using feature modelling to model context models allows the FBCOP designers to have a better visibility of the contextual knowledge and enhances their reusability in other context-oriented projects to reduce the design complexity of such systems if the context models are sufficiently generic [JLS10].

Context diagram Just like the feature model, the context model can be represented with a context diagram. A context diagram is also a tree-like structure, but where the nodes represent the contexts to which the system can adapt its features and the edges are the hierarchical or cross-tree constraints (also called dependencies) between the contexts of the diagram. Figure 3.2 depicts a context diagram of an excerpt of our messaging system.

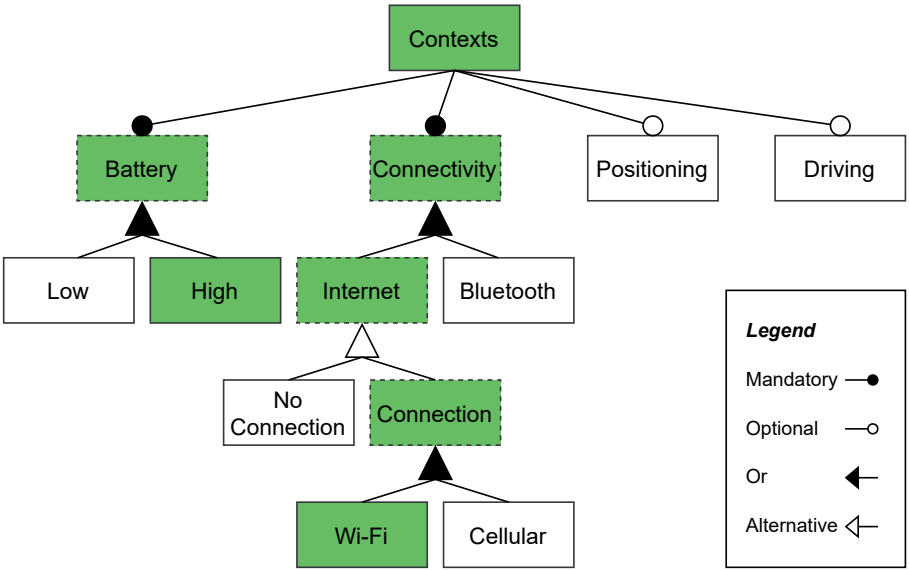


Figure 3.2: Context diagram representing a simplified version of the context model of a messaging system. Green boxes represent a valid configuration of the context model.

Examples of contexts are the abstract *mandatory* context *Battery* representing the battery level of the device on which the system is running with two possible sub-contexts in an *alternative* constraint: either the battery is *Low* or it is *High*. The context *Connectivity* is an abstract mandatory context to structure the different kinds of connectivity the system may have and has two sub-contexts designed with an *or* constraint: an abstract context *Internet* or a concrete context *Bluetooth*. *Internet* serves here to organise if the device has *No connection* or has a *Wi-Fi* or *Cellular Connection*. The last two contexts in our excerpt of the context model of Figure 3.2 are optional contexts to describe if the device can geolocalize the user (*Positioning*) and if the user is currently driving (*Driving*).

As for a feature diagram, a valid configuration of a context diagram is a selection of contexts that are currently active and that ensures all the constraints of the context model are satisfied. An example of such a valid configuration is: *Contexts*, *Battery*, *High*, *Connectivity*, *Internet*, *Connection*, *Wi-Fi*.

3.1.5 Context-feature mapping

Now that we have defined what are features and contexts, we need to define what is a context-feature mapping.

Definition *A context-feature mapping is a set of relations from the contexts to features to express what contexts trigger what features in order to adapt the behaviour of the running system.*

In our example of a messaging system, an example of a relation of the mapping is a link between the context *No connection* to the feature *Store message* since it is not possible to send a message over the Internet if we do not have a connection. A more complex relation connects the contexts {*Cellular*, *Bluetooth* and *Driving*} to the feature *Predefined*. In this specific situation, we consider the end-user is driving and has a device with cellular connection and Bluetooth to connect his car dashboard with his device so that when he wants to answer a message he can select easily a predefined message when he is driving to minimise interaction with his car dashboard.

In this mapping we do not allow relations from features to contexts because contexts should be the sole triggers of any adaptations. This design choice also simplifies the mindset of the control flow that will be explained in Section 3.2.

The mapping from contexts to features can be either implicit or explicit. For example the mapping is mainly implicit in the implementation of the layer-based context-oriented programming languages where the contexts are layers that contain the adapted behaviour for those contexts [HCN08; SGP12a]. In their modelling of context-aware systems, Capilla et al. [COH14] also suggest implicit mappings for a modelling strategy. They identify the contexts through context features, that are features that may (de)activated when the contexts are (de)activated and distinguish them from non-context features.

Hartmann and Trew [HT08] on the other hand propose an explicit mapping. Their *multiple-product-line-feature model* designs several variants of a same product depending on some contexts. As depicted schematically in Figure 3.3, they split the overall model in two separate submodels: a context variability model (representing the contexts and their intra-dependencies) and a traditional feature model. This allows them to model not only what the common and variable features are, but also how contexts affect what features should (not) become part of a product, by declaring explicit dependencies from the context model to the feature model. With this, we can easily model different products thanks to the variabilities in the context variability model.

As explained in the positioning of our FBCOP approach (see Section 2.1), we strongly believe that a clear separation between the contexts and features

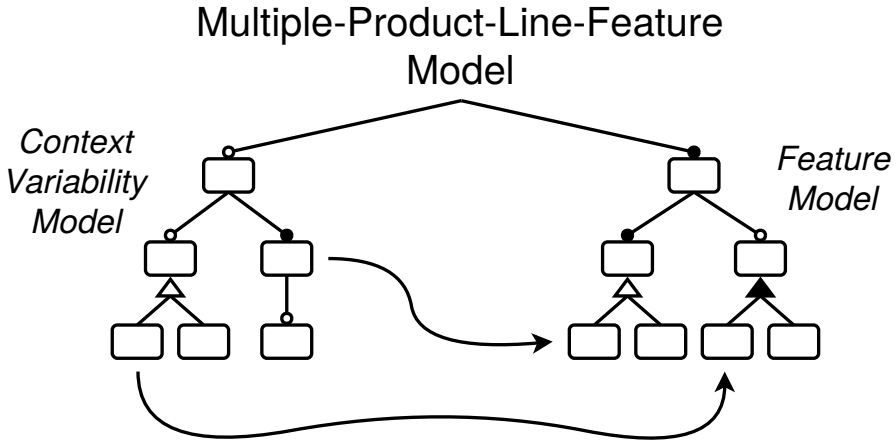


Figure 3.3: *Multiple-product-line-feature model suggested by Hartmann and Trew [HT08].*

with an explicit mapping between them can help the designers of context-oriented software systems when they model the contexts and features of their systems. Therefore we decide to reuse Hartmann and Trew’s approach [HT08] in our approach, to distinguish explicitly the context model and the feature model and for which an explicit mapping between the contexts and features is declared.

3.1.6 Mapping model

Now that we have explained the notion of the mapping between contexts and features in context-oriented systems and why we focus on an explicit mapping, we will describe in more detail how we can model this mapping.

Our mapping model consists of a conjunction of individual mappings, each representing a $N:N$ relation from contexts to features. Each individual mapping follows a key-value format where the keys are a set of contexts and values are a set of features. The features of an individual mapping are all activated if and only if all the contexts that identify this set of features are activated. This means that if a context of an individual mapping is deactivated, the features triggered by this set of contexts are deactivated. An example of such a mapping is illustrated in Table 3.1.

In this example, assume that initially the device on which the system runs has *No Connection*. The system cannot send nor receive any messages, and all unsent messages are just stored on the users’ device while waiting for an internet connection. As soon as the device senses a Wi-Fi connection, the system activates the *Wi-Fi* context and deactivates the *No connection* context. Because a mapping exists from context *Wi-Fi* to features *Send Message* and *Re-*

Contexts	Features
No Connection	Store Message
Wi-Fi	Send Message, Receive Message
Cellular	Send Message, Receive Message
Wi-Fi, High	Picture
Wi-Fi, Positioning	Position
Cellular, Bluetooth, Driving	Predefined

Table 3.1: Excerpt of the mapping model of our messaging system.

ceive message the system thus activates the features *Send Message* and *Receive Message* as well as it deactivates the feature *Store Message*. Nevertheless the features *Picture* and *Position* are not activated yet since the contexts *High* and *Positioning* are not activated yet. Later in the day, assume the user is driving and his device senses a *Cellular* connection (and loses its *Wi-Fi* connection) and has a *Bluetooth* connection to communicate with the car dashboard. The system refines the interaction to send messages by proposing only *Predefined* messages on the dashboard when he needs to answer someone, because such a mapping exists. Since a connection is still sensed the features *Send Message* and *Receive Messages* are not removed.

In addition to these explicit individual mappings we have also one implicit individual mapping that selects all the mandatory features that must be activated and installed in the system. These features are automatically triggered once by the root node of the context model when the application is launched. Hiding this individual mapping helps the developers to not worry about such features and to focus only on those contexts that dynamically trigger features. Moreover we think it increases the readability of the mapping model.

As we consider our overall approach relatively complex, we deliberately restricted ourselves to only allow such a simple mapping model in FBCOP, even though it may limit the expressiveness of our mapping somewhat. An example of such a limitation is we need to create an individual mapping for each sensed connection (*Wi-Fi* and *Cellular*) as illustrated in Table 3.1. We will see in future work (see Section 14.1) how we could enhance its expressiveness.

3.2 System architecture

So far, we described the underlying principles and concepts of FBCOP and how these concepts interact together. Now we will detail the FBCOP’s system architecture to provide a global overview of our approach.

The FBCOP approach revolves around our context-oriented software ar-

chitecture [MCD16] and Hartmann and Trew’s *multiple-product-line-feature model* [HT08]. In essence, our architecture and this model propose to manage and represent both the contexts to which the system adapts and the features it adapts in terms of separate independent feature models, as we explained in the previous section. Figure 3.4 presents its system architecture.

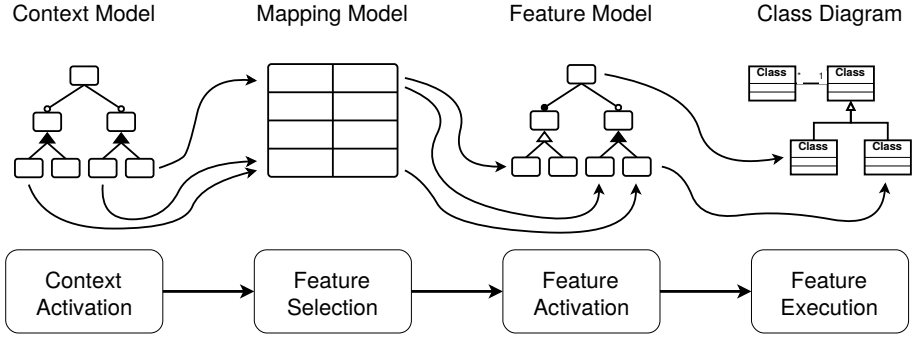


Figure 3.4: Overview of the FBCOP’s system architecture [DMD19].

The control flow of our system architecture goes as follows: whenever a change is detected in the system’s surrounding environment, either by user interaction or information received from external or internal sensors, the contextual information is reified as context objects. The **CONTEXT ACTIVATION** component tries to activate or deactivate these reified contexts in the configuration of the *context model*, while respecting the constraints imposed by that model. If the updated configuration does not satisfy the context model’s constraints the system rolls back to a previous valid configuration. Otherwise, the updated configuration is kept, based on which the **FEATURE SELECTION** component (un)selects the appropriate features thanks to a *mapping model*. The **FEATURE ACTIVATION** component then attempts to (de)activate these features in the configuration of the *feature model*, while ensuring that all constraints imposed by that model remain satisfied. Again, if the updated configuration does not satisfy the feature model’s constraints the system rolls back to a previous valid configuration. Finally when the features have been (de)activated, the **FEATURE EXECUTION** component installs or removes the code of these features in the system to refine and adapt its behaviour while the system is running.

Now that we have explained the control flow of our system architecture, let us exemplify it with a potential scenario of our messaging system. Assume at the launch time the user has a device with a high level of battery and does not sense any connection. The **CONTEXT ACTIVATION** component will try to activate the contexts *High* and *No Connection*. Since all the constraints of the context model are satisfied, this new configuration is kept. Based on the

newly activated contexts, the FEATURE SELECTION component picks up only the *Store Message* feature to activate since no feature can be triggered only by the *High* context, as mentioned in Table 3.1. The FEATURE ACTIVATION component then tries to activate this feature in the feature model. As the mandatory features are always activated at launch time, the features *Message*, *Text* and *Send* are also active at the same time as the *Store Message* feature. Again, as all the feature model's constraints are satisfied, the configuration of the feature model is updated. Finally the FEATURE EXECUTION installs and deploys these newly features in the system.

Now imagine the system detects a Wi-Fi connection. The system reifies it and then activates the *Wi-Fi* context and deactivates the *No Connection* context (because when the system senses a new connection, the *No Connection* context is no more valid). Once the activation and deactivation are executed, it unselects the *Store Message* and selects the features *Send Message*, *Store Message* and *Picture*. Then it deactivates and activates the corresponding features in the feature model since all the constraints are still ensured. Finally the system adapts its behaviour by first removing the *Store Message* feature and then installing the features *Send Message*, *Receive Message* and *Picture*.

3.3 Development methodology

Due to their high run-time adaptivity in terms of active contexts and features, conceiving FBCOP systems is hard. In addition to dedicated visualisation tools and others (such as for example testing tools), designers and developers of such systems can benefit from a supporting development methodology to put them in the right mindset and tell them what to focus on in each step of the life-cycle. This section present such a FBCOP-specific methodology.

At a high level, our methodology, summarised in Figure 3.5, is a typical iterative development process consisting of four phases: *Requirements analysis*, *Design*, *Implementation* and *Testing*.

3.3.1 Requirements

During the *requirements* analysis phase, by carefully analysing the domain of the application, developers are asked to come up with a list of features of the system and a list of contexts to which the system must adapt or refine its behaviour. To avoid misunderstandings and misinterpretations, a lexicon needs to be defined containing a description of each feature and context, as well as a rationale to justify the less trivial features and contexts. Wireframes must also be drawn to get a better idea of what the application or at least some UI layouts look like. This aims to better understand the requirements and to check if designers have well-understood the domain application.

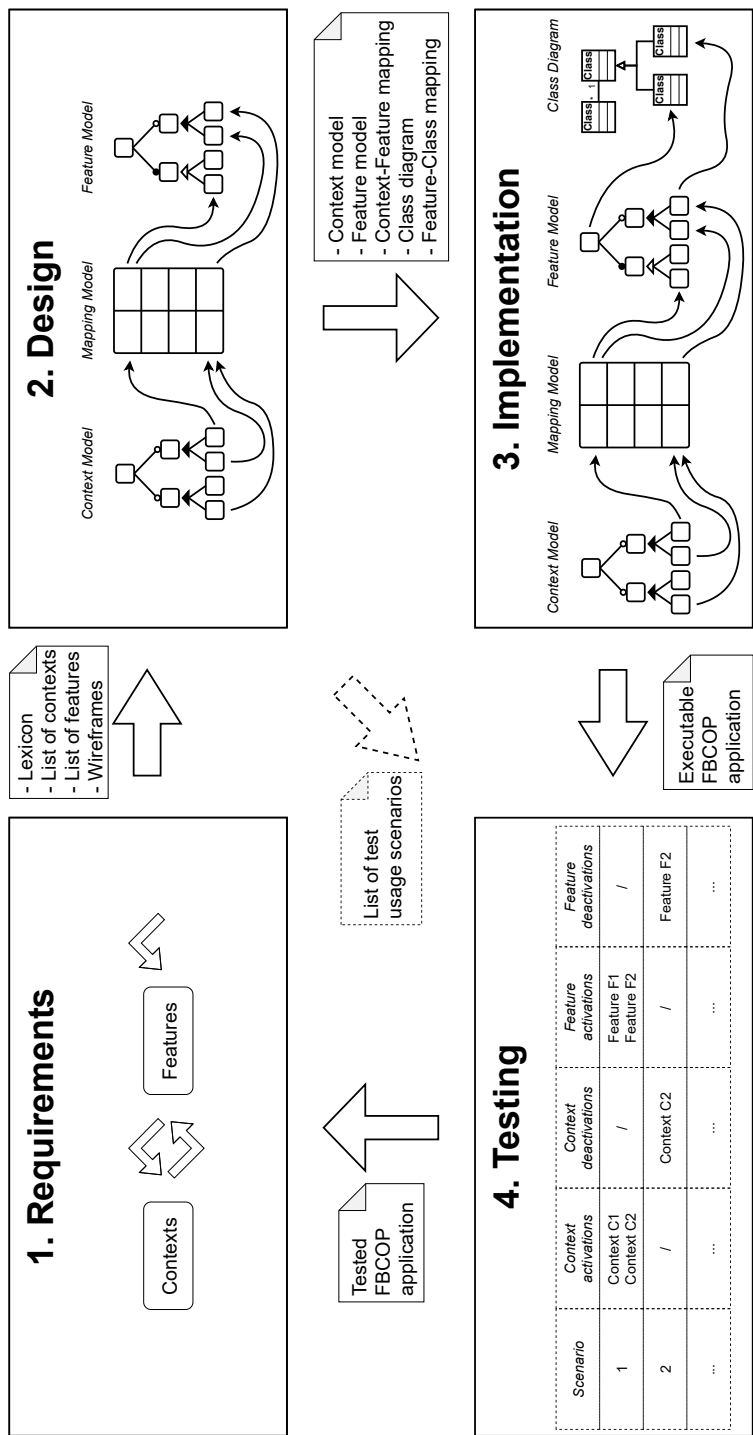


Figure 3.5: Overview of our FBCOP development methodology. The dashed content has been added by the Martou et al.’s work [Mar+21].

To get started, our methodology suggests to identify the main features of the system first. Two main features of our case study are *Sending* and *Receiving* messages. Using these features as starting point, developers are suggested to think about possible variants of these features and in what contexts these particular variants would be triggered. For example, designers could identify that the messages can be stored locally in the device if no Internet connection (i.e., a *Wi-Fi* or *Cellular* connection) is sensed.

These contexts can then be used as a next step to think about what other features or variants of existing features may be relevant to those or related contexts. Having thought about how the Internet connection can influence the feature *Sending*, designers could identify that different types of messages can be sent or received depending on the kind of sensed connection. For example, any type of messages (i.e., *Text*, *Picture* and more) can be sent or received when a *Wi-Fi* connection is detected while only *Textual* messages can be handled with a *Cellular* connection in order to save users' mobile data.

This process is repeated until the designer is happy with the set of features and contexts he has identified to include in a first version of the system to implement.

3.3.2 Design

During the design phase, the developers organise the contexts and features identified during the previous step in a context and feature model, respectively. Additionally, in the context-feature mapping they need to declare explicitly what contexts trigger what features. For the non-trivial relations in this mapping it is advised to provide a rationale to justify their purpose. Putting all these models together yields an overall model such as the one illustrated in Figure 11.5 (in Section 11.1). What still needs to be added to this model is what are the main application classes of the software system, in terms of a kind of class diagram, and what features will adapt what application classes.

3.3.3 Implementation

Now the programmers can finally start to implement their FBCOP application. More information on how to do so can be found in Chapter 4. The most important things to be implemented during this phase are the base application classes and their class hierarchy, as well as the individual features that will modify or add methods defined on these application classes or on previously activated features. During this implementation phase, to help developers understand or inspect the behaviour of the system they are implementing, they can make use of two supporting visualisation tools [DMD18; Duh+19b]. At the end of this phase, an executable FBCOP system is delivered.

3.3.4 Testing

As for all software development methodologies, programmers must test their FBCOP application. This explains why we added the testing phase.

During this step of the development methodology, designers and programmers may wonder what are the typical use scenarios of the system?, whether the system exhibits the expected behaviour in these scenarios? They may also ask whether the dynamic adaptations of the system are relevant, useful and acceptable? With all these questions, designers and programmers can release a list of typical usage scenarios, an assessment of the expected behaviour and an evaluation of the interest, relevance and acceptability of the adaptations. All these deliverables aim to test their FBCOP application.

In this dissertation, we focus our effort on the first three steps.

Nevertheless, another PhD student, Pierre Martou, has started to study this testing phase, and in particular how designers and programmers can generate a list of relevant test usage scenarios. Martou et al. [Mar+21] build their solution upon a pairwise combinatorial interaction testing approach from the domain of software product lines. They implement an algorithm to generate automatically a small set of relevant test scenarios, ordered to minimise the number of context activations between tests. This work explains the dashed content that has been added in Figure 3.5.

3.3.5 Iterative methodology

As stated before, our proposed methodology is incremental. After having done a first iteration through the four phases from requirements to testing, via design and implementation, another iteration can be done to extend the first version of the system with additional contexts and features.

3.4 Conclusion

In this chapter, we defined the underlying notions of the FBCOP approach. While contexts reify *any information to describe particular situations from the surrounding environment in which the system runs*, features are *implementation components, or parts thereof, that are visible and distinguishable to the end-user and which may be more or less relevant depending on the particular user or usage context*. Despite their duality they can be modelled in a similar way thanks to the feature modelling notation to have a context model and a feature model. In addition they are complementary since the contexts trigger the features to adapt dynamically the behaviour of the system. This complementary is characterized by the context-feature mapping in FBCOP.

With such an approach we can propose a clear separation between the

contexts and features. This aims to enhance the maintainability and reusability of these notions in context-oriented systems.

We have then described the FBCOP's system architecture and its control flow through different components: CONTEXT ACTIVATION, FEATURE SELECTION, FEATURE ACTIVATION and FEATURE EXECUTION.

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

Finally we presented a supporting development methodology to help designers and programmers to create their context-oriented applications with FBCOP. This development methodology is composed of four phases: *Requirements analysis*, *Design*, *Implementation* and *Testing*.

CHAPTER

4

FEATURE-BASED CONTEXT-ORIENTED PROGRAMMING FRAMEWORK

Designing a FBCOP approach without proposing a supporting programming language and framework is not really interesting for programmers since they cannot easily develop FBCOP systems that adapt their behaviour dynamically to the context of their surrounding environment.

In this chapter, we introduce, explain and illustrate the usage of the FBCOP programming language and framework. We describe what source code the programmers need to write to create a FBCOP application.

As we implemented our framework and language on top of the *Ruby* programming language, all the illustrated examples of our *smart messaging system* case study will be developed in *Ruby*, or rather, in *RubyCOP*, our context-oriented extension of *Ruby*.

An important part of *RubyCOP*'s language design is the explicit representation of contexts and features in terms of hierarchical tree structures that capture the structural constraints to be respected at runtime.

We will end this chapter with a discussion of whether *RubyCOP* should be regarded from a FBCOP developer's point of view as either a framework, a language or a domain-specific language.

4.1 Defining the application classes

Because context-oriented programming (and *RubyCOP* in particular) is a programming paradigm built on top of object-oriented programming (*Ruby* in our case), the programmers first need to implement the application classes of the application they want to develop. This corresponds to creating the skeleton of the FBCOP application by providing the different classes of the application. These application classes define the basic structure of the application that may be extended or refined at run-time by different features upon activation of certain contexts. Two examples of such application classes are illustrated in Listings 4.1 and 4.2. The application class `MessageModel` represents the model of a message, while the application class `MessageView` is the view of the message model. These application classes will implement a traditional *Model-View-Controller* pattern to keep a good level of separation of concerns of the implemented application to ease its maintainability.

```
1 class MessageModel
2   include Observable
3 end
```

Listing 4.1: Code snippet of the definition of the class `MessageModel` of our messaging system.

```
1 class MessageView
2 end
```

Listing 4.2: Code snippet of the definition of the class `MessageView` of our messaging system.

As opposed to traditional object-oriented programming practice, the programmers deliberately keep all the application classes as empty as possible. This design choice allows them to put all the behaviour, even the default one, in (mandatory) features, thus creating a kind of homogeneity between mandatory and non-mandatory features. The behaviour described by the mandatory features will be installed in the application at launch time.

Finally, the developers must also include the framework module `CodeExecutionAtLaunchTime` in the metaclass of one application class (preferably the main class) of the system. This framework module executes the default behaviour at the launch of the application after this behaviour has been installed in the application classes. The default behaviour consists of the mandatory features and those non-mandatory ones that are triggered by the default context describing a default environment at launch time. Without this framework module, the default behaviour of the system will not be executed. Listing 4.3 shows how developers must include this module (Line 3) in the application class `SmartMessagingSystem` of their messaging system.

```

1 class SmartMessagingSystem
2   class << self
3     include CodeExecutionAtLaunchTime
4   end
5 end

```

Listing 4.3: Code snippet of how developers include the module `CodeExecutionAtLaunchTime` in the main class `SmartMessagingSystem` to run the default behaviour of our messaging system.

4.2 Declaring contexts and features

Together with application classes (representing the system's structure that can be adapted dynamically), contexts and features are the primitive building blocks of our language. Due to their different nature, each of these notions is reified by a different class in our implementation framework: the class `Context` and the class `Feature`. While a context is just identified by its name, a feature is characterised by a name and a list of targeted application classes, that is, the application classes of which this feature can adapt the behaviour. Each of these two framework classes inherit, respectively, from their abstract counterpart `AbstractContext` and `AbstractFeature`. This extra abstraction level is motivated by the need to have both abstract contexts and concrete ones in the context model (and similar for features). Abstract contexts are just infrastructural entities to help structuring a context model, whereas concrete contexts correspond to an actual situation that may occur in the environment surrounding the application. Similarly, abstract features are just there for helping to structure the tree hierarchy, but are not attached to a particular application class that they can alter, as opposed to concrete features.

Listing 4.4 illustrates how application developers can create concrete and abstract contexts.

```

1 context :@no_connection , 'No connection '
2 abstract_context :@connection , 'Connection '
3 context :@wifi , 'Wi-Fi '
4 context :@cellular , 'Cellular '

```

Listing 4.4: Code snippet of the definition of some concrete and abstract contexts of our messaging system.

In this example (Listing 4.4) based on our messaging system, they need to declare some contexts to reify the kind of connectivity the device senses. Here the device can sense if it has *No connection* or a *Wi-Fi* or *Cellular* connectivity. All these contexts are concrete. The context *Connection*, on the other hand, is

an abstract one since it serves to structure the different types of connectivity the device can sense. In the context model, depicted in Figure 4.1, this abstract context will serve as a parent context of the two contexts *Wi-Fi* and *Cellular*.

Listing 4.5 shows how concrete and abstract features of the feature model shown in Figure 4.2 can be defined by application programmers. Based on our messaging system, we define a concrete feature *SendMessage* that will adapt the application class *MessengerService* with the behaviour of sending messages on the Internet. We also create two concrete features dedicated to different types of messages that can be handled by the messaging system. The types of messages are *Text* and *Picture*. These features adapt the application classes *MessageModel* and *MessageView* to modify the behaviour (*i.e.* the model and the view) of a message so that a message can have a text and a picture. Again to structure the feature model, we define an abstract feature *Rich* that will group all the richer types of messages (*i.e.* all but pure text messages) that application users can read and write.

```

1 feature :@send_message, 'SendMessage',
    ↪ [: MessengerService]
2 feature :@text, 'Text', [: MessageModel, : MessageView]
3 abstract_feature :@rich, 'Rich'
4 feature :@picture, 'Picture', [: MessageModel,
    ↪ : MessageView]
```

Listing 4.5: Code snippet of the definition of some concrete and abstract features of our messaging system.

4.3 Defining features

We just showed how programmers can declare abstract and concrete contexts and features. Nevertheless they also need to define how these contexts can be reified when a change is sensed in the surrounding environment and the source code that must be attached to features to create real context-oriented programs. While we have not yet addressed the definition of the contexts in our programming language and framework (see Section 14.2 for a discussion of how we could implement them), we tackled the feature definitions.

In this section we then describe how the features need to be defined by the programmers using our implementation framework to create FBCOP applications.

An important part in the declaration of a feature is that a concrete feature needs to have a list of targeted application classes of which the feature can adapt or refine the behaviour. This means that to such features we need to attach some code that defines how to adapt or refine the behaviour of these application classes when they are activated. Therefore, after having declared

the concrete features¹, the programmers need to implement each feature as a trait (*i.e.* a module) in Ruby. The trait contains the code of the feature to be installed in the application class(es) it adapts and must have the same name as the name of the declared feature.

Listing 4.6 illustrates how developers can implement a feature. In this example, we show the implementation code of the feature *Text* which adapts the *MessageModel* and *MessageView* application classes.

```
1 module Text
2   module Model
3     can_adapt :MessageModel
4
5     def initialize()
6       # Some code here
7     end
8   # More code
9 end
10
11 module View
12   can_adapt :MessageView
13
14   def create_message_widget(frame)
15     # Some code here
16   end
17
18   def update()
19     # Some code here
20   end
21 # More code
22 end
23 end
```

Listing 4.6: Code snippet of a concrete feature definition of the feature *Text* of our messaging system.

As defined in Subsection 3.1.1, a single feature can address several concerns and thus can have several feature parts, such as for example a component dedicated to the core logic of the application and a component for the user interface component. For each feature part the programmer needs to implement a module inside the trait, as illustrated on Lines 2 and 11 in Listing 4.6. The first feature part *Model* defines an adaptation of the core logic, the other feature part *View* adds a refinement for the user interface.

The developers also need to state explicitly which application classes the feature part adapts. This can be expressed by the method *can_adapt*. In List-

¹Note that for the abstract features there is no associated code as they are only used to structure the feature model.

ing 4.6, the feature part `Model` adapts the application class `MessageModel` as shown on Line 3. The feature part `View` on the other hand adapts the behaviour of the application class `MessageView` as shown on Line 12. (Intuitively, such an adaptation means that the code of that feature part will be textually included in that class, even though it is a bit more subtle and not exactly implemented like that as we will see in Section 9.2.)

In addition, a feature (composed by its feature parts) can be reused to adapt different application classes depending on the particular contexts in which the application runs. For that the programmers must mention for each feature part the other application classes that it can adapt with the method *can_adapt*. They must then declare different features in their feature model by precisising which application classes will be really adapted or refined when these features are activated. An example of this usage is illustrated in Listings 11.9 and 11.10 with a feature *Name* (see Section 11.1).

If the programmers do not precise an application class in the declaration of the feature or in the list of potential application classes that a feature part can adapt, our implementation framework raises an error to inform the programmers the feature cannot adapt the declared application classes in its declaration.

After that, the programmers can develop the entire behaviour of the feature, *i.e.* all the methods that compose each feature part of the feature, as exemplified by the methods in Lines 5, 14 and 18 of Listing 4.6. (Since for now our goal is mainly to show how the code is structured, we commented out the body of the methods. A concrete example with real source code is shown in Section 11.1.)

Installing features, *i.e.* their feature parts in the different application classes of the system, modifies the behaviour of such classes, but does not execute immediately this new behaviour. Such a behaviour will only get executed the next time one of these installed methods gets called. Nevertheless it is sometimes important to execute some code as soon as some features are (de)activated. For example when the programmers need to update or remove UI objects according to the new behaviour or when they need to initialise the connection to a server when an Internet connection is sensed. To do that, the programmers can set a prologue and an epilogue in the different feature parts with the methods *set_prologue* and *set_epilogue*, respectively, as inspired by Calvary et al. [CCT01a; CCT01b] and López-Jaquero et al. [Lóp+08]. This prologue or epilogue takes a list of methods as argument to describe what methods need to be executed at the activation or deactivation of the feature. While the prologue is executed immediately after activation of the feature, the epilogue is executed immediately before deactivation of the feature. The declared methods are executed in the order in which they were declared in the argument list. This means that programmers must pay attention to the

order of declaration of these methods in the prologue or epilogue so that they are executed in the correct order. An example of how to set the prologue is illustrated in Line 5 in Listing 4.7. (Again we commented out the body of these methods.)

```

1 module SendMessage
2   module Behaviour
3     can_adapt : MessengerService
4
5     set_prologue : init_connection , : send_stored_messages
6
7     def init_connection ()
8       # Initialise the connection to the server
9     end
10
11    def send_message (message)
12      # Send message
13    end
14
15    def send_stored_messages ()
16      # Send stored messages
17    end
18  end
19 end

```

Listing 4.7: Code snippet of a part of the feature definition of our messaging system that uses a prologue to execute two methods immediately after the activation of the feature *SendMessage*.

In the example of Listing 4.7, when the system activates the *SendMessage* feature because it has sensed an Internet connection, we first need to initialise the connection to the server and then send all the stored messages the user wanted to send before. The prologue allows to automatically trigger the execution of these methods as soon as the feature gets activated.

4.4 Declaring the context model

Now that we have explained how programmers can declare individual contexts and features and define features, we show how programmers building a FBCOP application can declare the *context model* of their application. To do so, the programmers need to extend the framework class `ContextModelDeclaration` of our programming framework. More information about the framework class `ContextModelDeclaration` will be discussed later in Section 7.2.

A code example of the context model declaration of our messaging system, represented graphically in Figure 4.1, is shown in Listing 4.8, with the

application class `AppContextModelDeclaration`.

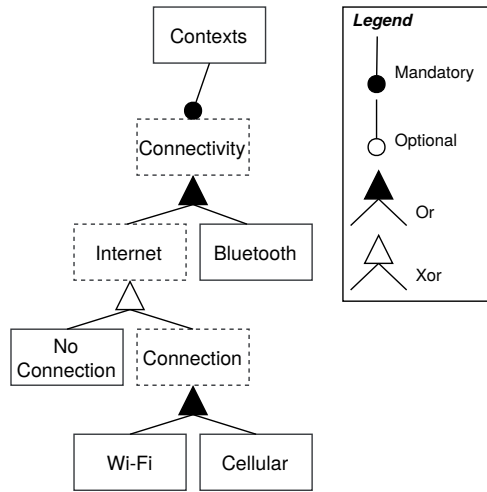


Figure 4.1: Excerpt of the context model of our messaging system that we declare in Listing 4.8.

```

1 class AppContextModelDeclaration <
    ↳ ContextModelDeclaration
2
3 include Singleton
4
5 def initialize()
6     super()
7     _define_connectivity_context()
8     # More code to declare the contexts and relations
9     @root_context.relation :Mandatory, [@connectivity]
10    # More code to attach contexts to the root node
11 end
12
13 def _define_connectivity_context()
14     abstract_context :@connectivity, 'Connectivity'
15     abstract_context :@internet, 'Internet'
16     context :@no_connection, 'No connection'
17     abstract_context :@connection, 'Connection'
18     context :@wifi, 'Wi-Fi'
19     context :@cellular, 'Cellular'
20     @connection.relation :Or, [@wifi, @cellular]
21     @internet.default @no_connection
22     @internet.relation :Alternative, [@no_connection,
    ↳ @connection]
23     context :@bluetooth, 'Bluetooth'
24     @connectivity.default @internet

```



```

25     @connectivity.relation :Or, [@internet, @bluetooth]
26   end
27   # More methods to declare the context model
28 end

```

Listing 4.8: Code snippet of the context model declaration of our messaging system.

The programmers creating this application class must first define it as a singleton, as illustrated in Line 3 of Listing 4.8, since a FBCOP application is supposed to have only one context model.

Then they must implement the method *initialize* (Line 5). This method must first call the *super* method (Line 6) which will initialise the root context, then declare their contexts as shown in Listing 4.8², and finally attach them to the root context.

To relate contexts with a constraint to their parent context in the context model, our language framework provides a method *relation* that takes two arguments: the name of the constraint and a list of the contexts. Examples of this usage are exemplified on Lines 9 and 20. The first example (Line 9) shows how we can declare *Connectivity* as mandatory child of its parent *Contexts* (i.e. the root context). A second example (Line 20) attaches the contexts *Wi-Fi* and *Cellular* as children of the parent context *Connection* under an *or* constraint.

For uniformity reasons we impose that the mandatory features (i.e. the commonalities) are also triggered by contexts, even if these features are only activated once, at launch time and will never be deactivated during the entire execution of the system. As a consequence of this design choice, we need to define some default contexts that will automatically become active at the launch time of the application. So we need to allow programmers to declare some contexts as default when they declare their context model. First of all, the root context named *Contexts* will always be active by default. For child contexts, the programmers can precise which child context will be chosen by default by the parent context. For that, they must assign what child context is the default context of its parent context, as showed on Lines 21 and 24. In the example shown on Listing 4.8, *No Connection* will be the default child of *Internet* (Line 21). *Internet* will be default child of *Connectivity* (Line 24) and this one is a mandatory child of *Contexts*, the root context which is activated by default at launch time. So by propagating the subcontexts, *Connectivity*, *Internet* and *No Connection* will always get activated at launch time. Declaring some contexts as default is specific to the context model. It is not useful for the feature model declaration since each feature needs to be triggered by at least a context in our approach. So the way to ensure a certain feature is active by default is to ensure that it is triggered by a context which is active by default.

²We partially commented out the declaration of the context model for conciseness.

We will come back to this discussion when we talk about the context-feature mapping (Section 4.6).

4.5 Declaring the feature model

Similarly to the declaration of the context model, the programmers must declare the feature model of their FBCOP application. For that they must create an application class that extends the `FeatureModelDeclaration` framework class and define it as a singleton since their application must have only one feature model. This framework class has some infrastructural scaffolding code just like its homologue `ContextModelDeclaration` for the context model. More information about the `FeatureModelDeclaration` framework class will be discussed later in Section 7.2.

An example of how programmers can declare the feature model of our messaging system, as depicted in Figure 4.2, is illustrated in Listing 4.9.

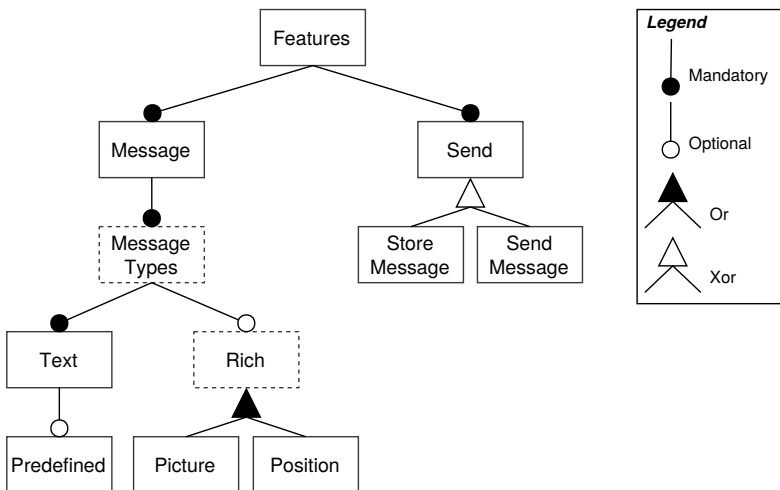


Figure 4.2: Excerpt of the feature model of our messaging system that we declare in Listing 4.9.

```

1 class AppFeatureModelDeclaration <
    ↪ FeatureModelDeclaration
2
3 include Singleton
4
5 def initialize()
6     super()
7     _define_send_features()
8     _define_message_features()
9     # More code to declare the features and relations

```

```

10     @root_feature.relation :Mandatory, [@message, @send]
11     # More code to attach features to the root node
12 end
13
14 def _define_send_features()
15     feature :@send, 'Send', [:MessengerService]
16     feature :@store_message, 'StoreMessage',
17     ↪ [:MessengerService]
18     feature :@send_message, 'SendMessage',
19     ↪ [:MessengerService]
20     @send.relation :Alternative, [@store_message,
21     ↪ @send_message]
22 end
23
24 def _define_message_features()
25     feature :@message, 'Message', [:MessageModel,
26     ↪ :MessageView]
27     _define_message_types()
28     @message.relation :Mandatory, [@message_types]
29 end
30
31 def _define_message_types()
32     abstract_feature :@message_types, 'MessageTypes'
33     feature :@text, 'Text', [:MessageModel,
34     ↪ :MessageView]
35     feature :@predefined, 'Predefined', [:MessageModel,
36     ↪ :MessageView]
37     @text.relation :Optional, [@predefined]
38     abstract_feature :@rich, 'Rich'
39     feature :@picture, 'Picture', [:MessageModel,
40     ↪ :MessageView]
41     feature :@position, 'Position', [:MessageModel,
42     ↪ :MessageView]
43     @rich.relation :Or, [@picture, @position]
44     @message_types.relation :Mandatory, [@text]
45     @message_types.relation :Optional, [@rich]
46 end
47 # More methods to declare the feature model
48 end

```

Listing 4.9: Code snippet of the feature model declaration of our messaging system.

As for the context model declaration, the programmers have to implement the method *initialize* (Line 5) and create the root feature (*Features*) of the model by calling the *super* method.

They must then declare all the features of the feature model and relate them with the method *relation* as explained in Section 4.4. For example, in

Listing 4.9, the features *StoreMessage* and *SendMessage* are attached to the parent feature *Send* as alternative child features (Line 18). This feature *Send* itself is a mandatory child of the root feature as shown on Line 10.

4.6 Mapping contexts to features

In addition to declaring a context and feature model with all the contexts and features of the application, the programmers still need to declare how the system should react to contextual changes, *i.e.*, which context (de)activations trigger which feature (de)activations. To declare this mapping from contexts to features, the programmers must extend the `MappingModelDeclaration` framework class for which they need to initialise the *mapping* instance variable.

This *mapping* is a hash data structure where the key is a list of contexts and the value is a list of features. The reason for using a list structure for the keys and the values is that the programmer cannot only create simple mappings (from one context to one feature) but also more complex ones³. Such more complex mappings are relations from many contexts to many features, indicating that if a set of contexts is simultaneously active, this would trigger the activation of all corresponding features listed. If at least a context of this set of contexts is deactivated, all the corresponding features of this mapping are deactivated.

Listing 4.10 gives an example of an excerpt of the mapping declaration as illustrated in Table 4.1. Note how such mapping model is declared as a singleton class, since just like the context and feature model, the FBCOP application is supposed to have only a single mapping model.

```

1 class AppMappingModelDeclaration <
    ↪ MappingModelDeclaration
2
3 include Singleton
4
5 def initialize ()
6     contexts = AppContextModelDeclaration.instance
7     features = AppFeatureModelDeclaration.instance
8     @mapping = {
9         [ contexts.no_connection () ] =>
        ↪ [ features.store_message () ],
10        [ contexts.wifi () ] => [ features.send_message () ],
        ↪ features.receive () ]

```

³Even more complex mappings than those currently proposed in our programming framework could be envisaged (see Section 14.1). But, given the overall complexity of our approach, for now we have preferred to keep the mapping relatively simple (*i.e.* just a N - N directional mapping of sets of contexts to sets of features).

```
11     }  
12   end  
13 end
```

Listing 4.10: Code snippet of a mapping declaration of our messaging system

Contexts	Features
No Connection	Store Message
Wi-Fi	Send Message, Receive Message

Table 4.1: Excerpt of the mapping model of our messaging system that we declare in Listing 4.10.

In this mapping declaration, the programmers must use the generated accessors of the contexts and features to create the mapping from contexts to features. In this particular mapping, when no connection is sensed, the context *No Connection* would be active, which would imply the selection of the feature *Store Message*, dedicated to temporarily store in memory all the messages the user would like to send to other people. However, when a *Wi-Fi* connection is detected, the context *Wi-Fi* would become active, which would imply the selection of the features *Send Message* and *Receive Message*, in order to be able to send and receive messages to and from people. If a connection is sensed, it means that the context *No Connection* must become inactive since it does not reify the current state of the surrounding environment. We assume that when a kind of connection is sensed (e.g. *Wi-Fi*), the sensors layer triggers the activation of the context *Wi-Fi* and deactivation of the context *No Connection* simultaneously. Such an activation and a deactivation of these contexts trigger the activation of the features *Send Message* and *Receive Message* and the deactivation of the feature *Store Message*.

4.7 Managing the activation order

An important thing we did not yet discuss is the activation order of contexts, and consequently the order of activation of features in the application classes since feature activation is triggered by context activation. In this section we first explain how developers can order the activation of the default behaviour, i.e., how the mandatory features and default behaviour that is triggered by the default contexts are installed. Then we will describe how programmers can order the activation of the features when contexts trigger them. Finally we will also explain and exemplify how they can define the order in which the feature parts must be activated.

Ordering default behaviour

As we have explained in Section 3.1.6, mandatory features are triggered by the root context of the context model at the launch time of the application. These features and their mandatory child features are installed according to the order declared by the developers when they attach the features to the root feature of their feature model. This order between all these mandatory features is determined by the depth-first search algorithm. In Listing 4.9, the activation order is defined on Line 10. This means that the feature *Message* will be installed first in the application classes. Then, the feature *Text* is installed in the application classes since it is a mandatory concrete feature of *MessageTypes*, a mandatory child feature of *Message*. The feature *MessageTypes* is not considered since it is an abstract one that serves only to structure the feature model. Finally the *Send* feature is installed in the application classes. This activation order assumes that the concrete mandatory features compose the full default behaviour, but it is rare. Therefore, when a mandatory feature has a child feature (non-mandatory) as default behaviour triggered at launch time by a default situation in the surrounding environment, this child feature is directly installed after its parent. For example, based on the context model declared in Listing 4.8 and the mapping model declared in Listing 4.10, the feature *Store Message* is deployed in the system's behaviour just after the feature *Send*. In this example, the features are thus activated in the following order (from the first activated feature to the last one): *Message*, *Text*, *Send* and *Store Message*. In a case where the feature *Picture* was also part of the default behaviour, the activation order would be *Message*, *Text*, *Picture*, *Send* and *Store Message*.

Ordering features in the mapping model

We have explained how programmers can order the installation of the default behaviour (*i.e.*, the mandatory features and the features triggered by the default situation at launch time). Nevertheless the programmers may also manipulate the feature activation order through the mapping model when the features are triggered during the execution. To do that the programmers may order the list of features in the mapping relation of the mapping model to precise in which order the features must be activated. The first declared feature in the mapping relation will be the first feature deployed and the last feature is the last deployed in the application classes. In Listing 4.10, if the context *Wi-Fi* is activated, the feature *Send Message* will be installed first and then the feature *Receive Message*.

Ordering feature parts

While the feature parts are deployed in their corresponding application classes, all feature parts of a feature are not always installed in the order in which they are implemented in the feature definition. In addition, as the prologues of the feature parts are executed following the order in which they are deployed, this can sometimes lead to errors. An example of such an error arises the execution of the behaviour of a feature part that uses an instance variable that was not correctly instantiated because the feature that initialises this variable was not executed yet. To avoid such issues, programmers can provide the order of the activation of feature parts. For that the programmers must define the order in which the feature parts are handled with the method `set_feature_part_order` that takes the list of feature parts as argument. Listing 4.11 depicts this usage. Line 2 illustrates that the feature part *Model* will be deployed first and the feature part *View* will be then deployed.

```
1 module Text
2   set_feature_part_order :Model, :View
3
4   module Model
5     # More code
6   end
7
8   module View
9     # More code
10  end
11 end
```

Listing 4.11: Code snippet of how programmers can define the order of the activation of the feature parts in the module *Text*

Without specifying the order of how the feature parts must be activated, all the feature parts will be executed in the order defined by the interpreter.

These different leverage points allow programmers to activate and deploy the features (and consequently, the feature parts) in a certain order. This order is not only used to deploy the features in their corresponding application classes. It is also used to execute the existing prologues (some code of a feature that will be executed automatically after its installation) of the newly deployed features. So it is crucial for the programmers to be aware of the activation order because if the features are not installed in the correct order, the system's execution could be not as expected or could lead to errors because certain methods do not exist yet.

4.8 Activating contexts

Now that we have introduced what the programmers need to provide to create a FBCOP application, *i.e.*, the context and feature model, the mapping, the application classes and the code of the features, we will see how programmers can use our FBCOP system architecture to (de)activate features depending on the detection of certain situations in the surrounding environment. (For example, by reading and interpreting the values of certain sensors. Note that, currently, our implementation framework provides no direct support for obtaining sensor information yet, as this may be very platform specific and can be considered as complementary to our approach. We will discuss about this improvement in the future work in Section 14.2.)

To explicitly activate or deactivate some contexts, programmers can use the methods *activate* and *deactivate*, respectively. These methods take a list of contexts as argument.

Listing 4.12 depicts an example of an activation and a deactivation of some contexts on Lines 1 and 6. In the example on Line 1, the system tries to activate the contexts *Bluetooth* and *Driving*. Such a situation could mean that the user is driving and has connected his smart device to his car dashboard through Bluetooth. The example of the deactivation shown on Line 6 could mean that the user has shut off his GPS on her device.

```

1 ContextActivation.instance.activate(
2     AppContextModelDeclaration.instance.bluetooth(),
3     AppContextModelDeclaration.instance.driving()
4 )
5
6 ContextActivation.instance.deactivate(
7     AppContextModelDeclaration.instance.positioning()
8 )

```

Listing 4.12: Code snippet of an activation and a deactivation of contexts in our messaging system.

As described in Section 3.2, when contexts are (de)activated in the context model, based on the mapping model the system selects the features that need to be (de)activated depending on these contexts and then tries to (de)activate these features. Finally it installs or removes them in the application classes in order to adapt or refine them.

As illustrated in Section 4.6, it could sometimes be useful to propose a method to activate and deactivate some contexts simultaneously. To do that our implementation framework proposes a method *alter* that takes at least one context wrapped in an action to activate or deactivate this context. The framework wrapper *act* serves to precise that the passed context must be activated while the framework wrapper *deact* is to precise that the passed

context have to be deactivated. Listing 4.13 shows an example of the use of the method *alter*. In this example, we want to activate the context *Wi-Fi* and deactivate the context *No Connection* simultaneously.

```

1 contexts = AppContextModelDeclaration.instance
2 ContextActivation.instance.alter(
3     act(contexts.wifi()),
4     deact(contexts.no_connection())
5 )

```

Listing 4.13: Code snippet of the activation of the context *Wi-Fi* and the deactivation of the context *No Connection* simultaneously in our messaging system.

4.9 Proceeding feature execution

Context-oriented applications are systems that adapt dynamically their behaviour with features depending on some particular contexts of the surrounding environment in which the system runs. In our programming approach, this means that the features adapt the application classes at runtime when new changes in the environment are sensed. Allowing the programmers to create features that only adapt the application classes could be really restrictive since the features aimed to adapt the behaviour of the application whatever its refinement or replacement. In addition, in our programming approach, we ask programmers to use features to define the default application's behaviour and that would mean the application could no more adapt it after the launch time and this is not the desired behaviour. Therefore, we consider that features can also refine or replace other features installed in a same class and that a chain of previously installed features will be kept so that new versions can build upon older ones, as proposed in context-oriented programming [HCN08].

When a feature refines the behaviour of a previous feature, it is sometimes useful to build upon that previous behaviour to profit fully from the power of dynamic adaptability and to obtain more reusable and extensible code. For that, Hirschfeld et al. [HCN08] suggested a *proceed* mechanism in context-oriented programming. Our language also implements such a *proceed* mechanism. Programmers can, inside a method of a feature, use a special keyword *proceed*, which will call a previously installed method with the same name provided by a previous feature. When the *proceed* is executed by a method of an application class installed by some feature, the system retrieves the previous version of this method installed by a previous feature, installs it in this application class, executes it, and then reinstalls the most recent version of the feature again. With such a mechanism, the programmers can refine the application's behaviour incrementally through different features.

Let us illustrate this *proceed* mechanism with a `create_message_widget` method to create the view of a message in the application class `MessageView`. Assume a user can write text and upload a picture in the message. After activations of the features *Message*, *Text* and *Picture* (in that order), the default behaviour is installed by the feature *Message*. This behaviour is then refined by the feature *Text* to add a text and finally, refined again by the feature *Picture* to add a picture to the message. As the last version of the method `create_message_widget` comes from the feature *Picture*, when the system calls `create_message_widget`, it executes the method of this feature. The activation order of features determines how features are chained together and thus which features refine or replace previously installed features on a same application class. Figure 4.3 graphically represents in what order the features were installed in the application class `MessageView`.

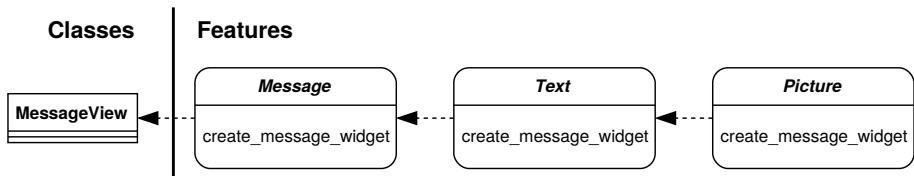


Figure 4.3: Activation order of the features *Message*, *Text* and *Picture* on the class `MessageView`.

The source code snippets of the different features *Message*, *Text* and *Picture* are shown in Listings 4.14, 4.15 and 4.16, respectively.

```

1 module Message
2   module View
3     can_adapt : MessageView
4
5     def create_message_widget(frame)
6       _frame = ... # create a frame for the message
7       # Create a label with the author name in the
8       ↪ frame _frame
9       return _frame
10    end
11    # More methods
12  end
13  # More feature parts
14 end

```

Listing 4.14: Code snippet of an example of a default behaviour (feature *Message*) of the method `create_message_widget` of the application class `MessageView` of our messaging system

```

1 module Text
2   module View
3     can_adapt : MessageView
4
5     def create_message_widget(frame)
6       _frame = proceed(frame)
7       # Create a label with the text of the message
7       ↪ below the author name in the frame _frame
8       return _frame
9     end
10    # More methods
11  end
12  # More feature parts
13 end

```

Listing 4.15: Code snippet of an example of a *proceed* mechanism to refine the view of the message with a text (feature *Text*) in the application class *MessageView* of our messaging system

```

1 module Picture
2   module View
3     can_adapt : MessageView
4
5     def create_message_widget(frame)
6       _frame = proceed(frame)
7       # Attach a picture below the text in the frame
7       ↪ _frame
8       return _frame
9     end
10    # More methods
11  end
12  # More feature parts
13 end

```

Listing 4.16: Code snippet of an example of a *proceed* mechanism to refine the view of the message with a picture (feature *Picture*) in the application class *MessageView* of our messaging system

Assume now that each of these features have been installed in the order depicted in Figure 4.3 and that we call the method *create_message_widget* on an object of the application class *MessageView*. The control flow of this method call then starts with the latter one, *i.e.*, the method of the feature *Picture*, but the *proceed* call immediately delegates to the previously installed feature *Text*, whereas the *proceed* call there delegates to the default version of the method defined by the feature *Message* on the application class *MessageView*. After the execution of the method of *Message* is finished, the con-

trol flows back in opposite order to execute the remainder of each of those methods. Figure 4.4 illustrates this execution flow.

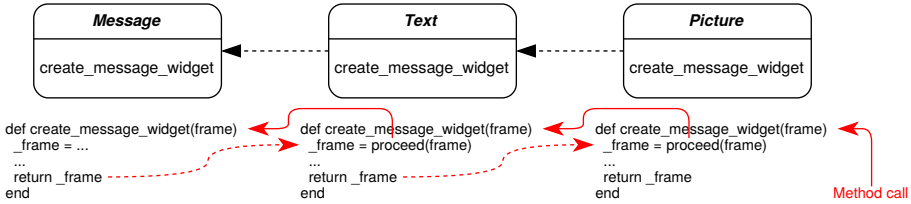


Figure 4.4: Method call semantics in presence of the *proceed* mechanism in our messaging system.

The result of this execution thus creates incrementally a message view object with the author of the message, followed by the text of the message, and followed by a picture. The result is visually represented in an incremental way with wireframes, after executing each method of each feature, in Figure 4.5.

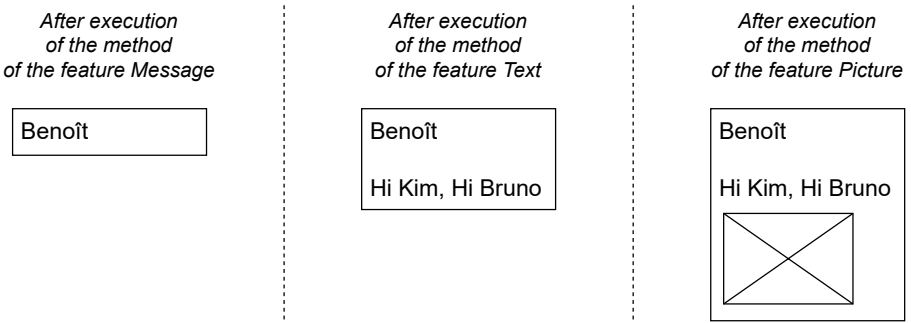


Figure 4.5: Wireframes drawing incrementally the result of the execution of the method `create_message_widget` in our messaging system.

4.10 Language versus framework

We have not yet addressed the question if our programming approach is rather a framework, a programming language, a library or even a domain-specific language from a context-oriented developer’s point of view. We argue that our implementation should be seen as an application framework for developers of FBCOP applications.

First of all, it proposes a generic implementation with some hotspots that developers need to specialise when they implement their FBCOP application. This generic implementation comprises a common architecture and structure for all FBCOP applications in order to facilitate the development of such systems. This implementation is a framework as it applies the principle of

inversion of control to call application-specific code provided by the developers. For example, developers need to declare a context model but its actual creation and usage when contexts need to be (de)activated are managed by the framework. This is also the case for our other models such as the feature model and mapping model. Another example concerns the installation or removal of feature definitions in the classes of the system. Whereas the programmers need to implement the different features definitions, it is the framework's responsibility to install or remove them in the system's classes when they are activated or deactivated. This leads us to consider our implementation as a framework rather than a library. Even when the programmers could call a framework-specific method to (de)activate certain contexts, we can consider it as an entry point of our framework that is itself in charge of selecting, (de)activating and (un)installing the features.

Even though we extended the Ruby programming language to add the *proceed* mechanism which is key to context-oriented programming [HCN08], we did not change the language or grammar of *Ruby*. Moreover we did not have to update any components of the *Ruby* interpreter to implement our FBCOP framework. We therefore cannot claim that our implementation is a programming language; at best it is an extension of an existing programming language, supported by a powerful implementation framework where all the magic of dynamic context activation, feature activation, deployment and execution happens.

Finally, although we occasionally introduce some syntactic sugar to make it easier for programmers to use our framework, this is not key to using the framework and more work needs to be done to make the syntax of using our framework even more compact.

We therefore conclude that our programming approach is a programming framework that helps programmers with all the abstractions we propose.

4.11 Conclusion

In this chapter we have explained what components programmers must implement when using our FBCOP programming framework to create context-oriented applications.

They need to implement the declaration of their context, feature and mapping model, as well as the feature definitions (*i.e.* the actual code of the concrete features that will be installed dynamically in the application classes) and the basic application classes of the system. We exemplified all of these components through snippets of source code.

We also described how they can manipulate the activation order through different leverage points by ordering the (mandatory and) default behaviour, the features in the mapping and the feature parts when they are triggered.

We also explained how the programmers can activate or deactivate contexts to trigger the installation or removal of features depending on these contexts in order to adapt dynamically the behaviour of the system.

Finally we explained and illustrated the important *proceed* mechanism we can find in context-oriented programming. With such a mechanism developers can incrementally build/refine the behaviour of their application at runtime.

CHAPTER

5

USER INTERFACE ADAPTATION LIBRARY

User interfaces are a must-have when conceiving applications for real users. Without them, applications would not be accessible to a wide audience having no knowledge in computer science, but only to experts for some specific tasks. Therefore we extend the FBCOP approach with a user interface adaptation (UIA) library to help programmers to build their user interfaces incrementally and allow them to adapt dynamically depending on the current situation in which the application runs.

Our UIA library relies on the *FXRuby* GUI library¹. *FXRuby* is a library that helps to create “*powerful and sophisticated cross-platform graphical user interfaces*”² for *Ruby* applications.

In this chapter, first we describe how the user interfaces are represented in a tree structure. Then we provide an overview of how a FBCOP application, our UIA library and the *FXRuby* GUI library are interconnected. Next we introduce the API we propose to build the user interfaces and exemplify its usage. Finally we will illustrate this usage in the features of a FBCOP application.

¹<https://github.com/larskanis/fxruby>

²<https://larskanis.github.io/fxruby/>

5.1 Representation of user interfaces

A user interface is a composition of layouts and UI objects. Figure 5.1 illustrates a wireframe of a user interface of our messaging system where end-users can see the different chats on the left and a chat content on the right. In this example, the end-user has four different chats and opens Kim’s chat.

A layout is a container with specific constraints on how it displays its different contained user interface objects (*e.g.*, a vertical or horizontal layout). For example, in Figure 5.1, a horizontal layout has been defined for the global view, so that end-users can see a list of chats on the left and the content of a chat on the right (see black box on the figure). A vertical layout has been used to list all the chats so that the different chats are displayed from top to bottom (see the left blue box on the figure). A vertical layout has been used to show the messages of a given chat and the user interaction below (see the right blue box on the figure). A vertical layout has also been used to show all the messages inside a given chat (see the orange box above on the figure) and a horizontal layout has been used to display the textfield and the button (see the orange box below on the figure). Finally, a vertical layout is also used to show the content of each message (see the yellow boxes on the figure).

A UI object is a final element in a user interface, *e.g.*, a button, a label, and so on. For example, we have a button for each chat so that when end-users click on a button of the chats list, the content of that chat opens on the right part of the user interface. We also have many labels such as the labels “Kim:” and “Can we discuss tomorrow about the LINGI2252 course?”, an input textfield and a “send” button.

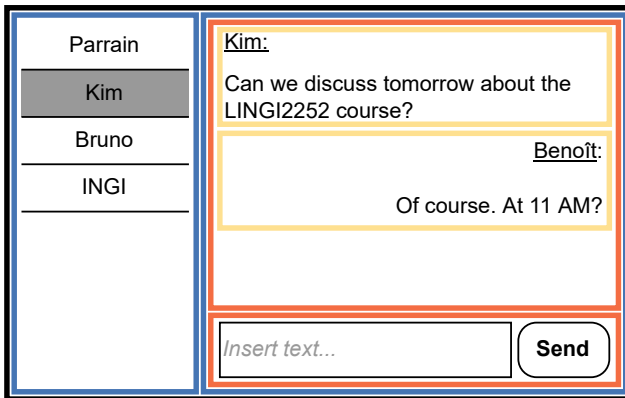


Figure 5.1: Wireframe drawing of a messaging system. Colours serve to show the different containers.

To allow the programmers to compose their user interfaces, we reify the user interfaces through a tree representation. This representation allows to

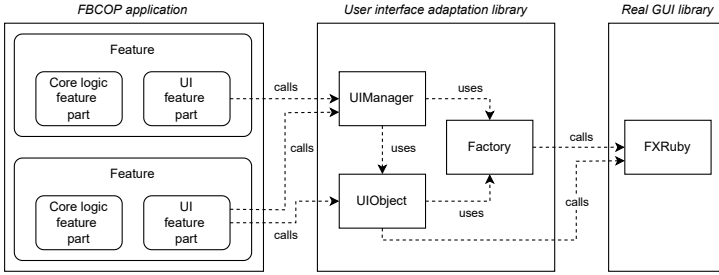


Figure 5.3: Overview of the interaction between the FBCOP application, our user interface adaptation library and the underlying *FXRuby* GUI library on which it relies.

in Figure 5.3. In fact, the `UIObject` library class has a method to delegate any method call to the *FXRuby* library.³

The design choice to create a library as intermediate component between the FBCOP applications and the real GUI library makes it easy to replace the GUI library by another one. In addition, by implementing only a single method in `UIObject` that delegates the method call to the real GUI library allows to implement our UIA library easily on top of an existing one, so that FBCOP application programmers could reuse as much as possible of what the fully implemented and documented real GUI library (*i.e.*, *FXRuby*) offers, such as accessing and modifying the properties of a UI object or attaching events to create user interaction.

5.3 API and usage of the UIA library

In this section, we will describe the full API of our UIA library and exemplify how programmers can implement their user interfaces with the UIA library. We will voluntarily hide the notion of features for a better understanding of the UIA library at this stage. We will explain in the next section how programmers can use it in the implementation of their context-specific features.

To create user interfaces, programmers must use the API of the class `UIManager` of the UI module. Using this API, programmers can create an application, a main window, create a UI object inside a container (*i.e.*, a layout) or in relation to another one (*i.e.*, above/below or on the left/right of another UI object), remove a UI object or all children of a UI object, and find

³Without entering in the details of the implementation, the *Ruby* programming language proposes an hook method that catches each method call for which no method is defined in the class. Thus, by overriding this hook method, we can easily delegate this method call to the real GUI library. This behaviour can be easily implemented in other languages through metaprogramming.

a specific UI object. Figure 5.4 depicts this API with the different methods it proposes for all the actions mentioned previously.

UIManager
<code>create_app(proc)</code>
<code>create_main_container(app, title, args)</code>
<code>create_ui_object_in(id, type, container, args)</code>
<code>create_ui_object_above(id, above, type, args)</code>
<code>create_ui_object_below(id, below, type, args)</code>
<code>create_ui_object_left_of(id, left_of, type, args)</code>
<code>create_ui_object_right_of(id, right_of, type, args)</code>
<code>remove_ui_object(ui_object)</code>
<code>remove_all_ui_children(ui_object)</code>
<code>find_ui_object(ui_object)</code>

Figure 5.4: API of our UIA library.

Creating an application is a concept that comes from *FXRuby* to handle the events loop [Lyl08]. Programmers must first create the application. Then they must create a main window in the application to be able to attach UI objects to it later.

Listing 5.1 depicts how programmers can create the application. Line 1 illustrates how programmers must access the instance of the class `UIManager`. This class is a singleton class since we cannot have two UI managers in a same application. Lines 3 and 6 show how the programmers have to create and run their application through our API, with the methods `create_app` and `run`. In fact, the method `run` is a method not implemented in our UIA library. However, as explained in Section 5.2, this method will be called on the UI object itself, that delegates in turn to the *FXRuby* GUI library.

```
1 ui_manager = UI::UIManager.instance
2
3 app = ui_manager.create_app() do |app|
4   SmartMessagingSystem.new(app)
5 end
6 app.run()
```

Listing 5.1: Code snippet illustrating how programmers can create the application.

Listing 5.2 depicts how programmers can create a main window with a size of `800*600` and as title “Smart messaging application”. Line 1 depicts how programmers must create and attach a main window to the application. This

is done with the method `create_main_container` that takes the application (*i.e.*, the `app` variable), a title, and the arguments imposed by *FXRuby*. In this example, the arguments we provide to *FXRuby* is the title, width and height.

```
1 ui_manager.create_main_container(app, "Smart messaging
  ↪ application", :width => 800, :height => 600)
```

Listing 5.2: Code snippet illustrating how programmers can create a main UI window.

Next, let us illustrate step by step how programmers can build a user interface with our library. Listing 5.3 shows how programmers can implement the left part of the wireframe (*i.e.*, the chats list) of Figure 5.1.

```
1 ui_manager = UI::UIManager.instance
2
3 chats_list_layout =
  ↪ ui_manager.create_ui_object_in(:list_of_chats ,
  ↪ :FXVerticalFrame , main_window)
4
5 chat_ids = ["Parrain", "Kim", "Bruno", "INGI"]
6 chat_ids.each do |chat_id|
7   chat_button =
  ↪ ui_manager.create_ui_object_in(chat_id.to_sym,
  ↪ :FXButton, chats_list_layout, chat_id, :opts =>
  ↪ LAYOUT_FILL_X|JUSTIFY_LEFT, :padLeft => 10,
  ↪ :padRight => 10, :padTop => 20, :padBottom => 20)
8   chat_button.backColor = "#FFFFFF"
9
10  chat_button.connect(SEL_COMMAND) do
11    # Display the content of the message
12  end
13
14  separator_id = "#{chat_id}-sep".to_sym
15  ui_manager.create_ui_object_in(separator_id,
  ↪ :FXHorizontalSeparator, chats_list_layout)
16 end
```

Listing 5.3: Code snippet illustrating how programmers can implement the chats list on the left part of Figure 5.1.

To create a UI object, programmers have to call the `create_ui_object_in` method on the instance of `UIManager`. This method takes the following arguments: an identifier, the type of UI object that programmers want, the layout in which the object must be placed, and the arguments imposed by *FXRuby*. With these arguments, the method creates a UI object of the passed type and delegates the creation of the UI object to *FXRuby* that needs the different arguments to build correctly the object and put the UI object in the

passed layout. For example, programmers create a vertical layout (also known as vertical frame in *FXRuby*) in the main window on Line 3.

With Lines 7 and 15, programmers create a button and a horizontal separator for each chat identifier in *chat_ids*. In addition to providing its identifier, type and the layout in which it must appear, they must also provide the different arguments imposed by *FXRuby*, i.e., the text of the button, some options (*:opts*) and the padding (e.g., *:padLeft*, ...).⁴ Here, the options indicate that the button must fill the space on the x-axis (*LAYOUT_FILL_X*) and that its text must be placed to the left (*JUSTIFY_LEFT*).

To update some properties of a UI object, we have exactly the same API as proposed by *FXRuby*. For example, when programmers want to update the background color of a UI object, they must call the setter of the *backColor* property⁵ as suggested by the *FXRuby* library. (More information on the different methods available for the different objects are available in the documentation of the *FXRuby* library.)

Finally, programmers can also attach behaviour when interactions are added on buttons. Again, they can implement such interactions as described in the *FXRuby* documentation. An example is shown on Line 10. Programmers must attach an event *SEL_COMMAND* on the button with the method *connect* and provide the behaviour that must be executed when the button is clicked. (More information on the different interactions on the different UI objects can be found in the *FXRuby* documentation.)

Now that we have presented how programmers create UI objects, we will explain how they can create UI objects in relation to one another and what the consequences of such creations are. For that, revisit the example of our wireframe in Figure 5.1 and consider only how a message is displayed in a chat (see the first yellow box in Figure 5.1). In this wireframe, only the author and the message itself were displayed. We will now add the date of the message to the right of the author, as depicted in Figure 5.5.

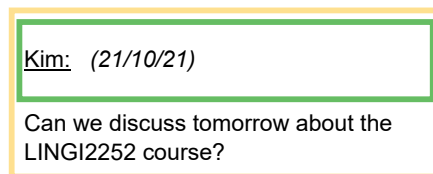


Figure 5.5: Wireframe illustrating how a message is displayed in a chat.

⁴More information on the arguments that take a button in *FXRuby* can be found on <https://larskanis.github.io/fxruby/Fox/FXButton.html>.

⁵In this case, the setter is the method *backColor=*. But in *Ruby*, we can call the setter as if we do a new assignment, as illustrated on Line 8.

Assume that programmers have already created a *chat_layout* to contain all the messages. On Line 2 of Listing 5.4, they define a vertical layout attached to *chat_layout* that contains the author and the content of the message with Lines 3 and 6, respectively.

```

1 ui_manager = UI::UIManager.instance
2 message_layout =
  ↳ ui_manager.create_ui_object_in(:message_layout_kim,
  ↳ :FXVerticalFrame, chat_layout)
3 author =
  ↳ ui_manager.create_ui_object_in(:message_author,
  ↳ :FXLabel, message_layout, "Kim:")
4
5 content = "Can we discuss tomorrow about the LINGI2252
  ↳ course?"
6 ui_manager.create_ui_object_in(:message_text, :FXLabel,
  ↳ message_layout, content)
7
8 ui_manager.create_ui_object_right_of(:message_date,
  ↳ author, :FXLabel, "(21/10/21)")
9 end

```

Listing 5.4: Code snippet illustrating how programmers can implement the view of a message by displaying the author, the date and the content of the message as depicted in Figure 5.5.

If programmers want to show the date of the message to the right of the author, they can do so by using the method *create_ui_object_right_of*, as shown on Line 8. This method takes as arguments an identifier for the new UI object, the related UI object (or the identifier of the related UI object), the type of the new UI object and the imposed arguments from *FXRuby*. The method creates the UI object and puts it on right of the related UI object (*i.e.*, the label “Kim:”). The usage of this method has the effect that our library generates a horizontal layout, puts both UI objects in the correct order in this new generated layout and puts the generated layout at the previous place of the related UI object. This result is represented in Figure 5.6.

To create a UI object on the left of another one, one may call the method *create_ui_object_left_of* that behaves as the method *create_ui_object_right_of*, except that the new UI object is placed on the left of the related one. Figure 5.7 illustrates the result of the wireframe, shown in Figure 5.7b, when programmers want to put the date sending to the left of the author on the initial wireframe as depicted in Figure 5.7a. Figure 5.7c represents the tree of this UI part after executing the method. The generated layout is shown with its dashed borders but no visual consequence on the real generated user interface. Similarly, when programmers want to create UI objects above or below another one, the new UI object is created above or below the related one in a gener-

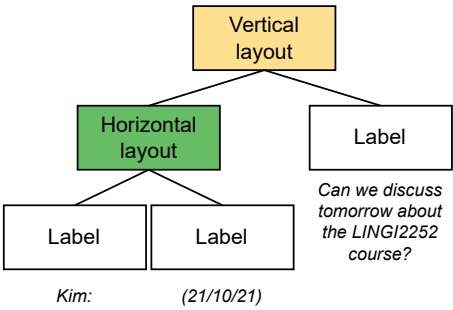


Figure 5.6: Tree representation of our second example illustrated in Figure 5.5.

ated vertical layout. Figure 5.8 illustrates the result of the wireframe, shown in Figure 5.8b, when programmers want to put the date sending above the author on the initial wireframe as depicted in Figure 5.8a. Figure 5.8c represents the tree of this UI part after executing the method.

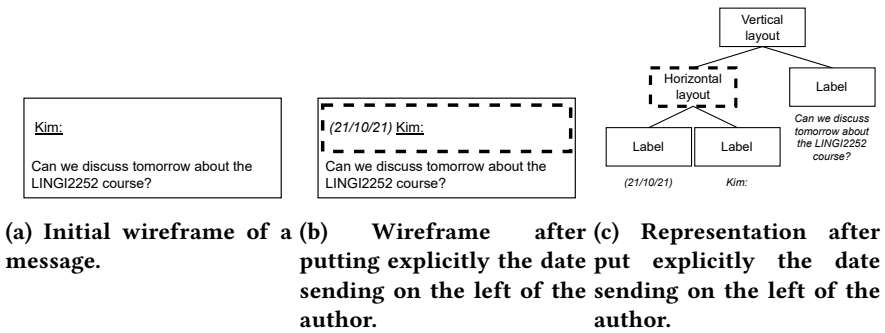


Figure 5.7: Illustration of the execution of the method `create_ui_object_left_of` where the date sending must be placed to the left of the author, and its tree representation after executed the method in the messaging system.

Now assume that programmers want to put another UI object to the right of the author, as for example, an emoticon to describe the current status of the profile, with another call to the method `create_ui_object_right_of`. Let us revisit the example of the wireframe in Figure 5.5. Since the author was already moved to a generated horizontal layout, the method will no longer create a new generated horizontal layout. Instead the method will insert the UI object to the right place, *i.e.* directly to the right of the author and before the date sending. Therefore its tree representation no longer contains two UI objects but three UI objects.

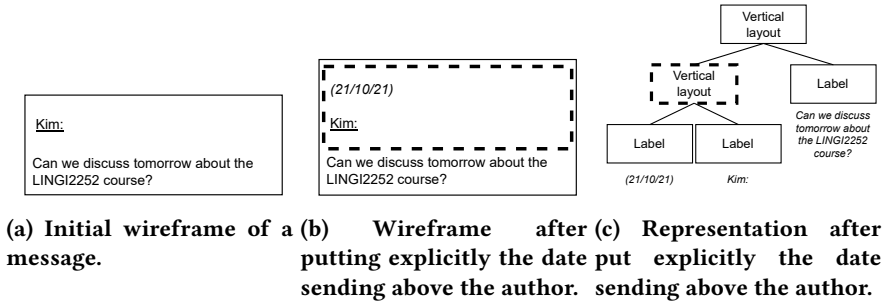


Figure 5.8: Illustration of the execution of the method `create_ui_object_above` where the date sending must be placed above the author, and its tree representation after executed the method in the messaging system.

To summarise this creation of a UI object in relation to another, the method will create a new corresponding layout at the first call and then the method will insert at the right place the new UI object if and only if the type of the layout is the same. For example, if the programmers want to put a UI object to the right of a UI object, the method call generates a horizontal layout in which both UI objects are correctly placed. Then, when they want to create another UI object on the left/right of any UI object of this generated layout, the UI object is added to this horizontal layout. But if the layout is another type of layout (e.g. vertical), the method will consider that the UI object is not yet attached to a generated corresponding layout. The method will thus generate the new corresponding layout as we have seen previously.

To remove a UI object or all the children of a UI object, programmers can call the methods `remove_ui_object` and `remove_all_ui_children`, respectively. These two methods take as argument a UI object. Nevertheless, a specific case appears in the method that removes a UI object when it is called: if the UI object was part of an auto-generated layout and its sibling remains alone, the auto-generated layout is also removed and the sibling is reattached to the parent of the auto-generated layout to take its place. Revisiting the wireframe of a message, depicted in Figure 5.5. When the sending date is removed, the generate layout contains only the author of the message. So the UI object representing the author is detached of the generated layout, this layout is also removed and the UI object is reattached to the previous parent layout. Figure 5.9 depicts the representation of this UI part after removing the UI object representing the date sending (and the generated layout). (As a reminder, the initial tree representation was shown in Figure 5.6)

Finally the method to find a UI object is the method `find_ui_object` that takes the identifier of a UI object.

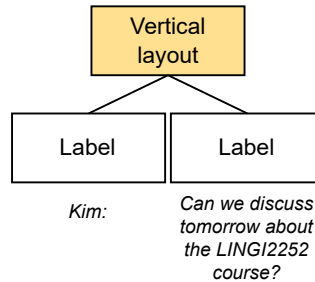


Figure 5.9: Tree representation after removing a UI object and its generated layout. The initial tree representation was shown in Figure 5.6.

5.4 UIA library and features

Now that we have exemplified the API of our UIA library, we will show how programmers can build their user interfaces through the feature definitions. Since each feature definition is on a specific concern, the user interface is split into different feature parts. The complexity thus lies in the fact that we must compose the user interface through dynamic binding and the *proceed* mechanism. In this section we provide an example of how a user interface can be built through the *proceed* mechanism.

To exemplify that, we revisit our example of how a message is incrementally composed in the chat layout, as illustrated in Figure 5.5 and represented as a tree in Figure 5.6.

Assume programmers have three features *Message*, *Text* and *Date*. Each of these features has a feature part *View* that can adapt and refine the application class `MessageView`. The feature *Message* is the default behaviour and creates the layout in which all the data of the chat messages will be displayed through UI objects. In addition, this feature also adds the author of the message⁶. The features *Text* and *Date* add the text content and the date of the message. Listings 5.5, 5.6 and 5.7 show the code snippets of the features *Message*, *Text* and *Date*, respectively.

```

1 module Message
2   module View
3     can_adapt : MessageView
4
5     attr_accessor : main_window
6
7   def create_message_widget(layout)
8     ui_manager = UI::UIManager.instance
  
```

⁶We should separate the author from the default behaviour to increase the modularity and separation of concerns. But we decided to group them to simplify the example.

```

9      message_layout =
    ↪ ui_manager.create_ui_object_in(:message_layout_kim,
    ↪ :FXVerticalFrame, layout)
10      # @message_model.from has the value "Kim:"
11      @author_label =
    ↪ ui_manager.create_ui_object_in(:message_author,
    ↪ :FXLabel, message_layout,
    ↪ "#{@message_model.from}:")
12      return message_layout
13  end
14  # More methods
15  end
16  # More feature parts
17 end

```

Listing 5.5: Code snippet of the feature part *View* of the feature *Message* adapting the application class *MessageView* in which a message layout is created with a label representing the author of the message in the messaging system.

```

1 module Text
2   module View
3     can_adapt :MessageView
4
5     def create_message_widget(layout)
6       message_layout = proceed(layout)
7       # @message_model.text has the value "Can we
    ↪ discuss tomorrow about the LINGI2252 course?"
8       ui_manager = UI::UIManager.instance
9       ui_manager.create_ui_object_in(:message_text,
    ↪ :FXLabel, message_layout, @message_model.text)
10      return message_layout
11    end
12    # More methods
13  end
14  # More feature parts
15 end

```

Listing 5.6: Code snippet of the feature part *View* of the feature *Text* adapting the application class *MessageView* in which the text is added to the message widget in the messaging system. Notice how a *proceed* call is used to generate the layout with the author of the message so that the text can be added to this layout.

```

1 module Date
2   module View
3     can_adapt :MessageView
4
5     def create_message_widget(layout)
6       message_layout = proceed(layout)

```

```

7      # @message_model.date has the value "21/10/21"
8      ui_manager = UI::UIManager.instance
9
10     ↪ ui_manager.create_ui_object_right_of(:message_date,
11     ↪ @author_label, :FXLabel,
12     ↪ "(#{@message_model.date})")
13     return message_layout
14 end
15 # More methods
16 end
17 # More feature parts
18 end

```

Listing 5.7: Code snippet of the feature part *View* of the feature *Date* adapting the application class *MessageView* in which the date of the message is added to right of the author of the message in the messaging system. Notice how a *proceed* call is used to generate the layout with the author and the content of the message so that the sending date can be added to the right of the author.

In this example, we assume the following activation order of these features: *Message*, *Text* and *Date*. In other words, the *MessageView* application class will be adapted first by *Message*, then by *Text* and finally *Date*. Whereas the first two features are mandatory, the feature *Date* is considered as optional because it could be hidden for small screens (e.g. smartphones).

Each *View* feature part implements the method *create_message_widget* that takes as argument the layout in which the message UI object must be part. As explained in Section 4.9, programmers can easily build in an incremental way the widget thanks to the power of the *proceed* mechanism. Figure 5.10 depicts the result of the incremental creation of this widget.

When the method *create_message_widget* is called, the version of the feature *Date* is first executed. When the *proceed* statement is encountered, the control is given to the version of this method of the feature *Text*. Finally the control is given to this method of the default behaviour (feature *Message*). Then a message layout is created by calling the method *create_ui_object_in* (Line 9 in Listing 5.5). The control flow continues by creating the label presenting the author of the message (Line 11 in Listing 5.5). The result of this execution is shown in Figure 5.10a. Once the default version of the method has finished executing, the control flow returns to the version of the method of the feature *Text* and creates the label that contains the text of the message (Line 9 in Listing 5.6). Its result is illustrated in Figure 5.10b. Finally the control flow returns to the version of the method of the feature *Date* and creates a label with the date of the message on right of the label of the author (Line 9 in Listing 5.7). Figure 5.10c shows the final result of this execution.

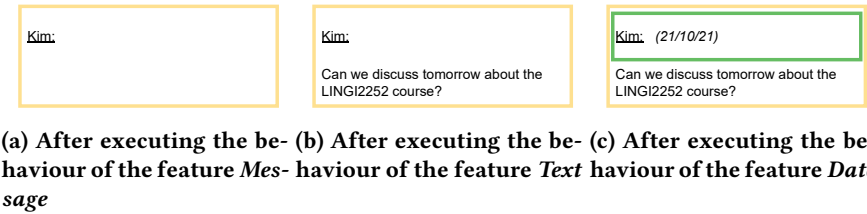


Figure 5.10: Results showing how the message widget is created incrementally by calling the method `create_message_widget` in our messaging system.

5.5 Conclusion

In this chapter, we first introduced how user interfaces can be represented internally to be easily manipulated. This representation is a tree-like structure where the root is the main window, the leaves are the widgets and the intermediate nodes are the different layouts that composes the user interfaces.

Then we provided an overview of how the components, *i.e.*, a FBCOP application are connected to the real GUI library (*FXRuby*) through our UIA library.

Next, we introduced the API of the UIA library by showing how programmers can use it to create UI objects in a layout, create UI objects in relation to others, as well as how they can remove and find UI objects.

Finally we exemplified how programmers can use the UIA library when they define their features dedicated to the user interface concern. With the combination of the proposed UIA library and the *proceed* mechanism, programmers are thus able to dynamically adapt the user interfaces (at runtime).

CHAPTER

6

VISUALISATION TOOLS

It comes as no surprise that designing and implementing FBCOP applications remains quite complex due to their highly dynamic nature and the exponential number of combinations we can have between the contexts and features they adapt. To help programmers achieve this complex task we created two separate visualisation tools: the `CONTEXT AND FEATURE MODEL VISUALISER` [Duh+19b] and the `FEATURE VISUALISER` [DMD18]. The former aims to provide a global overview of the system by visualising the context model, feature model, the system's application classes and their interdependencies. The second tool helps developers to inspect and visualise the complete process starting from context activation, via feature selection and activation to the actual deployment of those features in the system's application classes.

In this chapter we describe and illustrate both visualisation tools. For each tool, we detail its visualisation, present a meta-model, and explain its different functionalities through snapshots of the tool running on our simplified messaging system. Then we briefly explain why these visualisation tools can be considered as complementary from a programmers' point of view, when developing and debugging FBCOP applications.

6.1 `CONTEXT AND FEATURE MODEL VISUALISER`

Keeping track of all possible contexts, features and their intra- and inter-dependencies remains a daunting task for developers of FBCOP applications.

It is not easy for programmers to know what contexts or features are available, are currently active, what are the consequences of activating or deactivating them, or whether the system exhibits the intended behaviour in a particular situation.

We therefore developed the CONTEXT AND FEATURE MODEL VISUALISER tool [Duh+19b]. This visualisation tool was implemented by a master-level student, Hoo Sing Leung, during his master thesis [Leu19] we supervised. This visualisation tool can help developers to keep an overview of all existing contexts and features, by displaying the context and feature models and their dependencies. This tool allows a developer to examine how a hierarchical model (the context model) can statically or dynamically interact with another hierarchical model (the feature model). This visualisation also exposes which features adapt which application classes of the system. Figure 6.1 shows a snapshot of our CONTEXT AND FEATURE MODEL VISUALISER at work. To keep the picture readable, we deliberately hid some information like for example all inactive features, all non-impacted application classes, some of the impacted application classes and all mapping relations from the inactive contexts and features.

6.1.1 Visualisation

Figure 6.2 explains the structure of the CONTEXT AND FEATURE MODEL VISUALISER through its meta-model which emphasises the different entities (*i.e.*, contexts, features and application classes) involved in this tool. It consists of a context model containing nodes representing *contexts* and edges representing *constraints* between these contexts, as well as a feature model containing nodes representing *features* and edges representing *constraints* between these features. It also contains a *mapping* describing which *contexts* trigger which *features* and a mapping indicating which features *adapt* which *application classes* of the system. The tool will display only those *application classes* for which at least one *feature* exists that adapts that application class. Finally the visualisation indicates which *contexts* and *features* are *active* (*i.e.*, selected) or not in the running system, as well as which *application classes* are currently *adapted* (*i.e.*, refined) by a feature or not. Whereas the visualisation is mostly static, showing this activation and adaptation status adds a dynamic element to the visualisation.

6.1.2 Functionalities

Now that we have briefly introduced the CONTEXT AND FEATURE MODEL VISUALISER, we highlight the different design choices we took when creating this tool, corresponding to different usage scenarios focussed on programmers building a context-oriented system using FBCOP. All these functional-

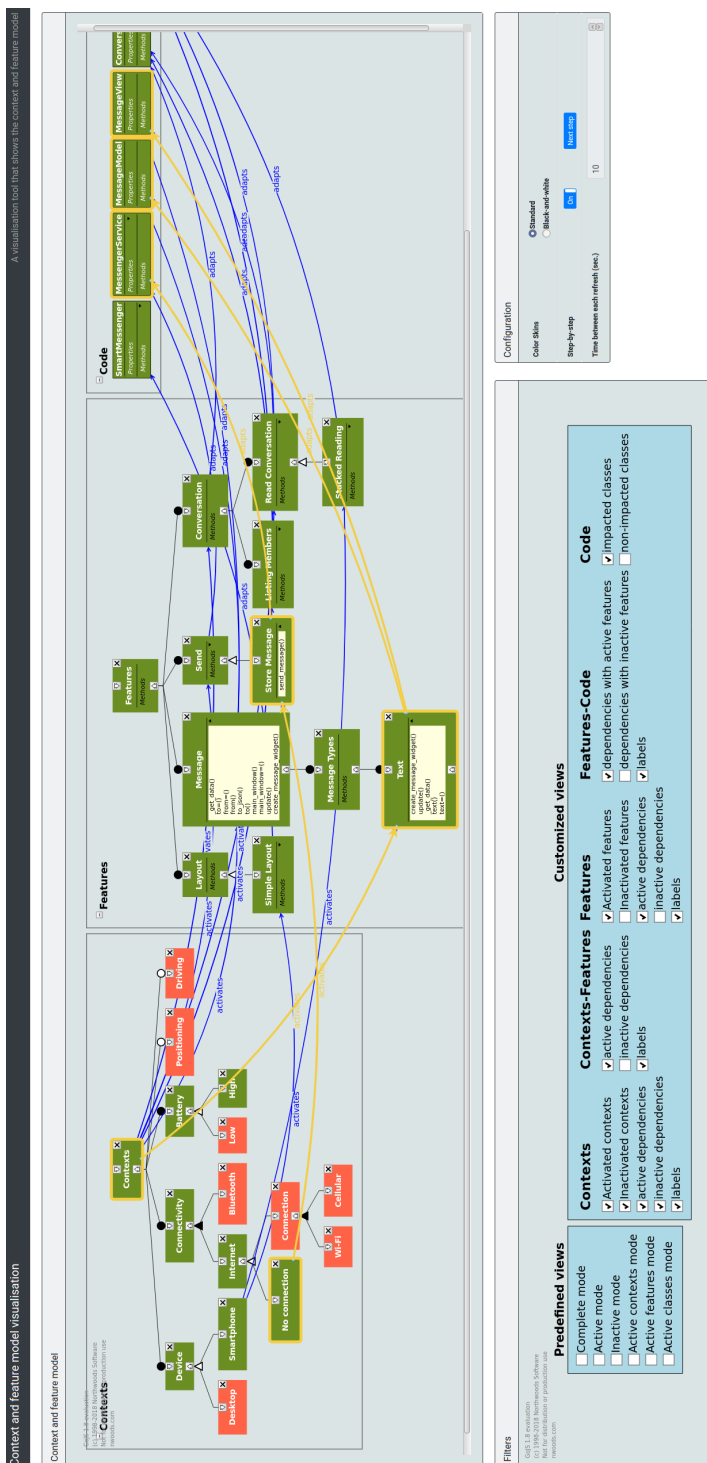


Figure 6.1: Snapshot of the CONTEXT AND FEATURE MODEL VISUALISER tool.

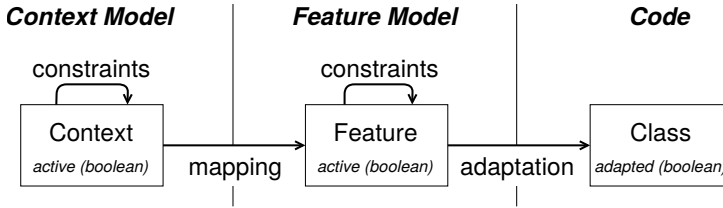


Figure 6.2: Meta-model of the CONTEXT AND FEATURE MODEL VISUALISER tool. The nodes represent the system entities visualised by the tool (i.e., contexts, features and application classes). The edges define constraints between contexts, constraints between features, a mapping from contexts to features, and which features adapt which application classes.

ities are also illustrated in a demonstration video with a simpler version of our messaging system¹.

Visualising statically the context and feature models

A first important usage scenario for a programmer is to get a global overview of the system, in terms of the different contexts, features and application classes of which it is composed. Figure 6.1 shows what such a visualisation looks like in the tool’s *Context and feature model* widget. This static snapshot can show all contexts, features and application classes of interest, regardless of whether they are currently active or not: the context model shown in Figure 6.1 contains both active contexts (colored in green) and inactive ones (colored in red). In this example, we can observe that the contexts *Smartphone* and *No Connection* are active in the context-oriented application while the contexts *Desktop* and *Cellular* are not active in the application.

The idea of including application classes in the visualisation as well is inspired by the *FEATURE VISUALISER*, presented in Section 6.2. Furthermore, our tool allows programmers to inspect in more detail the actual behaviour of features and application classes, as illustrated by yellow boxes inside some features (e.g., *Message* and *Text*) in Figure 6.1. An example of detailing the actual behaviour of the feature *Text* is shown in Figure 6.3. In its behaviour we can see some methods for the user interface concern such as for example the methods *create_message_widget* and *update* and some methods for the core logic concern such as the methods *text* and *text=*.

Exploring the dynamics of a context-oriented system

In addition to providing a static overview of a context-oriented system, the tool can support programmers in understanding and exploring the dynamic

¹Available at <https://www.youtube.com/watch?v=6XUrEkuvyA>.

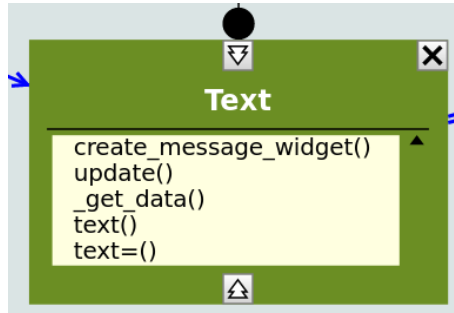


Figure 6.3: Snapshot of the feature *Text* for which we detail its actual behaviour.

aspects of such a system. More specifically the tool can help them inspect what contexts and features are currently active and how that affects the behaviour, in terms of what application classes are currently being adapted. For example, suppose that during testing and simulation of the system programmers discover that, when the device has *No Connection*, the system can still send and receive messages from and to other people. To understand such undesired behaviour they need to explore what contexts are currently active, what features were triggered in response to that, and how the application classes were then adapted by those features. A possible cause of this bug may be for example a wrong mapping relation between the context *No Connection* and its corresponding features.

The visualisation tool provides several ways of exploring the system dynamics. A first one, which was already mentioned above, is the use of colouring to show active contexts, features and classes in green. A second one, which will be explained next, is to use particular filters to show only activated contexts, features, relations, and currently adapted application classes. The final and probably most powerful functionality provided by the tool is to show changes to the diagrams as they occur. To explore these dynamic changes, the tool provides the ability to replay the changes step by step (by activating the *Step-by-step* mode and using the *Next step* button in the *Configuration* widget), so that programmers can inspect the state of the diagrams after each change. Alternatively, the programmers can configure the tool to update the diagrams automatically each x seconds (where a value of $x = 0$ would correspond to a *live* visualisation of the changes as they occur, whereas a value of $x > 0$ would yield a *delayed* visualisation). These settings can be changed easily in the tool's *Configuration* widget, depicted in Figure 6.4, which is located in the bottom right of the tool shown in Figure 6.1.

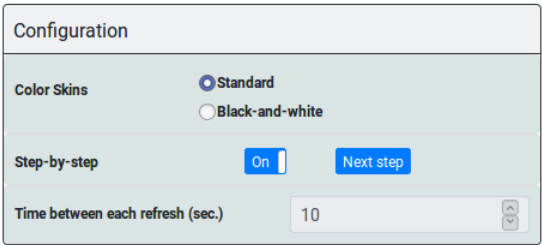


Figure 6.4: Snapshot of the *Configuration* widget of our **CONTEXT AND FEATURE MODEL VISUALISER**.

Filtering and predefined views

To help programmers manage the complexity of understanding large systems consisting of many different contexts and features, the tool also comes with a set of filters and predefined views that programmers can select to focus on particular concerns, as depicted in Figure 6.5. This widget is in the bottom left of the tool as shown in Figure 6.1.

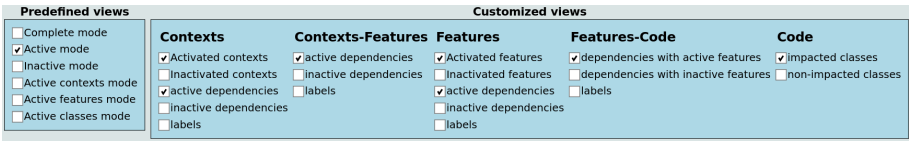


Figure 6.5: Snapshot of the *Filters* widget of our **CONTEXT AND FEATURE MODEL VISUALISER**.

These filters (called *Customized views* in the widget) allow programmers to indicate whether they are more interested in the contexts, the features, the application classes, or the dependencies (relations) between them, and whether they are currently more interested in exploring the active or inactive entities or dependencies. The filters can be combined in many different ways. In addition to that a few predefined views are provided, which are predefined combinations of filters that are often selected together. For example the *Complete mode* selects all filters and shows all possible entities and dependencies, whether they are active or not. Picking the *Active mode* shows all entities and dependencies that are currently *active*, and Figure 6.6 illustrates what the effect of applying this view is. Not surprisingly, after choosing this view only green (meaning *active*) contexts, features and application classes remain.

Highlighting specific elements of interest

As the number of possible contexts, features, application classes and dependencies can become quite large, in addition to filtering the diagrams to only

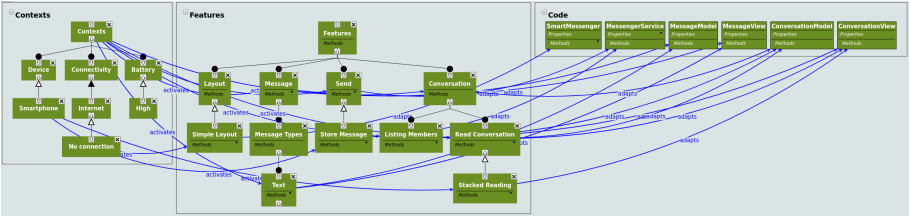


Figure 6.6: Snapshot of the *Context and feature model* widget in which we apply the *active mode* of the predefined views of our **CONTEXT AND FEATURE MODEL VISUALISER** to visualise only all the active contexts, features, application classes and the active dependencies.

show certain elements of interest, *highlighting* is another interesting way to help programmers navigate through the diagrams, letting them trace the behaviour of a particular feature.

Suppose for example that programmers are trying to understand why a particular feature, say the feature *Store Message*, does not seem to exhibit the expected behaviour. By simply clicking on that feature, it will be highlighted in yellow, together with the contexts that triggered its selection (by following the dependencies that have this feature as target) and the application classes it adapts (by following the dependencies that have this feature as source). In this example, the contexts *No Connection* will be highlighted, as well as the application class *MessengerService*. This highlighting is illustrated by the yellow borders and yellow arrows in Figure 6.7.

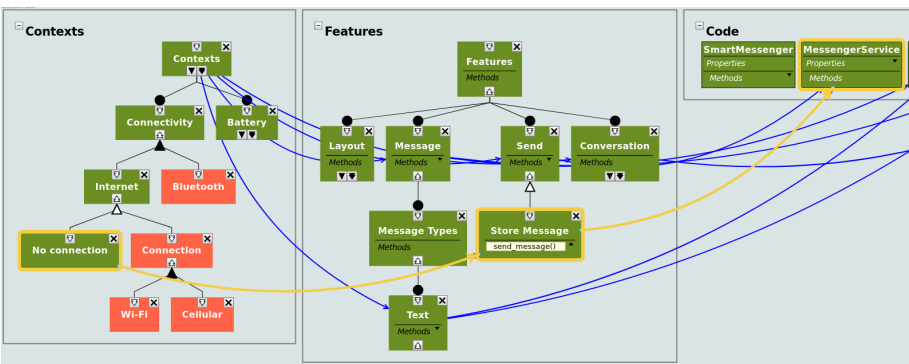


Figure 6.7: Snapshot of the *highlighting* functionality of our **CONTEXT AND FEATURE MODEL VISUALISER**. (For readability reasons, some non-relevant contexts, features, application classes and dependencies for this figure have been hidden.)

Hiding and collapsing information

Finally, programmers can also customise their visualisation at an even more fine-grained level through the actions of *hiding* and *collapsing* particular elements. For example, Figure 6.8a depicts a relatively dense feature model with many subtrees, some of which go four levels deep. But perhaps the programmers are currently not interested in one of these subtrees, in which case they can simply hide it by collapsing the children into its root node with the button at the bottom of that node and then clicking its X-button on the top-right corner. Similarly, if they are not interested in any of the subfeatures of a given feature they can collapse all nodes below it, by clicking on the collapse button at the bottom of that feature. Similar buttons exist to collapse all nodes above a certain node. Figure 6.8b shows a reduced version of Figure 6.8a obtained by just hiding and collapsing some features.

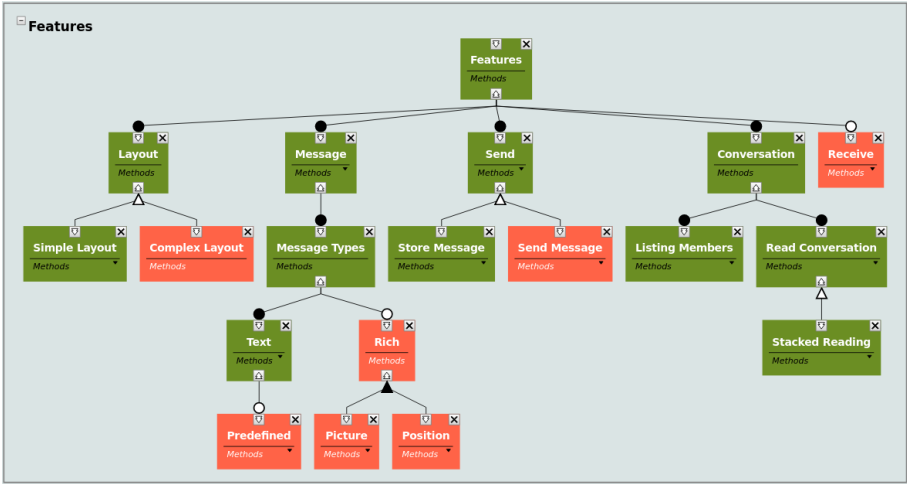
6.2 FEATURE VISUALISER

Whereas the previously described visualisation tool helps developers get a global overview of the context and feature model and its mapping, as well as which features adapt which application classes, some key information is still missing. It does not show in what order the different contexts and features are activated and in what order the different features are deployed. It also misses detail on which parts of which features adapt which application classes of the system. The FEATURE VISUALISER tool [DMD18] attempts to address these issues. A snapshot of the FEATURE VISUALISER applied to the messaging system is shown in Figure 6.9.

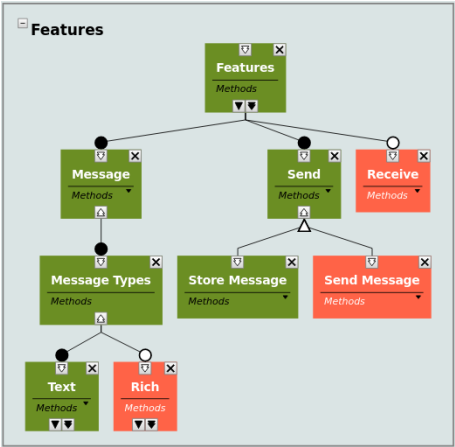
6.2.1 Visualisation

Before sketching the functionalities of the FEATURE VISUALISER, we first describe its visualisation with its meta-model. Figure 6.10 shows the meta-model of the different entities (contexts, features parts and application classes) involved in this tool and in what order they would appear in the visualisation. As opposed to the previous one, this meta-model emphasises the more dynamic aspects of the running system, as the purpose of this alternative visualisation is to show how the context-oriented system dynamically evolves during its execution. The nodes of this model represent the different entities that get dynamically activated, selected and adapted by the system: contexts, feature parts and application classes.

This visualisation shows feature parts instead of features since a single feature can be subdivided in different parts that can each adapt different application classes to get a better separation of concern as explained in Section 4.3, contrary to the previous visualisation.



(a) A feature model



(b) Simplified feature model obtained by hiding and collapsing some features

Figure 6.8: Tool support for hiding and collapsing nodes in a diagram.

The edges essentially correspond to the different phases of the FBCOP system architecture described in Figure 3.4 in Section 3.2. Let us explain the meta-model of Figure 6.10 in more detail based on the control flow of that system architecture. When the system senses changes in the surrounding environment, it reifies the newly sensed contextual information into context objects that it tries to (de)activate, which are then visualised in the tool (deactivated contexts are greyed out first, before they are removed from the visualisation as they are no longer active). The system then (un)selects the features that these contexts trigger and relays this information to the visualisation tool to display the feature parts of these features. This selection then triggers the

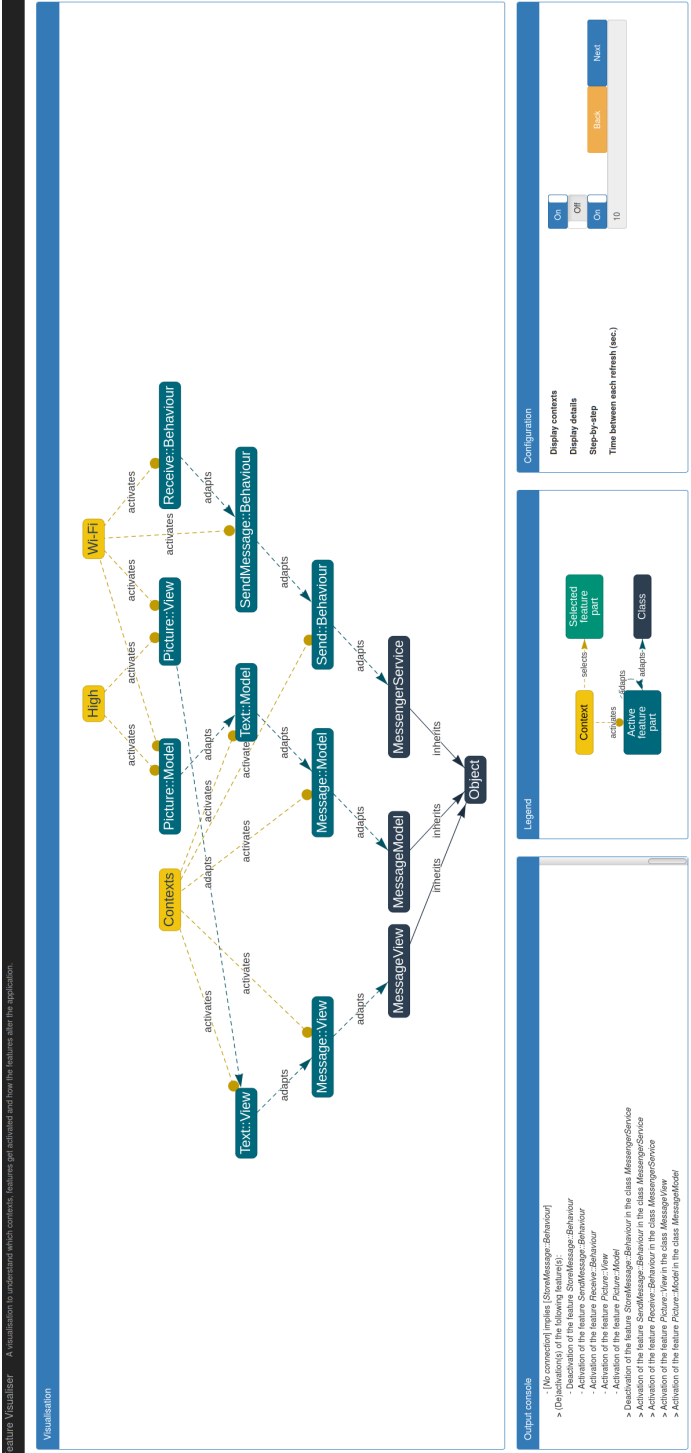


Figure 6.9: Snapshot of the FEATURE VISUALISER tool.

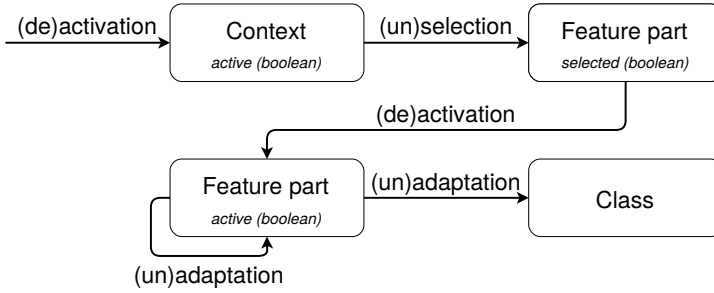


Figure 6.10: Meta-model of the visualisation of the FEATURE VISUALISER tool. The tool visualises how the FBCOP system architecture dynamically activates, selects and adapts the different system contexts, feature parts and application classes during the system’s execution.

(de)activation of those feature parts. When a feature gets (de)activated, each of its parts get (de)activated and visualised in the tool. Finally the system deploys the code of the activated feature parts in the application classes they adapt, and asks the tool to visualise this as well.

6.2.2 Functionalities

Now that the visualisation has been explained, we present the functionalities of this tool. With such functionalities, programmers can visually inspect in detail which active contexts trigger which active features parts (and by definition which features), how the active feature parts adapt the application classes of the system and in which order these feature parts are installed. All of these are also illustrated in a demonstration video with a simpler version of our messaging system².

Inspecting visually the dynamic aspect of the system

The main functionality of this tool is to inspect dynamically the system at runtime. This can be done through the main widget: the *Visualisation* widget. Figure 6.11 sketches a visualisation of our running messaging system illustrating how programmers can inspect how the context-oriented messaging system has dynamically evolved depending on the current surrounding environment.

To start with, we can observe that by default (represented by the context named *Contexts*), the system provides behaviour to send text messages, as implemented among others by the feature parts *Send::Behaviour* that adapts the application class *MessengerService*, *Message::Model* and *Text::Model*

²Available at <https://www.youtube.com/watch?v=JuJc1f2Pmzk>.

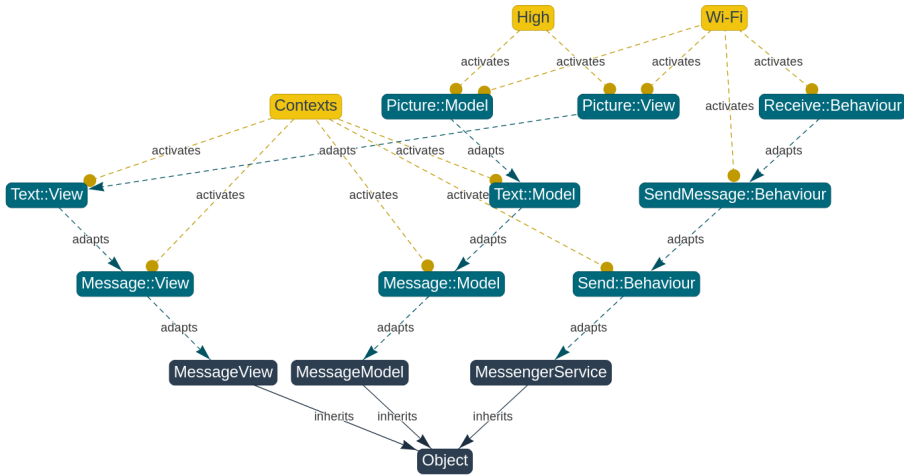


Figure 6.11: Snapshot of the *Visualisation* widget of our **FEATURE VISUALISER**.

that adapt `MessageModel` and, `Message::View` and `Text::View` which adapt `MessageView`. When a Wi-Fi connection is detected, the context `Wi-Fi` appears which activates both the feature parts `SendMessage::Behaviour` and `Receive::Behaviour` since having a Wi-Fi connection means that the system will allow users to both send and receive text messages. If the context `High` (representing a high battery level) is active as well, the system will allow users to send and receive pictures too (`Picture::Model` and `Picture::View`). The developers can also observe that `SendMessage::Behaviour` and `Receive::Behaviour` feature parts adapt the default behaviour implemented by `Send::Behaviour`. Similarly, they can observe that `Picture::Model` and `Picture::View` adapt the corresponding `Text::Model` and `Text::View` parts to allow for pictures to be included in the text messages.

To help the programmers in their inspection, the *Legend* widget has been added as a mental reminder of what the different coloured nodes and edges in the picture represent. Figure 6.12 shows the legend that this visualisation uses and this widget is displayed in the bottom-middle of the tool as depicted in Figure 6.9.

As for the previous visualisation tool, the developers can also configure how they can explore the dynamic evolution of the application depending on the contexts. The visualisation can be refreshed either manually by activating the *Step-by-step* mode and then by navigating through the steps with the *Next* and *Back* buttons or automatically each x seconds. They can set the visualisation refresh in the *Configuration* widget, depicted in Figure 6.13, in the bottom-right of the tool as shown in Figure 6.9.

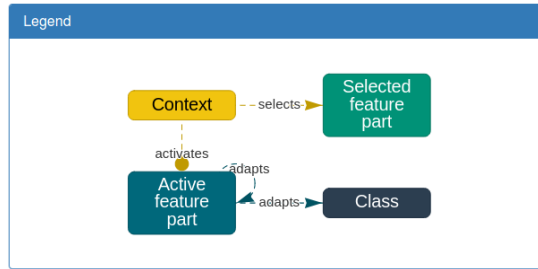


Figure 6.12: Snapshot of the *Legend* widget of our FEATURE VISUALISER.

Tracing the dynamic evolution

Since this visualisation constantly changes when contexts and features get (de)activated, it is sometimes difficult for the developers to keep track of all these (de)activations. The *Output console* widget can help them by providing a textual trace of all these activations and deactivations. This widget is on the bottom-left of the tool as depicted in Figure 6.9.

A textual version of the *Visualisation* widget provides the benefit of keeping a chronological trace of what contexts and features were activated or deactivated before. This can provide valuable information to the developer when debugging, for example to understand in what order certain features were added or removed and upon what contexts. Reconsider the example of the visualisation we have illustrated in Figure 6.11 when a *Wi-Fi* connection is sensed. The textual version of Figure 6.11 is illustrated in Listing 6.1. In this example, we can observe that before a *Wi-Fi* connection has been sensed, the contexts *No Connection* and *High* were activated.

```

1 > (De)activation(s) of the following context(s):
2   - Activation of the context Wi-Fi
3 > (De)activation(s) of the following context(s):
4   - Deactivation of the context No Connection
5 > Selection of the following rules (contexts implies
   features):
6   - [Wi-Fi] implies [SendMessage::Behaviour, Receive::Behaviour]
7   - [Wi-Fi, High] implies [Picture::View, Picture::Model]
8 > Unselection of the following rule (contexts implies
   features):
9   - [No connection] implies [StoreMessage::Behaviour]
10 > (De)activation(s) of the following feature(s):
11   - Deactivation of the feature StoreMessage::Behaviour
12   - Activation of the feature SendMessage::Behaviour
13   - Activation of the feature Receive::Behaviour
14   - Activation of the feature Picture::View
15   - Activation of the feature Picture::Model
  
```

```

16 > Deactivation of the feature StoreMessage::Behaviour in the
    class MessengerService
17 > Activation of the feature SendMessage::Behaviour in the
    class MessengerService
18 > Activation of the feature Receive::Behaviour in the class
    MessengerService
19 > Activation of the feature Picture::View in the class
    MessageView
20 > Activation of the feature Picture::Model in the class
    MessageModel

```

Listing 6.1: Snippet of the trace of Figure 6.11 of our FEATURE VISUALISER.

Customising the visualisation

Finally, the *Configuration* widget, depicted in Figure 6.13, allows the developers to also customise the visualisation tool to his needs.

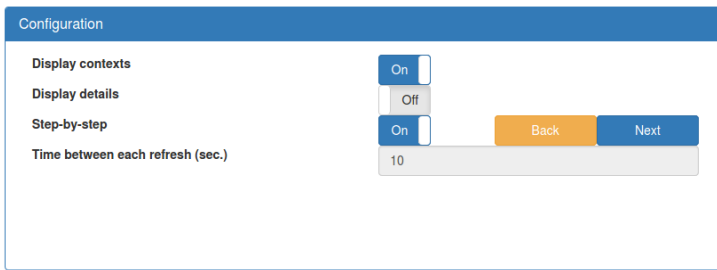


Figure 6.13: Snapshot of the *Configuration* widget of our FEATURE VISUALISER.

In this configuration, the programmers can decide whether to show the contexts or not (if not, only the features and application classes will be shown), or to show more detail on the behaviour (*i.e.*, the methods) of the different feature parts and application classes of the system. Figure 6.14 depicts an example of an excerpt of the visualisation where the contexts are hidden and the behaviour of the feature parts and application classes are detailed. In this example, we can see how the application class *MessageView* is adapted by the feature parts *Message::View*, *Text::View* and *Picture::View* in that order. Considering the method *create_message_widget*, we can see that the feature part *Message::View* has first adapted the behaviour of the application class *MessageView*, then this version has been refined by the feature *Text::View* to finally be adapted a last time by the feature part *Picture::View*. Therefore, the application class *MessageView* contains in its behaviour the latest installed version of the method *create_message_widget*, *i.e.*, the version of the feature part *Picture::View*. For each method the tool precises from which latest

installed feature part the method is, as shown in Figure 6.14. This also illustrates in which order the *proceed* mechanism if existing must be executed.

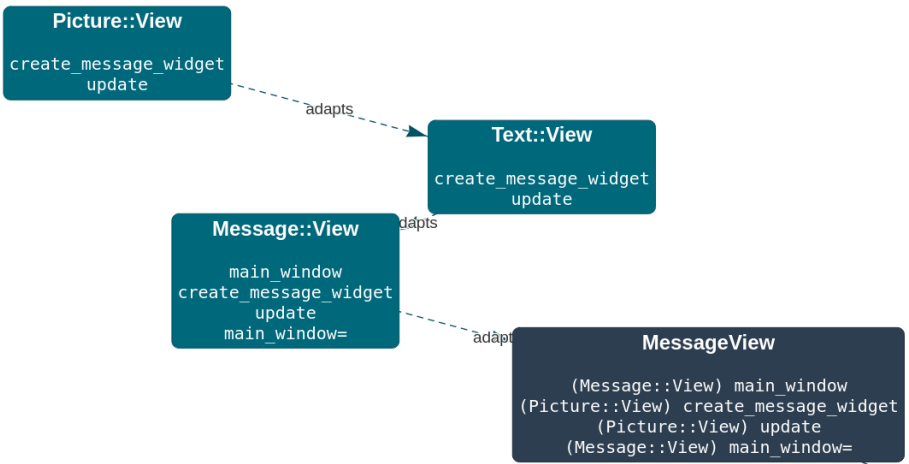


Figure 6.14: Snapshot of an excerpt of the *Visualisation* widget of our **FEATURE VISUALISER** for which we have hidden the contexts and detailed the behaviour of the feature parts and application classes.

6.3 Conclusion

In this chapter we described two visualisation tools: the **CONTEXT AND FEATURE MODEL VISUALISER** and the **FEATURE VISUALISER**.

With the **CONTEXT AND FEATURE MODEL VISUALISER** tool programmers can visually analyse the different models of their **FB COP** application, *i.e.*, the context model, the feature model, the mapping from the context model to the feature model, as well as the application classes and by which features they are adapted. Such a graphical representation allows them to quickly observe if they declare correctly the models of their application. In addition to this static visualisation, developers can also observe how the models evolve dynamically during the execution of their **FB COP** application. This dynamic aspect can be useful for developers to better understand why a context or feature cannot be (de)activated due to a constraint of their model that is not ensured. This tool also contains some functionalities to facilitate the programmers' exploration of the models. For example they can filter or hide and collapse some less relevant aspects of the models to focus on the more interesting ones. Finally they can also highlight an entity to better see its sources and targets.

The **FEATURE VISUALISER** tool aims to help programmers to inspect in more detail their **FB COP** application since it displays the feature parts and not the features as for the first visualisation tool. In this visualisation, they can

see what contexts are activated, what feature parts are triggered by these contexts, what application classes they adapt and in which order the feature parts are installed in the different application classes. In addition to this graphical visualisation, a textual version keeps a history of all the (de)activations of the different entities. This tool also helps to spot some errors in the order of the (de)activations of the feature parts.

To conclude this chapter, even though we have introduced these two visualisation tools separately, they can be considered as complementary for the programmers when they implement or debug their context-oriented application. Indeed, while the *CONTEXT AND FEATURE MODEL VISUALISER* proposes a more static visualisation that focuses on the structure of the underlying feature diagrams used by the context and feature model, the *FEATURE VISUALISER* is more dynamic and more fine-grained and focusses on the activation dynamics of the system. With both tools at hand developers using the FBCOP programming framework could switch between both visualisations to examine in more detail the particular concerns of the system they are interested in analysing.

Part III

Implementation

The previous part of this dissertation described the overall FBCOP approach. After having introduced its underlying key notions, we described its system architecture and a supporting development methodology to guide designers and programmers when they conceive FBCOP applications. Then we explained and exemplified how programmers can develop such applications with the FBCOP programming framework, including how they can build their user interfaces such that they are also dynamically adaptive. Finally, to help them in their development and debugging tasks, we introduced two visualisation tools to explore and inspect how their applications dynamically evolve at runtime.

The focus of this third important part of this dissertation will be on how we implemented the FBCOP programming framework. We explain how we implemented the building blocks of our programming framework (Chapter 7), how we integrated the modelling of contexts and features from an implementation point of view (Chapter 8) and how the control flow was implemented (Chapter 9). We also describe how we extended our programming framework so that we can easily develop external tools to support programmers in their various tasks when conceiving their FBCOP applications (Chapter 10).

We implemented our overall programming framework within the *Ruby* programming language. Nevertheless, the choice of *Ruby* is not essential even though we did rely on some of its powerful metaprogramming capabilities. This *Ruby* implementation mainly serves as a proof of concept that demonstrates how such a framework can be implemented on top of any sufficient powerful object-oriented programming language.

CHAPTER

7

ENTITIES

The key notions in our FBCOP approach are contexts, features and a context-feature mapping. They capture the main kind of entities that programmers need to declare and define: the contexts to which the application can adapt, the features describing the application behaviour specific to certain contexts, and the mapping declaring what specific behaviour is triggered by what particular contexts.

In this chapter we will explain the design choices we made to implement each of these kinds of entities. To better visualise our design choices, Figure 7.1 shows a class diagram of the ENTITIES part of our implementation architecture.

7.1 Context and feature entities

As explained in Section 4.2, since contexts and features are different concepts, they have their own classes in the implementation framework. While concrete contexts are instances of the framework class `Context`, concrete features are instances of the framework class `Feature`.

In addition to these classes, the framework also implements the classes `AbstractContext` and `AbstractFeature`, where the former is a generalisation of `Context` and the latter is a generalisation of `Feature`. These framework classes allow programmers to differentiate between concrete and abstract contexts and features when they declare their context and feature

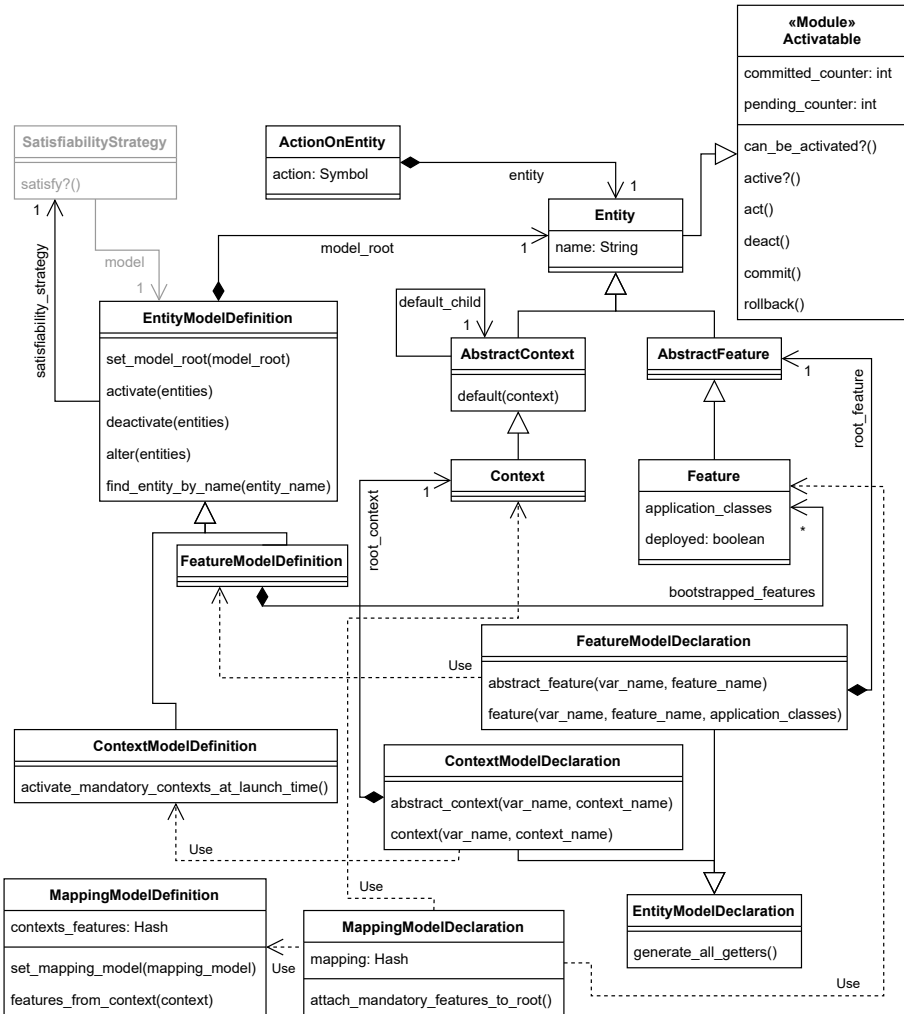


Figure 7.1: Class diagram of the ENTITIES part of our framework implementation. Some information (constructors, getters and setters) is voluntarily omitted for readability reasons. As the framework class **SatisfiabilityStrategy** is not part of ENTITIES we greyed its box and the arrow from this framework class. However we added it in this class diagram to better understand our explanation about the (de)activation of entities.

models. As we ask programmers to define a default situation in the surrounding environment to initialise a default behaviour in the application (see the explanation in Section 4.4), we add a reference to a *default child* context in the framework class **AbstractContext**. This reference tells us which child is the default one among the children. Such a reference is not relevant for the features because the default behaviour is automatically triggered by the

default contexts.

Despite their conceptual difference, we consider (concrete or abstract) contexts and features as a special kind of entity (*i.e.*, an instance of the framework class `Entity`), as illustrated in Figure 7.1. This framework class captures the fact that both contexts and features are simply entities among which similar kinds of constraints and dependencies can be declared to create a context and a feature model, respectively, as we will see in Chapter 8. In addition, as each context or feature is identified by a name, we add an instance variable *name* to their parent class `Entity`.

As developers must provide a list of each application class that a concrete feature adapts, we add an instance variable to the features (instances of the framework class `Feature`) to describe which *application_classes* they adapt. This variable represents a 1-*N* mapping from each feature to the application classes it adapts. (Note that it is allowed for a same application class to be adapted by different features.)

Inspired by the work of Cardozo et al. [Car+15], we also implement activation counters for any kind of entity. This is useful to recall how many times an entity is activated. For that we implement a trait `Activatable` that is included in the framework class `Entity`. This trait has two instance variables: *committed_counter* and *pending_counter*. The committed counters represent the number of times an entity is really activated in the system. The pending counters are auxiliary counters used when we try to (de)activate the entities in the model. With these pending counters we can ensure, without breaking the current configuration of the model, that all the constraints of the entity model are satisfied when the system attempt to modify it. The usage of such a pending status is also inspired by the work of Cardozo et al. [Car+15], where they use such status to verify the consistency of their context models before activating them. This `Activatable` trait also provides behaviour to query and update the real state of an entity with the methods *active?()* and *commit()*, respectively. For the pending status of an entity, the methods *act()*, *deact()* and *rollback()* modify this pending status. The method *can_be_activated?()* returns if the entity can be activated based on its current value of the pending counter.

Maintaining accurate counters for each entity guarantees that some features are not inadvertently removed from the system behaviour. Consider an excerpt of the design of our messaging system, depicted in Figure 7.2. In this example, we can see that the activation of the context *Wi-Fi* triggers the activation of the features *Send Message* and *Receive*. But these same features can also be activated when the context *Cellular* is activated.

Assume that both these contexts are activated. Users can send and receive messages over the Internet. As the system keeps a trace on how many times the features are present in the system, these features are activated twice, once because they were triggered by the context *Wi-Fi* and once because they are

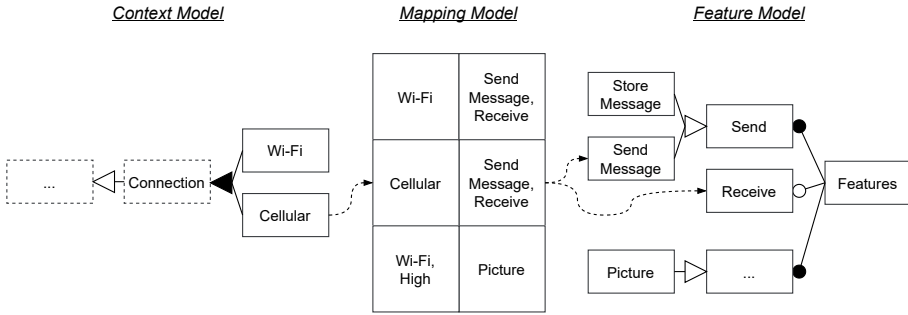


Figure 7.2: Excerpt of the context and feature model with their mapping from our messaging system. Some arrows are voluntarily omitted for clarity reasons.

triggered by the *Cellular* context. (Note that even if the features are activated many times, they are only installed once in the behaviour of the application. The reason of this design choice is explained later in this section.) Imagine that at some point, the *Wi-Fi* connection is no longer sensed. This implies that the *Wi-Fi* context will get deactivated. Without the counters of the entities, we would then have to deactivate the features *Send Message* and *Receive* since the context that triggered these features got deactivated. However this would give rise to an unexpected behaviour since *Cellular* is still active and thus we expect to still be able to send and receive messages. The purpose of the counters is to avoid such unexpected behaviour. Rather than deactivating the features immediately, their counter decremented by one. Only when their counter reaches zero will they actually be deactivated. A more detailed explanation on how the (de)activation of entities works and how the counters are managed will be given in Section 7.3.

Since entities can be activate multiple times, and features in particular, we need to analyse what the impact of that will be. If a same feature gets activated multiple times, could that inadvertently cause incorrect or wrong behaviour? As it turns out the answer to that question is affirmative. A problematic issue is illustrated in the next paragraph.

Consider an excerpt of the feature model of our messaging system dedicated to the feature *Message* with all the *MessageTypes* a message can be, as depicted in Figure 7.3. At launch time of the system, the system activates and deploys the mandatory default behaviour, *i.e.*, the features *Message* and *Text*. The feature *Message* provides the author and receivers of the message.¹ The feature *Text* adds a text to the message. As *Text* is a refinement of *Message*, some methods of *Text* (*e.g.*, the constructor and the method to display

¹We merged the author and receivers into the feature *Message* for conciseness reasons, but should split to increase the modularity in real application.

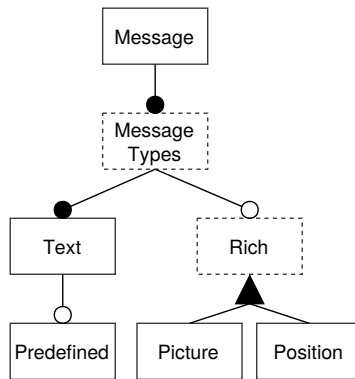


Figure 7.3: Excerpt of the feature model of our messaging system.

the information) uses the *proceed* mechanism to refine the default behaviour of these methods in *Message*. This means that the corresponding application class is first adapted by the feature *Message* and then by the feature *Text*. So when a call is executed on a method of the feature *Text*, this feature delegates at some point a part of its behaviour to the same method of the feature *Message*. After the execution of the methods in *Message* is finished, the control flow returns to the version of the method *Text*. Now, assume that users use their devices with a *High* level of battery and are connected to a *Wi-Fi* connection. In such a particular situation, the users can also add a picture to their messages or receive pictures as described in the excerpt of the mapping model shown in Table 7.1 and visually depicted in Figure 7.2².

Contexts	Features
Wi-Fi, High	Picture

Table 7.1: Excerpt of the mapping model of our messaging system.

Therefore the activation of *Wi-Fi* and *High* triggers the activation of the feature *Picture*. As the (de)activation of a feature triggers the (de)activation of these parent features, the activation of *Picture* triggers the activation of the features *Rich*, *Message Types* and *Message*. Because *Rich* and *Message Types* are abstract features, they are not attached to a feature definition. So they are activated in the feature model, but not installed in the system behaviour. However, the parent feature *Message* is a concrete one and thus should be activated and installed in its corresponding application classes. Therefore,

²For conciseness, we hide the features between the root feature *Features* and the feature *Picture*. An example of the expanded version of this part of the feature model is illustrated in Figure 7.3.

Message (as a parent feature) would first be deployed a second time and then *Picture* would refine *Message* in the corresponding application classes. In such a case, the corresponding application classes will be adapted with the features *Message*, *Text*, *Message* and *Picture* in that order, as illustrated in Figure 7.4. However this raises an issue since *Message* will override the previously installed behaviour. Because *Message* is a default behaviour (*i.e.*, no implementation of methods uses the *proceed* mechanism), when a method of *Picture* is called, the method is called on the version of *Picture*, that calls in turn the version of the same method of *Message*. After the execution of the method of *Message*, the control statement comes back to the version of the method of *Picture*. In such a case, it ignores the version of *Text* and the first deployed version of *Message*, as illustrated in Figure 7.4. This leads to an undesired, incorrect or erroneous behaviour because a part of the mandatory behaviour (provided by the *Text* feature) is no more executed.

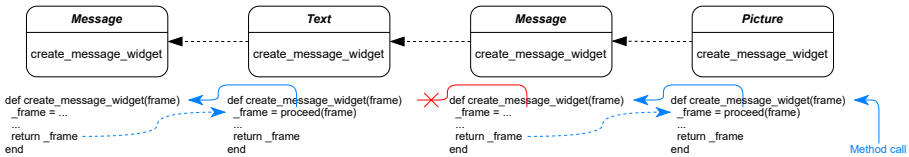


Figure 7.4: Example of how the features *Message*, *Text* and *Picture* would be activated without the usage of the deployed status in the framework class **Feature**.

A solution to avoid such issues is to activate a feature only once, even if it has an activation counter higher than 1. This can be achieved by adding a *deployed* status in the framework class *Feature*, as illustrated in Figure 7.1. With such a boolean, the feature *Message* will not be deployed a second time since it was already deployed in the system behaviour before. With this solution, only the feature *Picture* will be installed in its corresponding application classes to adapt and refine the behaviour of the application. In that case, the features *Message*, *Text* and *Picture* will be installed in that following order. (Figure 4.3 depicts how these features are installed in a common application class.)

This deployed status is also interesting to maintain properly the source code of the application. Assume different contexts that can be activated together trigger the same set of features. In the example based on our messaging system, when a connection is sensed, users can send and receive messages over the Internet, as shown in Figure 7.2.

When a *Wi-Fi* is sensed and activated, the features *Send Message* and *Receive* are activated. After the activation, the features *Send*, *Send Message* and *Receive* are installed in the system, as illustrated in Figure 7.5. (As the feature *Send* provides some default behaviour for its children, it will be installed first.)

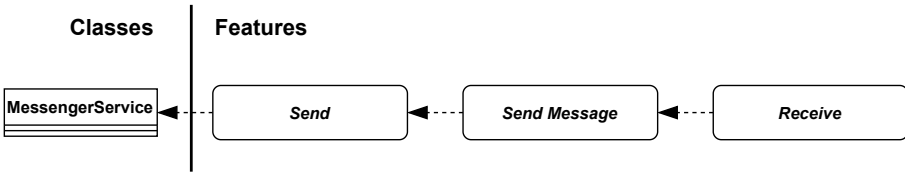


Figure 7.5: Activation of the features *Send*, *Send Message* and *Receive* in the application class *MessengerService*.

Now assume a *Cellular* connection is also sensed. This activates *Cellular* and triggers the activation of the features *Send Message* and *Receive* a second time. Without the deployed status, the system would have installed again these features in the application class *MessengerService*. This application class would then have had the following features installed in that order: *Send*, *Send Message*, *Receive*, *Send*, *Send Message* and *Receive*. Even if the second feature *Send Message* relies on the second feature *Send*, *Send* and *Receive* provide a default behaviour to send and receive messages, respectively. Therefore the first features *Send*, *Send Message* and *Receive* are no more executed. Nevertheless the behaviour is still the expected one and we do not encounter the same issue as explained previously. But keeping them in the application class clutters our understanding of what is going on as the class would contain several unused features. This could confuse programmers when they analyse how their application classes are adapted and by what features in their debugging tasks. Having this deployed status thus prevents to unnecessarily install features again and to maintain a clear source code of the system behaviour.

7.2 Context and feature model declarations

When programmers build a FBCOP application they must declare the context and feature model of their application. For that they need to extend the framework classes *ContextModelDeclaration* and *FeatureModelDeclaration* as explained in Sections 4.4 and 4.5. Each of these framework classes creates the root node of their respective model. *ContextModelDeclaration* creates the root node of the context model and *FeatureModelDeclaration* creates the root node of the feature model of the application. While the root node of the context model is a concrete entity, the root node of the feature model is an abstract entity, as shown in Figure 7.1. The reason for the root context being concrete is to ensure that at least one context is activated and triggers the activation of the default behaviour of the application. The root feature on the other part is an abstract feature. Although it may seem counter-intuitive, having the root feature being abstract is linked to the fact that we cannot develop application-specific behaviour for the context-

oriented programmer and then attach code to the root feature. Furthermore we think it will allow him to design the default behaviour of the application in a good separation of concerns.

These framework classes propose some syntactic sugar to create concrete and abstract nodes. `ContextModelDeclaration` offers the methods *context* and *abstract_context* to create a concrete context and an abstract context, respectively. Similar methods are available for the concrete and abstract features with the methods *feature* and *abstract_feature* in `FeatureModelDeclaration`. These four methods take a name used to create the instance variable of the entity in the system and the name of the entity being created. With such parameters we create dynamically an instance variable and assign it the entity created with the corresponding framework class. An additional parameter is added for the method *feature*. This last parameter allows programmers to describe which application classes this feature must adapt when it is deployed in the system behaviour.

Just like the framework classes `Context` and `Feature` inherited from a common ancestor framework class `Entity`, for consistency, the framework classes `ContextModelDeclaration` and `FeatureModelDeclaration` inherit from a common parent framework class `EntityModelDeclaration`. This parent class contains some infrastructural scaffolding code that can be applied to both models. An example of such scaffolding code is the code that automatically generate getters for all the entities of the model so that programmers have to use the getters of the different entities to create the context-feature mapping model (as illustrated in Section 4.6).

7.3 Context and feature model definitions

Model definitions

At the instantiation of the programmers' declared subclasses of `ContextModelDeclaration` (resp. `FeatureModelDeclaration`), a context (resp. feature) model definition, instance of the framework class `ContextModelDefinition` (resp. `FeatureModelDefinition`), is created so that the framework can interact with the model definition to (de)activate entities in the model. This creation is made with the method *set_model_root(model_root_node)* accessible in the framework classes `*ModelDefinition`. Since we cannot have multiple context (feature) model definition, these two framework classes are singleton classes to ensure a single instance of each model definition at runtime.

We think this separation between the declarations and definitions leads to a better separation of concerns in their corresponding behaviour. With it we can easily distinguish the behaviour of declarations used by programmers

when they have to express (declare) their model of FBCOP applications and the behaviour of definitions used by the framework to (de)activate entities in the models.

As the behaviour of these `*ModelDefinition` framework classes is relatively similar, again they inherit from a common parent framework class `EntityModelDefinition` that contains the implementation of this common behaviour as illustrated in Figure 7.1. This framework class `EntityModelDefinition` offers the methods `activate(entities)`, `deactivate(entities)` and `alter(actions_on_entities)`. The first two methods are called when aiming to attempt to activate or deactivate the list of the entities passed as argument. The method `alter(actions_on_entities)` is a more generic method that tries to activate and deactivate the passed entities. To know which action (activation or deactivation) the framework must perform on each passed entity, an action is associated to the entity. This can be expressed with the framework class `ActionOnEntity` that contains an instance variable describing the *action* the framework must execute on the other instance variable that is the *entity* object. Finally a last method `find_entity_by_name(entity_name)` returns the real entity object that perfectly match the passed name of the entity.

In addition to this common behaviour, more specific behaviour is needed for each of these `*ModelDefinition` framework classes.

For the framework class `ContextModelDefinition`, the framework programmer must add a method to activate at launch time the default situation of the surrounding environment for which the system must run. The method `activate_mandatory_contexts_at_launch_time` is thus used during the bootstrap phase of the application.

The framework class `FeatureModelDefinition` on the other hand contains the list of the features (*bootstrapped_features*) that are installed at launch time. Because the objects are not yet created when the mandatory and default features are installed in the system behaviour, these methods cannot be executed at the deployment phase. Therefore these features are maintained to be executed later when the objects are created. (More information on how this instance variable is initialised will be discussed in Subsection 9.1.3.)

(De)activation of entities

These model definition framework classes serve as a bridge between the ARCHITECTURE and MODELLING parts of our implementation architecture. Whenever some component of our implementation architecture wants to activate or deactivate an entity, it sends a message (e.g. with a call to the method `activate(entities)`) to the definition of this entity model. Then this model attempts to activate or deactivate the entities with a transaction. This transaction ensures that the modifications in the model (*i.e.*, a new configuration) satisfy all

its imposed constraints before either committing or ignoring them.

Now we will explain in more detail how the activation of an entity works. For each entity, we first increment or decrement its pending counter by one when it is an activation or a deactivation, respectively. To maintain a coherence in the model, we must also propagate these changes in the pending counters of all the parents of this entity. This ensures that the parents are also (de)activated if their child is (de)activated since a child cannot exist without its parents in a feature model. Through this explanation, we will illustrate the activation of an entity with the feature *Picture* in the simplified version of the feature model shown in Figure 7.3. As a reminder, *Picture* is a richer message type allowing users to add a picture in their messages or read a picture in their chats.

Figure 7.6a depicts the initial configuration in which we will attempt to activate *Picture*. In Figure 7.6, each box represents a feature (*i.e.*, an entity) and shows the committed counter and pending counter of the feature on the bottom left and on the bottom right, respectively. In the initial configuration, we can observe that the mandatory concrete features (*Message* and *Text*) have already been activated. As *Message* has been activated first, its counters have been incremented by one, as well as the counters of *Features*. *Text* has then been activated. This implies that the counters of *Text*, *Message Types*, *Message* and *Features* have also been incremented by one. As a result, the counters of *Text* and *Message Types* are set to 1 and the counters of *Message* and *Features* are set to 2.

From this initial configuration, we attempt to activate the feature *Picture*. Therefore we increment by one its pending counter and all of its parents' pending counters, *i.e.*, the features *Rich*, *Message Types*, *Message* and *Features*. These modifications are shown in orange in Figure 7.6b.

After updating the pending configuration, we must verify the satisfiability of this model. This means that we must ensure that all the constraints are still respected for this new configuration. We delegate this to a dedicated *satisfiability strategy* as shown in Figure 7.1. More information on this current implementation of the strategy will be discussed in Section 8.2. If this satisfiability strategy returns that the new configuration is valid, all the pending counters of updated entities are committed. In other words, for each modified entities we set the committed counter to the value of its pending counter. This results is shown in Figure 7.6c where the updated committed counters are highlighted in green. Otherwise, if the new configuration is not valid, a rollback is executed. In such a case, each pending counter of each updated entity is reset to the current value of its committed counter.

Now that we described how an activation (and a deactivation since the mechanism is the same) of a feature works, we will illustrate how a deactivation can lead to a rollback. Suppose we try to deactivate the feature *Text*.

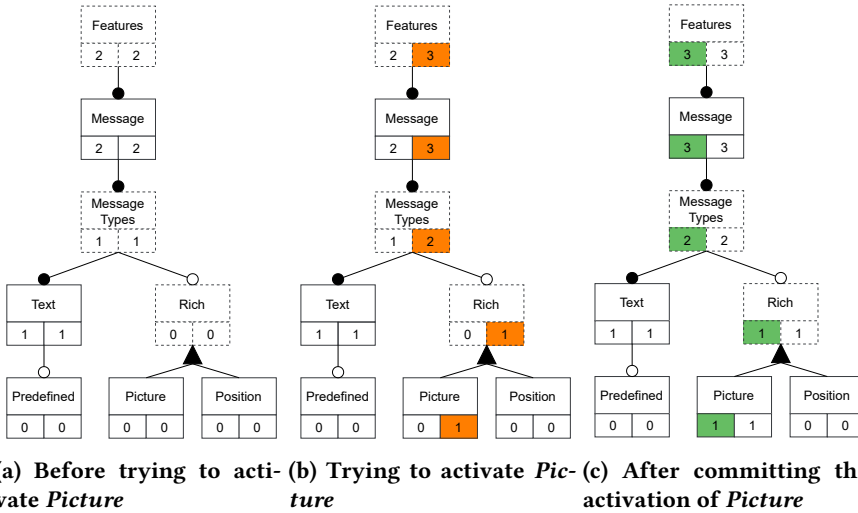


Figure 7.6: Example of activating the feature *Picture* in a simplified version of the feature model of the messaging system. For each entity (feature in this example), we precise its *committed counter* on the bottom left of the box and *pending counter* on the bottom right of the box. Orange boxes highlight the modified pending counters of the concerned features and green boxes show the updated committed counters of these features.

From an initial configuration depicted in Figure 7.7a, we try to deactivate the feature *Text*. For that the pending counter of each entity from *Text* to the root entity *Features* is decreased by one. Figure 7.7b sketches such a scenario. However, since *Text* cannot be deactivated due to the mandatory constraint, this new configuration will be considered invalid by the satisfiability algorithm. A rollback is thus executed to reset the pending counter of each updated entity. This is illustrated in Figure 7.7c where the pending counters are reset and highlighted in red in the figure.

7.4 Mapping model declaration

Similar as for the context and feature model, FBCOP programmers must declare their context-feature mapping model by extending the framework class `MappingModelDeclaration`. The programmer's subclass must be a singleton class since only one mapping model must exist. As explained in Section 4.6, programmers must set the *mapping* instance variable in their subclass. This instance variable is a hash data structure where the keys are lists of contexts and the values are lists of features. A particularity in this hash data structure is that the keys are compared by their identity. In other words, this means that two different references pointing to equivalent objects are not

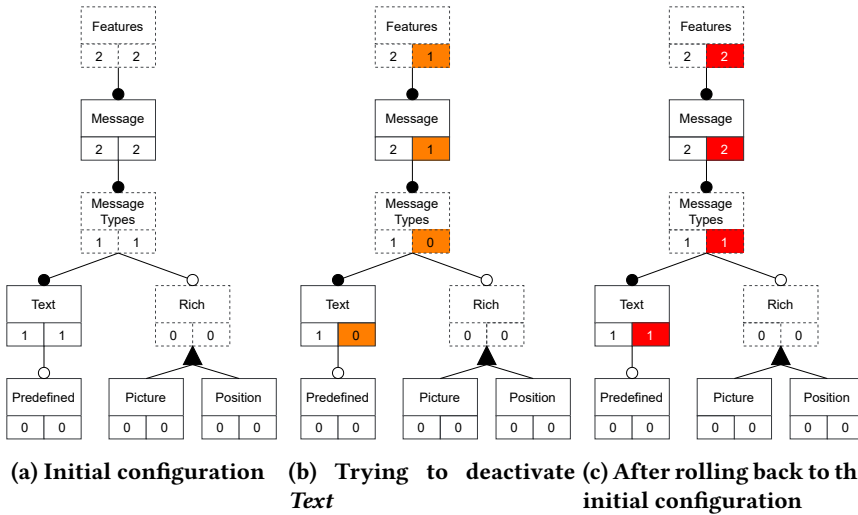


Figure 7.7: Example of the deactivation of the feature *Text* that leads to a rollback based on an excerpt of the feature model of the messaging system. For each entity (feature in this example), we precise its *committed counter* on the bottom left of the box and *pending counter* on the bottom right of the box. Orange boxes highlight the modified pending counters of the concerned features and red boxes show the reset pending counters of these features.

the same keys³. This ensures that we use the same entity objects declared by feature-based context-oriented programmers.

At the creation of the mapping model declaration, the framework is able to generate the implicit individual mapping from the root node of the context model to the mandatory concrete features as described in Section 3.1.6. This mapping relation serves to install the mandatory features at the bootstrap of the application.

7.5 Mapping model definition

As for the context and feature model definitions, when the programmers' mapping model declaration (`MappingModelDeclaration`) gets instantiated, the mapping model is set in the framework class `MappingModelDefinition` by calling the method `set_mapping_model(mapping_model)`.

This framework class is used by the framework to find the features that need to be activated (resp. deactivated) when contexts got activated (resp. deactivated) with the method `features_from_activation(contexts)` (resp. `features_from_deactivation(contexts)`). The method `features_from_activation(contexts)` is shown in Algorithm 1 through pseudo-code. The intuition of this

³https://apidock.com/ruby/v2_5_5/Hash/compare_by_identity

method is as follows. The method analyses the mapping model to find all the potential mapping relations for which the passed contexts are implied. Then the algorithm selects only the features for which all the contexts of each mapping relation are active. This second part of the algorithm is illustrated in Algorithm 2. The union operator is used in both algorithms to avoid duplicate features. Similarly, the method *features_from_deactivation(contexts)* follows the same approach but returns only the features for which at least one context in each mapping relation is inactive.

Algorithm 1 Main algorithm to find all the features to activate when contexts got activated

Input: activated and deactivated contexts

Output: features

```

1: features  $\leftarrow []$ 
2: for all context  $\in$  contexts do
3:   _relations  $\leftarrow$  _FIND_POTENTIAL_RELATIONS(context)
4:   features  $\leftarrow$  features  $\cup$  _FIND_FEATURES_TO_ACTIVATE(_relations)
5: return features

```

Algorithm 2 Algorithm to find all the features to activate based on the filtered mapping

Input: mapping relations

Output: features

```

1: features_to_activate  $\leftarrow []$ 
2: for all contexts, features  $\in$  mapping_relations do
3:   if _CONTEXTS_ARE_ALL_ACTIVE?(contexts) then
4:     features_to_activate  $\leftarrow$  features_to_activate  $\cup$  features
5: return features_to_activate

```

7.6 Conclusion

In this chapter, we discussed all the design choices we took for designing and implementing the ENTITIES part of our framework implementation. We explained how we designed and implemented the building blocks of the FBCOP programming framework, *i.e.*, the contexts and features as well as the mapping between them. We also described how the context (feature and mapping) model declaration are mapped to their context (feature and mapping) model. Finally we also exemplified how we can (de)activate entities in its corresponding entity model with a transaction.

CHAPTER

8

MODELLING

Now that we have explained how to implement the main building blocks of our programming framework, *i.e.*, the ENTITIES part of our implementation architecture, we turn our attention to the MODELLING part. The main purpose of this part is to provide a generic implementation architecture for building context or feature models. Since both context and feature models will have the same structure, we will refer them as entity models. Figure 8.1 shows the class diagram for the MODELLING part of our implementation architecture. In this chapter, we will present this part from two different angles: the structure of an entity model (Section 8.1) and the satisfiability strategy to ensure a configuration of such models respects all the constraints imposed by the model (Section 8.2).

8.1 Structure of an entity model

As stated before, both context and feature models will be represented as feature diagrams. Since the entities on these diagrams can be either contexts or features we will refer to them as entity models. An entity model is a tree-like structure where the nodes are entities (*i.e.*, contexts or features) and the edges are relations between those entities. By including the trait¹ Node, an entity can be part of the entity model as illustrated in Figure 8.1.

¹Implemented as a module in the *Ruby* programming language.

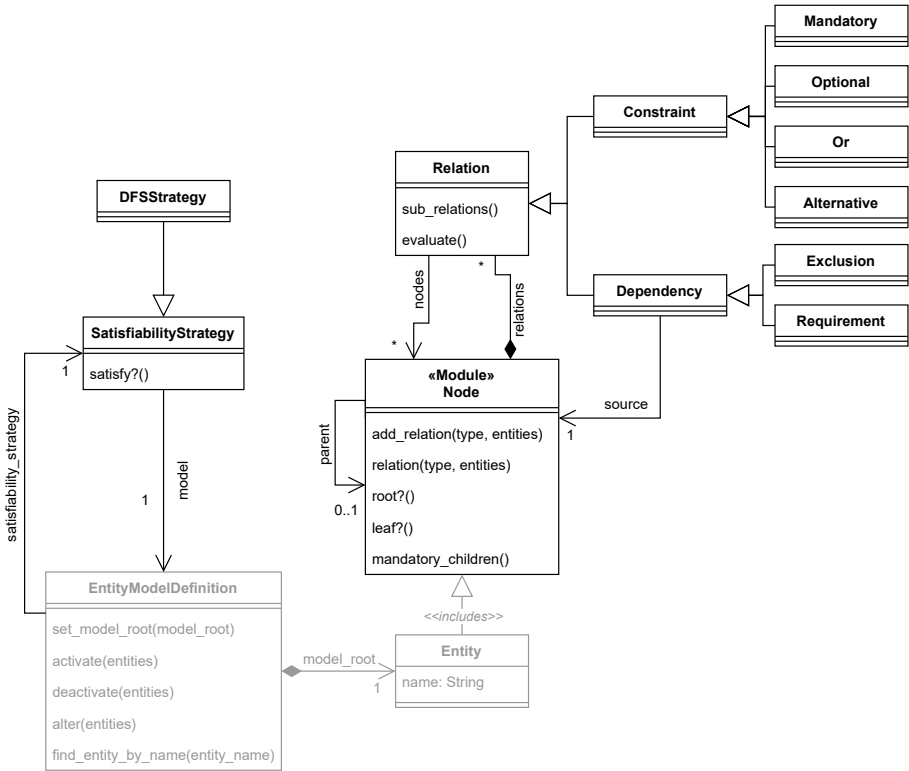


Figure 8.1: Class diagram of the MODELLING part of our framework implementation. Some information (constructors, getters and setters) is voluntarily omitted for readability reasons. Some parts are greyed out since they are part of ENTITIES. However we reduced them to facilitate the understanding of the interactions between the different classes on our implementation framework.

When programmers create a context or feature model, they can link the nodes through relations (instances of the framework class *Relation*). As explained in Sections 4.4 and 4.5, they can use the method *relation* (an alias for the method *add_relation*) of the trait *Node* to add a specific relation with some other entities to the entity. Such relations can be either constraints or dependencies.

Constraints, which are instances of the framework class *Constraint*, represent a hierarchical constraint between an entity and its child entities. These constraints can be instances of the framework classes *Mandatory*, *Optional*, *Or*, or *Alternative*.

Cross-tree relations, also known as dependencies, are instances of the framework class *Dependency*. They represent non-local dependencies between nodes in the tree that do not necessarily have a direct hierarchical

relation. For reusable reasons, the target nodes are the nodes of the (dependency) relation and we add a *source* node as an instance of the framework class `Dependency`, as illustrated in Figure 8.1. Two types of such dependencies are currently available in our programming language: `Exclusion` and `Requirement`.

(As the meaning of these constraints and dependencies have already been introduced in Section 3.1.2, we do not discuss them in more detail here.) Other types of dependencies could also be implemented in the future (see Section 14.1).

Each relation has two methods *sub_relations()* and *evaluate()*. More information about these two methods will be provided in Section 8.2 since they are used by the satisfiability strategy when the framework verifies whether the new configuration is valid or not.

Furthermore each entity of the model, except the entity model's root, has a reference to its direct *parent*. This is useful when we must propagate an entity's (de)activation through the model since the propagation goes up to the root of the entity model as explained in Section 7.3.

The trait `Node` also has the methods *root?()*, *leaf?()* and *mandatory_children()* to query the entity of a model. These methods are mostly interesting for the programmers of the programming framework. For example, *mandatory_children()* is used to find the direct mandatory sub-entities of an entity. This method is useful when we must find the initial situation of the surrounding environment through the context model. It also serves to get all the mandatory features (from the root) to generate the implicit mapping relation that must be triggered to activate the default behaviour at launch time of the application (as explained in Section 3.1.6).

8.2 Satisfiability algorithm

As explained in Section 7.3, when we try to (de)activate entities in an entity model, we first modify the pending status of the concerned entities to create a new pending configuration. Then we must ensure that this new pending configuration respects all the constraints imposed by the entity model. Finally the new configuration is committed or ignored depending on the validity of the configuration. Even though we already sketched this algorithm, we did not yet explain how we verify the consistency of the constraints. This is what we will discuss in this section.

To determine whether a new pending configuration of an entity model is valid or not, the model (instance of `EntityModelDefinition`) uses its instance variable *satisfiability_strategy* (instance of the framework class `SatisfiabilityStrategy`). As a reminder, a valid configuration of an entity model is a set of active and activated entities of the model that respects all

its constraints. Many mechanisms exist to verify the consistency of a model, from using a depth-first search algorithm to the usage of sat solving [Bat05] or constraint programming techniques [BTR05]. To keep things simple, we decide to implement a depth-first search algorithm. It will visit the entire model to ensure that each constraint is respected. Obviously when a constraint is no longer satisfied, the algorithm is stopped to avoid useless computations. To do that we implemented a `DFSStrategy` framework class as a subclass of the framework class `SatisfiabilityStrategy` that overrides the method `satisfy?()`. Such a design choice allows us to potentially extend our implementation framework with other strategies. We will discuss about other possible strategies in the future work (see Section 14.4).

A pseudo-code of the method `satisfy?()` of the `DFSStrategy` framework class is shown in Algorithm 3. As we can see on Line 1, we initialise a boolean variable with the pending status of the root node. If the root node can be activated, the variable is set to true. Otherwise it is set to false. In other words, if its pending counter is strictly greater than 0, this means that it can be activated. This design choice implies the root node must be always activated to get a valid configuration of the model. This is needed because of an entity model has always a default valid configuration composed by a non-empty set of entities of the model to describe either the default environment in which the application runs or the default behaviour of the application.

Algorithm 3 Algorithm to verify an entity model's consistency based on a pending configuration

Input: /

Output: True/False

```

1: sat?  $\leftarrow$  CAN_BE_ACTIVATED?(root node)
2: queue  $\leftarrow$  relations attached to root node
3: visited  $\leftarrow$  empty set
4: while sat?  $\wedge$  queue is not empty do
5:   relation  $\leftarrow$  pop a relation of queue
6:   if relation was not yet visited then
7:     add relation in visited
8:     sat?  $\leftarrow$  sat?  $\wedge$  EVALUATE(relation)
9:     push SUB_RELATIONS(relation) in queue
10: return sat?

```

To ensure the model is valid, the framework must traverse the model through the relations and not the nodes because of the constraints are on the relations and not on the nodes. For that, we traverse the different relations with an iterative algorithm. This explains the usage of the *queue* data structure on Lines 2, 4, 5 and 9. While the queue is not empty, we will con-

tinue to traverse the relations to evaluate them. An optimisation is still implemented to avoid useless computations when the model is not longer valid, as shown on Line 4. To feed this queue, the framework relies on the method *sub_relations()* to get the direct relations under the nodes of the current manipulated relation, as depicted on Line 9.

From an algorithmic point of view, we must avoid infinite loops when visiting the entity model. Such infinite loops can from errors in the declaration of the entity model or from dependencies (*i.e.*, cross-tree constraints). To do so we ensure we visit only once each relation through the usage of a set of *visited* relations on Lines 3, 6 and 7. When a relation has been evaluated, we add it to the *visited* relations.

To verify whether a relation respects its constraint with the new pending configuration, the algorithm delegates it to the relation itself by calling the method *evaluate()* as depicted on Line 8. For that to work, we must implement this method on each relation as illustrated in Figure 8.1. As we are interested to verify the new pending configuration, we must check the pending counters of the nodes of the relation to avoid breaking the current semantics of the constraint.

A pseudo-code example of the method *evaluate()* of the framework class *Mandatory* is illustrated in Algorithm 4. In this algorithm we ensure that all the nodes of the relation can be activated. When a node cannot be activated, we stop the computation by returning that the constraint is no more satisfied. The semantics of all other constraints and dependencies have been already illustrated in Section 3.1.2 and follow the same structure for their algorithms.

Algorithm 4 Algorithm to verify whether the *mandatory* constraint is respected on a pending configuration

Input: /

Output: True/False

```

1: for node  $\in$  nodes do
2:   if  $\neg$ CAN_BE_ACTIVATED?(node) then
3:     return False
4: return True

```

As such we can verify if the new pending configuration is valid or not since we visit all the constraints and check if they respect the semantics of the model.

8.3 Conclusion

In this chapter, we first introduced how we implemented the structure of an entity (*i.e.*, context or feature) model. As each entity is considered as a node,

we can compose a model by connecting its nodes through relations. The relations can either be constraints (*i.e.*, *mandatory*, *optional*, *or* and *alternative*) or dependencies (*i.e.*, *exclusion* or *requirement*).

We also described a verification algorithm that ensures the consistency of the semantics of the model is always respected. This algorithm is used each time the framework tries to (de)activate entities in a model. We implemented it as a depth-first search strategy algorithm. It allows to visit all the relations of the model and check if the relations are still satisfied or not. After finishing its traversal this algorithm can determine if the new pending configuration is a valid or invalid one.

CHAPTER

9

ARCHITECTURE

In this chapter we explain how we implemented the control flow of our framework from the (de)activation of the contexts to the (un)adaptation of the features in the application behaviour, via the (un)selection and (de)activation of the features triggered by those contexts. Then we discuss the mechanism we implemented to adapt dynamically the application behaviour. Finally we describe how we implemented the *proceed* mechanism which is a key concept in context-oriented programming, to achieve dynamic adaptation of previously activated behaviour. As a summary of the overall architecture, Figure 9.1 illustrates the class diagram for the ARCHITECTURE part.

9.1 Control flow

As the overall control flow of our framework has already been explained and illustrated at multiple occasions throughout this dissertation, and in particular in Section 3.2, we emphasise here only the key design choices we made to implement the different components of this control flow.

Each component of the control flow is implemented with a dedicated framework class. The CONTEXT ACTIVATION, FEATURE SELECTION, FEATURE ACTIVATION and FEATURE EXECUTION components each have their own framework class: `ContextActivation`, `FeatureSelection`, `FeatureActivation`, `FeatureExecution`. These four framework classes are singleton classes to ensure that we have only one instance of each component in the

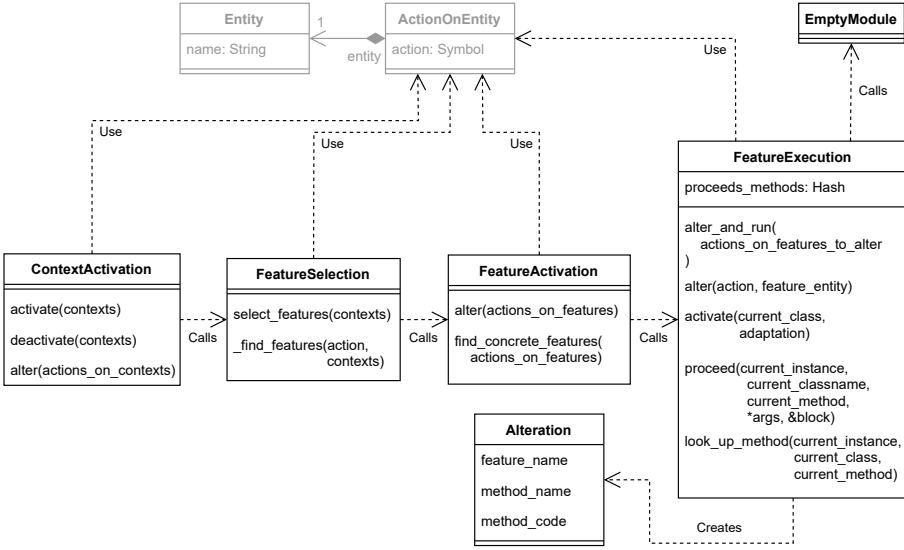


Figure 9.1: Class diagram of the ARCHITECTURE part of our implementation. Some information (constructors, getters and setters) is voluntarily omitted for readability reasons. Some parts are greyed out since they are part of ENTITIES. We reduced them to facilitate the understanding of the interactions between the different classes of our implementation framework.

execution of a FBCOP application. Below we describe the design choices for each of these different components and the order in which they are executed.

9.1.1 CONTEXT ACTIVATION

The framework class `ContextActivation` serves as entry point of the control flow to adapt the application behaviour when contexts change. For that, this framework class proposes three methods *activate(contexts)*, *deactivate(contexts)* and *alter(actions_on_contexts)*.

As their names suggest, the methods *activate(contexts)* and *deactivate(contexts)* are used to try to activate or deactivate the contexts passed as argument. The method *alter(actions_on_contexts)* is more generic and either activates or deactivates the different contexts passed as arguments according to the kind of action associated for each context. As a reminder, Section 4.8 illustrated how we can call these three methods.

When one of these methods is called, the component first delegates the activation or deactivation of the contexts to the context model definition of the application, as explained in Section 7.3. For example, when the method *activate(contexts)* is called, the component calls in turn the *activate(contexts)* method on the context model definition. Then the component sends the con-

texts that are activated and/or deactivated to the FEATURE SELECTION component with the method *select_features(contexts)*.

Algorithm 5 illustrates the pseudo-code of the method *activate(contexts)*.

Algorithm 5 Algorithm of the method *activate(contexts)* in the `ContextActivation` framework class.

Input: contexts to activate

Output: activated contexts

- 1: `ContextModelDefinition.instance.ACTIVATE(contexts)`
 - 2: `FeatureSelection.instance.SELECT_FEATURES({activated: contexts})`
-

9.1.2 FEATURE SELECTION

The FEATURE SELECTION component aims to find the features that must be activated and deactivated with the method *select_features(contexts)* called by the `ContextActivation` framework class. This method is depicted in pseudo-code in Algorithm 6.

Algorithm 6 Algorithm of the method *select_features(contexts)* in the `FeatureSelection` framework class.

Input: activated contexts

Output: features to (de)activate or an error is raised

- 1: *selection* \leftarrow `_FIND_FEATURES("activation", activated_contexts)`
 - 2: *unselection* \leftarrow `_FIND_FEATURES("deactivation", deactivated_contexts)`
 - 3: `REVERSE(unselection)`
 - 4: *features_to_activate* \leftarrow `_GENERATE_ACTIONS_ON_FEATURES("activate", selection)`
 - 5: *features_to_deactivate* \leftarrow `_GENERATE_ACTIONS_ON_FEATURES("deactivate", unselection)`
 - 6: *features* \leftarrow *features_to_deactivate* + *features_to_activate*
 - 7: `FeatureActivation.instance.ALTER(features)`
-

The component first selects the features to activate and deactivate with the method *_find_features(action, contexts)* as illustrated on Lines 1 and 2.

For that, the component delegates this query to the mapping model definition as explained in Section 7.5. The method *_find_features* returns either the selected features or an empty list if no context has been (de)activated. While we keep the order in which the features to activate are selected, we reverse the list of the features to deactivate as shown on Line 3. This design choice comes from the fact that we cannot remove a feature if the next feature relies on it with a *proceed* statement. We consider this as a caution to preserve the

order in which the features must be deactivated. In addition it is more logic to deactivate first the most recent feature.

For the lists of activated and deactivated features, we encapsulate each feature into an instance of the framework class `ActionOnEntity` by associating each action that must be executed for this feature, as depicted on Lines 4 and 5. Then we merge these two lists so that the new list contains first the deactivated features followed by the activated features as illustrated on Line 6. This design choice helps us to generalise the behaviour of the (de)activation of the features in the `FEATURE ACTIVATION` component. Finally this merged list is sent to the `FEATURE ACTIVATION` component with the method `alter(actions_on_features)`.

9.1.3 FEATURE ACTIVATION

As opposed to its homologue, the framework class `FeatureActivation` proposes only the method `alter(actions_on_features)` to (de)activate features. This generic implementation is possible thanks to the manipulation we did in the previous component. This allows us to keep only one method to do all the actions.

The pseudo-code of this method is shown in Algorithm 7.

Algorithm 7 Algorithm of the method `alter(actions_on_features)` in the `FeatureActivation` framework class.

Input: features to deactivate and activate

Output: (de)activated features

- 1: `FeatureModelDefinition.instance.ALTER(actions_on_features)`
 - 2: `actions_on_concrete_features` \leftarrow `_FIND_CONCRETE_FEATURES(actions_on_features)`
 - 3: **if** \neg `bootstrap_done?` \wedge \neg `EMPTY(actions_on_concrete_features)` **then**
 - 4: `bootstrapped_features` \leftarrow `GET_ALL_FEATURES(actions_on_concrete_features)`
 - 5: `FeatureExecution.instance.ALTER_AND_RUN(actions_on_concrete_features)`
-

The first step is to delegate the (de)activation of the features to the feature model definition (Line 1).

As we can have concrete features in the parents or ancestors, we must find all the concrete features (Line 2) that are (de)activated to ensure these features are also (de)activated in the application behaviour by the `FEATURE EXECUTION` component. To do that, we traverse the list of the (de)activated features and for each feature we look for potential concrete parent features. If we encounter concrete parent features, we add them before (resp. after) the

feature in the list in case of an activation (resp. a deactivation) to ensure that they are deployed (resp. uninstalled) in the correct order in the next component. This allow us to maintain the hierarchy between the features in the feature model since parent concrete features contain more generic behaviour while the child features refine/specialise it. To verify if the parent features must be activated for an activation, we ensure that the parent feature was not deployed yet (as described in Section 7.1). In the case of a deactivation, we ensure that the parent feature is totally deactivated (*i.e.*, its committed counter is equal to zero) and its behaviour has been already deployed in the application behaviour.

Before letting the framework adapt the application behaviour, we must ensure that we store the mandatory and default behaviour that must be executed at the launch of the application (Lines 3-4). This is a crucial step for bootstrapping the application. Without this, we will encounter runtime errors at launch time. In fact, after the framework has deployed the features thanks to the next component (*i.e.*, `FEATURE EXECUTION`), it runs each necessary method on the corresponding instances to create the user interface or to load information, for example. Because the objects of the application are not created yet at the first activation of features, the framework cannot execute the different prologues on the inexistent application instances. Therefore we must store this mandatory and default behaviour in the feature model definition. This behaviour will be executed later when the application will have been launched thanks to the module `CodeExecutionAtLaunchTime`, as described in Section 4.1.

Finally all the features, encapsulated as instances of the framework class `ActionOnEntity`, are sent to the singleton instance of the framework class `FeatureExecution` with the method `alter_and_run(_actions_on_features)`.

9.1.4 FEATURE EXECUTION

The `FEATURE EXECUTION` component deploy and undeploy the features in the application behaviour. Algorithm 8 sketches the method `alter_and_run(actions_on_features)` in pseudo-code.

For each feature passed to the method `alter_and_run`, the framework calls a method `alter(action, feature)` on Line 7 that modifies the application behaviour by (de)activating the feature depending on its associated action. This method loads all the feature parts (each dedicated to a specific concern to increase modularity) of the passed feature. This method then installs or uninstalls each feature part in its corresponding application class defined by the feature declaration depending on if the feature has to be activated or deactivated. More information on how we propose to adapt dynamically the application behaviour will be given in Section 9.2.

Algorithm 8 Algorithm of the method *alter_and_run(actions_on_features)* in the `FeatureExecution` framework class.

Input: actions on features

Output: /

```

1: active_features_to_execute  $\leftarrow []$ 
2: for action, feature  $\in$  actions_on_features do
3:   if action is a deactivation then
4:     RUN_EPILOGUE(feature)
5:   else
6:     PUSH(feature) on active_features_to_execute
7:   ALTER(action, feature)
8:   UPDATE_DEPLOYED_STATUS(action, feature)
9: if bootstrap_done? then
10:  for feature  $\in$  active_features_to_execute do
11:    RUN_PROLOGUE(feature)

```

Once the feature is installed in the application behaviour, we must update its deployed status as illustrated on Line 8 for the reasons explained in Section 7.1.

Last but not least, adapting the application behaviour implies the framework must also run the prologue or epilogue of each feature part activated or deactivated. For logical reasons, the framework must execute the epilogue of each feature part before removing the feature of the application behaviour (Line 4). Otherwise the epilogue could not be executed because the dedicated source code will not longer be part of the dedicated application class. This would raise a runtime error or an unexpected behaviour. For the prologue, we decide to run all prologues after having activated all features for simplicity reasons, as shown in Lines 10-11. This design choice also ensures that the prologues are executed on a stable version of the source code, especially when the prologues run methods using the *proceed* mechanism. This explains the use of a list of active features to keep a trace of each activated feature on Lines 1 and 6). Nevertheless the prologues can only be executed when the application is already running as explained in Subsection 9.1.3. This justifies why we guarded the execution of the prologue in Line 9.

9.2 Dynamic adaptation

Now we will explain how we treat the dynamic adaptation when features are installed to or removed from the application behaviour. Many mechanisms exist to implement such a dynamic adaptation. For example, we can see in the literature, and more specifically in Cardozo and Mens's work [CM22]

that some approaches rely on context-dependent method dispatch [CH05; GMH07; GMC08; HCN08], metaprogramming [Gon+11; SMH17], or incremental composition techniques [Gon+13].

In our implementation, we use an *unbinding and binding methods* mechanism through metaprogramming. This mechanism aims to retrieve the unbound methods of a feature part and then attach or remove them to the corresponding application class.

To illustrate this mechanism let us revisit the messaging system. Assume we have two application classes, `MessageModel` and `MessageView`, that represent the model and the view of a message, as depicted in Listing 9.1. We also have a feature `Message`, shown in Listing 9.2, that provides the default behaviour of a message in our messaging system. (In this example, we reduced the default behaviour for readability reasons.)

```

1 class MessageModel
2   include Observable
3 end
4
5 class MessageView
6 end

```

Listing 9.1: Code snippet of the definition of the application classes `MessageModel` and `MessageView` of the messaging system.

```

1 module Message
2   module Model
3     can_adapt : MessageModel
4
5     def initialize()
6       # Some code here
7     end
8     def from()
9       # Getter of the instance variable 'from'
10      ↪ (representing the author of the message)
11    end
12    def from=()
13      # Setter of the instance variable 'from'
14    end
15  end
16 end
17
18 module View
19   can_adapt : MessageView
20
21   def initialize()
22     # Some code here
23   end
24   def create_message_widget(frame)

```

```

23      # Return the created layout attached to the frame
    ↪ with the author in the layout
24  end
25  def update ()
26      # Update the layout or the author if something
    ↪ has been changed.
27  end
28 end
29 end

```

Listing 9.2: Code snippet of the feature definition of the feature *Message* in the messaging system.

When the feature *Message* is activated, we retrieve all the methods of a feature part as unbound methods (since they are not longer attached to a class or a module). In our example, for the feature part *Model* which is the part of the feature *Message* that will be installed in the application class `MessageModel`, we have three unbound methods: *initialize*, *from*, and *from=*. The first method is the constructor and the two last methods represent the getter and setter of the instance variable *from*¹, respectively. Then we attach these unbound methods to the corresponding application class. As a reminder, the application class is defined by the intersection of the targeted application classes declared in the feature declaration (as shown on Listing 9.3) and the macro *can_adapt* in the current feature part (as shown on Line 3 of Listing 9.2). Therefore, the application class for the feature part *Model* in the feature *Message* is the application class `MessageModel`. Then we continue to iterate like this for each feature part of the activated feature to install completely the feature *Message* in the application behaviour. In particular, in this example, the unbound methods *initialize*, *create_message_widget* and *update* of the *View* feature part will get attached to the `MessageView` application class.

```

1 feature :@message, 'Message', [:MessageModel,
    ↪ :MessageView]

```

Listing 9.3: Code snippet of the declaration of the feature *Message* of our messaging system.

Figure 9.2 illustrates incrementally how the behaviour of the application classes is thus adapted when we activate the feature *Message*. Before the activation of the feature in the behaviour, the two application classes `MessageModel` and `MessageView` are empty, as depicted in Figure 9.2a. Then after installing the feature part *Model*, the behaviour of the application

¹To simplify the understanding of our dynamic adaptation mechanism, we have explicitly written the getter and setter of the instance variable *from*. In real *Ruby* source code, however we would use the *Ruby* macro, *attr_accessor*, to generate its getter and setter.

class `MessageModel` is updated with the three methods *initialize*, *from* and *from=*, as shown in Figure 9.2b. Finally after installing the feature part *View*, the behaviour of the application class `MessageView` is modified with the methods *initialize*, *create_message_widget*, and *update*, as illustrated in Figure 9.2c. Figure 9.2c also represents the application behaviour available after the full activation of the feature *Message*.

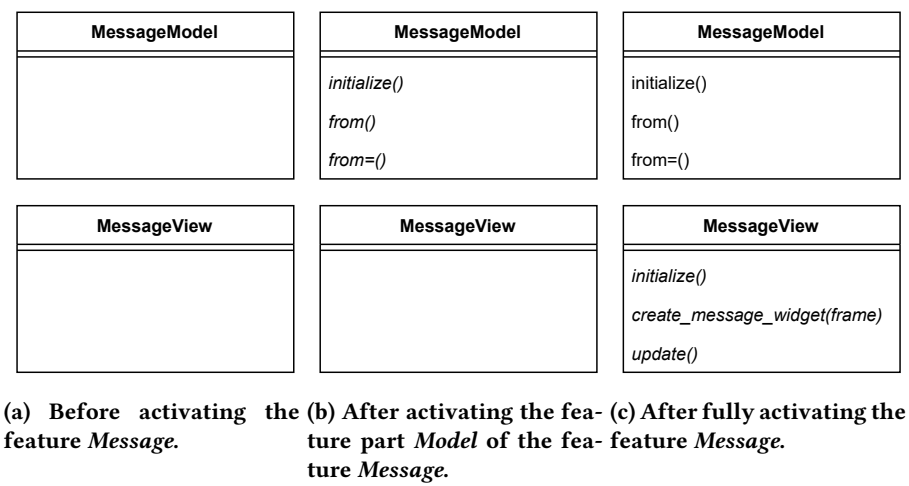


Figure 9.2: Example on how the application classes `MessageModel` and `MessageView` are incrementally adapted by the feature *Message* when it gets activated. The methods just activated are in italics.

The previous example explains how the application behaviour is adapted when a feature is activated and for which the behaviour is a new one. But how does our mechanism adapt the system behaviour when an activated method has a new version or a refinement? To address this issue, we use the same mechanism but we store a stack for each known method of each application class and only the last adaptation (*i.e.*, the adaptation on the top of the stack) is installed in the application class so that when this method is called, the latest version of the method is executed. To store all the variants of each method of each application class, we use the hash data structure *proceed_methods*, the instance variable of the framework class `FeatureExecution`, as shown in Figure 9.1. In this data structure, the keys are identified by an identifier composed of the name of the application class and the method name and the values are a stack data structure that contains all the versions of the method. Each version of the method is implement with a data class `Alteration`, that contains the application class for which this method is an adaptation, the name of the method and the (unbound) method itself, as shown in Figure 9.1.

Let us illustrate in more detail how this mechanism works based on our messaging system. Assume we already installed the feature *Message* in the

application classes `MessageModel` and `MessageView`, as shown in Figure 9.3a. Let us now activate the feature *Text*, declared in Listing 9.4 and defined in Listing 9.5, in the application behaviour.

```
1 feature :@text, 'Text', [:MessageModel, :MessageView]
```

Listing 9.4: Code snippet of the declaration of the feature *Text* of our messaging system.

```
1 module Text
2
3   module Model
4     can_adapt :MessageModel
5
6     def initialize()
7       # Some code here
8     end
9     def text()
10      # Getter of the instance variable 'text'
11    end
12    def text=()
13      # Setter of the instance variable 'text'
14    end
15
16  end
17
18  module View
19    can_adapt :MessageView
20
21    def create_message_widget(frame)
22      # Add the textual content in the layout created
23      ↪ in the default behaviour
24    end
25    def update()
26      # Update the textual content if it was modified
27    end
28  end
29
30 end
```

Listing 9.5: Code snippet of the feature definition of *Text* of our messaging system.

After the installation of the feature part *Model* of this *Text* feature, the behaviour of the application class `MessageModel` gets adapted by adding the methods *text* and *text=* and have updating the constructor *initialize* with its new version. When the feature part *View* has been installed, the behaviour of the application class `MessageView` is also modified since the methods

create_message_widget and *update* have been refined. The result of the activation of the feature *Text* in the application behaviour is shown in Figure 9.3b. The method in italics are just activated and those in bold are a refinement or a new adaptation of a previously installed adaptation for this method in the application class.

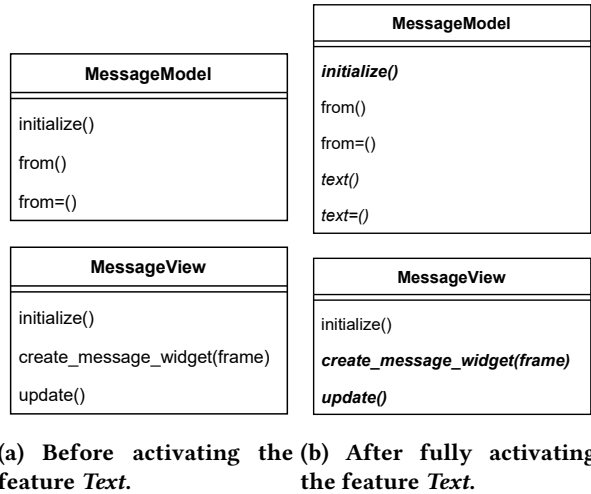


Figure 9.3: Example of how the application classes **MessageModel** and **MessageView** are further adapted by the feature *Text*. In this example, we assume the feature *Message* had already been activated before. The methods in italics are just activated and those in bold are a refinement or a new adaptation of an already activated adaptation of this method in the application class.

To better visualise how the adaptations are stored in the *proceed_methods* data structure after the activation of the features *Message* and *Text*, Figure 9.4 displays what this structure contains. We can observe that for each method of each application class, we keep a stack of all the alterations (*i.e.*, all the versions) of their method.

9.3 Proceed mechanism

Now that we have explained how our architecture can adapt the application behaviour, we need to explain in more detail how it handles the *proceed* mechanism to incrementally adapt application behaviour.

As a reminder, when programmers want to refine an existing behaviour by extending it, they must use the *proceed* mechanism as exemplified in Sections 4.9 and 5.4.

To ensure the *proceed* method is available at any place in the code for programmers, we add this method in the class `Object` of the programming lan-

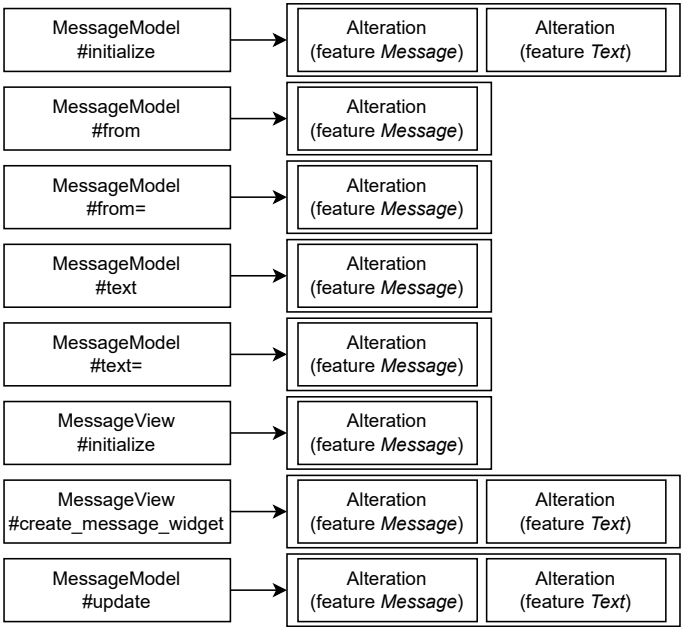


Figure 9.4: State of the *proceed_methods* variable when the *Message* and *Text* features are activated (in that order) in the application behaviour.

guage. With that, this method is thus callable in each new instantiated object since all instances inherit the behaviour from the class *Object*. This method has the following parameters: the name of the application class within the method is deployed and the list of arguments that represents the arguments of the method in which the *proceed* is called. The name of the application class is important since this serves to find the previous adaptation of the right method. As a reminder, a method contains a stack of adaptations and it is identified by the name of the application class and the name of the method.

Revisiting the feature part *View* of the feature *Message* that adapts the application class *MessageView*. As a reminder, Listing 9.6 shows the code snippet of this feature part. To use the *proceed* method, as illustrated on Line 1, the application programmers must pass the argument *layout*, that is the parameter of the method *create_message_widget*. However they do not pass the application class within the method is deployed. This argument is automatically added when the method is deployed in its application class. Therefore each *proceed* call is translated so that the name of the application class is automatically added by the programming framework as the first argument. Listing 9.7 shows the translation of the *proceed* call depicted on Line 6 of Listing 9.6. This translation is hidden for the programmers to avoid they must mention at each time the application class when calling the *proceed* method,

but it is the translated version that will really execute at runtime.

```

1 module Text
2   module View
3     can_adapt : MessageView
4
5     def create_message_widget(layout)
6       message_layout = proceed(layout)
7       ui_manager = UI::UIManager.instance
8       ui_manager.create_ui_object_in(:message_text,
9         ↪ :FXLabel, message_layout, @message_model.text)
9       return message_layout
10    end
11    # More methods
12  end
13  # More feature parts
14 end

```

Listing 9.6: Code snippet of the feature part *View* of the feature *Text* adapting the application class *MessageView* in the messaging system.

```

1 message_layout = proceed("MessageView", layout)

```

Listing 9.7: Code snippet showing the translation of the *proceed* call shown on Line 6 of Listing 9.6.

The *proceed* method called by programmers is the method of the *Object* class and not the method *proceed* of the *FeatureExecution* framework class, while the real implementation of the *proceed* mechanism is developed in the *FEATURE EXECUTION* component. We made this design choice because it simplifies the *proceed* call from application programmers' perspective and is syntactically similar to the *super* method when programmers want to call a generic version of their method. So, when a *proceed* call is executed at runtime, the *Object* class intercepts the message, retrieves the needed required arguments of the real implementation of the *proceed* mechanism and then delegates to the implementation of the *proceed* method of the framework class *FeatureExecution*. As depicted in the class *FeatureExecution* in Figure 9.1, the method *proceed* has the following parameters: the current instance on which the method has been called, the name of the application class, the name of the current method that calls the *proceed* mechanism and the arguments of this current method. Algorithm 9 depicts the pseudo-code of the *proceed* method of the *Object* class. On Line 1, the current method is retrieved by inspecting the call stack for example, and the instance on which the method is executed can be easily retrieved by the *self* instance as shown in Line 2.

When the framework calls the *proceed* method in the *FEATURE EXECUTION* component, the framework loads the application class passed as argument,

Algorithm 9 Algorithm of the method *proceed* of the class *Object*.

Input: classname, args

Output: Result of the *proceed* method

```

1: current_method  $\leftarrow$  FIND_NAME_CURRENT_METHOD()
2: return FeatureExecution.instance.PROCEED(self, classname, cur-
   current_method, args)

```

finds the adaptation stack for the current method of the application class and pops twice the stack to get first the current version and then the previous adaptation. It then deploys the previous adaptation in the application class to erase the latest version of the method and executes the previous version of this method. After running the previous adaptation, it redeploys the latest adaptation of the method and returns the result of the previous adaptation. Algorithm 10 illustrates the pseudo-code of this *proceed* method in FEATURE EXECUTION.

Algorithm 10 Algorithm of the *proceed()* method in the *FeatureExecution* framework class.

Input: current_instance, classname, current_method, args

Output: Result of the previous adaptation

```

1: current_class  $\leftarrow$  GET_CLASS(classname)
2: last_adaptation  $\leftarrow$  POP_ADAPTATION(current_class, current_method)
3: previous_adaptation  $\leftarrow$  POP_ADAPTATION(current_class, current-
   _method)
4: ACTIVATE(current_class, previous_adaptation)
5: method_to_execute  $\leftarrow$  LOOK_UP_METHOD(current_instance, current-
   _class, current_method)
6: result  $\leftarrow$  method_to_execute.CALL(args)
7: ACTIVATE(current_class, last_adaptation)
8: return result

```

Let us revisit again our example of the *proceed* mechanism of Section 5.4, but we will only show how it works with the features *Message* and *Text* for conciseness reasons. The implementation of the features *Message* and *Text* are again shown in Listings 9.8 and 9.6, respectively.

```

1 module Message
2   module View
3     can_adapt : MessageView
4
5     attr_accessor :main_window
6
7   def create_message_widget(layout)

```

```

8      ui_manager = UI::UIManager.instance
9      message_layout =
    ↪ ui_manager.create_ui_object_in(:message_layout_kim,
    ↪ :FXVerticalFrame, layout)
10     @author_label =
    ↪ ui_manager.create_ui_object_in(:message_author,
    ↪ :FXLabel, message_layout,
    ↪ "#{@message_model.from}:" )
11     return message_layout
12   end
13 end
14 end

```

Listing 9.8: Code snippet of the feature part *View* of the feature *Message* adapting the application class *MessageView*.

Figure 9.5 depicts how the adaptation stack evolves when *proceed* is called in the method *create_message_widget* of the feature *Text* in the application class *MessageView*. After installing these two features for the method *create_message_widget*, we obtain the stack as described in Figure 9.5a. The last adaptation of the method comes from the feature *Text*. When the *proceed* statement is called, the stack of the method is empty, as illustrated in Figure 9.5b, after running the pop operations, as depicted on Lines 2-3 of Algorithm 10. Then we activate the previous adaptation, as illustrated on Line 4 of Algorithm 10. This modifies the adaptation stack by adding the version of the feature *Message* as depicted in Figure 9.5c. As a reminder, this feature is the default behaviour and does not thus contain a *proceed* statement for the method *create_message_widget*. Finally, after executing the previous adaptation, we reactivate the last adaptation, as shown on Line 7 of Algorithm 10. This manipulation results to come back in the initial state, as shown in Figure 9.5d.

In this example, we show how the *proceed* mechanism works with two features (a default behaviour and a refinement). When many refinements adapt the default behaviour through the *proceed* mechanism, our algorithm works recursively to handle these longer chains of *proceed* calls.

However a problem can arise if we run the previous method, as depicted on Line 6 of Algorithm 10, on the current instance. Assume programmers define two application classes with hierarchy between these two classes, and the behaviour of a method is build incrementally with a *proceed* mechanism in the superclass. When the *proceed* statement is called, this algorithm activates the previous adaptation and then reactivates the last adaptation in the superclass correctly. However when the previous adaptation is executed on the current instance, the method will be executed in the method of the subclass and not the method of the superclass. With a *super* call, the method of the superclass

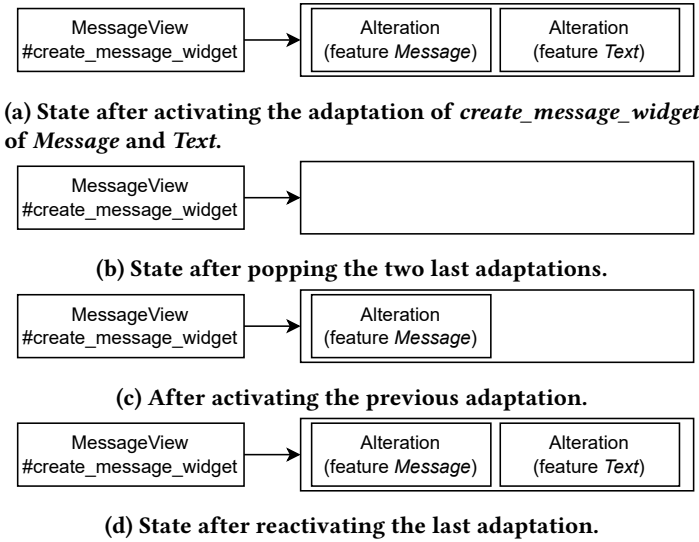


Figure 9.5: Example on how the adaptation stack evolves when the *proceed* mechanism is called for the method `create_message_widget` of the application class `MessageView`.

will be then executed. Therefore an unexpected behaviour is raised since the method of the subclass should not be executed at this moment.

To solve this issue, we ensure that we find the expected method in the hierarchy of the application classes to execute the right method, without executing the submethods. This is performed with the method `look_up_method()` that takes as arguments the current instance, current class and the current method that must be executed, as shown on Line 5 of Algorithm 10. In this `look_up_method` method, we traverse the hierarchy of the application classes for which they have the method as part of their behaviour till we reach the method of the current application class passed as argument and we return this method in order to execute the right method. With this solution, we can easily manage some issues related to the hierarchy of application classes.

9.4 Conclusion

In this chapter we described how we implemented the control flow of our architecture, the dynamic adaptation of the application when features get activated, and the *proceed* mechanism.

We illustrated the implementation of each component in the control flow of the framework's architecture with pseudo-code and explained the design choices we made to implement them.

Then we discussed *unbinding and binding methods mechanism* to adapt

dynamically the application behaviour. We exemplified this mechanism with examples from the messaging system.

Finally we described how we implemented the *proceed* mechanism that allows features to dynamically and incrementally adapt the behaviour of previously installed features.

CHAPTER

10

TOOL SUPPORT

So far we have discussed the implementation choices of the different parts (ENTITIES, MODELLING and ARCHITECTURE) that compose our FBCOP application development framework.

When building FBCOP applications, programmers can get lost in their development and debugging tasks due to the inherent complexity of creating such applications. This complexity comes from the high dynamicity of such systems but also from the exponential number of combinations when designing the contexts, the features and the mapping between them.

To help programmers in their development and debugging tasks, there is a need to provide them with dedicated programming support tools, such as for example visualisation tools. In this thesis, we developed two such visualisation tools, the CONTEXT AND FEATURE MODEL VISUALISER and the FEATURE VISUALISER, which we presented in Chapter 6.

In this chapter, we explain how we extended our implementation architecture to make it easy to connect such tools to it. We start by showing an overview of such connections at a high level. Then we explain how we implemented the TOOL SUPPORT part in the implementation architecture, and how and when it communicates with our supporting tools. In addition to discussing how we can communicate with the visualisation tools, we also introduce a new command line tool (the CONTEXT SIMULATOR) and explain how it can communicate with the implementation framework. Finally we discuss the extensibility and evolution of this TOOL SUPPORT component to support

other kind of tools we have experimented with throughout different master theses.

10.1 Overview of the communication

Figure 10.1 sketches how the implementation architecture can communicate with external supporting tools (whatever technologies and programming languages they use) through a server. With such a separation between the implementation architecture and the tools, we can easily evolve our approach. For example we could rewrite the implementation architecture in another programming language for some reason, without losing our previous effort in the development of the external tools. In addition we could also add new supporting tools easily, to enrich the help we could provide to programmers with a minimum effort in the implementation architecture.

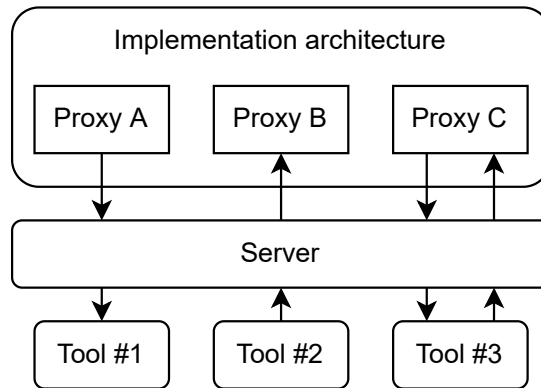


Figure 10.1: Overview of how external tools can communicate with the implementation architecture through a server.

To facilitate the communication between both, we add the notion of communication channels that are maintained by the server. Since we extend the implementation architecture to send and receive information to and from tools, the communication handling is decoupled from the implementation of our FBCOP architecture for maintainability and evolution reasons. For that we add a system of proxies in the implementation framework so that it can communicate easily through communication channels to communicate with the right tool.

To illustrate the different kinds of communication possible we exemplify them with the different tools we implemented for FBCOP programmers.

Tool #1 (the FEATURE VISUALISER tool) helps programmers inspect how the application classes are adapted by the features. When an application

class is adapted by a feature part, proxy *A* sends what feature part has been (de)activated for this class through a dedicated communication channel. The server then relays this information to tool #1 so that it can refresh its visualisation.

Tool #2, a simple command line tool (called the *CONTEXT SIMULATOR* tool), allows programmers to (de)activate some contexts to simulate particular situations to study how their application reacts at runtime. When programmers send a (de)activation command, the server resends the information to the implementation architecture through a communication channel for which the proxy *B* is a client. Then this proxy interprets and executes the command to launch the process to (de)activate the passed contexts.

A more complex tool #3 (the *CONTEXT AND FEATURE MODEL VISUALISER*) provides an overview of the different models and could allow the programmers to interact with it to (de)activate a set of contexts.¹ When the application is launched, the proxy *C* sends the models to the visualisation of the tool #3, via a dedicated communication channel. When programmers click on some contexts and decide to (de)activate them, this information is sent to the proxy *C* via the communication channel so that it can (de)activate the contexts in the application.

These three examples are external tools we already implemented in the FBCOP approach as supporting tools for programmers. We will also discuss other supporting tools that will use this proxy architecture later on Section 10.3.

10.2 **TOOLSUPPORT in the implementation architecture**

Now that we gave an overview of how the implementation architecture can easily communicate with external tools, and inversely, we go deeper in the implementation architecture to better describe how that communication is designed. Figure 10.2 depicts the class diagram of the *TOOLSUPPORT* part.

As the supporting tools must communicate with the implementation architecture through communication channels, each tool has its own framework class. The *CONTEXT AND FEATURE MODEL VISUALISER* tool and the *FEATURE VISUALISER* tool each have their own framework classes *CFMVCommunication* and *FVCommunication*, respectively. The *CONTEXT SIMULATOR* tool also has its own framework class *ContextSimulatorCommunication*. To be easier to understand, we thus have created three different tools with

¹This interaction that programmers could have with this tool to activate and deactivate contexts was not yet implemented in the tool but we could easily imagine this interaction as a future work to improve the tool (see Section 14.3). Here, we mention this interaction only to demonstrate that we can design tools that receive and send information from and to the implementation architecture.

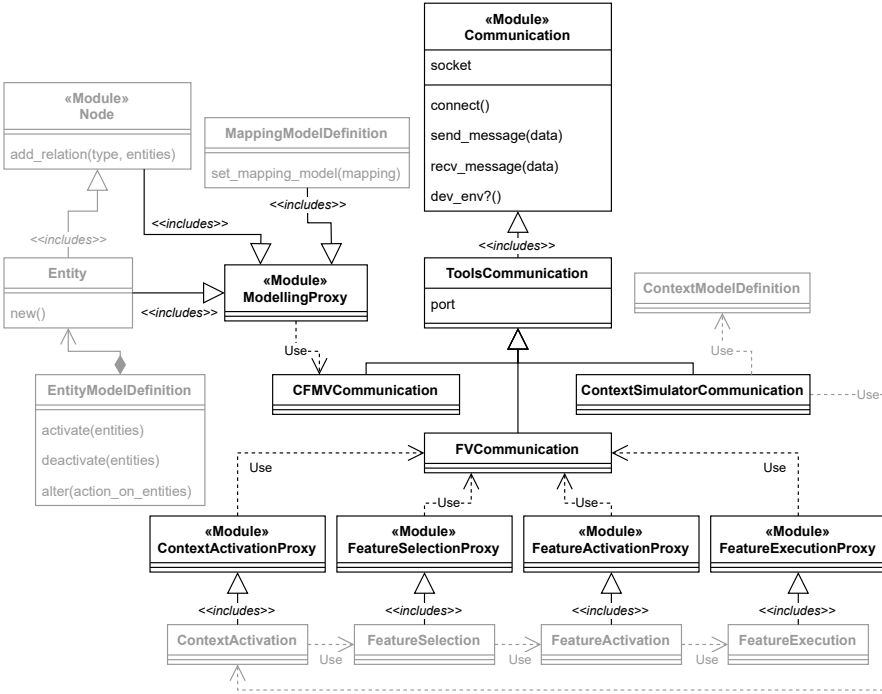


Figure 10.2: Class diagram of the ToolSupport part of our implementation architecture. Some information (constructors, getters and setters) was voluntarily omitted for readability reasons. Some parts are greyed since they are not part of ToolSupport. However we added them to facilitate the understanding of the interactions of the different framework classes.

three different communication channels.

Each of these communication classes defines a *port* on which they must send or listen information. This *port* is an instance variable of their parent framework class **ToolsCommunication**. This parent class itself includes the trait **Communication** that creates the connection with the method **connect()** and that allows to send and receive messages with the **send_message(data)** and **recv_message(data)** methods, respectively.

To help programmers visualise the models and adaptations with our two visualisation tools, we define a specific environment '*dev*'. This allows us to avoid sending messages dedicated to the visualisation tools when programmers are in production or automated testing stage, for example. Therefore we implement the method **dev_env?()** in the trait **Communication** as a guard that serves for our visualisation tools.

Next we will describe how and when the implementation architecture sends/receives information to/from the supporting tools.

10.2.1 CONTEXT AND FEATURE MODEL VISUALISER

When visualising the models of a FBCOP application with the CONTEXT AND FEATURE MODEL VISUALISER tool, all messages are sent through the framework class `CFMVCommunication` via the proxy `ModellingProxy`, as illustrated in Figure 10.2.

When the context and feature model are created, for each entity and each relation, a message is sent to the communication channel to inform the visualisation tool. For the entity, the message is sent when the method *new* is called in the framework class `Entity`. For the relation, the message is sent when the method *add_relation* in the trait `Node` is called.

To visualise the mapping between the contexts and features, the implementation architecture sends a message when the method *set_mapping_model* of the framework class `MappingModelDefinition` is executed. While we inform explicitly the visualisation tool that a mapping between the contexts and features exist, the mapping between the features and application classes is naturally provided to the visualisation tool since each feature knows what application classes it adapt. This design choice allows us to stay coherent with the explicit mapping between the context and feature model and the implicit mapping between the features and their application classes of our FBCOP programming paradigm.

Finally, for each (de)activation of entities called through the methods *activate*, *deactivate* and *alter*, a message is sent to inform the visualisation tool when such a (de)activation occurs.

10.2.2 FEATURE VISUALISER

As Figure 10.2 depicts, each class of the ARCHITECTURE part of the FBCOP framework has a corresponding trait to send information to our FEATURE VISUALISER tool.

For example, The framework class `ContextActivation` has a trait `ContextActivationProxy` that tells the tool which contexts are (de)activated. The framework class `FeatureSelection` has a trait `FeatureSelectionProxy` that sends to the tool what features are (un)selected. As its homologue `ContextActivation`, The framework class `FeatureActivation` with its trait `FeatureActivationProxy` tells the tool which features are (de)activated. Finally the framework class `FeatureExecution`, through the trait `FeatureExecutionProxy`, tells the tool which application class is adapted by which feature part.

10.2.3 CONTEXT SIMULATOR

When programmers execute their FBCOP application and want to simulate some changes in the surrounding environment, they can use the `CONTEXT SIMULATOR` tool to tell the application how the surrounding environment has evolved. To do that, they send a context (de)activation command. This command is first verified syntactically. If valid, the command is sent to the implementation architecture. Otherwise the tool reports an error message to the programmers and reinvites them to write another command.

When the implementation architecture receives a command to (de)activate contexts, the framework class `ContextSimulatorCommunication` first transforms the command into a list of instances of `ActionOnEntity` that describes a list of actions to be performed on those contexts. For that, the contexts are first reified into context objects with the *find_entity_by_name* method of the framework class `ContextModelDefinition`. Then the communication class forwards this list to the `ContextActivation` framework class to (de)activate the passed contexts. However, if an error is detected either due to a wrong name of a context or due to a model satisfiability issue, currently no message is sent back to the `CONTEXT SIMULATOR` tool to inform the programmers. However, this error message appears in the output console of the application. Having better programmer feedback providing more explicit error handling could be an improvement to implement as future work (see Section 14.3).

10.3 Extensibility and evolution

Up to now we have explained how our implementation architecture can communicate with three supporting tools that we have already implemented. With three different tools we already demonstrated that our design allows to easily add new tools that can help programmers in their different tasks when they conceive FBCOP applications. To show that this design is really powerful, we briefly mention some other supporting tools that some master students explored during their master theses to enrich the toolbox we can propose to programmers.

Interaction modality The first example is about the adaptation of the interaction modality that can depend on the surrounding environment in which the application runs. In fact, revisiting our messaging system, assume car drivers who receive messages and want to answer them. In such a situation, drivers may prefer to listen to these messages thanks to a vocal assistant, and answer them with voice commands. Other modalities such as gestures can also be explored. In the messaging system, we could imagine that end-users

could swipe (from right to left) to erase the message when the keyboard is missing. Even if such tools addressing such modalities are not addressed for the programmers, our master-level students illustrated in another case study (see Section 11.4) that the tooling architecture discussed in this chapter can be reused to implement supporting tools dedicated to end-users as well. With such an architecture and design, they could easily reuse their tools in other applications that deal with the same modalities. In addition they can more easily decouple the complexity to implement such tools since they can more easily build their solution incrementally. Finally it allows these external tools to be more independent of the programming language on top of which we build our architecture.

Sensor input In our system architecture, the control flow of our architecture starts by (de)activating contexts. These contexts must be triggered by something in the real world, such as sensors, for example. But how can we catch easily the different information coming from sensors in a maintainable and reusable way? To address this issue, with a master-level student, we propose to define a generic communication channel (following the architecture presented in this chapter) to receive all messages from all interesting sensors. This solution allows to easily receive information from the real world via the server. Even if the programmers must still provide a way to reify this information so that they can manipulate it more easily, the generic behaviour to catch all messages through a single communication channel is already implemented in the implementation architecture.

10.4 Conclusion

In this chapter we first introduced how the implementation architecture can be easily connected with external tools through a server. We exemplified this with our three supporting tools: the CONTEXT AND FEATURE MODEL VISUALISER, the FEATURE VISUALISER and the CONTEXT SIMULATOR.

Then we presented the design choices we made to implement the TOOL-SUPPORT component in the implementation architecture in a maintainable way. We also detailed how each of our three tools are linked to the implementation architecture.

Then we discussed the extensibility and evolution of this component with three other external tools. The examples prove that the design choices we made for external tooling are interesting since we can easily extend our toolbox with other kinds of tools, whatever the technologies used, and integrate them into our FBCOP approach to enrich it.

Part IV

Validation

The previous part illustrated how we have implemented the FBCOP programming framework and explained which design choices we made to build it. We have first explained how the building blocks of our programming framework have been developed. We have then described how we have integrated feature modelling in our programming framework implementation. Next we have also detailed the implementation choices we made to develop the control flow of the architecture, the dynamic adaptation and the *proceed* mechanism. Finally we have also explained how we can extend our approach to easily integrate external tools support in order to help programmers in their development and debugging tasks.

This fourth part aims to validate the FBCOP approach and its design. We will first validate its expressiveness with the help of five case studies: two variants of a *smart messaging system*, a *smart risk information system*, a *smart meetings system* and a *smart city guide* (Chapter 11). Then we will evaluate its design qualities and will discuss its usability with the cognitive dimensions of notations framework (Chapter 12). Finally we will validate the usefulness and usability of the FBCOP approach with real designers and programmers that must create context-oriented applications (Chapter 13).

CHAPTER

11

VALIDATING FBCOP'S EXPRESSIVENESS

In this chapter we will validate FBCOP's expressiveness through five case studies. Each case study will study various aspects of its expressivity.

1. The first case study (Section 11.1) is the *smart messaging system* which we used throughout this dissertation to exemplify different concepts notions and code snippets. We use this study to illustrate the conception of an application following the supporting development methodology, from the *requirements* phase until the *implementation* phase. With this we validate the feasibility of our development methodology, the modelling approach and the implementation framework.
2. The second case study (Section 11.2) is a variant of the *smart messaging system* developed by two master-level students. This also validates the FBCOP modelling and programming framework. We will describe their variant and show some snapshots of it to demonstrate how they succeeded to conceive a context-oriented application with FBCOP.
3. The third case study (Section 11.3) is a *smart risk information system*. We will describe the system and explain its context-feature model. This case study is another demonstration that we can design such dynamic systems with our approach.

4. The fourth case study (Section 11.4) is a *smart meetings system*, also developed by two other master-level students. We will describe their application, show their context-feature model, and illustrate their working prototype with some snapshots. In addition to illustrating FBCOP with another type of smart system, they demonstrate the extensibility of our approach by integrating multimodality interaction in their FBCOP application and our approach.
5. The last case study (Section 11.5) is a *smart city guide* designed by Cardozo and Mens [CM22]. We will briefly describe the system, before displaying its context-feature model. Again, this work serves to show how FBCOP can be used to design another type of context-oriented system.

Finally we will conclude this chapter (Section 11.6).

11.1 Smart messaging system

Throughout this dissertation we exemplified the different FBCOP concepts and code snippets with a *smart messaging system*. Now we will design and implement this *smart messaging system* following our supporting development methodology as explained in Section 3.3. For each phase (*i.e.*, *requirements*, *design* and *implementation*), we will show the results of our design and implementation choices.¹

This case study demonstrates the life-cycle and process of analysing the *requirements*, *designing* and *implementing* a context-oriented application with the FBCOP approach.

Requirements

As a reminder, the *smart messaging system* allows users to exchange messages and is smart in the sense that it can adapt or refine its behaviour depending on some contextual situations.

In the *requirements* phase, we first define the vocabulary of such a system. Table 11.1 shows the lexicon of the *smart messaging system*.

Then we elicit the contexts and features of the system, that are listed in Tables 11.2 and 11.3.

¹As we did not work on *testing* for this dissertation, we will not perform the *testing* phase. This is, in fact, the topic of another PhD dissertation on which is currently being conducted in our research lab. However, we will test our application as end-users to verify if it exhibits the expected behaviour.

Table 11.1: Lexicon of the *smart messaging system*.

Term	Definition
User	Person who uses the <i>smart messaging system</i> .
Chat	Conversation between two or more persons and is composed of messages.
Message	Information that users send or receive. It has an author and a content. More information can be added such as the date and time the message sent.
Message content	Content can be textual or richer such as a picture, an emoticon, a video, a position, or a combination of these elements.

Table 11.2: Contexts of the *smart messaging system*.

Context	Definition
User age	Users can be children, adults or elderly persons.
User disability	Some users may have certain disabilities such as sight problems, hearing problems and so on.
User activity	Users can be either occupied (<i>i.e.</i> , in a meeting or driving) or available.
Device	Users may run the system on their smartphone, car dashboard or desktop.
Internet connection	Users may or may not have an Internet connection. Possible Internet connections are Wi-Fi, cellular or both.
Bluetooth	Users may have a Bluetooth connection so that their smartphone can communicate with their car dashboard, or with other devices.
Positioning	Users may activate their GPS to obtain their exact location.
Noise	Users may be in a quiet place, a place where the ambient noise is acceptable or in a very noisy place.

To simplify the case study, for this prototype we decided not to include some other useful features, such as editing a chat (*e.g.*, its name), removing a user of a chat, or deleting a message.

Finally we draw some wireframes of the *smart messaging system* for which we illustrate the scenario for each:

- Figure 11.1 illustrates what the desktop version should look like when

Table 11.3: Features of the *smart messaging system*.

Term	Definition
Send	Users can send messages over the Internet.
Store	Users may write messages that will be stored locally when no Internet connection is sensed.
Receive	Users may receive messages when they have an Internet connection.
List chats	Users may see the list of all ongoing or past chats.
Read chat	Users may open a chat to read all its messages of this chat.
Create chat	Users may create new chats with other selected users. For each new chat, users must provide a name to identify it.
List members	Users may see the members of a chat.
Create message	Users may write a new message in a chat.
Message	Messages have an author.
Message content	Messages can be of type text, picture, emoticon, and more.
Sending date	Messages have a date when the message was sent.
User preferences	Users may provide their age, preferences and disabilities (e.g. sight problems).
Profile picture	Users may have a profile picture.
Text sizing	The size of the text is either normal or enlarged.
Notify	When users receive new messages, notifications are sent to the users through a sound alarm or vibration. Notifications may be muted.
Layout	The main layout is stacked or side-by-side.

users have a cellular connection. The layout is a side-by-side view where the chats list is on the left part of the layout and the selected chat is displayed on the right part of the layout. In this example, Benoît uses his desktop and opens his direct chat messages with Kim.

- Figure 11.2 depicts what the smartphone version should look like when users have a cellular connection. Taking the same example of Figure 11.1, we show its smartphone version. On the left, a first view shows the list of all chats. When a chat is selected, a new view replaces the previous one to display the chat content. This other view is on the right of Figure 11.2. To let the users come back to the chats list, a ‘back’ button is added on the top left of the view.

Parrain	<u>Kim:</u> Can we discuss tomorrow about the LINFO2252 course? <u>Benoît:</u> Of course. At 11 AM? <input type="text" value="Insert text..."/> <input type="button" value="Send"/>
Kim	
Bruno	
INGI	

Figure 11.1: Wireframe showing what the desktop version of a *smart messaging system* should look like.


Parrain	 <u>Kim:</u> Can we discuss tomorrow about the LINFO2252 course? <u>Benoît:</u> Of course. At 11 AM? <input type="text" value="Insert text..."/> <input type="button" value="➤"/>
Kim	
Bruno	
INGI	

Figure 11.2: Wireframe showing what the smartphone version of a *smart messaging system* should look like.

- Now assume users launch the *smart messaging system* on their desktop. With a Wi-Fi or ethernet connection, richer messages can be sent such as a picture, an emoticon or a video. Figure 11.3 illustrates this scenario. In this example, Benoît is on his laptop. After sending a picture, he writes a new message.
- However, when the Internet connection is lost, users must be informed that their messages will be sent only later when a new connection is sensed. Figure 11.4 shows such a case. In this example, Benoît wants to

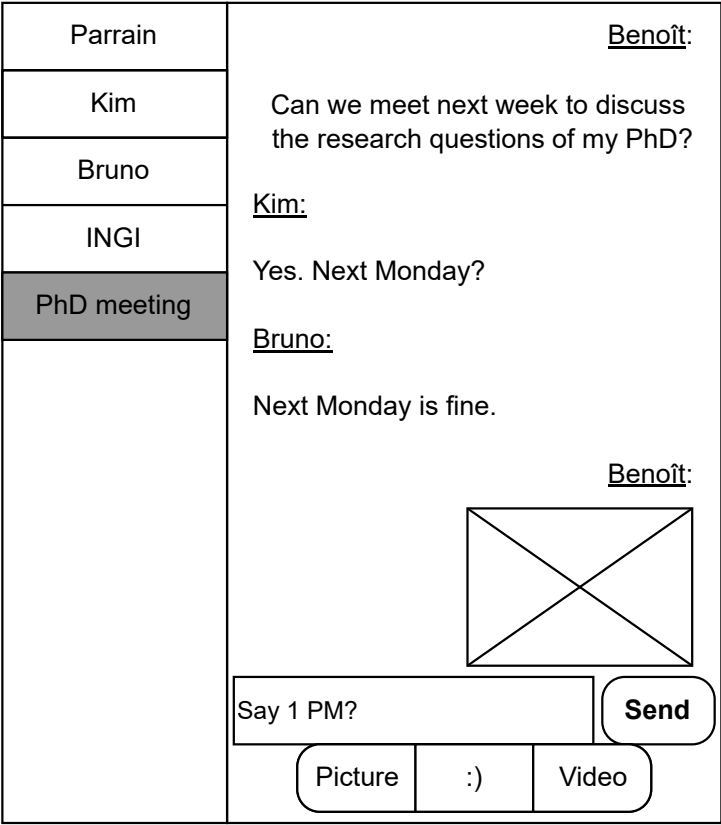


Figure 11.3: Wireframe showing what the desktop version of a *smart messaging system* should look like when users have a Wi-Fi connection.

send a message to Kim. But he temporarily lost his Internet connection, so a new message appears in the chat to inform him that the message will be sent later.

More wireframes were conceived during our analysis of the *smart messaging system*. However we decided not to show all of them here since most were similar to those we displayed here.

Design

After having completed the *requirements* phase, we now design the context-feature model and sketch a class diagram which will serve as skeleton for the application classes and their interactions. We also mention what features adapt what application classes.

Parrain	<u>Kim:</u> Can we discuss tomorrow about the LINFO2252 course? <u>Benoît:</u> Of course. At 11 AM? <i>!!! The message will be sent when an Internet connection will be sensed.</i> <input type="text" value="Insert text..."/> <input type="button" value="Send"/>
Kim	
Bruno	
INGI	

Figure 11.4: Wireframe showing what the desktop version of a *smart messaging system* should look like when users want to send messages while they do not have a Wi-Fi connection.

Context-feature model Figure 11.5 depicts the context-feature model we draw for this system based on the list of contexts and features we elicited after the *requirements* phase. In Figure 11.5, the context (resp. feature) model is on the left (resp. right) and the mapping model is drawn with a table between both models. To keep the image readable, we deliberately omitted many arrows in the mapping model from context model to feature model.

As the description of this case study has been already detailed (see Section 1.5), we will no longer explain this context-feature model.

Applications classes, interactions and features Next we draw a kind of conceptual class diagram that shows the application classes with their interactions of the *smart messaging system*, as shown in Figure 11.6. Since the full behaviour is separated into features, we will also indicate what features may adapt each application class instead of precisising the attributes and methods in each application class.

The main application class is the `SmartMessagingSystem`. This application class may be adapted by the features *Layout*, *Stacked* and *Side-by-side*. *Layout* is a structural implementation on which *Stacked* and *Side-by-side* rely on to implement the master/detail pattern. The features *Normalised* and *Enlarged* may also adapt this application class and is in charge of adjusting the text sizing. `SmartMessagingSystem` is also adapted by the features *Create chat* and *List chats* to allows users to create a new chat and displays the list all chats to the users in the main window. As `SmartMessagingSystem` is in charge of the menu bar of the application, the different features (*Name*,

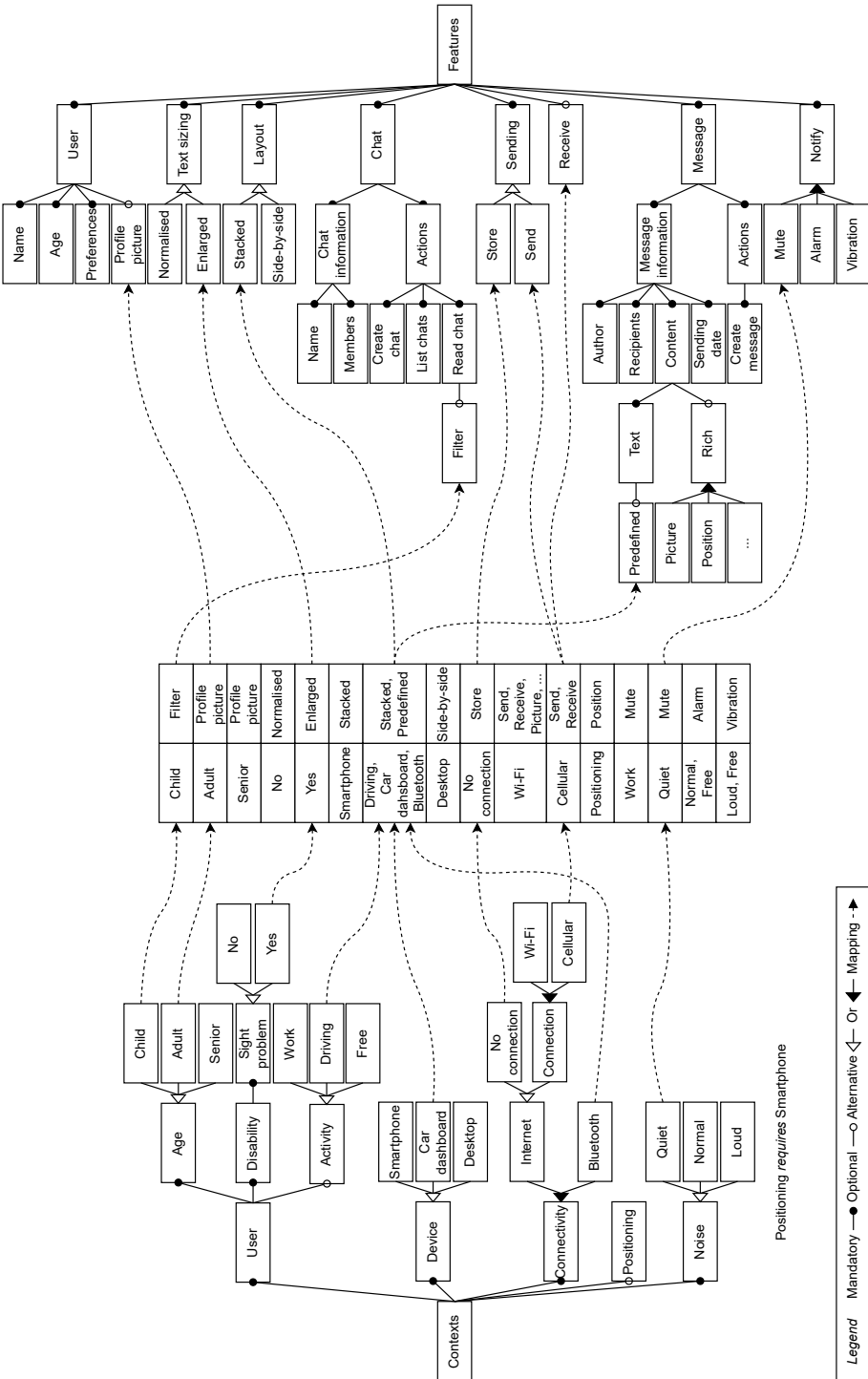


Figure 11.5: Context-feature model of the *smart messaging system*. Some mappings arrows are voluntarily omitted for readability reasons.

Age, *ProfilePicture* and *Preferences*) that compose the menu also adapt this application class. (To simplify, we decided to create a menu item for each attribute that the user can modify, but nothing prevents us to merge them to create a menu item dedicated to the user profile in which they can update their information.) To know the current user, *SmartMessagingSystem* has an instance variable of the application class *UserModel*.

The *UserModel* is adapted by the features *Name*, *Age* and *Preferences* and may be adapted by the feature *Profile picture*. Its view, *UIView* is adapted by the same features and retains an instance variable to its model, an instance of *UserModel*.

The application class *ChatModel* is used by *SmartMessagingSystem* to create a chat or display the chats list. This *ChatModel* is adapted by the features *Chat information*, *Name*, and *Filter*. The feature *Chat information* is more a structural behaviour that provides the default behaviour of the application class *ChatModel*. *Name* adds behaviour to this class to identify the chat. *Messages* also adapts *ChatModel* to add an instance variable that contain all the messages (instances of the application class *MessageModel*). Nevertheless, *Filter* refines the behaviour of *Read chat* by censoring unappropriate words or sentences in *ChatModel*. Finally, the feature *Members* adds an instance variable in *ChatModel*. As for the user view and its model, *ChatView* is the view of *ChatModel* and has a reference to its model. This *ChatView* class is also adapted by the feature *Chat information* to add the structural default behaviour, the features *Stacked* and *Side-by-side* that read the chat, the feature *Messages* and *Members* that allow to refresh the view when a new message is created or a new member is added. The feature *Read-Chat* also adapts this application class to display all the messages of the current chat. Finally, the feature *CreateMessage* adds the user interaction to create a new message and this feature is refined by the different types of message a user can write (i.e., the features *Text*, *Predefined*, *Picture*, *Position* and ...). The *ChatView* class uses the application class *UIView* to show the members of the current chat model.

The application class *MessageModel* is naturally adapted by the features *MessageInformation* and *Text* that implements a structural default behaviour and the default textual content that a message can have, respectively. In addition to *Text*, this class may also be adapted by the features *Predefined*, *Picture*, *Position* and ... that refine the types of content a message can have. For each message, a message must also have an author, an instance of the application class *UserModel* and this behaviour is added by the features *Author*. The application class *MessageModel* is also adapted by the features *Sending date* to have a sending date in each message. Finally, the feature *Filter* also adapts this application class to hide inappropriate keywords or sentences in the message itself. Again, its view, i.e., the application class *MessageView*, retains

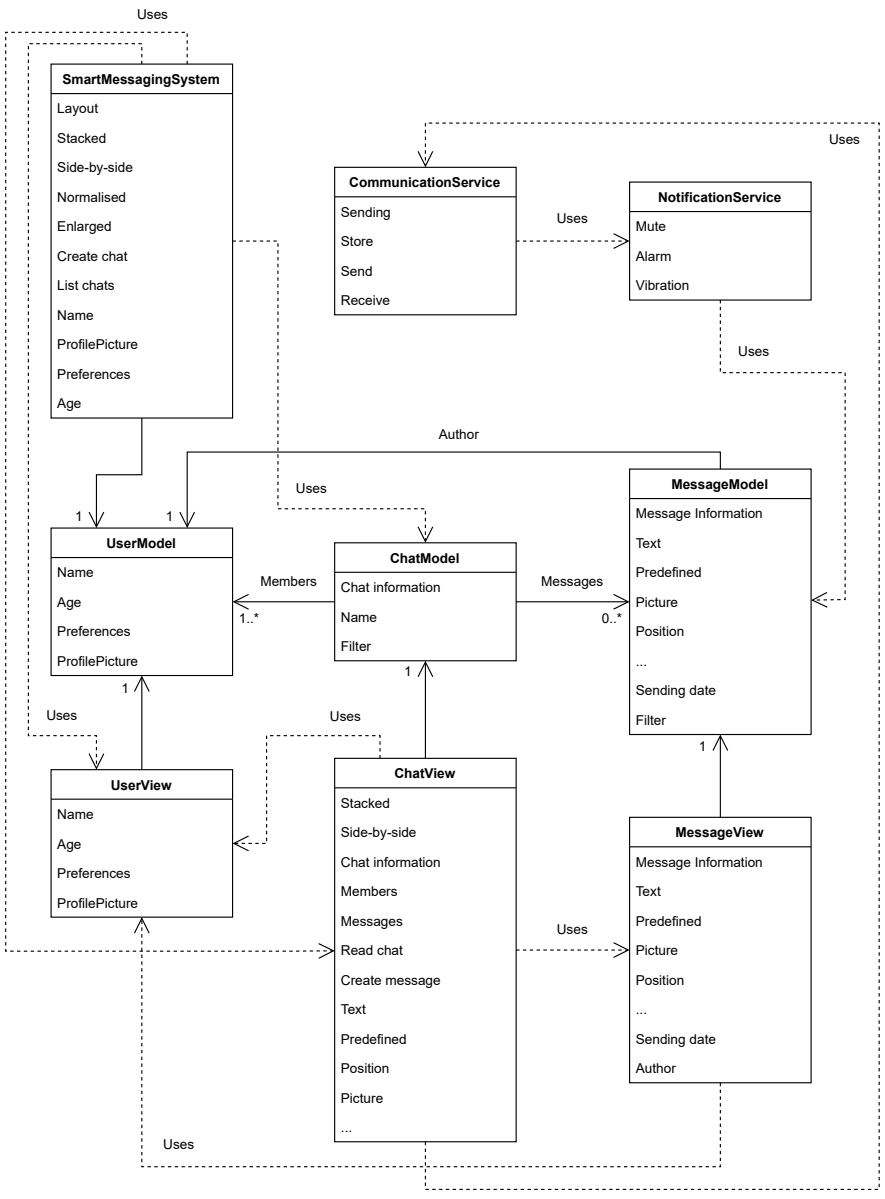


Figure 11.6: Kind of class diagram that shows the application classes and their interactions of the *smart messaging system*. In each application class or interaction, we indicate what features (may) adapt it.

its model (instance of `MessageModel`) and is adapted by almost all the same features that *MessageModel* (i.e., *MessageInformation*, *Text*, *Predefined*, *Picture*, *Position*, *Author*, *Sending date* and ...). As for `ChatView`, `MessageView` uses `UIView` to instantiate the author of the message.

When users want to create a message, the feature *Create message* in the application class `ChatView` is executed to first create a `MessageModel` instance, then to set the different information of the message and finally to send it through the application class `CommunicationService`. This explains why `ChatView` uses `CommunicationService`. As its name suggests, `CommunicationService` is adapted by *Sending* suggesting structural code for its child features and may be adapted by the feature *Store*, *Send* and *Receive* to store, send and receive messages.

On receipt of new messages, `CommunicationService` uses the application class `NotificationService` to inform the users if this latter class is adapted by the features *Alarm* or *Vibration*. The feature *Mute* may also adapt `NotificationService` when it is needed. When a new message is received, this new message is directly reified into a `MessageModel` instance.

The two application classes `CommunicationService` and `NotificationService` are naturally singleton classes since we do not need two different instances of each of these services.

Implementation

Now that we have designed the context-feature model and drew a kind of class diagram to visualise which application classes we need, how these classes interact together, and how they are adapted by what features, we will show some code snippets to illustrate the implementation of the *smart messaging system*. For length reasons, we only show how we implemented the version for smartphones with a few selected of features. The full source code is available in the repository.

Listing 11.1 depicts the context model declaration in which the contexts *Smartphone* and *Desktop* are declared under their parent context *Device*.

```

1 class AppContextModelDeclaration <
    ↪ ContextModelDeclaration
2   include Singleton
3
4   def initialize ()
5     super ()
6     _define_device_context ()
7     # More code
8     @root_context.relation :Mandatory, [@device]
9   end
10
```

```

11 private
12
13 def _define_device_context()
14   abstract_context :@device, 'Device'
15   context :@desktop, 'Desktop'
16   context :@smartphone, 'Smartphone'
17   @device.default @smartphone
18   @device.relation :Alternative, [@desktop,
19     ↪ @smartphone]
19 end
20 # More code
21 end

```

Listing 11.1: Snippet of the context model declaration of the *smart messaging system*.

A part of the feature model declaration is shown in Listing 11.2.

```

1 class AppFeatureModelDeclaration <
2   ↪ FeatureModelDeclaration
3
4   include Singleton
5
6   def initialize()
7     super()
8     _define_layout_features()
9     _define_chat_features()
10    _define_message_features()
11    # More code
12    @root_feature.relation :Mandatory, [@layout, @chat,
13      ↪ @message]
14  end
15
16 private
17 def _define_layout_features()
18   feature :@layout, 'Layout', [:SmartMessagingSystem]
19   feature :@stacked, 'Stacked',
20     ↪ [:SmartMessagingSystem, :ChatView]
21   feature :@side_by_side, 'SideBySide',
22     ↪ [:SmartMessagingSystem, :ChatView]
23   @layout.relation :Alternative, [@stacked,
24     ↪ @side_by_side]
25 end
26
27 def _define_chat_features()
28   abstract_feature :@chat, 'Chat'
29   _define_chat_information_features()
30   _define_chat_actions_features()
31   @chat.relation :Mandatory, [@chat_information,
32     ↪ @chat_actions]
33 end

```

```

27
28 def _define_chat_information_features()
29     feature :@chat_information, 'ChatInformation',
    ⇨ [:ChatModel, :ChatView]
30     feature :@chat_name, 'Name', [:ChatModel]
31     feature :@chat_messages, 'Messages', [:ChatModel,
    ⇨ :ChatView]
32     @chat_information.relation :Mandatory, [@chat_name,
    ⇨ @chat_messages]
33 end
34
35 def _define_chat_actions_features()
36     abstract_feature :@chat_actions, 'ChatActions'
37     feature :@list_chats, 'ListChats',
    ⇨ [:SmartMessagingSystem]
38     feature :@read_chat, 'ReadChat', [:ChatView]
39     feature :@filter_messages, 'FilterMessages',
    ⇨ [:ChatModel, :MessageModel]
40     @chat_actions.relation :Mandatory, [@list_chats,
    ⇨ @read_chat]
41     @chat_actions.relation :Optional, [@filter_messages]
42 end
43
44 def _define_message_features()
45     abstract_feature :@message, 'Message'
46     _define_message_actions_features()
47     _define_message_information_features()
48     @message.relation :Mandatory, [@message_actions,
    ⇨ @message_information]
49 end
50
51 def _define_message_actions_features()
52     abstract_feature :@message_actions, 'MessageActions'
53     feature :@create_message, 'CreateMessage',
    ⇨ [:ChatView]
54     @message_actions.relation :Mandatory,
    ⇨ [@create_message]
55 end
56
57 def _define_message_information_features()
58     feature :@message_information,
    ⇨ 'MessageInformation', [:MessageModel,
    ⇨ :MessageView]
59     feature :@author, 'Author', [:MessageModel,
    ⇨ :MessageView]
60     _define_content_message_features()
61     @message_information.relation :Mandatory, [@author,

```

```

    ↪ @content_message ]
62 end
63
64 def _define_content_message_features ()
65     abstract_feature :@content_message , 'Content'
66     feature :@text , 'Text' , [:MessageModel ,
    ↪ :MessageView , :ChatView ]
67     @content_message.relation :Mandatory , [@text]
68 end
69 # More code
70 end

```

Listing 11.2: Snippet of the feature model declaration of the *smart messaging system*.

In this feature model, we declare the three main features: *Layout*, *Chat* and *Message*.

The feature *Layout* has two child features *Stacked* or *SideBySide*. All these three features adapt the application class *SmartMessagingSystem* to implement the main layout and more precisely the master part. The detail part is implemented in the application class *ChatView* by the child features (i.e., *Stacked* or *SideBySide*).

The feature *Chat* has two child features *Chat Information* and *Chat Actions*. *Chat Information* is refined by the features *Name* and *Messages*. These three features adapt the application class *ChatModel*, and the features *Chat Information* and *Messages* adapt the application class *ChatView*. *Chat Actions* is an abstract feature that has three child features *Lists Chats*, *Read Chat* and *Filter*. *Lists Chats* adds a behaviour in the class *SmartMessagingSystem* as entry points to visually list all the chats. *Read Chat* is a feature adapting the application class *ChatView*. *Filter* adapts the application classes *ChatModel* and *MessageModel*.

The feature *Message* is again specialised with its actions (*Message Actions*) and information (*Message Information*). *Message Actions* has the feature *CreateMessage* that adapts *ChatView* to add the user interaction in order to create a new message. *Message Information* contains a feature *Author* and a feature *Content* that is refined by the feature *Text*. The feature *MessageInformation* adapts the application classes *MessageModel* and *MessageView* to implement the structural code on which we can build a message and its view. The feature *Author* refines the behaviour of these classes by adding the concept of author in a message. *Text* adapts logically the application classes *MessageModel* and *MessageView* to add a textual content in a message. This feature also adapts the application class *ChatView* to add the interaction to create a new textual message.

Almost all the features we presented in Listing 11.2 are mandatory. Nevertheless, the kind of layout must be different depending on the device. List-

ing 11.3 illustrates how we declared the fact that the context *Desktop* triggers the feature *SideBySide* and the context *Smartphone* triggers the feature *Stacked*.

```

1 class AppMappingModelDeclaration <
    ↳ MappingModelDeclaration
2   include Singleton
3
4   def initialize ()
5     contexts = AppContextModelDeclaration.instance
6     features = AppFeatureModelDeclaration.instance
7     @mapping = {
8       [ contexts.desktop () ] => [ features.side_by_side () ],
9       [ contexts.smartphone () ] => [ features.stacked () ],
10      # More mapping relations
11    }
12  end
13 end

```

Listing 11.3: Snippet of the mapping model declaration of the *smart messaging system*.

Listing 11.4 depicts the main application class of the *smart messaging system*.

```

1 class SmartMessagingSystem
2   class << self
3     include CodeExecutionAtLaunchTime
4
5     attr_reader :instance
6
7     def run ()
8       if @instance.nil? ()
9         app = UI::UIManager.instance.create_app () do
10           |app|
11             @instance = SmartMessagingSystem.new(app)
12           end
13         app.run ()
14       else
15         return @instance
16       end
17     end
18   end
19
20   def initialize (app)
21     @main_window =
22       ↳ UI::UIManager.instance.create_main_container (app,
23       ↳ "Smart messaging application", :width => 500,
24       ↳ :height => 500)
25     # More code

```

```

23     @main_window.show(PLACEMENT_SCREEN)
24   end
25   # More code
26 end

```

Listing 11.4: Snippet of the application class `SmartMessagingSystem` of the *smart messaging system*.

Listings 11.5 and 11.6 show what the skeleton of the application classes `ChatModel` and `ChatView` look like. These classes are empty because the default behaviour is designed in the features as the variability points.

```

1 class ChatModel
2   include Observable
3 end

```

Listing 11.5: Snippet of the application class `ChatModel` of the *smart messaging system*.

```

1 class ChatView
2 end

```

Listing 11.6: Snippet of the application class `ChatView` of the *smart messaging system*.

The application classes `MessageModel` and `MessageView` follow the same empty structure that this shown in Listings 11.5 and 11.6.

Listing 11.7 illustrates the *Stacked* feature definition. This definition consists of two feature parts: *MainLayout* and *ReadChat*. The *MainLayout* feature part adds its behaviour in the application class `SmartMessagingSystem` to create a vertical frame that will be used later when we must list all the chats. The *ReadChat* is the default behaviour in the application class `ChatView` when users read a chat. This default behaviour depends on the device. As a reminder, when users read a chat, we must replace all the UI objects by others for smartphones whereas only a part of the user interface must be replaced for desktop.

```

1 module Stacked
2   set_feature_part_order :MainLayout, :ReadChat
3
4   module MainLayout
5     can_adapt :SmartMessagingSystem
6
7     def list_all_chats()
8       ui_manager = UI::UIManager.instance
9       return
10      ↪ ui_manager.create_ui_object_in(:list_of_chats,
11      ↪ :FXVerticalFrame, @main_window, :opts =>
12      ↪ LAYOUT_FILL)

```

```

10     end
11 end
12
13 module ReadChat
14     can_adapt :ChatView
15
16     def read_chat()
17         ui_manager = UI::UIManager.instance
18         _content_layout =
19         ↪ ui_manager.find_ui_object(:content_layout)
20         ui_manager.remove_all_ui_children(_content_layout)
21         @chat_layout =
22         ↪ ui_manager.create_ui_object_in(:chat_layout ,
23         ↪ :FXVerticalFrame , _content_layout , :opts =>
24         ↪ LAYOUT_FILL)
25         back_button =
26         ↪ ui_manager.create_ui_object_above(:back_button ,
27         ↪ @chat_layout , :FXButton , '<- Back')
28         back_button.connect(SEL_COMMAND) do
29         ui_manager.remove_all_ui_children(_content_layout)
30         SmartMessagingSystem.instance.list_chats()
31     end
32 end
33 end
34 # More code
35 end

```

Listing 11.7: Snippet of the feature *Stacked* adapting the application classes *SmartMessagingSystem* and *ChatView* of the *smart messaging system*.

Listing 11.8 depicts the implementation of the feature *ListChats* with two features parts. Its feature part *Model* serves to fetch all the chats in *SmartMessagingSystem*. Its feature part *View* refines the default behaviour installed previously by the feature part *MainLayout* of the feature *Stacked* in the application class *SmartMessagingSystem*. When calling the *list_all_chats* method, it executes first this method of the feature *Stacked*. Because the first statement is a *proceed* call (Line 18), the system calls the previous version of this method (*i.e.*, the version of the method of the feature part *MainLayout* of *Stacked*). This execution of this *proceed* call results in the creation of a vertical frame (see Lines 7-10 of Listing 11.7). After the *proceed* call is over, the control flow returns to the version of the method of the feature part *View* of the feature *ListChats*. It then creates the different UI objects that will be attached to this vertical frame. This separation of concern for the method *list_all_chats* allows having a default behaviour that differs from the type of device and only one implementation to list all chats.

```

1 module ListChats
2   set_feature_part_order :Model, :View
3
4   module Model
5     can_adapt :SmartMessagingSystem
6     set_prologue :load_chats
7
8     def load_chats()
9       @chat_models = _create_chat_models()
10    end
11  end
12
13  module View
14    can_adapt :SmartMessagingSystem
15    set_prologue :list_chats
16
17    def list_chats()
18      _list_chats_layout = proceed()
19
20      ui_manager = UI::UIManager.instance
21      @chat_models.each do
22        |chat_model|
23          chat_view = ChatView.new(chat_model)
24          chat_view.main_window = @main_window
25          chat_id = chat_model.name.to_sym
26          chat_name_label =
27            ↳ ui_manager.create_ui_object_in(chat_id,
28            ↳ :FXButton, _list_chats_layout, chat_model.name,
29            ↳ :opts => LAYOUT_FILL_X|JUSTIFY_LEFT, :padLeft =>
30            ↳ 10, :padRight => 10, :padTop => 20, :padBottom =>
31            ↳ 20)
32            chat_name_label.backColor = '#FFFFFF'
33
34            chat_name_label.connect(SEL_COMMAND) do
35              chat_view.read_chat()
36            end
37
38            separator_id = "#{chat_model.name}-sep".to_sym
39            ui_manager.create_ui_object_in(separator_id,
40            ↳ :FXHorizontalSeparator, _list_chats_layout)
41          end
42        end
43      end
44    end
45  end
46 end

```

Listing 11.8: Snippet of the feature *ListChats* adapting the application class *SmartMessagingSystem* of the *smart messaging system*.

When the system creates the UI object of a chat name by displaying the name of the chat, it assumes that the system has already activated the feature *Name* in the application class `ChatModel`. The implementation of the feature *Name* is shown in Listing 11.9.

```

1 module Name
2   module Model
3     can_adapt : UserModel, [: ChatModel
4
5     attr_accessor :name
6
7     def initialize ()
8       proceed ()
9       @name = " "
10    end
11  end
12  # More code
13 end

```

Listing 11.9: Snippet of the feature *Name* adapting the application class `ChatModel` of the *smart messaging system*.

As we can observe, this feature *Name* can also adapt the application class *UserModel* (Line 3 of Listing 11.9). This allows to define only one feature definition for several adaptations in different application classes. Therefore, this improves the reusability of this feature definition. However, programmers must declare in the feature model declaration for which application class the feature definition must adapt, as illustrated in Listing 11.10. While the first feature declaration adapts the application class *UserModel* with the feature *Name*, the other declaration adapts the application class *ChatModel* with the same feature.

```

1 feature :@username, 'Name', [: UserModel]
2
3 feature :@chat_name, 'Name', [: ChatModel]

```

Listing 11.10: Snippet of the feature model declaration to illustrate we can reuse the same feature definition in the *smart messaging system*.

Listing 11.11 exposes the implementation of the feature *ReadChat* that adapts the application class `ChatView`. In the feature part *View*, the method *read_chat* refines the default behaviour installed by the feature part *ReadChat* of the feature *Stacked* (or *SideBySide*). Again, we use the *proceed* mechanism to build this view.

```

1 module ReadChat
2   module View
3     can_adapt : ChatView

```

```

4
5     def read_chat()
6         proceed()
7         @chat_model.messages.each do
8             |message_model|
9             message_view = MessageView.new(message_model)
10            message_view.create_message_layout(@chat_layout)
11        end
12        self.create_new_message_layout()
13    end
14 end
15 end

```

Listing 11.11: Snippet of the feature *ReadChat* adapting the application class *ChatView* of the *smart messaging system*.

With this design and implementation of how we built the view to list all chats or this view to read a chat, we can implement the features *ListChats* and *ReadChat* so that they are independent of the type of device, since they refine the default behaviour added by the feature corresponding to the layout.

As already explained above, the feature *MessageInformation* adds the structural code of the application classes *MessageModel* and *MessageView*, as shown in Listing 11.12.

```

1 module MessageInformation
2     module Model
3         can_adapt :MessageModel
4
5         def initialize()
6             end
7         # More code
8     end
9
10    module View
11        can_adapt :MessageView
12
13        attr_accessor :main_window
14
15        def initialize(message_model)
16            @message_model = message_model
17            @message_model.add_observer(self)
18        end
19
20        def create_message_layout(layout)
21            ui_manager = UI::UIManager.instance
22            return ui_manager.create_ui_object_in(:Message,
23            ↪ :FXVerticalFrame, layout, :opts => LAYOUT_FILL_X)
24        end
25    end
26 end

```

```

24     # More code
25 end
26 end

```

Listing 11.12: Snippet of the feature *MessageInformation* adapting the application class *MessageView* of the *smart messaging system*.

Listing 11.13 shows the implementation of the feature *Text* that adapts the application classes *MessageModel* and *MessageView*. In the feature part *Model* that adapts *MessageModel*, we only add the instance variable *@text*. In fact, the full behaviour is build incrementally with the different features with the *proceed* mechanism. The feature part *View*, adapting *MessageView*, aims to only display the text of the message in the layout creating originally by the feature part *View* of the feature *MessageInformation*. Again, we use the *proceed* mechanism to create incrementally this user interface in order to have a good separation of concerns. In addition, this feature *Text* has a last feature part to refine the user interaction to create a textual message. (We omitted this last feature part for conciseness.)

```

1 module Text
2   module Model
3     can_adapt :MessageModel
4     attr_accessor :text
5     def initialize()
6       proceed()
7       @text = nil
8     end
9     # More code
10  end
11
12  module View
13    can_adapt :MessageView
14    attr_accessor :main_window
15    def create_message_layout(layout)
16      _layout = proceed(layout)
17      ui_manager = UI::UIManager.instance
18      ui_manager.create_ui_object_in(:MessageText,
19      ↪ :FXLabel, _layout, @message_model.text, :opts =>
20      ↪ self.position_message_content())
19      return _layout
20    end
21  end
22 end

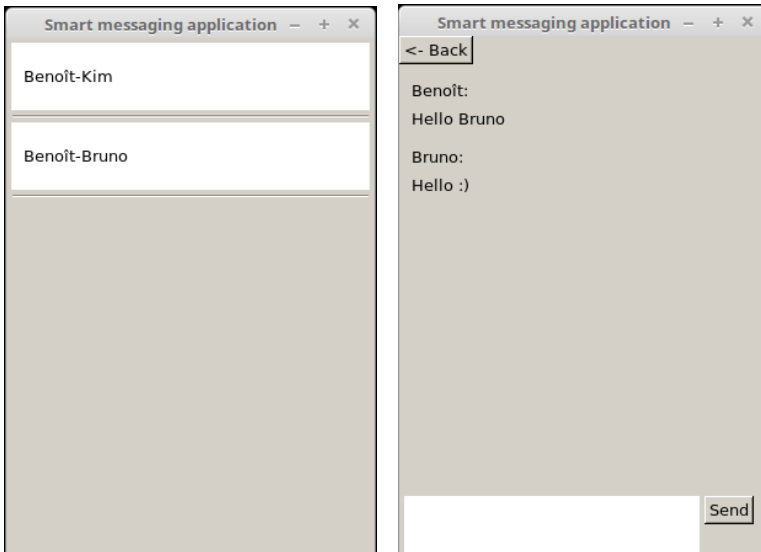
```

Listing 11.13: Snippet of the feature *Text* adapting the application classes *MessageModel* and *MessageView* of the *smart messaging system*.

Execution

Now that we implemented the *smart messaging system*, we will show some snapshots of its execution. (These snapshots are from an intermediate version, but a more complete version is available by running the latest source code.)

Figure 11.7 shows how the application behaves for users use their smartphone. Assume Benoît launches the application on his smartphone. As his smartphone has a small screen, he first sees his list of chats, as depicted in Figure 11.7a. When he opens a chat (here, the chat identified with the name 'Benoît-Bruno'), he sees all the messages of this chat, as shown in Figure 11.7b. If Benoît wants to return on its list of chats, he can click on the '<- Back' button on top of the window.



(a) Snapshot of the master part that displays all the chats. (b) Snapshot of the detail part that displays a chat.

Figure 11.7: Execution of our implementation of the *smart messaging system* when users use their smartphone.

Later in the day, Benoît turns on his desktop and launches the application. Figure 11.8 illustrates what the application looks like on his desktop.

11.2 Another smart messaging system

In the previous section we have conceived our own smart messaging system.

Now we will show a variant *smart messaging system* designed and implemented by two master-level students, Julien Lienard and Céline Nardi, during

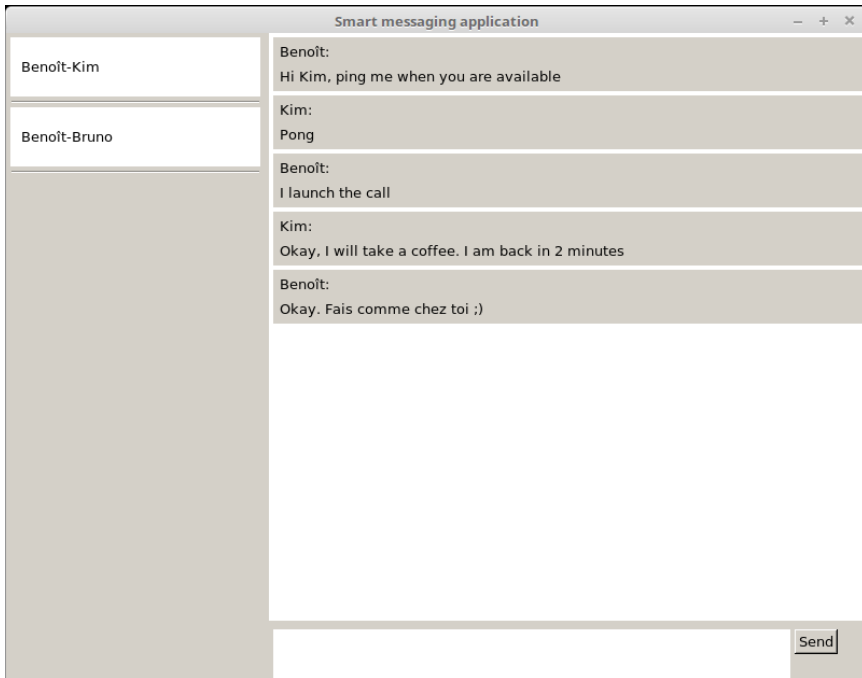


Figure 11.8: Execution of our implementation of the *smart messaging system* when users use their desktop.

a user study in which they participated (see Section 13.5).

As we already detailed a design and implementation of the *smart messaging system*, we will only describe their interpretation of a *smart messaging system*, and then we will show an execution of their application with some snapshots.

This case study aims to demonstrate that other designers and programmers can use the FBCOP approach to create context-oriented applications.

Description

In their *smart messaging system*, users can send and receive messages to and from other users.

The default type of message are text. Depending on the users' activity, other types of message can be enabled such as audio, video or picture. When users are in a meeting or in a library, users can also send and receive pictures and videos, but cannot send and receive audio messages. If the users are driving, they can only send predefined or audio messages and receive audio messages. But when users are available, they can send and receive any type of message.

On receipt of a new message, the system notifies textually the users. In addition, it can also inform the users in a different way according to their activities. They can be notified with a vibration while they are in a library. When users are driving or are available, a sound is played and a vibration is emitted.

Depending on their users' activities, different inputs are enabled. For example, the voice is allowed when users drive while a keyboard is shown when users are in a library or during a meeting. If the user is available, the voice and the keyboard are allowed as input.

Depending on the time of day, the application can be either in a light mode or in a dark mode. During the night, the application is in a dark mode to make reading less painful in nighttime.

Finally, the type of keyboard can be different depending on the language set in the user's preferences. For example, the *azerty* keyboard is available when users set a Latin language while the *qwerty* keyboard is for users having set a Germanic language in their preferences.

Execution

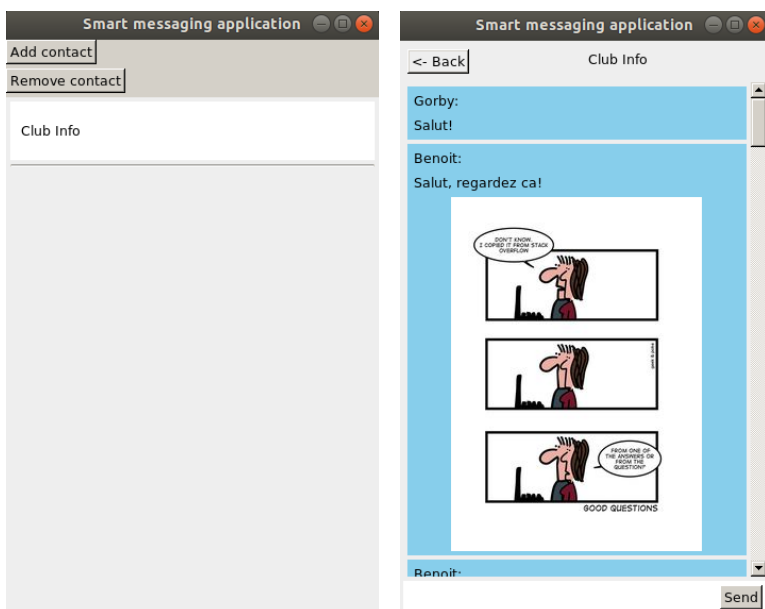
Now we have a rough idea of their design of a *smart messaging system*, we will illustrate its execution with some snapshots in Figure 11.9. These snapshots have been provided to us by these two students.

Assume a user called Benoît uses this application in the daytime during his free time. After launching the application, Benoît sees his lists of chat, as depicted in Figure 11.9a. Then he clicks to open the chat called 'Club Info' because he must check an information about the next meeting they have for his club. Figure 11.9b shows what Benoît sees when he opens this chat. After finding his information, he wants to return to the main window (*i.e.*, the window that display the list of chats) and clicks on the '<- Back' button on top of the window.

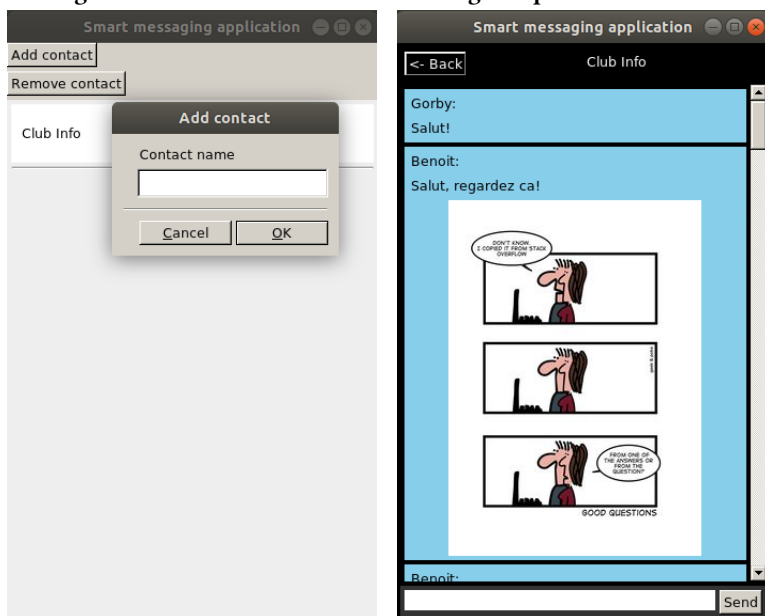
Later in the day, Benoît wants to send a message to his advisors about a new question he has for is research. Then he adds his advisors with the 'Add contact' button on top of the window of Figure 11.9a. A dialog box is thus displayed to add a new contact, as shown in Figure 11.9c.

During the evening, Benoît wants to share some new information with the 'Club Info', then he opens again this chat. As he uses the application during the nighttime, the application switches the mode to dark mode, as illustrated in Figure 11.9d.

This scenario illustrates only a subset of what their application can do.



(a) Snapshot of their application showing the list of chats. (b) Snapshot of their application showing an opened chat.



(c) Snapshot of their application showing how to add a contact. (d) Snapshot of their application showing an opened chat in a dark mode.

Figure 11.9: Some snapshots of the execution of the *smart messaging system* designed and implemented by Julien Lienard and Céline Nardi.

11.3 Smart risk information system

In addition to the *smart messaging system*, we also designed a *smart risk information system* with FBCOP.

As we already provided a complete design based on the supporting development methodology for the *smart messaging system*, we will summarise the *smart risk information system* without detail all the deliverables expected in the FBCOP development methodology. Therefore we will first provide a description of what is a *risk information system*. To better understand this system and to define its functionalities, we met with business experts, and more specifically with experts working at the Belgian Crisis Centre² of the Federal Public Service of the Interior. Combining their expertise on such systems and our expertise on context-oriented systems, we could create a prototype of a *smart risk information system*. We will then show the context-feature model we designed for this case study.

This case study aims to demonstrate that we can design another context-oriented applications with FBCOP.

Description

A *smart risk information system* is a system for informing citizens about ongoing emergencies such as earthquakes or floods, providing them with instructions on how to act in such emergencies.

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

The instructions given to citizens in case of an emergency can depend on many contexts: the user's age, the weather, the status of the emergency and so on. The status of an emergency describes whether the emergency is about to happen, is ongoing or is finished. According to this status, an instruction could for example be "*Limit journeys and avoid the danger zones*" if a flood is happening or "*Ventilate rooms adequately, but also heat them adequately to dry off the moisture*" when the flood is over.³ For an earthquake, an instruction could be "*Hide under a table, desk, bed or any sturdy piece of furniture*" if an earthquake is happening or "*Be prepared for after-shocks*" when the earthquake is over.⁴ Such instructions could be given in textual form for adults, or

²<https://ibz.be/fr/centre-de-crise>.

³<https://www.info-risques.be/en/hazards/naturals-hazards/floods>

⁴<https://www.info-risques.be/en/hazards/naturals-hazards/earthquake>

using pictograms when the citizen is a child.

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

Context-feature model

Now that we have described what a *smart risk information system* has to offer, we show the context-feature model for a *smart risk information system*, as depicted in Figure 11.10.

Based on the description, we can easily detect the two main contexts: *User profile* and *Emergency*.

For the *User profile* context, we have some specific contexts that allow to define the current user, such as the user's *Age*, its *Location* or for what *Risks concerns* the user is interested. The *Age* context allows to adapt the display of the instructions. While pictograms will be shown for children, the text of instructions will be enlarged for seniors. The *Location* context allows to activate the GPS so that the system can determine if the user is in the impacted zone of the emergency. The *Risk concerns* context is a user's preference that indicates to the system that the user is interesting to find out more information on a risk and instructions to follow in case of it will happen, occurs and when it is finished.

About the *Emergency* context, some information of an emergency can also be contextualised. The contexts *Type* (i.e., *Earthquake* or *Flood*) and *Status* (i.e., *Before*, *During* and *After*) define what kind of emergency is coming soon, occurs or is finished. Depending on the *Status*, only the instructions for this specific time and for the specific emergency will be displayed depending on the user's *Age*. In addition more information of the current emergency is also displayed to the users as for example its severity and impacted zone. Finally the severity of the emergency will also adapt how users are notified. For example, when the severity is *High*, the system alerts the users through different channels communications, as for example a SMS, via the application and more. For a *Medium* severity, users are warned only by the application. In case of a *Low* severity, users are not notified but if they open their application, a dialog box appears to inform them about the minor emergency.

11.4 Smart meetings system

During their master thesis [DT22], two master-level students, Erwan Delhove and Ho Yien Tsang, have designed and implemented a *smart meetings system* with the FBCOP approach.

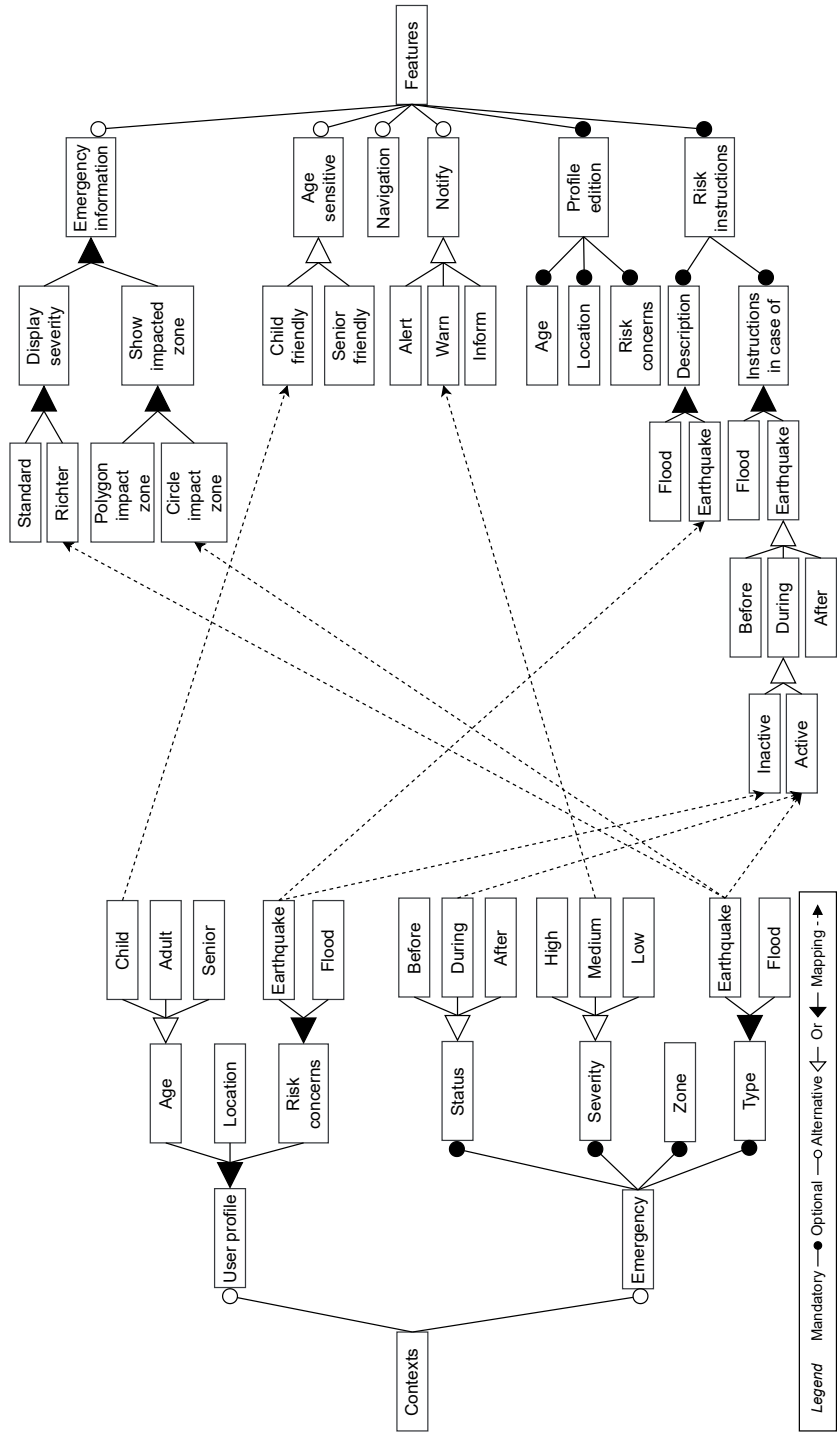


Figure 11.10: Context-feature model of the *smart risk information system*. Some mappings arrows are voluntarily omitted for readability reasons.

We will start this section by describing their smart application, then will show their context-feature model and finally will display some snapshots of their application.

In addition to demonstrate the feasibility to conceive another context-oriented application with FBCOP, they also shown that the FBCOP approach can be extended to treat the multimodal interaction that has not been explored before in our approach.

Description

A *smart meetings system* is an application that allows users to manage their remote meetings. When users have a meeting, they can create a virtual meeting, add and remove participants, and define the time allotted for it. Furthermore they can also send and receive messages in the dedicated chat. Once the meeting is finished, users can no longer modify the meeting but can always open it to see the exchanged conversation during the meeting.

This application is also smart in the sense that the input interaction can be different depending on the surrounding environment. In a normal environment where a mouse and a keyboard are sensed, users can perform all the actions through them. But, when a micro is added, users can also provide commands using voice to write, send or delete a message, and to add or remove participants of a meeting. Another considered modality is gesture when a camera is added to the system. For example, users can delete unsent messages by a swiping gesture from right to left. In addition these both modalities can be combined to write a message by telling the message content in the finger pointing textfield.

The output modality has been also a bit worked in their application. The feedback is either displayed through a screen for users having no vision troubles or dictated by a vocal assistant for blind users. The vocal features require that speakers have been added.

Context-feature model

The context-feature model they designed is depicted in Figure 11.11. Their model has been redrawn with some renaming while preserving all their design choices.

Depending on the vision of users, a different *Set color palette* will be chosen. If users are *Daltonian*, text is adapted with *Daltonian safe* colors. Otherwise the *Default color* palette is applied.

The interaction can vary according to the *Users*, *Environment* and available peripherals (*i.e.*, the different hardwares added to the system in order to provide *Output* and *Input*). For example, when users are *Blind* and *Speak-*

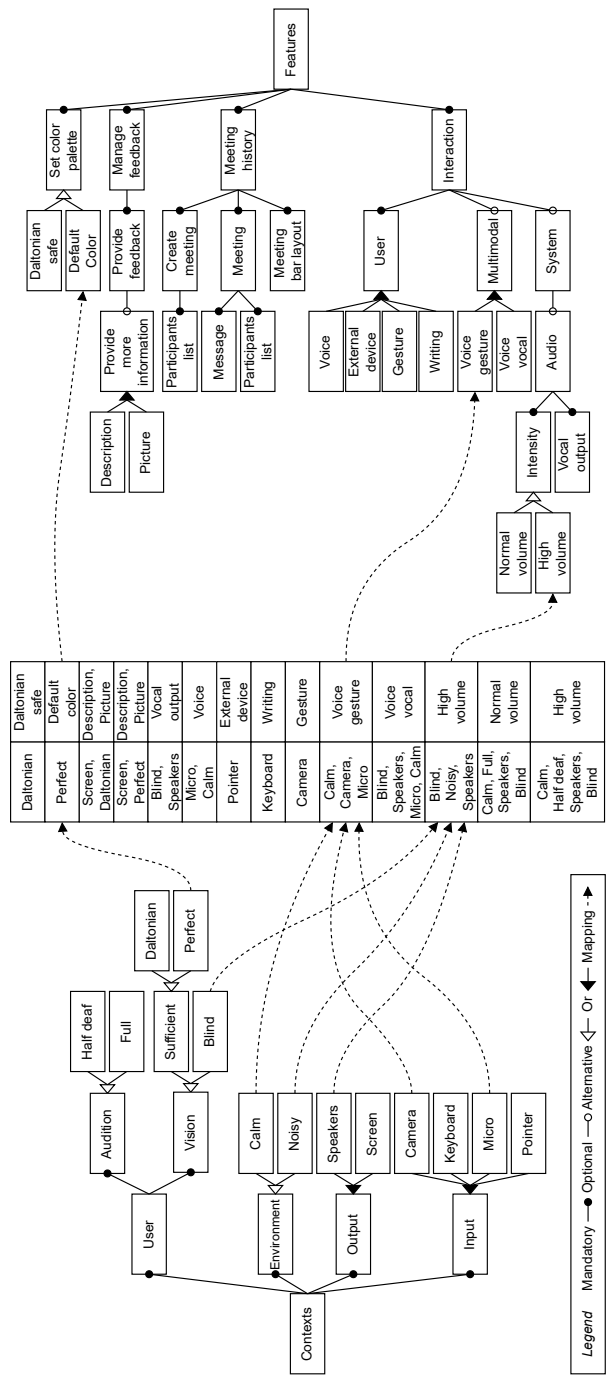


Figure 11.11: Redrawn context-feature model of the *smart meetings* system designed and implemented by Delhove and Tsang for their master thesis [DT22]. Some mapping arrows are voluntarily omitted for readability reasons.

ers are added, the *Vocal output* is applied to read the messages and provide feedback through a vocal assistant.

The inputs are more complex. When a *Micro* is added and the *Environment* is *Calm*, users may interact with the system through *Voice*. Users may write on the application if a *Keyboard* is added and may use *Gesture* when a *Camera* is added. The *Pointer* refers here to the mouse that users can use and defined as *External device*. However the feature *External device* is more a feature to interact with a laser since the mouse behaviour is handled by the computer.

In addition to these modalities, users can also take benefits of the multi-modality. The *Voice gesture* feature simulates the well-known *Put-That-There* paradigm [Bol80]. With it, users can write something in the pointed textfield by telling what they want to write. For that, the environment must be *Calm* and the system must have a *Camera* and *Micro*. Another *Voice vocal* feature is dedicated for *Blind* users in a *Calm* environment with a *Micro* and *Speakers*.

However the *Intensity* of the *Audio* can also be set to a *Normal volume* or a *High volume* depending on the *Audition* disability of *Blind* users (i.e., either a *Full* or *Half deaf* audition) and if they are in a place that is *Calm* or *Noisy*. If the place is *Noisy* or users are *Half deaf*, the volume will be set to a *Higher volume*. Whereas the *volume* will be set to *Normal* for users having *Full* audition if the place is *Calm*. However this feature implies *Speakers* are added to the system.

Finally, the feedback provided can also enrich with a *Description* or a *Picture* when a *Screen* is added and users have a *Sufficient* vision (i.e., *Daltonian* or *Perfect*).

Execution

After designing their context-feature model, they also implemented a prototype to validate the expressiveness of the FBCOP programming framework. We show some snapshots of their application in this subsection.

Figure 11.12 depicts the creation of a new meeting entitled 'Meeting'. The user with a *Sufficient* vision is in a *Calm* place and has a *Micro* and a *Screen*. In this creation, the user sends a vocal command to add a new participant 'Benoit'. In the box on top of the window, the user sees what the system understood (here, 'add Benoit') and the feedback on the execution of this command (here, 'Participant called Benoit has been added'). The execution of this command also results by adding the participant 'Benoit' in the window 'Create new meeting'.

Figure 11.13 illustrates that the user dictates a message in the textfield in the 'Chat' box. Again, the user sees what the system understood and the feedback of this command on top of the window. Furthermore the message is written in the textfield in the 'Chat' box. The situation in which this execution

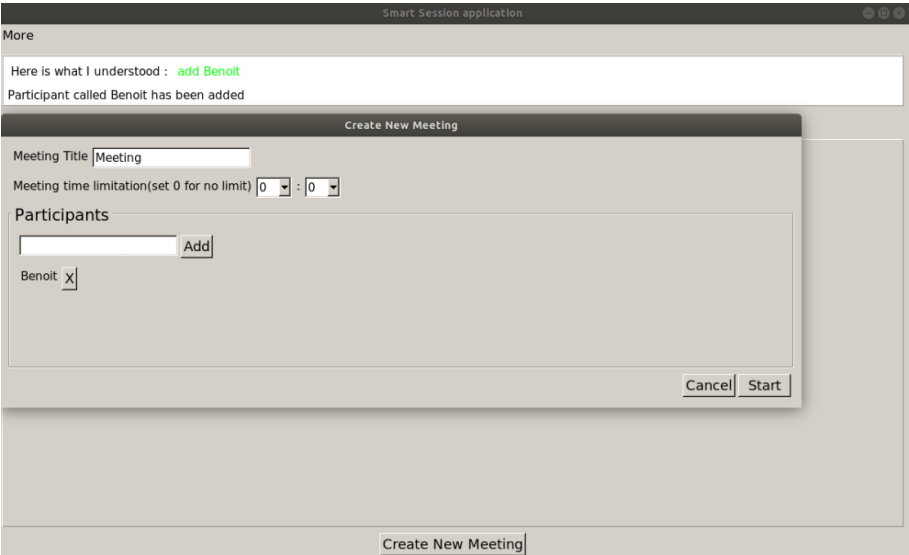


Figure 11.12: Snapshot of the *smart meetings* system where users create a new meeting.

runs is similar that the previous snapshot.

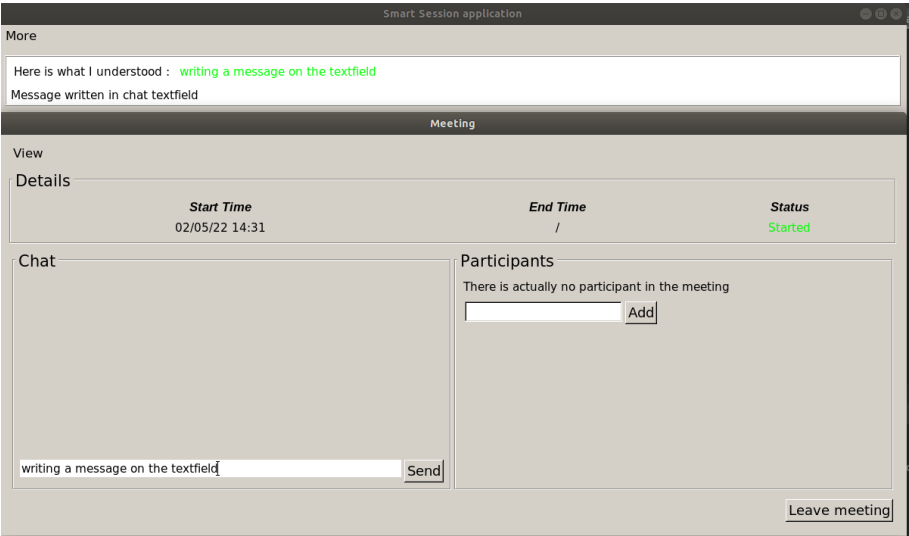


Figure 11.13: Snapshot of the *smart meetings* system where users dictate message.

Assume that an additional *Camera* has been added that allows the user to communicate via gestures. Figure 11.14 shows how the system has been adapted. First, a new UI object has been added on the top-right of the win-

dow to return to the user which kind of gesture the system understood. This feedback is displayed through a picture that represents the type of gesture (*i.e.*, pointing or swiping from right to left). The scenario of this snapshot is that the user wants to add a new participant in the participants list of the new meeting. But the user does a mistake in the name of the participant. To erase the wrong name, the user swipes from right to left the participant's name. We can observe that the gesture understood by the system is the correct one as shown in the dedicated box in Figure 11.14.

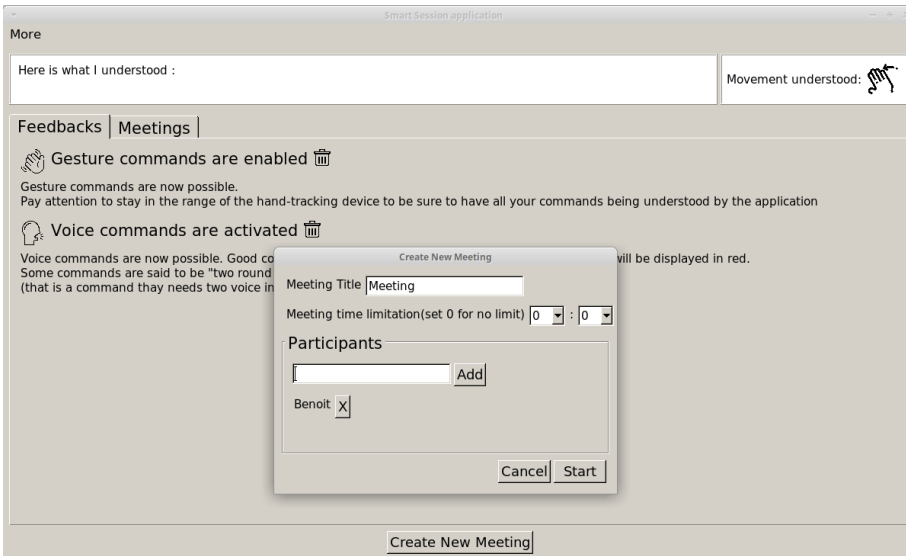


Figure 11.14: Snapshot of the *smart meetings* system where users use a swipe movement (from right to left) to erase the content of a textfield.

11.5 Smart city guide

The FBCOP approach has also been used to model a *smart city guide* in the work of Cardozo and Mens [CM22].

In this section, we will describe what is a *smart city guide* for the authors and then describe their context-feature model.

This demonstrates again the feasibility to design one more dynamically adaptive software system with FBCOP.

Description

A *smart city guide* is an application that offers guided tours of points of interest (POIs) that allow users to discover the beauty and history of a city. Users can choose between two different tours: a guided city tour or a free tour. The

guided city tour suggests different circuits stored in the system. The free tour allows users to walk around the city. However, the free tour indicates when certain interesting POIs are close to the users and can guide them to reach these POIs.

In addition to the selection of POIs by users, the application is smart in that it adapts its behaviour according to the users' preferences and the environment in which the application runs. For example, POIs can be different depending on the time of day in the guided tour (taking into account their opening hours). The time of day can also change the figures of the POIs. Navigation depends on the type of tour and whether a compass or GPS is activated. Finally the users' preferences and users' age can also adapt the information that is displayed for the different POIs. More contexts and features will be discussed in the context-feature model.

Context-feature model

After giving a rough idea of what their application could offer, we will analyse their context-feature model [CM22], shown in Figure 11.15. As illustrated everywhere in this dissertation, the context model is on the left part of the figure and the feature model is on the right part of it.

Users may choose a *GuidedTour* or *FreeTour*. Depending on the *MemoryLevel* of the device, the *Itineraries* could be different. More the memory level is *High* (resp. *Low*), more the itineraries are large (resp. small) and/or contain more (resp. less) information.

The POIs information can display in *English* or *French* according to the preferred user's language or the language of the current country. Furthermore the description of each POI can be simplified for *Children* or complexified for *Adults*. Another context that can adapt POIs is the *TimeOfDay*. During the *Day* (resp. *Night*), all figures are daytime (*nighttime*) pictures. For that, the authors define a feature *DownloadStrategy* that will chosen the right pictures according the *TimeOfDay*.

The *DownloadStrategy* feature is also used to load additional information for POIs when a *Wi-Fi* connection is sensed.

Depending on some peripherals, the navigation is also adapted. For example, when a *Compass* is activated, the application displays a *Map* for users and if the *GPSSAntenna* is activated, users can follow the provided *Directions*.

Finally the *BatteryLevel* can also have an impact on the displayed information, the *Navigation*, and the *DownloadStrategy*. More the *BatteryLevel* is *High* (*Less*), more (resp. less) options are available for users.

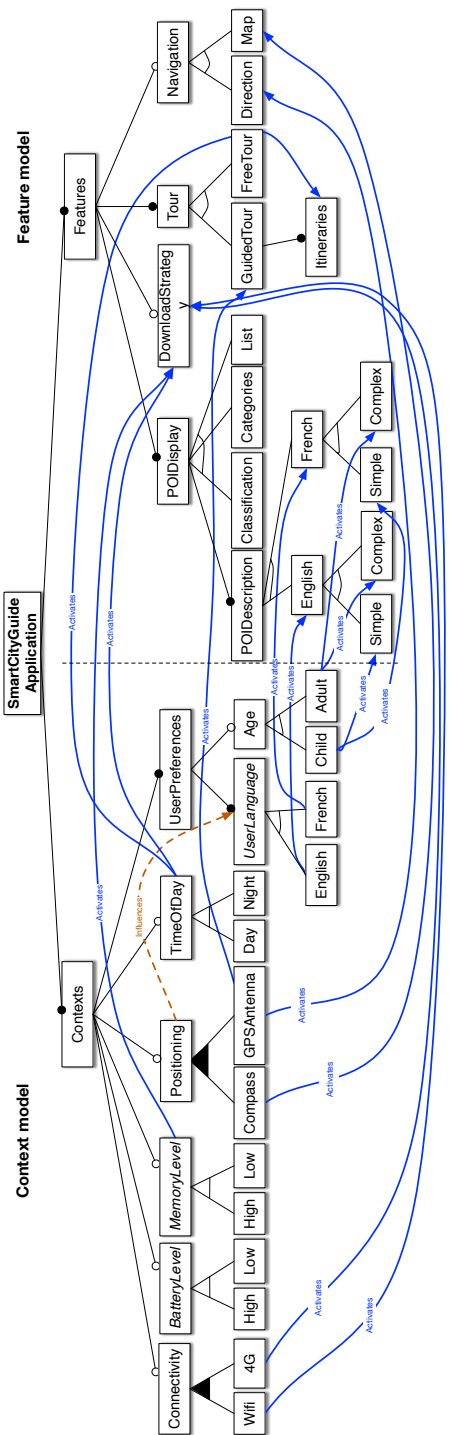


Figure 11.15: Context-feature model of the *smart city guide*. Taken from [CM22].

11.6 Conclusion

In this chapter, we validated the expressiveness of the FBCOP approach with five case studies: two variants of a *smart messaging system*, a *smart risk information system*, a *smart meetings system* and a *smart city guide*. Each of these case studies was conceived either by us or by others.

Through this validation, we can conclude that FBCOP has an interesting expressiveness to create high dynamic adaptive software systems, and in particular context-oriented applications. Indeed, all the five case studies show that designers can model their context-oriented applications with the expressiveness of the FBCOP modelling. Three case studies, both *smart messaging system* and the *smart meetings system*, demonstrate that programmers can implement their context-oriented applications with the expressiveness of the FBCOP programming framework. Three case studies show that the supporting development methodology of FBCOP is enough expressive to analyse the *requirements*, *design* and *implement* context-oriented applications. Even if we only detailed once this development methodology in this validation, the two case studies developed by our master-level students (*i.e.*, the second variant of a *smart messaging system* and the *smart meetings system*) claimed that they followed it to create their prototypes.

Although the expressiveness of FBCOP is therefore sufficient for designing and implementing systems, we could improve some parts of its expressiveness, such as the mapping model, keywords in the programming language for feature definitions, and more, as discussed in Section 14.1.

CHAPTER

12

VALIDATING FBCOP'S DESIGN

Previously we validated FBCOP's expressiveness through five case studies. Now we will examine the design of our approach. First we will discuss the design qualities of our programming framework to demonstrate if its implementation is maintainable, extensible, adaptable, readable and scalable. Then we will position the programming framework based on the cognitive dimensions of notations framework [GP96; Bla+01] in order to assess its usability. This allows us to present our point of view on its usability before evaluating it with real users.

12.1 Design qualities

In this section, we demonstrate that the implementation of our programming framework is maintainable, extensible, adaptable and readable. Then we explain why we think it may not be scalable to very large and complex case studies.

Maintainability We consider that our implementation is maintainable for the following reasons.

First we emphasise the modularity of our implementation. Indeed we divided the implementation of our programming framework into four separate parts (ENTITIES, MODELLING, ARCHITECTURE and TOOLSupport) to decrease the coupling between them (*i.e.*, the entities, modelling, control flow and tool-

ing support). For example, the ARCHITECTURE part is not directly connected to the MODELLING part. When a component of the ARCHITECTURE part wants to update or query the context and feature model, it sends a message to the model in the ENTITIES part, that delegates itself the message to the MODELLING part. Then the result is returned first to the ENTITIES part, that returns it to the component of the ARCHITECTURE part.

We also increased the cohesion of each concern with this separation. Indeed, all the entities (contexts, features and mapping), the declarations and definitions of the different models are in a same part (ENTITIES). All the needed framework classes to create a generic feature model are also implemented in a separate part (MODELLING). Finally, all the components of the control flow have their dedicated part (ARCHITECTURE) and all the framework classes for the communication between the implementation framework and the different supporting tools compose their own part (TOOLSUPPORT).

For the ENTITIES part, we also separated the model declaration from the model definition to better ease the understandability of what (context and feature) model is provided by FBCOP's programmers and used by the system.

For the ARCHITECTURE part, each component is implemented in its own folder. To get a more maintainable code, we also separate the core logic of each component from its proxy or bridge that only sends relevant information to the dedicated supporting tool.

In addition we used several other concepts and mechanisms to ensure a good maintainability in our implementation such as inheritance, polymorphism, and design patterns (of which the proxy and bridge mentioned above are but one example).

We used the concept of inheritance to implement the different relations between the contexts and features in the context and feature model. Another example of the use of inheritance is the implementation of an abstraction of the similar state and behaviour of contexts and features.

As for design patterns, we used the singleton design pattern for the framework classes that require exactly one instance, the strategy design pattern to implement the satisfiability algorithm (in order to be able to allow for multiple variant implementations of that algorithm), the proxy or bridge design pattern to send relevant information from the implementation architecture to the supporting tools, and more.

Extensibility Our programming framework is also extensible.

In addition to provide a programming framework that programmers must specify and use in the creation of their models, they could also extend the behaviour of the framework classes depending on their needs. For example, they could add more relations (*i.e.*, constraints or dependencies) in the context and feature model as suggested by Cardozo et al. [Car+15] or implement a

new satisfiability strategy based on SAT(isfiability) solvers [Bat05].

Furthermore, they could also extend the `TOOLSUPPORT` part of the environment by adding for additional external tools to assist either programmers or end-users. We already demonstrated its feasibility for the programmers with our two different visualisation tools and the `CONTEXT SIMULATOR` tool. Some master-level students also extended this part to add new tools, dedicated for end-users, as discussed in Section 10.3.

Finally, with minor changes, programmers can also extend the other components of the programming framework such as the `CONTEXT ACTIVATION` component or others.

Adaptability Our approach is adaptable by definition since it allows to modify the application behaviour at runtime depending on the surrounding environment. The behaviour of `FBCOP` applications is by purpose adaptable so that the different application classes are adapted according to the features that must be active for a particular situation. Moreover, nothing prevents even the framework classes to be adapted by features depending on some particular situations. But for that programmers have to implement a particular feature for the framework class and precise what contexts trigger this feature. For example, assume programmers want a different logging system according to the current mode (*e.g.*, development, testing and production) in which they are. For that, they must implement the different contexts, features, mapping and which framework classes are adapted by these features. Then, once the application is launched, they can precise in which mode they are, and the framework classes are refined to adapt the logging system.

Readability We also consider that our implementation is readable for future programmers that want to extend our approach. To guarantee the readability of our code, we did our best to divide all the code into small methods, where each method is responsible of one small and single task. In addition we named our methods by giving them an intention-revealing name to describe their real intention. The modularity explained above could reduce the readability of our code somewhat. But, as we used clear names for the instance variables, the local variables, the methods and the framework classes, we consider that the modularity does not decrease the readability of our implementation. We also drew the different class diagrams to support programmers in their reading. Finally we implemented many design patterns which also increases the readability of our code for programmers who know these patterns. Nonetheless metaprogramming may reduce the readability of the code due to its high level of abstraction. Nevertheless, this cannot be avoided as it is a need to perform some operations (*e.g.*, adding or removing a new method in an application class).

Scalability Our approach is certainly viable for small applications that have tens of contexts and features, as proven by our case studies.

However it may not be scalable for industrial case studies with much larger and more complex context-feature models.

A first reason for this is the satisfiability strategy we used. With our depth-first search strategy we must traverse the entire model in the worst-case, even for small changes in the configuration. For that we think that our approach is less scalable than others with better optimisations. But this problem could perhaps be solved, at least partly, by implementing a more efficient satisfiability solver.

Another problem with scalability is that it becomes tedious to implement large and complex applications due to the number of application classes and features that programmers must create. Nevertheless it would be a problem anyway regardless of the approach.

Finally a last problem is the scalability of the visualisation tools. Their visualisations were not designed to display so many entities. Alternative visualisation should be thought of to scale up to industrial-sized systems.

12.2 Cognitive dimensions framework

Now that we have explained the design qualities of our programming framework, we will debate its usability from a programmers' perspective according to the cognitive dimensions of notations framework [GP96; Bla+01]. For that we will discuss the following cognitive dimensions: viscosity, visibility, premature commitment, hidden dependencies, role-expressiveness, error-proneness, abstraction, secondary notation, closeness of mapping, consistency, diffuseness, hard mental operations, provisionality and progressive evaluation.

Viscosity The viscosity of a system is the resistance to refactor an implementation for changes. In other words, the more viscous a system is, the more changes must be made to add, remove or modify a functionality.

The viscosity in FBCOP is intrinsically related to the type of changes that FBCOP programmers must perform.

For example, to add or remove a context, update a constraint or move a context in the context model, programmers only have to update the context model declaration. While for renaming a context they have to propagate changes in the context and mapping model declarations.

To add or remove a feature, they must modify the feature model declaration and also implement or delete its feature definition. Updating a feature depends on the kind of change that a programmer wants to make. For refactoring a constraint in the feature model or moving a feature in the feature

model, only the feature model declaration should be updated. If the modification is about the feature definition (*i.e.*, the source code of the feature), programmers should update its definition. Renaming a feature declaration means programmers must update the feature and mapping model declarations. Renaming a feature definition or changing an application class adapted by the feature means that they have to update the feature declaration and feature definition.

For the application classes, adding a new application class implies that programmers must create a new class. Then they must reference it when declaring and defining the features that adapt this application class.

In addition to being able to easily refactor the code of FBCOP applications, programmers do not have repetition in changes because of the clear separation of contexts and features. But some repetition could appear when they implement the context and feature model declarations because these models follow the same notation.

We thus conclude that our programming framework is not viscous since the changes do not imply big changes.

Visibility Visibility is when the information or components are easily visible. For example, less the information is encapsulated, more the information is visible.

In FBCOP, while the separation of contexts, features and application classes has some advantages (*e.g.*, to ease the maintainability and evolution of the source code), programmers may lose in visibility when some information is spread over various components. For example, when programmers have to inspect what application classes are adapted by what features in a particular situation (*i.e.*, a context), they must analyse the different entities (*i.e.*, at least the context, mapping and feature models). Another example is about the *proceed* mechanism for which they must inspect the different features parts that compose the full behaviour of a functionality.

Despite this visibility reduction, our visualisation tools help programmers to increase it. For example, to visualise what contexts have an impact on what application classes, our two visualisation tools help programmers in this task. For the chain of *proceeds*, programmers can use the FEATURE VISUALISER tool to visually inspect this chain.

Therefore we conclude that the visibility of FBCOP is reduced due to its clear and explicit separation between the different components, but the visualisation tools partially address this issue.

Premature commitment Some decisions need to be made early on by designers or programmers. This dimension is called premature commitment.

In FBCOP, programmers must already think about the application classes at the design step to know which application classes they need. Since they have to take such decisions before the implementation step, we consider that a case of premature commitment. Another example of premature commitment is the activation order. Because many features refine features representing the default behaviour, the activation order of the features is also important. For example, a refinement cannot be installed before the default behaviour, otherwise the system will raise an unexpected behaviour or error. Nonetheless these premature commitments are intrinsically related to the approach and domain problem of conceiving such systems.

We thus conclude that FBCOP has some premature commitments, but they are necessary to develop context-oriented applications.

Hidden dependencies Relevant dependencies between entities might be hidden, such as for example class hierarchies that are too complex. But this makes it more difficult to read and understand the application's source code.

An example of such hidden dependencies in FBCOP is the chain of *proceeds* between the features to compose a functionality. But our visualisation tools, and in particular the FEATURE VISUALISER tool, help programmers to visualise these hidden dependencies since the different chain of calls are explicitly visualised in the FEATURE VISUALISER, even if it could be done at an even more fine-grained level.

Role-expressiveness Assessing how programmers can easily infer the purpose of an entity refers to the role-expressiveness dimension.

With the different concepts in FBCOP and building blocks (*i.e.*, context, feature, mapping, and application class) in the programming framework, programmers can easily infer the meaning and distinguish the role of each concept. Nevertheless some keywords at the implementation level could be more role-expressive, such as a feature definition or a feature part. These two concepts are defined with a same keyword “module” (as defined in the *Ruby* programming language) which may lead to some confusion.

We therefore conclude that FBCOP is mostly role-expressive, but could still be improved.

Error-proneness Error-proneness is when the notation of a programming language or framework may invite to errors. But the system can inform programmers about the problem as a preventive measure.

In FBCOP, designing context-feature models may lead to design errors. Such design errors may arise from the complexity of previewing the complete model before execution. But the CONTEXT AND FEATURE MODEL VISUALISER tool allows to overview the complete model to assist designers and

programmers. Furthermore the test generation tool proposed by Martou et al. [Mar+21] also helps to detect design errors or inconsistencies before implementation.

Implementing the context and feature models may also invite errors due to their repetitive structure. For example, when programmers have to declare a context model, they must declare contexts and link them together to create the model. But programmers may write more easily errors for large models due to a lack of attention. Indeed, they might easily link a context to another one while these contexts must not be linked together. Again, the `CONTEXT AND FEATURE MODEL VISUALISER` also helps programmers to visualise their model declaration to ensure their models are well-declared.

When implementing FBCOP systems, the context and feature interaction problems may also lead to errors. An example of this problem is when two different contexts trigger two different features that have incompatible behaviour. For example, assume in a home automation system a feature that must trigger the sprinklers and another feature that turns off the main water supply. In case of the activation of the two features, the sprinklers cannot continue to extinguish a fire if the main water supply is turned off. Such issues may arise from the high dynamicity of such systems because it is tedious to visualise all the interactions between the components and how the system dynamically evolves. These errors are then intrinsically related to the problem domain. But our visualisation tools may partially help to better understand such errors.

Another kind of errors may also arise from the activation order if the features are not installed in the right order. Indeed, if a feature is a refinement of a default behaviour but the latter is deployed after the former, an error is logically raised.

We conclude that FBCOP is error-prone, but our visualisation tools partly help programmers to detect some errors.

Abstraction This dimension concerns the abstraction level that a programming language or framework can have, through notations such as types, data structures or more.

FBCOP has different abstractions for the contexts, features and application classes and this is needed because the concepts are different by nature. These abstractions allow programmers to better understand the different concepts when they use them. Whereas too many abstractions may become hard to understand and may confuse programmers, our abstractions are relatively straightforward since they are intrinsically related to the approach and the problem. Moreover we do not have too many abstractions which simplifies its understandability.

Therefore we consider our programming framework is abstract enough without leading to confusion or difficulties.

Secondary notation This dimension discusses additional information that programmers can provide to better understand what they are making. An example of a secondary notation are comments in a programming language.

FBCOP has the lexicon and rationale of contexts, features and mapping as secondary notations. These notations are useful to justify some design choices that are made during the *requirements* and *design* phases. With that, designers and programmers may understand more easily or remember the design choices made for the project. Nonetheless these notations are not yet integrated into the programming framework itself, which means that such information is decoupled from the implementation. Therefore, it might be interesting to extend the model declarations or add annotations in the feature definitions to be able to explicitly document this information as part of the code.

Another secondary notation available in FBCOP are the comments that programmers can write to describe their source code. This notation is naturally present in FBCOP since the programming framework is built on top of the *Ruby* programming language that proposes such a feature.

We thus conclude that we could improve the secondary notation in FBCOP to help programmers to better understand their code and remember their design choices.

Closeness of mapping The closeness of mapping is a dimension to explain how close the notations of the programming language are to the problem domain.

The FBCOP notations are really close to the problem domain. We have contexts that represent a particular situation and features that describe the behaviour of the system. With these notations, we followed the same ideology proposed by context-oriented programming, without forgetting that we clearly separate both to get closer to the problem domain.

Consistency A notation is consistent when a same syntax is used for a same semantics.

FBCOP propose a similar notation to ease the modelling and implementation of both the context and feature models.

We are thus consistent for the similar semantics in our approach.

Diffuseness The verbosity of a programming framework may also be evaluated with the cognitive dimensions of notations framework. This dimension is called diffuseness.

FBCOP is somewhat verbose in the declaration of the context and feature models, but also in the declarations and definitions of features. For example, programmers have to express what application classes the feature must adapt when declaring a feature and make explicit which application classes a feature part can adapt when defining it. Such a redundancy may be verbose for programmers.

Hard mental operations The dimension about hard mental operations assesses which user actions may be cognitively complex to perform.

We observe mostly two complex tasks in FBCOP.

The first one is about the development of such high dynamic systems. It may be tedious for programmers to overview all the different interactions between all the concepts.

The other one is related to the *proceed* mechanism and its chains of calls. Indeed programmers must do cognitive effort to visualise the different chains of *proceed* calls and in the right order. It is not straightforward to keep in mind how a functionality is composed through the different feature parts because they are split in fine-grained features to obtain a better maintainability and reusability.

Nevertheless both of these hard mental operations are intrinsically related to the problem of conceiving such context-oriented applications. Furthermore, programmers may rely on our visualisation tools to help them for these complex tasks.

Provisionality It is sometimes useful to make premature commitments to move ahead with the conception of applications. This is called provisionality.

FBCOP developers must foresee some design choices at the modelling level that will have an impact at the implementation level. Indeed they must think up front what contexts trigger what features, in what order and for what application classes.

We thus conclude that some premature commitments must be provisioned in FBCOP to move ahead with the project.

Progressive evaluation This dimension assesses to what extent the designers and programmers may evaluate their progress at any moment when using an approach, a programming framework or language.

In FBCOP, designers and programmers can progressively evaluate their progress if they follow the incremental and iterative methodology we proposed (see Subsection 3.3.5) to get frequent releases of their application.

Based on this cognitive dimensions of notations framework, we can conclude that our approach and programming framework seems usable with the

help of our visualisation tools. Some dimensions are good or excellent in our approach, such as the viscosity, role-expressiveness, abstraction, closeness of mapping, and consistency. The dimension of progressive evaluation is neutral since this depends on the methodology that designers and programmers will follow for their conception. Other dimensions are lesser but our supporting visualisation tools partially help programmers in their actions. These dimensions are the visibility, hidden dependencies, error-proneness and hard mental operations. Some dimensions could be improved such as the visibility, role-expressiveness, error-proneness, secondary notation, and diffuseness. Finally other dimensions may be problematic such as the premature commitment, hard mental operations and provisionality. But these last dimensions are intrinsically related to the problem of conceiving context-oriented systems that are highly dynamic.

12.3 Conclusion

In this chapter we discussed which design qualities our implementation of the programming framework has. We can conclude that our implementation is maintainable, extensible, adaptable and readable. However it is not yet fully scalable.

Then we evaluated the usability of our programming framework following the cognitive dimensions of notations framework. We can say that our programming framework is usable with the help of our visualisation tools based on the different dimensions. However some dimensions could be enhanced to make life easier for programmers and yet other dimensions are complex but this complexity is intrinsically related to the problem of developing context-oriented applications. However, even if we consider our programming framework as usable, we need to evaluate its usability with real users, and this will be the purpose of the next chapter.

CHAPTER

13

VALIDATING THE FBCOP APPROACH WITH USERS

The previous chapter reviewed the design qualities of the implementation of our programming framework and the usability of our approach through the cognitive dimensions of notations framework. However our own appreciation of its usability and the overall approach is not necessary the same as that of real users. (Even if we are also real users of our approach, we are mainly the developers of the FBCOP approach.) So, besides the various technical challenges we had to confront when creating a new approach, there are also the challenges that users of the approach have to face, such as its learning complexity, its comprehensibility, its usefulness and its usability. Having a rich and powerful approach is really interesting, but if users cannot understand and cannot use it, this new approach will be never used. Therefore, in this chapter, we will assess the FBCOP approach with real designers and programmers in order to evaluate its usefulness and usability. For that we conducted four user studies over the years.

We first evaluated our two visualisation tools as preliminary user studies in order to get a first indication on how programmers may perceive FBCOP through these visualisation tools and whether these tools help them in understanding FBCOP. The first preliminary study concerns the `FEATURE VISUALISER` (Section 13.2) and the second preliminary study is of the `CONTEXT AND FEATURE MODEL VISUALISER` (Section 13.3). Analysing these preliminary user

studies on the visualisation tools, we observed that FBCOP seems interesting and understandable for users. However, in these two first user studies, we did not evaluate the complete FBCOP approach we have built, *i.e.*, we did not assess whether the modelling and programming framework are useful and usable for users designing and implementing FBCOP applications, nor did we evaluate the supporting development methodology and visualisation tools in detail that we suggest to help them when conceiving such applications. Therefore we conducted a first complete user study (Section 13.4) with participants to evaluate whether our complete approach was useful and usable for real designers and programmers. To confirm or not and to better understand the strengths and weaknesses of our approach, we conducted another complete user study (Section 13.5) the following year with another group of participants. In this last complete user study, we updated the previous user study to gather more feedback by asking them to justify their answers and simplify some parts of the questionnaires to avoid they had to face an overload of work.

In this chapter, we will first present the setup we followed for all the user studies (Section 13.1). Next, for each user study, we will follow the same structure: we will first introduce the user study, followed by a description of the user study itself we conducted with the participants, then we will show the raw results we gathered and we will finally discuss them and conclude with the lessons we have learnt with the user study.

13.1 Setup for all user studies

Each user study was conducted with master-level students in computer science and engineering enrolled in a software engineering course. Since the participants were students enrolled in a course, we explicitly and repeatedly clarified that this user study was not taken into account for their course evaluation but only dedicated to research. Furthermore, the studies were completely anonymous to limit potential bias.

For each user study we performed, the full list of questions and responses of our (pairs of) participants are available on this accompanying repository <https://github.com/bduhoux/PhDThesisUserStudiesData>.

13.2 Preliminary user study of the FEATURE VISUALISER

This section presents a first qualitative study of the FEATURE VISUALISER tool that supports developers to visually inspect the dynamics of the FBCOP application they develop. The goal of this evaluation was to gather feedback from potential developers on the tool's overall understandability and relevance, as well as feedback for further improvement.

We presented it to a set of master-level students with significant programming experience and collected their feedback using a questionnaire. In the remainder of this section, we will start by describing the user study itself, then will present the raw results of this study before analysing and interpreting these results. Finally we will conclude this preliminary study of the FEATURE VISUALISER tool.

User study

The participants of this study carried out in 2018 were 25 master-level students aged between 21 and 27 years old. During a two-hour classroom session, the students were asked to play the role of a potential developer using our programming approach and tool.

The user study was conducted as follows. We first provided a set of questions to gather information on the participants' software development skills. After this first set of questions, we briefly explained the idea of FBCOP and introduced the case study of a *risk information system*.

We then explained and illustrated the dynamic feature adaptation of our approach when their contexts get activated using a small scenario taken from the case study. In this scenario, we assumed an earthquake was detected and that the authorities announced its severity and its location (*i.e.*, its epicenter and radius). For this scenario, we showed the program code and its execution but not yet the visualisation tool. This was to avoid a bias in the understanding of our approach and in the perception of the visualisation tool before assessing it.

We then extended this scenario into a didactic scenario to assess the understandability of our programming approach. In this extended scenario, before setting the severity and location of the earthquake, we assume that an adult end user wants to consult information on what to do in case of an earthquake. For this second scenario, we again showed the code but not the visualisation tool. The participants had to answer some questions regarding the expected behavior of this scenario. After having answered these questions, we explained and showed them the results of the second scenario. The main purpose of this scenario was to ensure that the participants had a good comprehension of the underlying language approach, *before* showing them the visualisation tool.

Once the participants had understood the second scenario, finally we presented the FEATURE VISUALISER tool and ran the two first scenarios again, but now with the visualisation tool running.

We then applied the FEATURE VISUALISER tool to a more realistic third scenario. In this scenario, the user was a child and the scenario involved a generic instruction followed by an early earthquake warning. Messages were

tailored to different age populations, including children. For this scenario, we asked the same set of questions as for the second scenario, but now the participants could see not only the program code but also the visualisation rendered by the tool. The main purpose of this scenario was to crash test the participants' ability to understand more complex programs built using our approach, thanks to the visualisation support.

After having assessed their understanding of the approach using each of these three scenarios, finally we asked the participants for their feedback on our tool via a questionnaire. Participants were to rate the dynamic representation, usability, usefulness for program understanding and usefulness for debugging purposes of the visualisation tool on five-level Likert scales. Two optional open-ended questions allowed participants to leave additional feedback if needed.

Raw results

Our questionnaire was composed of five sections: an introductory page, questions about the participants' background knowledge, questions about the scenario without use of the visualisation tool, with use of it, and finally feedback on the tool.

Divergent stacked bars All closed questions were phrased in such a way that the participants could answer whether they agreed or not on a five-level Likert scale (ranging from “strongly disagree” to “strongly agree”, or synonyms). We followed this approach for all the closed questions in the different surveys we gave to the participants throughout all our user studies. In order to easily see a tendency in the participants responses to these questions, we summarise them using divergent stacked bar charts, such as depicted for example in Figure 13.1. In addition to showing the frequency of each possible response to a question, as opposed to traditional stacked bar charts, divergent stacked bars position the replies horizontally. Negative responses are stacked to the left of a vertical baseline and coloured red (dark red for ‘strongly disagree’, pale red for ‘disagree’). Positive responses are stacked to the right of the vertical baseline and coloured blue (dark blue for ‘strongly agree’, pale blue for ‘agree’). Neutral responses are wrapped symmetrically around the vertical baseline and coloured grey.

Participants' background Figure 13.1 summarises the answers regarding the participants' background knowledge. For each question, participants were to state their skill on a five-level Likert scale (from “non-expert” status to “expert” status). All 25 participants responded to each of these questions. We can see that our participants had some skills in *programming in general* (44%

with positive values against 4% of negative answers.) and were really confident with *object-oriented programming* (80% of positive values and 0 negative answer). Despite these skills, they were not at all comfortable with the *Ruby programming language* (80% of negative values) and not really with *feature modelling* (56% of negative values against 12% of positive ones).

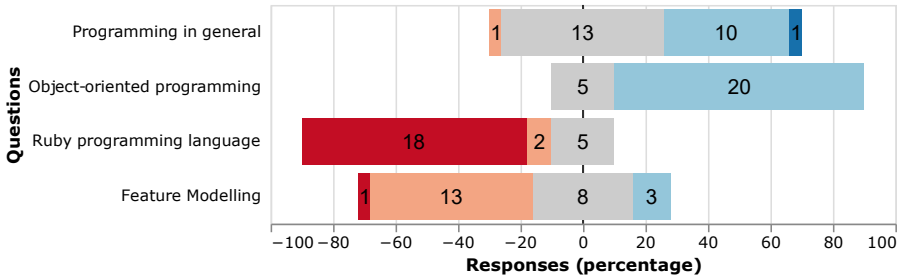


Figure 13.1: Divergent stacked bar presenting the background of our 25 participants. For each question we indicate the absolute frequency of each answer textually.

Participants' comprehension Figure 13.2 collects the results of the 25 participants' comprehension of our FBCOP approach, based on the simpler second scenario without access to the visualisation tool yet. For this didactic scenario, we asked three open-ended questions, which focussed on whether the participants understood which features were activated (Figure 13.2a) and in which order (Figure 13.2b), as well as if they could ascertain the execution of the program at a given point (Figure 13.2c). We evaluated and divided the participants' answers in four categories: incorrect answers (*Fail*), many inaccuracies because of missing information (*Many*), mostly correct answers with some errors only (*Some*) and completely correct answers (*Success*). Participants who did not answer at all were considered as having given an incorrect answer. We can see that many of them understood well what features are active (76% of them succeeded or made some errors only) for this didactic scenario. A similar tendency (68%) is true for the activation order and expected execution questions. Going further by correlating the different answers, we observed that 12 participants succeeded or made only some errors for the three questions while 7 participants failed or made many errors in one of the three questions.

Figure 13.3 then collects the results of the participants' comprehension, but now *with* use of the visualisation tool, based on the more complex third scenario. The same three open-ended questions were asked as for the simpler second scenario, to assess whether the visualisation helped participants to understand what is going on, thanks to the visualisation, on a realistic sce-

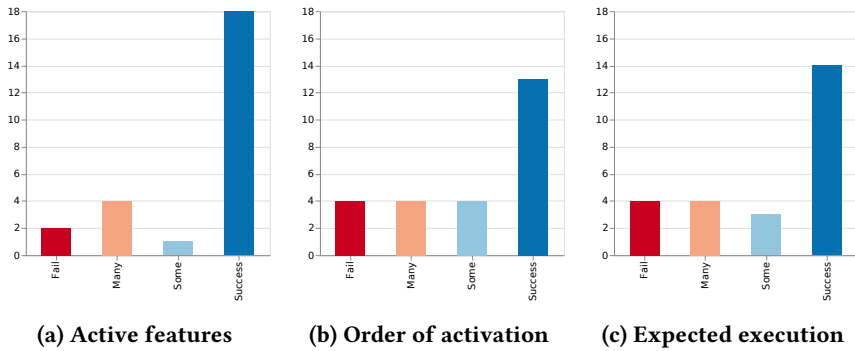


Figure 13.2: Participants’ comprehension of the FBCOP approach on a didactic scenario, without use of the **FEATURE VISUALISER** tool. The graph depicts how well our 25 participants answered the comprehension questions asked: *failure*, *many* errors, *only some* errors or *success*.

nario. For this harder exercise, we can see that only 60% (counting only the blue bars) answered correctly for the question “*which features are currently activated?*”. While for the other questions, the results were worse: 64% failed or made many errors wen they had to predict the activation order of the features and 88% gave wrong answers for the expected execution. All participants for this exercise failed or made many errors for at least one of the three questions.

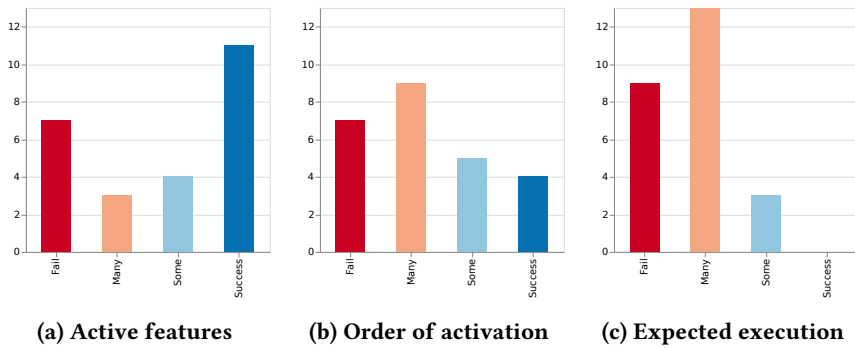


Figure 13.3: Participants’ comprehension of the FBCOP approach on a more complex and more realistic scenario, *with* use of the **FEATURE VISUALISER** tool. The graph depicts how well our 25 participants answered the comprehension questions asked: *failure*, *many* errors, *only some* errors or *success*.

Participants’ feedback Finally, Figure 13.4 summarises the participants’ feedback on our tool. This was evaluated through four questions assessing the tool’s *dynamic representation* of feature (de)activation, whether the visualisation could *help to understand* or *help to debug* a program, and its *ease of*

use. In their answers we can see that they (80% with only positive values) appreciated the *dynamic representation* of the (de)activation of the features. They also considered that this tool *helped them to understand* our proposed programming approach (80% taking only positive values). 68% (with only positive values) believed that the tool could be valuable for debugging purposes (*help to debug*). 72% (taking only positive values) of all also claimed that our tool were easy to use.

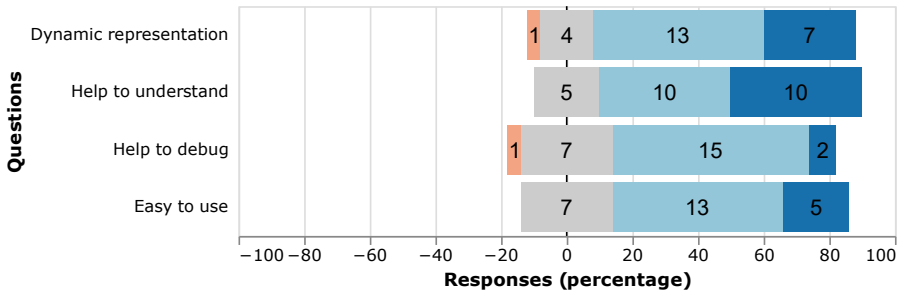


Figure 13.4: Divergent stacked bar presenting the participants’ feedback on the FEATURE VISUALISER tool.

Two additional open questions were asked regarding what functionalities offered by the visualisation tool could or should be improved in order to enhance a developer’s experience with our programming approach. In the next subsection we provide a detailed analysis of the raw results and the participants’ responses to all other questions.

Discussion

Although the participants of our study were students, they were master students in computer science and computer science engineering, following a study program with a strong emphasis on programming skills. Despite they had been exposed to a variety of programming languages of which the most important are Java, C and Python, they had some difficulties with *Ruby*. This could be explained by the fact that they were not familiar with its syntax.

At first glance, from Figure 13.4 we could derive that the FEATURE VISUALISER tool is rather comprehensible and useful for developers of our novel programming approach. The lower value for debugging is likely due to the fact that none of our scenarios included a debugging activity. Since our visualisation tool is only a first prototype, we were quite satisfied with this perceived positive feedback. In the responses to the open questions, we also received some suggestions and constructive feedback to further improve the relevance and usability of the tool for developers. They suggested us to improve the FEATURE VISUALISER tool by adding some filters to display only

some specific behaviour or only a specific feature. Another request was the addition of a ‘back’ button to enable to rollback the visualisation without re-launching the tool to inspect what was happening previously. Finally they also proposed to improve the automatic layout algorithm to get a smoother visual experience.

Nevertheless this positive feedback is mitigated strongly by the more negative results of Figure 13.3, which seems to show that the participants still had a hard time understanding the approach even when using the visualisation tool. In fact, the results were even more negative than without using the visualisation tool. It should not be forgotten, however, that the third scenario was significantly more complex than the second one, which was of more didactic of nature and contained only a very limited amount of features. The results of Figure 13.3 can and should thus not be compared directly with the results of Figure 13.2. Where the purpose of the second scenario was just to have a kind of ‘green light’ check to verify that the participants understood the approach before showing them the visualisation tool, the third scenario was deliberately made more complex to have a more realistic scenario in which the visualisation would show its real value. In that sense we are content that the more negative results of Figure 13.3 seem to be compensated by the more positive results of Figure 13.4, where the participants do confirm they believe in the value of such a visualisation tool, even though this feedback was given after their more negative experience with scenario 3.

Conclusion

This user study to assess the FEATURE VISUALISER tool was conducted with the help of 25 participants.

Based on the Ledo et al.’s work [Led+18], we assessed our visualisation tool with the following evaluation strategies: *demonstration* and *usage*. Our demonstration strategy relied on a *risk information system as example* to illustrate the FBCOP approach and visualisation tool. In addition we also used the ‘*how to*’ scenarios technique to explicit the different steps of our visualisation tool. We also conducted a small *usability study* with a *Likert scale questionnaire*. To be sure they understood how they could use the visualisation tool, we also did a *walkthrough demonstration* to explain its functionalities.

We conclude that the FEATURE VISUALISER tool has at least the potential to help developers better understanding and debugging programs written in our FBCOP approach. However such a user study was insufficient to really assess the real usability and usefulness of our visualisation tool. Indeed such a tool should have been used more deeply by participants, for example by asking them to really implement and debug a full context-oriented application with our tool. But for that we should have taken more time to better explain our

FBCOP approach and then trained them more extensively before conducting this user study. Therefore, even if some of the preliminary results have a positive tendency, we need to confirm it by conducting another more complex user study.

13.3 Preliminary user study of the CONTEXT AND FEATURE MODEL VISUALISER

After the previous preliminary user study of the FEATURE VISUALISER tool, we also conducted preliminary user study to get an initial insight in the usability and usefulness of our CONTEXT AND FEATURE MODEL VISUALISER tool.

This user study was carried out in 2019 with all new participants. The subjects of our study were 34 master-level students, aged 20 to 27 years old. To evaluate this second visualisation tool, we asked them to play the role of programmers working on a FBCOP system.

As we felt that we did not take enough time to introduce the different underlying technologies in the past, we organised two preparatory sessions of two hours each to initiate and train our participants to *Ruby* and our FBCOP approach, respectively. Then we started the user study itself.

In the remainder of this section, we first describe the user study itself, present the raw results we gathered through the study, discuss the results and conclude this study.

User study

We performed the user study during a two-hour session where the students were asked to assess the usability and usefulness of our CONTEXT AND FEATURE MODEL VISUALISER tool. Participants were first asked to report some information about themselves: their age, general background in programming, knowledge of object-oriented programming, context-oriented programming and so on.

Then we asked them to perform two tasks during a 25-minute time slot. These tasks aimed to extend the earthquake-specific variant of the *risk information system* (see previous user study) with a new kind of risk and emergency: that of floods. *Task 1* concerned the characteristics of a flood. In this task, they had to implement some fine-grained features. *Task 2* was about implementing the flood-specific instructions (either static or dynamic) that citizens must follow before, during or after a flood.

We split the programmers in two separate groups (A and B) to perform their first assigned task. Group A had to start implementing *Task 1* whereas group B had to develop *Task 2*. During their first task, they were not allowed to use the visualisation tool. Afterwards they were asked to answer three ques-

tions about the task they performed, to verify if they understood well what was asked of them. The questions for both tasks have the same structure, *i.e.*, introducing a scenario followed by a question. Through these questions we evaluated whether they could list the contexts that are currently activated, the features that are triggered (*i.e.*, selected) and the features that are currently activated. While the first question about what contexts are activated suggested a different scenario, the two other questions followed the same scenario. For example for the *Task 1*, the questions are the following: “Assume a citizen wants to consult instructions about an earthquake risk (when no earthquake is currently occurring), can you tell us which context(s) is(are) active at that moment?”, “When the government announces the occurrence of a flood, it will inform the citizens about its characteristics (*i.e.* its severity and impacted zone). Can you indicate which feature(s) will be **selected** thanks to the context-feature mapping?” and “When a citizen launches the application, it detects that a new flood is active and receives information about this flood. This information describes that the severity is medium and its impacted zone is near the river. The system informs the citizen about the characteristics of the flood. Can you indicate **all** features in the feature diagram that are currently active?”.

Next, they received a quick introduction to the CONTEXT AND FEATURE MODEL VISUALISER tool as a preparation for their second task. To train them we asked them to answer the three same questions for the same task but with the visualisation tool now.

Then we switched the tasks for this second exercise. Group A now had to develop *Task 2* while group B had to implement *Task 1*. Again, both groups received at most 25 minutes to finish this task and then had to answer two new questions to verify their understanding of the task. These questions are the same questions we asked to the other group for their first assigned task. However we focussed only on the questions about what features are selected and which features are currently activated.

Finally, all subjects were asked to answer some questions regarding how they perceived the usability and usefulness of the visualisation tool.

Raw results

We now present the results of this user study. Figure 13.5 illustrates the background knowledge of our participants at the start of our user study. For each of these questions, they had to assess their skills on a five-level Likert scale (From “no expertise” to “expert”). Despite only 33 participants (of 34) answered to this part of the survey, we can clearly see that our participants had a good background in *programming in general* (almost 50% of positive responses) and in *object-oriented programming* (~55% of positive answers). However they (~94% of negative values) did not consider to be expert in the

Ruby programming language. They also were not confident about their skills in *feature modelling* (~64% of negative answers versus ~12% of positive responses), *context-oriented programming* (~70% of negative values versus ~12% of positive values) and in our *FBCOP* approach (~73% of negative answers versus ~6% of positive responses).

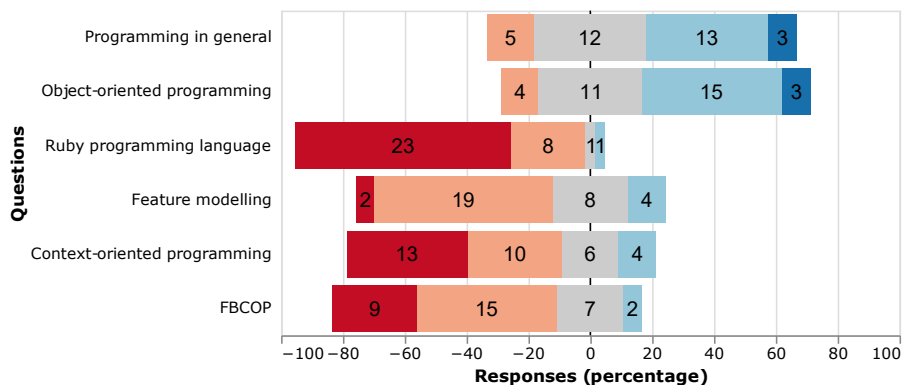


Figure 13.5: divergent stacked bar diagram summarising the results of our closed questions about the background knowledge of our 33 participants.

To assess if they understood our approach, we asked them some questions. As opposed to the previous user study, we noticed that almost all of them failed by comparing their responses with the correct ones. This is true for the first exercise without using the tool, but also for the other questions when they had to use our tool. Having observed these results still indicating a lack of understanding of the complexity of the underlying FBCOP approach, at this point we adapted the goals of the user study to only assess the usability and usefulness of our CONTEXT AND FEATURE MODEL VISUALISER tool (and no more on the underlying approach). For that we showed them our solution including the implementation of the two tasks. Then we asked them to explore and understand our solution with the tool. As such we tried to still get some preliminary results about the CONTEXT AND FEATURE MODEL VISUALISER tool and if it could help them to better understand our approach.

Their opinions about our visualisation tool are depicted in Figure 13.6. All 34 participants answered this part of the survey. We see that the static and dynamic representations offered by the tool are understandable for our participants (*static representation*: ~59% of positive values against ~20% of negative values; *dynamic representation*: ~59% of positive values against ~26% of negative values). The five values ranged from “hard to understand” to “easy to understand”. But they had a preference for the *dynamic* visualisation (~50% counting the positive responses) over the *static* one (~21% counting the negative answers). When asked the following question “Does such a visualisation

tool help you to better understand such an approach when you develop context-aware programs?”, a positive tendency (~56% of positive values) was emerging (against ~15% of negative values). Finally the feelings were more mixed about the ease of use of our CONTEXT AND FEATURE MODEL VISUALISER tool. While ~38% found it *easy to use*, there were still ~26% that disagreed and ~35% that had no opinion about that.

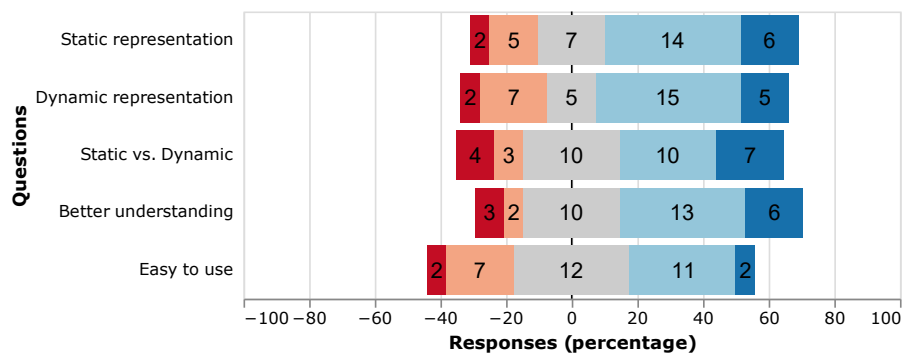


Figure 13.6: divergent stacked bar diagram presenting the results of our closed questions about the usability and the usefulness of our visualisation tool.

Discussion

Even if our participants had good knowledge of programming and object-oriented programming in particular, they were not comfortable with more dynamic approach such as context-oriented programming and feature-based context-oriented programming. Their weak knowledge of the *Ruby* programming language can be justified by the fact that only a few of the students had prior experience (beyond what they saw in a two-hour preparatory session) in *Ruby*. Despite we taught them the feature modelling notion during the course, they were not confident with this modelling. This could be explained by the complexity of this new modelling to which they are not yet accustomed. In addition, as the lecture and exercises about feature modelling were given at the beginning of the semester and the study was conducted at the end of it, maybe they did not yet revise or revisit this notion yet. (The study was carried out **before** the exam session where they had to study the material in more depth.)

Regarding the fact that nearly all of them failed the questions to assess their understanding of our approach, we can infer that our underlying approach was not well-understood yet. This probably stems from a poor preparing phase due to timing constraints. In addition to the missing time, our FB-

COP approach remains intrinsically complex. Such an approach with many new notions (as the contexts, features, mapping and so on) can be heavy to learn and probably too complex to get fully acquainted with in only a two-hour session.

Nevertheless, despite the perceived difficulty of our approach, our participants did seem to be interested by how this visualisation tool could help them understand our approach and how FBCOP applications are implemented. Furthermore, they found the visualisation strategy interesting and liked its dynamic aspect. In the open-ended question about which functionality of the tool was regarded as most useful, many participants answered that the ability to replay *Step-by-step* changes dynamically was a strength of the tool and was really useful.

However two weaknesses were raised as well. The first one is that it missed a button to step back in the dynamic visualisation. The second is that our current visualisation does not really scale well to support larger context and feature models.

Finally the mixed results on the ease of use of our tool can probably be explained by the high complexity of the underlying approach as well. Indeed, assessing the usability of such a visualisation tool is intrinsically linked to the understandability of the underlying programming approach. Therefore, when they had to learn how to use our tool and our underlying approach simultaneously, they could be a bit confused on its ease of use. The distinction between ease of use of the underlying programming approach and ease of use of the visualisation tool may have been hard to make in their minds.

Conclusion

To assess the usability and usefulness of our CONTEXT AND FEATURE MODEL VISUALISER tool, 34 programmers participated to this user study

Similar to the previous user study, we used the same evaluation strategies (*i.e.*, *demonstration* and *usage*) to perform our user study [Led+18]. However this user study did not go as planned. Therefore we provided our solution and asked them to understand it to still provide their feelings about the visualisation tool. For this user study, they had to answer a Likert scale questionnaire.

Despite the clear complexity of our FBCOP approach, the participants in this study did seem to agree that our visualisation tool can be useful for developers when learning our approach. However, as for the previous preliminary user study which we conducted on our other visualisation tool, it is clear that we should perform a more elaborate user evaluation in which the participants will deeply use the visualisation tool in their implementation and debugging tasks. However, when doing so we should put much more effort in teaching and training our participants to better understand this new paradigm, before

asking them to evaluate the visualisation or other tools that support it.

13.4 First complete user study

Despite the limited initial evaluations of both of our visualisation tools, we considered they had not yet been sufficiently validated from a usability and usefulness point of view. We came to the conclusion after these initial validations that the participants of our studies still needed a better understanding of FBCOP as well as some specific clarifications on our approach. Indeed, we observed that the results from these initial validations were not only influenced by the usability and usefulness of the tools, but also biased by the participants' core understanding (or lack thereof) of the underlying FBCOP software development approach. Furthermore, in these initial studies both tools were evaluated in isolation by different participants in different years. Yet, we believe that both tools are complementary in the sense that each visualisation completes the other one. Finally the focus of both initial studies was solely on the visualisation tools, but we never validated our entire FBCOP approach itself in the past.

Therefore we decided to conduct a first more complete user study with master-level students, during a software engineering course in the year 2020. As for the previous user studies, the participants played the role of FBCOP designers and developers to create a context-oriented application with our approach.

From 41 participants, a large majority (35 out of the 41 students) was aged between 20 and 25 years old. The others were just a bit younger or older.

In the next subsections we will first describe the set-up of our user study. Then we will present the gathered results before analysing and discussing them. To conclude this section we will summarise this user study.

User study

Before starting to introduce any FBCOP concept, we collected the participants' background to assess their prior design and modelling skills, how comfortable they are with object-oriented programming and who had already heard about (feature-based) context-oriented programming before.

Next, regardless of their background, to avoid our participants having insufficient understanding of the underlying FBCOP programming paradigm, they received 27 hours of training (both theoretical lectures and hands-on sessions) over several weeks. Contrary to the past user evaluations where dedicated only a limited amount of time to teach our approach, for the current study we entirely oriented the course around it. Table 13.1 summarises the

different topics we taught and how many hours were spent on each topic, for both theory and practice sessions.

Table 13.1: Time spent to introduce background material to participants in the user study of 2020.

Topic	Theory	Practice
Feature modelling	2	2
Introduction to COP and FBCOP	3	6
Programming in Ruby	2	4
Feature-based context-oriented programming	3	5
Total	10	17

For the entire user study (except the participants' background questionnaire), we asked the participants to work in pairs, doing pair programming to build the case study, but also when completing the questionnaires we gave them. This allowed them to solve some of the problems they encountered by themselves without us having to intervene and thus risk biasing the results.

Throughout this study several questionnaires were taken from our participants to gather their feedback. We always ensured that all the notions assessed in the questionnaire were previously presented and worked on to guarantee that our participants had the minimal required background before responding to the questionnaire.

As our approach relies on feature modelling, we first taught this modelling approach and gave a practical session on it. Then we introduced the context-oriented programming and FBCOP paradigms. We followed this lecture with a quick questionnaire to assess whether our participants understood well what is a context-oriented system, a context and a feature. We concluded it by asking them to explain what is the difference between context-oriented programming and FBCOP.

After this introduction, we organised a context-feature modelling workshop. In this workshop, they had to design two context-feature models for two assigned case studies. For each model, they had to define and structure the contexts, features and define the mapping between the contexts and features.

For the first context-feature modelling exercise, we assigned a case study and asked them to model it without any guidance, *i.e.*, without our supporting development methodology. Half of the pairs were asked to model a *smart calculator system* while the other half were to model a *smart messaging system*.

For the second modelling exercise, they had to design the other case study but following our development methodology which we presented to them be-

fore asking to complete the exercise. To avoid bias because they already knew the application domain of their previous assigned case study, we swapped the case study between the groups. The pairs who had to model a *smart calculator system* before now had to design a *smart messaging system* and inversely.

After each modelling exercise, we asked them to complete a questionnaire to collect their feelings about the complexity of designing such systems without or with the help of our proposed development methodology. In this workshop, we focussed only on the design of the context-feature model without paying attention yet to how the features should adapt the application classes.

Once the modelling workshop was finished, we taught our participants the *Ruby* programming language and gave them some exercises on the specific language constructs they should understand and use during the study.

Then we introduced the FBCOP programming language with a toy example. We followed this exercise by asking them to use it to implement the last case study they designed (*i.e.*, either their *smart messaging system* or their *smart calculator system*). During this implementation workshop, we also presented how they should build and compose at runtime their user interfaces with the UIA library. At the end of this implementation workshop, we asked our participants to respond to a questionnaire to gather their feedback about the expressivity and complexity of our programming language.

Finally we introduced both of our visualisation tools to assess their usability and usefulness. Before filling in the questionnaire on our tools, each pair of participants had to perform several tasks to learn how to use and exploit these tools. We first asked them to use the tools to inspect a toy example we gave them. Later we asked them to use our tools again, but now on their implemented case study to examine whether all contexts and features got activated when expected, but also whether the features adapted the correct application classes of their system and the system changed its behaviour as desired.

Then we asked them to find and explain two bugs that we had purposefully introduced in an extension of our toy example. The first bug concerned the ordering of the adaptations of an application class: a same method of a same application class was adapted by two different features, but because these adaptations were not applied in the correct order the behaviour was not as desired. The second bug was a design error related to the fact that a context could not get activated correctly, since another context was already active and these contexts were in an alternative constraint. We designed the bugs in such a way that the we expected the *FEATURE VISUALISER* tool to be more useful to find the former bug and the *CONTEXT AND FEATURE MODEL VISUALISER* for the latter. Nevertheless, we did not provide this hint to the participants, and they were allowed to use either tool at will to look for the bugs.

Once they finished these tasks, each group had to fill in a questionnaire about the usability and usefulness of both visualisation tools. For that we used the *System Usability Scale* (SUS) [Bro96], a widely used ten-items questionnaire measuring generic aspects of usability. The disadvantage of using a SUS questionnaire was that the questions were not specific to our particular approach. We therefore complemented the questionnaire with additional more specific closed questions, dedicated to our approach, using a Likert scale. To avoid interpretation bias of these closed questions we made sure to provide open-ended questions as well to allow participants to elaborate on their answers.

Finally, we interviewed six volunteers¹ to better understand their opinion of the tools.

Raw results

Now that we have described how we set up this user study, we show the results we collected through the different questionnaires. We first show the background of our participants, followed by their understanding of the different notions of FBCOP paradigm. Next, we summarise their feedback about our development methodology, our programming framework and finally our visualisation tools.

Also note that, whereas 41 students participated in the background survey, they worked in pairs for the remaining user study. Therefore the number of responses for the other questionnaires is divided by two: we thus had maximum 21 answers for these questionnaires (not all pairs of participants answered all questionnaires).

Participants' background Figure 13.7 summarises the background of our 41 participants. For both questions about their skills on *design and modelling* and in *object-oriented programming*, they had to answer on a five-item Likert scale (in the range from “no expertise” to “expert”). When we asked them the question “*what is your level of expertise in designing and modelling object-oriented applications?*”, only ~30% (counting only positive values) considered that they had a good level in design and modelling. But for the question “*what is your level of expertise in object-oriented programming (regardless of your preferred object-oriented programming language)?*”, almost half (~46% taking only positive values) said that they had good knowledge of object-oriented programming. When asked whether they had already heard about context-oriented programming, only ~20% claimed that they had, but when

¹Since keeping anonymity was impossible during these oral interviews, we only worked with students who volunteered to partake in the interview, and we did not conduct any interviews before having fully encoded and analysed the anonymous questionnaires.

asked to explain it in a few sentences, only ~15% of them managed to define it correctly.

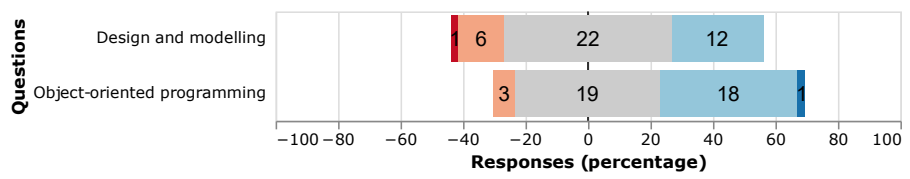


Figure 13.7: Divergent stacked bar presenting the background of our 41 individual participants.

Comprehension of FBCOP notions For the questionnaire on their understanding of important FBCOP concepts, we received 20 responses. The questions for which they had to explain what is a context-oriented system, contexts and features they mainly succeeded. Even if the definitions they provided were sometimes not totally exact, it was clear that they understood the different concepts. The last question about the difference between context-oriented programming and feature-based context-oriented programming was a bit more tricky. In many responses the main difference was not mentioned.

Design methodology After a first context-feature modelling exercise without using a specific design methodology, we asked their feeling about the perceived difficulty of designing a context-oriented system for a first time. All 21 pairs answered to this question through a five-item Likert scale (from “very complex” to “very easy”). Figure 13.8 shows their feelings and no clear tendency emerges. While ~28% found it easy, ~24% suffered from the complexity of designing such an application and ~48% did not have a specific opinion.

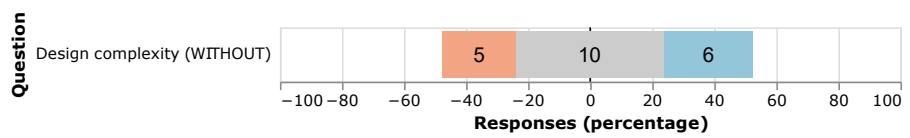


Figure 13.8: Divergent stacked bar presenting the participants’ feelings about the perceived difficulty of a first experiment in designing a context-oriented application without any supporting design methodology.

As we did not propose a particular design methodology yet, we asked them an extra question to describe the kind of structured approach or series of steps they used to come up to design their first context-oriented application. In their responses, we can observe two different methodologies.

A first methodology was to start by designing the contexts. Once the contexts were defined, they followed their application design with the features. During this step or afterwards, they also defined the mapping to describe for which contexts which features had to be present in the system. The time when the context (feature) model was built depended on the groups. Some pairs designed the context model after defining the contexts and before continuing with the features. Other pairs preferred to list the relevant contexts and features before they modelled them into a context and feature model, respectively. The methodology to start by designing the contexts was used by 5 pairs. While one pair found it was complex to design their first application, the other pairs (4) did not have a clear opinion.

The other methodology was the opposite, *i.e.*, they started thinking about the features before thinking about the contexts. 8 pairs applied this methodology. As opposed to the previous approach, the opinions were more clear. While 3 pairs found it was complex to design an application like this, 4 groups found it was easy with this methodology. One pair who found it was complex explained in their response that they did not quite understand how they could relate the contexts and the features. On the other hand, another pair who stated that designing such a system was easy said that with such a methodology “*the mapping between the two felt natural*”.

The 8 remaining pairs did not follow a specific methodology. However one of these pairs explained that they played the role of users to discuss about the features and what different users could want such a system. But they did not explicitly describe in which order they tackled the problem, *i.e.*, if they started with the contexts or the features. Nevertheless, they found it was easy to design a first context-oriented application.

Table 13.2 summarises the ad-hoc methodologies we can observe from our participants’ feedback to the open-ended question. For each of these methodologies used, we show how the participant pairs felt (*complex, no opinion, easy*) about using their ad-hoc methodology to design their first context-oriented application.

Table 13.2: Summary of the different approaches our participant pairs used to design their first context-oriented application, and how they experienced this approach.

	<i>Complex</i>	<i>No opinion</i>	<i>Easy</i>
No specific approach	1	5	2
Starting with contexts	1	4	0
Starting with features	3	1	4
Total	5	10	6

Next we asked them to design another context-oriented application but now using our specific design methodology (cf. the *requirements* and *design* phases of our development methodology). After this they again had to answer a questionnaire on the design methodology we suggested. Figure 13.9 summarises the answers of the 21 participant pairs to the closed questions on a five-item Likert scale. For the question “*did the proposed development methodology help you as context-oriented designers to better elicit the contexts and features of your context-oriented system?*”, ~52% (positive values only) confirmed that our methodology allows a better (*contexts / features*) *elicitation* against ~24% (negative values only). However they did not have a clear opinion about the relevance of explaining *contexts and features rationale*. A positive tendency appears for the question whether our design methodology allowed to better *structure the models*. While ~70% (positive values) of the pairs agreed with this statement, only ~19% (negative values) disagreed. ~52% (positive values) stated that it also helped them to define the *mapping model* against ~28% (negative values). Many pairs (~66% positive values) found the *complexity of our design methodology* easy, while only ~10% (negative values) did not. As they could compare the design of an application without and with a provided design methodology, we asked the question “how do you assess the *complexity of designing a context-oriented system without* using our development methodology?”. Unfortunately from their responses we cannot infer if it is more easy or more complex to design a context-oriented system with or without our provided design methodology. Nevertheless there is a clear tendency that our methodology helped them to *better comprehend* our underlying approach (~66% in favour versus ~10% against).

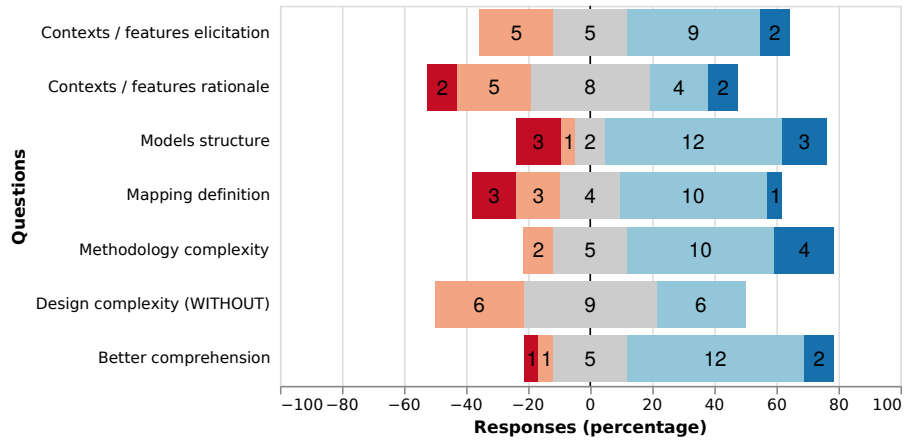


Figure 13.9: Divergent stacked bar showing the results of our closed questions regarding the design methodology (the requirements and design phases of our development methodology).

When asked “*can you mention one or more positive aspects of our proposed development methodology?*”, they often responded that our methodology helped to structure and organise their ideas in order to save time. One group using the methodology by starting designing the contexts for the first exercise also claimed it was more easier to start by designing the features to discover the contexts then. Another pair using the methodology by starting to think about the features for the first exercise said “*it really enabled us to structure our thoughts in a more rigorous and natural way*”. A last pair that answered many times with a “not at all” answer (worst value) to our closed questions justified their choices by the fact that they already followed a similar methodology instinctively (*i.e.*, by starting to think about the features). Therefore using our methodology had not a real added value for this pair. Yet they did confirm that our methodology is an “*easy-to-follow methodology*”. When asked to mention some less interesting aspects, they mainly answered that describing a rationale for all contexts and features was painful. (In this first version of our methodology, we asked to provide a rationale for *all* contexts, features and mapping relations.)

Going in more depth, when we correlate the answers of the questions about the complexity to design a context-oriented system without using our methodology (in Figures 13.8 and 13.9), only 8 pairs changed their opinion after using our methodology. For these pairs, 4 pairs found that it is easier to design without a methodology (even if they agreed that our methodology is easy). In the open-ended questions, 3 of these pairs explained that they already followed such a methodology before we provided it and thus saw little added value in using that methodology. Nevertheless they explicitly agreed that our supporting methodology did provide a more formal process and a “*clear cut path to follow*”. The 4 other pairs said it is more complex to design such a system without a methodology. They explained that our methodology adds more formalism and clear steps to follow. One group precised that it allows to better structure the information to facilitate the integration of new stakeholders in the project.

Programming framework After their design and modelling tasks our participant pairs had to implement an application with our programming framework, followed by another questionnaire. The purpose of this questionnaire was to assess the expressivity and complexity of our programming framework. For this questionnaire, 18 pairs of participants provided their feedback.

Figure 13.10 displays the responses for the closed questions. Many pairs (~77%) appreciated the expressiveness to declare a *context and feature model*. They also liked the expressivity of how to declare a *mapping model* (~83%). Their feeling was different for the expressiveness of our programming framework to *define the features and application classes* of a system. While ~39%

found it was not expressive enough, only ~28% considered the expressiveness sufficient to define them. When asked to evaluate the *file structure* when creating a context-oriented system and if it is readable enough for them to know which files must be modified, a clear negative tendency was sketched (~55% negative values versus ~11% positive values). ~66% also found that our *programming language* was complex to use to create a context-oriented application (against ~11% of positive values). Finally when we asked about the potential *complexity to develop such a system without* using our programming framework, half of our pairs thought it would be complex whereas only ~11% thought it would be easier.

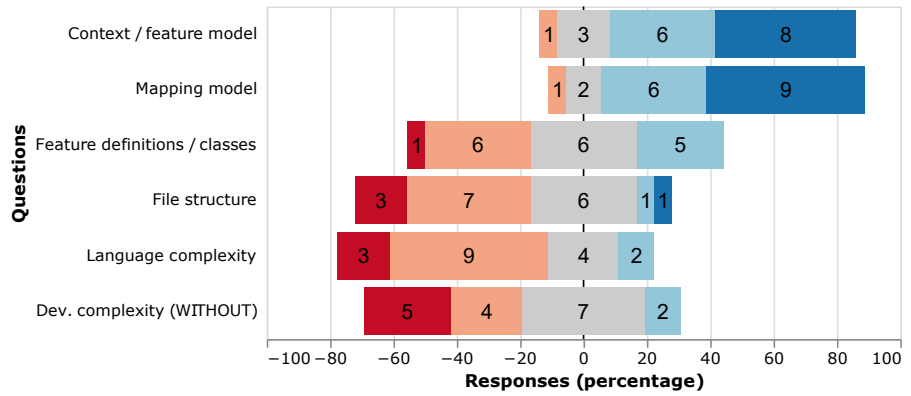


Figure 13.10: Divergent stacked bar showing the results of our closed questions regarding the programming framework used during the implementation phase.

In the open-ended questions, when asked about the positive points of our framework many pairs mentioned the simplicity to declare the context and feature model. They liked it since they could simply translate their designed model with our programming framework. One pair precised that with our programming framework “*the source code of applications, once they are built, is clean and readable*” and added that “*the decomposition of responsibilities by feature makes it easier to create an application*”.

The questionnaire also asked what aspects of our programming framework could be enhanced. One main remark was that the learning curve for learning such a new framework was steep. It takes time to get into the right mindset and even after a practical session, it is sill complex to use. One pair said explicitly that they suffered “*to represent the global application and to know how it works*”. Another important comment was that sufficient documentation was lacking. Furthermore, two pairs found that the error messages were not suited for their debugging tasks. Two other pairs precised that keeping all the features in the same folder (without splitting them into subfolders)

was not suitable for building larger applications. Finally two groups did not like the underlying *Ruby* programming language and some others found that some functionalities were missing from our UIA library. (This UIA library they used was a first prototype we already evolved it after getting their feedback.)

Visualisation tools At the end of the user study, we introduced our visualisation tools. To assess our participants' understanding we asked them to find and explain two bugs we deliberately created in the toy example. Despite all pairs did the exercise, only 18 responded to the questionnaire on the visualisation tools.

CONTEXT AND FEATURE MODEL VISUALISER Figure 13.11 outlines the participants' answers to a SUS (System Usability Scale) survey about the usability of the CONTEXT AND FEATURE MODEL VISUALISER tool. In a SUS survey, positively phrased questions like "I like to use this system more often" are alternated with negative ones like "I find this system to be more complicated than it should be". For each positive question, we got high score. ~67% of the pairs would like to *use* this visualisation tool *frequently*, ~95% found the tool *easy to use*, ~72% found the *functions well integrated*, ~83% would imagine that most people would *learn* to use it very *quickly*, and ~72% felt *very confident* using it. Their positive answers were confirmed by the fact that they responded negatively to most negatively phrased questions. ~67% disagreed that the visualisation tool was *unnecessarily complex*. All pairs thought that *no support* from a technical person is needed to use this tool. ~78% confirmed that there was not *too much inconsistency* in our visualisation tool. ~72% did not find our tool *very cumbersome to use*. ~83% did not need to learn a lot of things before they could get going with this visualisation tool thus indication a **low learning curve** for the visualisation tool.

In addition to the standardised SUS questions we asked more specific questions to better highlight the strengths of our CONTEXT AND FEATURE MODEL VISUALISER tool and points we should improve. Whereas this tool shows mostly a static representation of the contexts, features and application classes and the dependencies between them, it also features some dynamic aspects by indicating which of those are currently activated, selected or adapted. Figure 13.12 displays a divergent stacked bar of the participants' answers to the additional closed questions.

The first two questions asked whether the *static* or *dynamic representation* of the visualisation is understandable; for the third the participants were asked whether they considered the *static* (negative answers) or *dynamic* (positive answers) aspects as more interesting to visualise. We can clearly observe that both aspects were largely considered as understandable. Never-

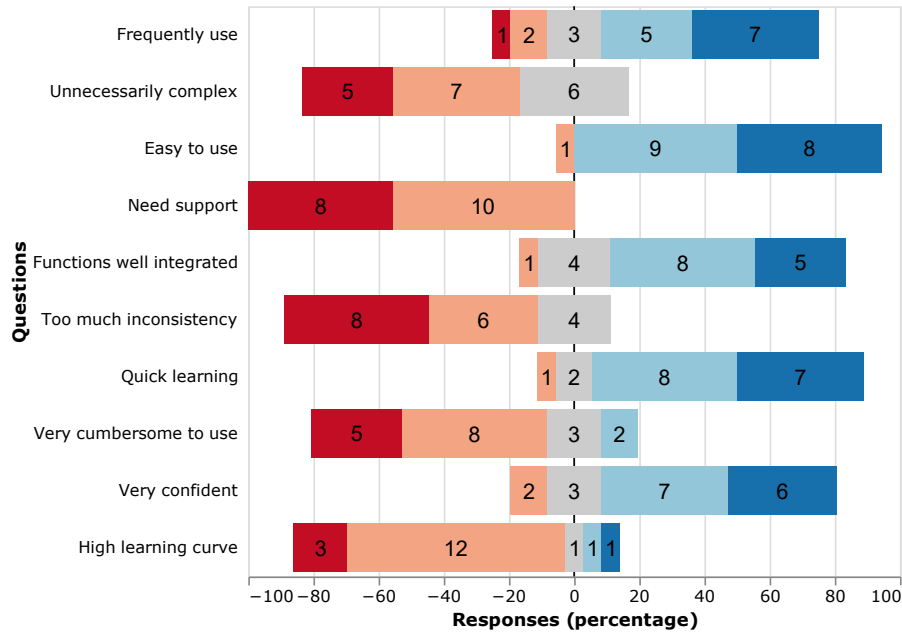


Figure 13.11: Divergent stacked bar presenting the results gathered from the SUS questions about the CONTEXT AND FEATURE MODEL VISUALISER tool.

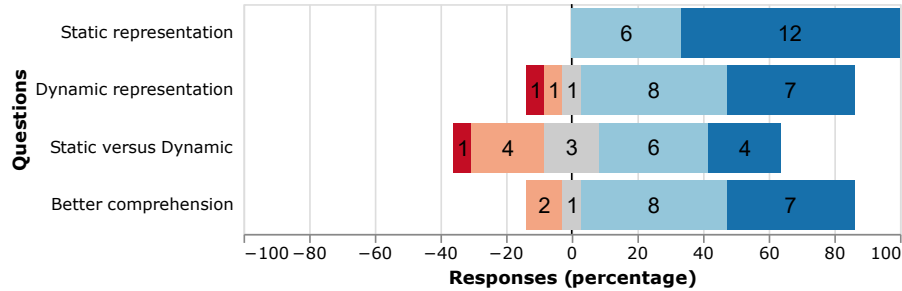


Figure 13.12: Divergent stacked bar showing the results of our additional closed questions regarding the CONTEXT AND FEATURE MODEL VISUALISER tool.

theless the dynamic aspect was considered as more interesting (~56% of positive responses versus ~28% against). To our final question asking whether this tool would help them *better understand* the approach when developing feature-based context-oriented programs, the response was also largely positive (~83%).

Finally, analysing the gathered answers to the more open-ended questions we can observe the following strengths for this visualisation tool. The tool

is considered useful since it helps to better understand and see explicitly the different models and how they evolve dynamically throughout the execution of a FBCOP application. Several groups stated that this tool is “*easy to use*” and “*intuitive*”. Some participants clarified that it helped them to spot quickly errors in the models or when a context or feature is (de)activated while it should not be (de)activated. They also indicated that the dynamic aspect of the visualisation was more interesting to see the evolution of the configuration of the models at runtime. Some participants said that they liked the *Step-by-step* navigation for inspecting dynamically in what order the different contexts and features got (de)activated.

Nevertheless they did identify two weaknesses of this tool. The first concerns the scalability of this visualisation, causing readability issues when the diagrams become too large. The other issue was that the mapping between the contexts and the features was considered hard to read, because of the many overlapping links from the context model to the feature model. A same remark was made for the links between the feature model and the application classes.

FEATURE VISUALISER Figure 13.13 presents the participants’ answers to the SUS survey about the usability of the FEATURE VISUALISER tool. As for the previous tool we can immediately observe that the tendencies are alternating for each question; Again, this is encouraging as it means that positive questions gather more positive answers while negatively phrased questions collect more negative answers. ~61% would like to *use* this visualisation tool *frequently* when developing FBCOP systems. They (~61%) found the *functions* in our tool *well integrated* and only some (~11%) found there was *too much inconsistency* in our visualisation tool. Most (~78%) found our tool *easy to use*. This was confirmed by the fact that ~72% disagreed that our tool was *unnecessarily complex* and ~89% thought that *no support is needed* from a technical person to be able to use it. ~83% of our pairs could imagine that this tool can be *quick to learn* to use it. And ~67% thought they did not need to learn a lot of things before they could get going with it (*learning curve*). Half of our pairs confirmed that our visualisation tool was not *very cumbersome to use*. However we did get a more mixed result when asking they felt *very confident* using it. Only ~45% were very confident, but ~22% had the opposite feeling.

One group had the feeling that using this tool for small programs was “too much”, because they found that our architecture was already sufficiently easy to understand. So these participants considered the tool as too cumbersome to use as well. Nevertheless, the same group claimed that such a tool could be “more beneficial” when applied to larger programs and thus still believed in the potential of using our tool on larger cases.

In addition to this SUS survey, again we polled in more depth for the main

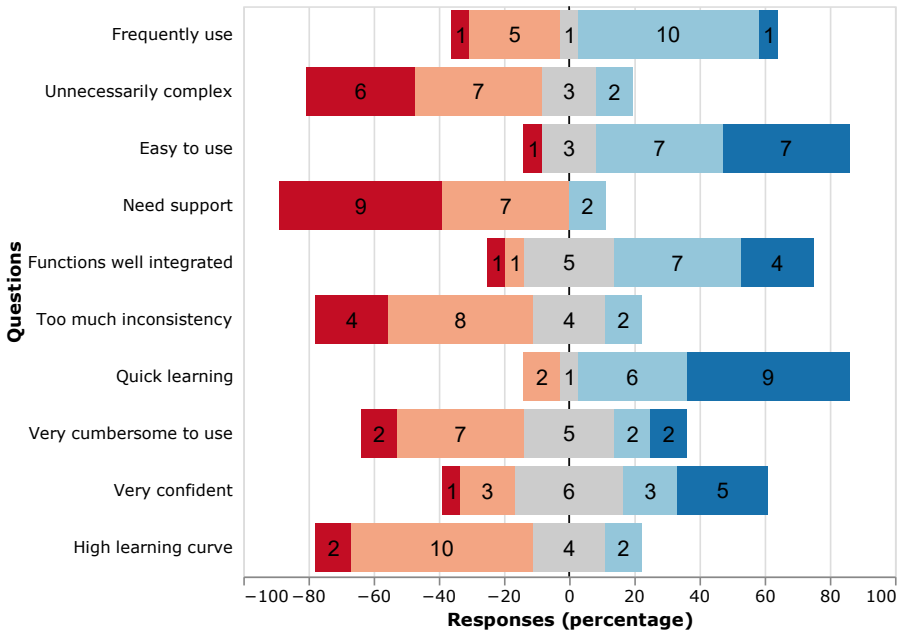


Figure 13.13: Divergent stacked bar presenting the results gathered from the SUS questions about the FEATURE VISUALISER tool.

strengths and weaknesses of the FEATURE VISUALISER tool, through additional closed and open-ended questions. Figure 13.14 shows a divergent stacked bar that summarises the participants’ answers to the closed questions. The *dynamic representation* of the activated entities (contexts, features, ...) in the visualisation was mostly understandable. Responses were more mixed to the question whether this visualisation made the *control flow* of the FBCOP architecture sufficiently clear. The two last questions asked if this visualisation tool could help them, either to *better understand* a program developed using this programming approach, or to *help to debug* such a program. They nearly unanimously confirmed this to be the case.

Overall, the participants to the user study agreed to say that the FEATURE VISUALISER tool is useful. They confirmed that it helps to easily visualise what feature parts adapt what application classes and in which order the feature parts are (de)activated. They also felt that with the help of this tool they could easily spot errors in the order of the (de)activations of the features.

Again, scalability was considered as the first weakness of this visualisation tool as the current layout may not be the most appropriate when dealing with larger programs consisting of many contexts, features parts and application classes. Another issue that was mentioned concerning the layout was the jiggling effect on the entire visualisation whenever the visualisation is

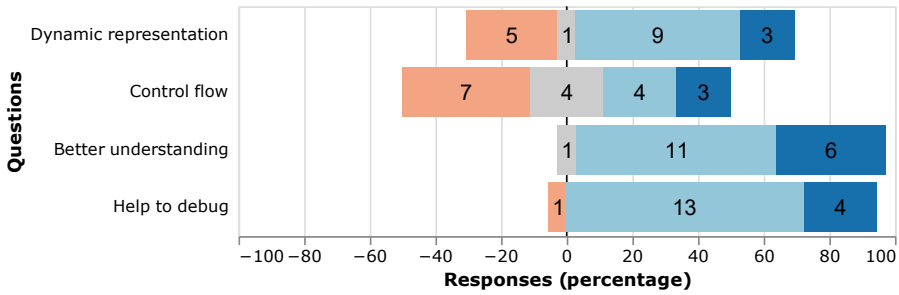


Figure 13.14: Divergent stacked bar showing the results of our additional closed questions regarding the FEATURE VISUALISER.

refreshed. Some also considered that it displays too many entities without giving clear visual clues about new ones being added when the visualisation is updated. The participants also regretted that the visualisation did not allow them to move objects around, for example to fix or avoid crosscutting edges. Finally, participants regretted the absence of a ‘back’ button in the visualisation because this would really have improved their user experience when navigating through this visualisation. (This ‘back’ button was implemented later than this user study.)

Complementarity and usefulness of the visualisation tools We finally polled our participants’ opinion about the complementarity and usefulness of both visualisation tools. As can be seen in Figure 13.15 there was a positive tendency (50% of positive values versus ~22% of negative values) for considering that both tools were *complementarity*. Some participants, however, considered the tools too similar and suggested combining them in a single integrated tool. Whereas these participants agreed on the usefulness of our visualisation tools, they did not find an interest in the separation of their functionalities into two different tools.

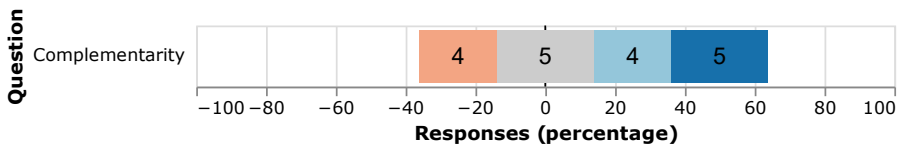


Figure 13.15: Divergent stacked bar exposing the results of more closed-questions about the complementarity of both visualisation tools.

For ~89% of the pairs, these tools were useful to get a *better comprehension* of our FBCOP approach as depicted in Figure 13.16. Out of curiosity, we also asked their opinion about whether we should have provided them access to

these visualisation tools earlier in their development process. Figure 13.16 shows their feedback. ~61% of our pairs thought that they could have gotten a *better comprehension* (*early use*) of our architecture and approach if we would have provided them earlier. However the complexity of using them could have been harder, looking at the results (*tools complexity* (*early use*)). Despite the slightly positive tendency, only ~39% would have considered them as easy to use while ~22% disagreed with that.

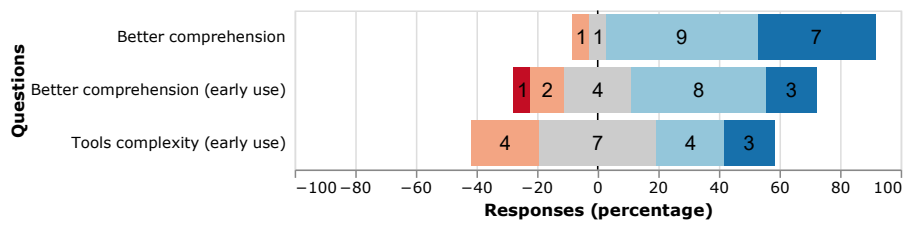


Figure 13.16: Divergent stacked bar exposing the results of more closed-questions about the usefulness of both visualisation tools.

Discussion

Now that we have shown the raw results from our participants, we will discuss and interpret them.

Design methodology Generally speaking, we do get the feeling that our methodology helped our participants to design their applications. Firstly, our proposal to start by designing the features and continuing with contexts seems a bit more intuitive. Almost an half of the pairs (10 pairs) followed it naturally for their first task. 8 pairs explicitly stated it and 2 more pairs claimed later that they followed a similar methodology for their first task. Many of them also agreed to say that this approach served to better structure their modelling. Moreover three pairs that did not instinctively follow our methodology (for the first task) did precise that our design methodology would allow them to save time and effort. Of course these results must be taken with caution because they already had a first experience on modelling such systems without following our methodology.

Nevertheless the complexity of designing a context-oriented application without using a specific methodology seems to be relatively constant before and after using our design methodology. Maybe this unexpected result could be intrinsically related to the complexity of designing such applications that were novel for them. It is true that when learning a new paradigm, it often takes time and experience before we get sufficiently confident to go beyond the initial perceived complexity.

Finally as our methodology was only a first prototype we must continue to improve it to simplify some of its less interesting aspects. For example we should define the rationale only for non-trivial contexts, features and mapping relations. In addition, even if it is a risk to complexify the methodology we must also integrate the design of application classes and how they are adapted by what features. Without forgetting that we should also draw rough wireframes of the user interface aspects to get a more complete analysis and design methodology. Many of these aspects were not addressed yet by the methodology.

Programming framework When we analyse the results we got for our programming framework, we observed a negative tendency. While the expressiveness of our programming framework to declare the models seems really good, the complexity appears when our participants had to code concretely the features and application classes. This can be explained by the complexity to learn a new programming approach in a restricted amount of time, the different features not yet fully present in our suggested programming framework, and the fact that we used a new programming language with which they were not fully comfortable yet.

A first reason that could explain the perceived complexity to implement feature definitions and application classes is they did not work on the design of these classes (with a kind of class diagram or similar). In addition they did not yet think about what features could adapt what application classes. This design step was not tackled in this user study due to time constraints because of the number of sessions we have already used for the study.

Another reason could be the steep learning curve of our programming framework. This seemed really complex for our participants despite the introduction and the tasks we gave them. But this does not seem too surprising because harnessing a new programming paradigm is not straightforward at all. This takes time and requires gaining experience through practice. For example one does not become an expert in object-oriented programming by just studying the theory. Maybe we would have been spent even more time on the practice of using our approach before asking them to use it. However we were limited in the time since we did this user study as part of a course during a semester.

To better help future developers, we must first take into account our participants' comments to further improve our programming framework, such as proposing a more complete documentation with concrete and specific examples. With such examples they would have concrete code snippets to better understand our programming framework and mimick them. We should also improve the error messages so that they can better understand the different errors they can encounter.

Another way to help them during their implementation stage could be to provide our visualisation tools earlier. They stated the providing these tools earlier could help them to better understand our architecture and approach. One pair also precised that such visualisation tools could have helped them to better find and understand the errors. For example, when they met an error in the (context and feature) models due to an unsatisfied constraint of the model at the (de)activation of some entities. With our visualisation tool, they could have pinpointed directly the problem without having to waste time to try and understand the error. However another pair warned that such visualisation tools should not be provided too early otherwise they would not see “*how and why they are useful*”.

Another explanation about the complexity to implement feature definitions and application classes could come from the *Ruby* programming language on top of which we built our programming framework. Despite their good skills in object-oriented programming and in programming languages in general (they mainly had expertise in the *Java*, *C* and *Python* programming languages), using a new programming language can be another obstacle in the implementation of software systems. This could come from the new syntax or the language constructs that are less familiar to them (such as for example *Ruby*’s mixin modules). We could observe they were not really comfortable with *Ruby* because many pairs criticised the fact that the folder containing the features definitions could become quickly large for bigger applications. If they would have been more fluent in *Ruby*, they could have quickly observed that they could solve this problem by splitting this folder in sub-folders with a minimal modification in an application file. So, even if we taught the basics of *Ruby* during a lecture and provided them some exercises, we could have gone further in the teaching of *Ruby* to our participants or offering more documentation with concrete examples to help them in their implementation of their context-oriented application.

Despite this negative tendency, we can conclude that our programming framework remains interesting to reduce the complexity of implementing context-oriented systems with our programming framework. Indeed an half of the pairs thought it would have been more complex to develop them without using our programming framework. This seems to suggest that the programming framework can be of help in creating such systems and that much of the perceived complexity comes from the intrinsic complexity of such context-oriented applications. This could also be explained by the different abstractions and tools the framework offers them, such as for example the verification of the consistency of the different models.

Visualisation tools To assess the usefulness of our visualisation tools, we relied on the *System Usability Scale* (SUS) [Bro96]. A system that is considered

usable according to this usability scale will exhibit the alternating tendencies that are clearly present on Figure 13.11 for the CONTEXT AND FEATURE MODEL VISUALISER tool and on Figure 13.13 for the FEATURE VISUALISER tool. All positively phrased questions tend to have a largely positive tendency in the responses received, meaning that the respondents agreed with the statement. Conversely, most participants tend to disagree with the negatively phrased questions.

In addition to this visual representation of the alternating results, we can also compute a score [Bro96] to determine how usable our visualisation tools are. Following this computation, we obtained an overall SUS score of 76.4 for the CONTEXT AND FEATURE MODEL VISUALISER tool. According to Bangor et al.'s adjective rating scale [BKM09], this score can be interpreted as somewhere between *good* and *excellent*. This means that we can indeed consider that our tool is usable from a developer's point of view. Nevertheless our participants found two weaknesses. To deal with the scalability issue, the tool offers various filters and provides ways of hiding and collapsing contexts or features so that the developer can focus only on specific parts of the models he is interested in. Nevertheless, this functionality did not seem to suffice to address the perceived scalability issue, even for this smaller case study. Other graphical representations could be considered to attempt to solve this issue as future work. By changing its visual representation, the other issue about the visualisation of the different mappings could be also tackled. To recall this issue is about the readability of the mapping between the contexts and features, and between the features and application classes.

The positive assessment seems more mitigated for the FEATURE VISUALISER tool. When we calculate the SUS score for this tool, we get a score of only 69. Nevertheless, even with this lower score, according to Bangor et al. [BKM09] and Sauro [Sau11], we can still conclude that this visualisation tool is usable. The lower score of this tool as compared to the previous one is related to the fact that more participants considered this tool as *cumbersome to use* and felt *less confident* using it. Another reason to explain this lower score could be the fact that this tool had an implicit visual representation of our underlying architecture as opposed to CONTEXT AND FEATURE MODEL VISUALISER. We can also notice that the responses were more mixed when they assessed if the visualisation of the *control flow* of our architecture was sufficiently clear or not (see Figure 13.14), in spite of the additional training they received. This could perhaps mean that our programming architecture was still not fully understood by some participants. We are aware that our phased architecture is somewhat complex and since this visualisation is more technical and strongly linked to our architecture, their understanding of the tool may have been hindered if they did not fully understand the architecture and its flow. This might also explain why they found this tool less usable. The

participants also considered an issue in the FEATURE VISUALISER view about too many entities without clear visual clues about new ones being added. To address this issue we proposed a console showing textually what happened upon each update. However the tool users wanted an animation on the new visualised contexts, feature parts and application classes, so that their attention gets drawn to these new entities.

Finally even if they had a slight preference for the CONTEXT AND FEATURE MODEL VISUALISER tool, when asked again whether both visualisation tools are helpful to better understand our feature-based context-oriented approach, a large majority of the participants replied affirmatively.

Conclusion

We conducted a first complete user study with 21 participant pairs to assess our FBCOP design methodology, programming framework and visualisation tools.

Again, we followed the same evaluation strategies (*i.e.*, *demonstration* and *usage*) for this user study [Led+18]. However we better applied the ‘*how to*’ *scenarios* technique since we clearly divided the different steps of our workflow to conceive a FBCOP application.

By analysing this user study, we can conclude that our approach seemed to be interesting to design and implement FBCOP applications. Its strengths seems to be in the design of context-feature models and the visualisation tools we suggested to help the FBCOP programmers. However some aspects must be enhanced and added to simplify its comprehension and usage, as for example the proposed programming framework. We also noticed that our approach is intrinsically complex for our participants. Despite the effort we made to teach our approach to our participants (with respect to the previous user studies), this could have from its novelty and the different technologies in which we rely on to build it. This could have a potential reason of why its learning curve is steep. Finally, even if we gathered interesting results with this user study, we need to conduct other similar user studies to better identify the strengths and weaknesses of our approach.

13.5 Second complete user study

After the first complete user study which we conducted with many participants, we improved the design methodology and programming framework of our FBCOP approach. For the design methodology, we included a step to think about application classes and how they are adapted by the features. We also revised the rationale that the designers must describe. About the programming framework, we upgraded it by enhancing the modelling aspect

and fixing some bugs that were reported during the last user study. Since we improved our FBCOP software development approach, we wanted to reassess it. This new user study allowed us to confirm the previous observations of the first study and to better identify where the complexity of our FBCOP approach lies.

This new study was conducted in 2021 with all new participants, aged between 20 and 25 years old. Based on the feedback we collected from the previous study, we updated the new study of 2021 to reduce the workload for our participants. Indeed the user study of 2020 was considered heavy due to the work we asked them. We also adapted the user study to avoid having too many questionnaires as was the case for the previous user study. Finally, we deliberately skipped doing another full user study of our visualisation tools (*i.e.*, we did not conduct other SUS evaluations), but we still asked small questions without going deeply in the assessment of their usability.

As previously, we asked our participants to play the role of FBCOP designers and programmers to conceive a highly dynamic software system using our approach. There were 36 students (of 45) who participated voluntarily to this user study.

This section will follow the same structure used for the previous user studies. We first explain the study itself. We then present the raw results we collected and discuss these results. Finally we conclude this new user study.

User study

At the start of the user study, we gathered the participants' background. We evaluated their expertise level in design and modelling object-oriented applications, and in implementation skills to develop object-oriented applications. We also verified whether they have already heard about the context-oriented programming paradigm.

Regardless of their background, to ensure all our participants had a comparable knowledge about context-oriented programming and FBCOP, we gave them 26 hours of lectures and practical sessions over many weeks. Table 13.3 outlines the different topics we taught and the time we spent on each topic. In this table we emphasise in bold the changes we did with regard to the previous user study in 2020.

Again we asked them to work in pairs for the rest of the user study. To simplify how we gathered their feedback, we provided them only one questionnaire at the end of the user study.

We started the user study by introducing the domain modelling concept in which we present feature modelling, followed by a hands-on session. In this practical session they had to design a feature model of a case study that served as toy example. Then we taught context-oriented programming, FB-

Table 13.3: Time spent to introduce background material to participants in the user study of 2021. We emphasise in bold what are the updates with regard to the time we spent in 2020. (The old values are in parentheses.)

Topic	Theory	Practice
Feature modelling	2	2
Introduction to COP and FBCOP	4 (3)	6
Programming in Ruby	2	2 (4)
Feature-based context-oriented programming	3	5
Total	11 (10)	15 (17)

COP modelling and our supporting methodology. To ensure they understood the key notions of our approach, we took one hour more to explain all these topics. We followed this lecture with a context-feature modelling workshop that was a bit different from the previous one. As a first exercise, we asked them to extend their feature model of the toy example with a context model and a mapping between the models in a two-hour session. This exercise aimed to better understand our context-feature modelling approach. After that we asked them to design a context-feature model of a *smart messaging system* from scratch during two sessions of two hours. This choice to model only one case study allowed them to better think about their design since they had more time. The other difference is that we asked them to use directly our supporting design methodology. Finally we also asked them to think about the application classes (with a conceptual class diagram) and what features should adapt what application classes.

As previously, we followed this modelling workshop by introducing the *Ruby* programming language. Instead of four hours, we only dedicated a two-hour session to practice *Ruby* due to timing constraints. Then we presented an upgraded version of our feature-based context-oriented programming framework (with regards to the previous user study). This upgraded version added more expressivity to the (context and feature) modelling and fixed some errors reported by our past participants. We also presented and showed a demonstration of our two visualisation tools to help them in their implementation phase. These visualisation tools also aimed to reduce the number of times we had to intervene to solve simple bugs as they could visualise with our tools. After giving them some time to understand how our programming framework and visualisation tools work, we presented a new version of our UIA library to help them build their dynamic user interfaces.

At the end of the experiment, we provided our participants a questionnaire to assess our design methodology, programming framework and UIA library. We also asked some questions about our visualisation tools to un-

derstand whether they used them and why they were useful in their cases. In addition to these questions, we also asked our participants to justify their answers in open-ended questions.

Raw results

We now present the raw results collected in this user study. We first show the background of our participants. We then present their answers to the closed questions about our design methodology, programming framework and user interface library.

Participants' background On a five-level Likert scale (ranging from “no expertise” to “expert”), before the user study and before their 27 hours of training, we asked the participants to rate their object-oriented design, modelling and programming skills. As summarised in Figure 13.17, only ~18% of the participants agreed to have some or strong skills in design and modelling. Almost half of the participants affirmed they have some or strong expertise in object-oriented programming. Only ~22% of them claimed that they already had heard about context-oriented programming, but when asked to explain it in a few sentences, only ~11% described correctly what is context-oriented programming.

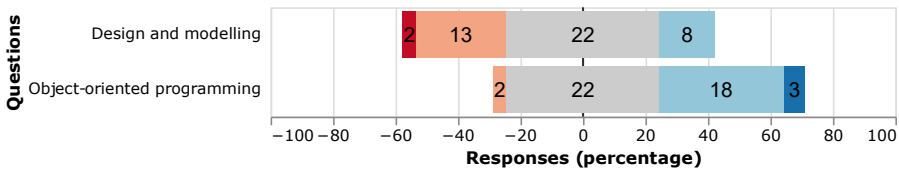


Figure 13.17: Divergent stacked bar presenting the background of our 45 potential participants.

Whereas 45 students participated in this participants' background survey, only 36 students eventually participated in the main user study, working in pairs, which explains why we will only have 18 responses in total for our next questions. Furthermore we also removed the neutral answers for the next questions to force them to take a decision.

Design and development methodology Figure 13.19 presents the gathered feedback of our 18 pairs of participants to the closed questions. We observe a positive tendency for our design methodology. They were ~90% to say that this methodology helped them to better *elicit the contexts and features*. Furthermore, ~78% confirmed it helped them allowed to better *structure the models*. And ~72% found it helped to define the *mapping definition*

from contexts to features. Nevertheless, despite our effort to reduce the rationale they had to describe, half of the pairs found it was not *relevant* to provide *rationale* for each non-trivial context, feature and mapping relation. In the open-ended questions they confirmed this took much time. They also mentioned that creating the lexicon was time-consuming and redundant and should be postponed later in the design phase. Even if we can see that our proposed design methodology helped them in the *requirements* and *design* phases, they suffered to design a conceptual class diagram in which they had to sketch the interactions between the application classes and how they are adapted by which features. They were also mixed about the *complexity* of using our development methodology (including the *implementation* phase) to design FBCOP systems (as opposed to not using such a methodology). As shown in Figure 13.18, ~55% assessed it as easy to use and ~45% did not. Despite this mitigated result, we can observe in the open-ended questions they confirmed our supporting development methodology was interesting since it guided them incrementally towards a solution. Some also claimed that it forced them to think about the requirements before taking actions. However some groups were a bit confused when they had to continue to the *implementation* phase because we did not propose enough guidance for this transition. Similarly, one group thought their efforts during the *design* phase were not rewarded for the *implementation* phase because their features were not well-modelled (*i.e.*, they got bad interaction between the features and had useless features in real-world usages).

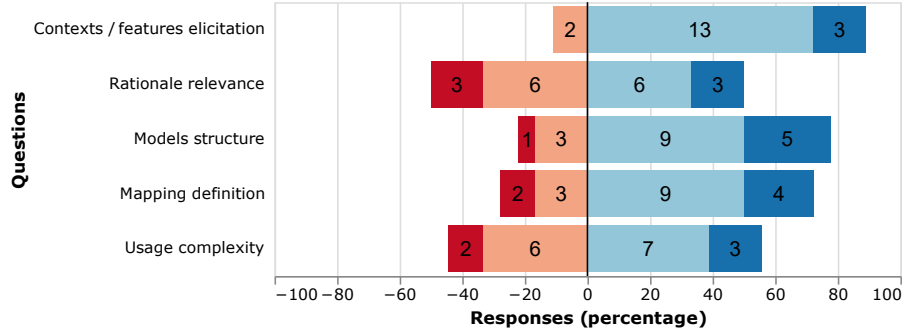


Figure 13.18: Divergent stacked bar showing the results of our closed questions regarding the design and development methodology.

Programming framework Figure 13.19 depicts the results of the closed questions about the programming framework.

As we can observe, ~83% agreed that our programming framework has a good expressiveness to declare a *context and feature model*; ~77% found it

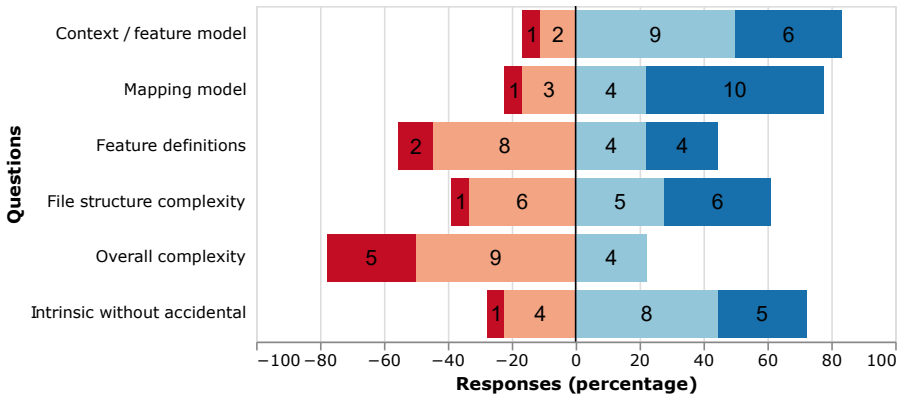


Figure 13.19: Divergent stacked bar showing the results of our closed questions regarding the programming framework used during the *implementation* phase.

was also sufficiently expressive to declare the *mapping model*. They found the syntax was *clear*, *short* and *easy to write*. Nevertheless two groups mentioned that such a code could quickly become very long.

They had a more mixed feeling regarding the implementation of the *feature definitions*. While $\sim 44\%$ considered that the implementation of feature definitions was easy, $\sim 56\%$ did not agree with this statement. This could be explained by the fact that implementing feature definitions was complex for large applications. It took time to understand how they had to code the feature definitions. Some groups precised that it was not straightforward to understand the *proceed* mechanism and the variable scoping with the interactions between the features and the application classes.

A positive trend was observed for the *file structure complexity*. Indeed, $\sim 60\%$ found the file structure of a FBCOP application was *easy to understand* and suggested a *good* and *coherent* structure. The others considered that it had too many files and they could get lost in all the files. Furthermore three groups did not find *intuitive* the name of the folder (“*Skeleton*”), that is supposed to contain all the application classes of the application.

Even if some points are encouraging, we can see a negative trend ($\sim 80\%$) about the *overall complexity* after they implemented a first FBCOP application. The main issues they raised were about the error messages, the *Ruby* programming language, the debugging, the documentation and the learning curve. They considered the error messages were not explicit enough for the (de)activation of entities in the different models. Six groups found that the *Ruby* programming language on which we have built our programming framework was complex. Furthermore a few groups mentioned that the debugging could become really complex due to the many interactions they could

have between the features and application classes. One group precised that such an approach required much effort when they had to debug a feature because they also had to develop its contexts, mapping and all the links between these entities. Some also considered we did not provide enough documentation and some more elaborate examples were missing to better illustrate how they could develop some concerns. They also precised that the learning curve was *very steep*. However some positive comments were also received about our programming framework. For example, one group said that our approach “*makes FBCOP possible*”. Two groups precised that our approach allowed to have good practices in their implementation with a well-structured code and a natural code cut across several files. Two groups found that *Ruby* was not a barrier at all to implement such systems.

Despite the overall complexity of our approach for implementing FBCOP applications, ~70% considered the complexity was intrinsic for developing such systems. Some groups precised our programming framework was complex but necessary. Other groups specified that the complexity to write such systems was reduced thanks to our programming framework, especially with the abstractions to (de)activate the contexts and features, to verify the model consistency, but also with the *proceed* mechanism. In the same spirit, one group said “*If we had to start from scratch, it would be very complex to connect all the modules together*”. However one group did not agree with this at all and found that letting everything in the hands of the developer could cause accidental complexity. For example, when they had to name a feature in its declaration, they had to name its definition with the same name since without a perfect match between both, an error was raised.

User interface adaptation For our integrated UIA library to help them build and compose user interfaces, the results were clearly mixed. When we asked our participants whether the library was sufficiently usable and easy to use, only two pairs confirmed this statement. They explained this complexity by a lack of documentation and that they lose much time to understand how to use it. From an implementation perspective they found that each method of the library took too many arguments. However some more positive points were also highlighted. Some of them found it was useful and usable for simple user interfaces. They liked the fact that our library was itself based on a well-documented GUI library. Furthermore some groups claimed that the *proceed* mechanism was really interesting to compose and build the user interfaces: “*The proceed mechanism is a neat way to enable modifications of the user interface based on contexts.*”. Finally 50% of the pairs considered that the proposed application programming interface was complete while the other 50% did not have the same feeling, as illustrated in Figure 13.20.

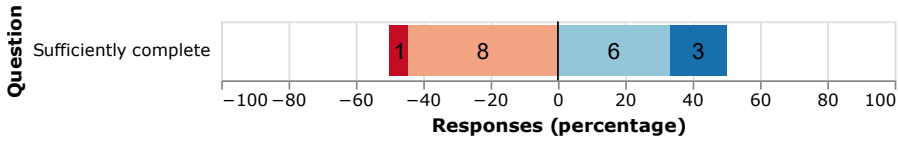


Figure 13.20: Divergent stacked bar showing the result of our closed question regarding our UIA library dedicated for building user interfaces.

Visualisation tools As opposed to the first full user study, we did not assess our visualisation tools in detail since we already assessed them deeply and due to timing constraints. But as we introduced them to our participants in order to help them in their development and debugging tasks, we asked to explain in which situations they used these tools. Eleven pairs exploited our CONTEXT AND FEATURE MODEL VISUALISER tool and only three groups used the FEATURE VISUALISER tool. For those using the CONTEXT AND FEATURE MODEL VISUALISER, they used it to visualise the interactions between the different models and if they declared correctly their models. They also took benefit of this visualisation to try to understand when an (de)activation was ignored (due to an invalid configuration). Even if it is not the main goal of this tool, two groups precised they used it to see the activation order of the contexts and features. The usage of the FEATURE VISUALISER tool was more used to inspect if the (de)activation has been done and to see the connection between the contexts and features. One group claimed that this tool proposes “*a clear view of which application classes are adapted*”. Unfortunately some information was missing in the FEATURE VISUALISER tool for a group but they did not explain which kind of information was missing. One group regretted that no tutorial was provided for the FEATURE VISUALISER tool, which could also explain why only three groups used that tool.

Discussion

We will now discuss all results we collected via our questionnaire.

Development methodology As for the previous user study, our development methodology seemed to interest our FBCOP designers and programmers when they had to conceive their application. It allowed them to save time if we analyse their answers to the open-end questions. It also guided them incrementally through the development process, from the *requirements* to the *implementation*, via the *design* of their application. This could be probably explained by the clear workflow we attempted to propose them to create FBCOP systems.

Even if we simplified the description of rationale in the design method-

ology for this last user study, half of our pairs were still doubtful about its relevance. In addition they considered this step as time-consuming. This mitigated result could perhaps be explained by the simplicity and limited size of the case study (a *smart messaging system*). Therefore we can understand this task could take much time for a small project. However we think this step would remain useful for complex and larger FBCOP applications and for which many stakeholders (e.g., designers and programmers) have to interact together during a long time (due to the complexity of the project), which was not the case for our participants since they only worked in pairs. Such rationale should aim to ensure that all non-trivial contexts, features and mapping relations could be understood by any stakeholder at any moment of the project once they were defined. But such a claim remains to be tested with a large and complex application with many stakeholders.

Several pairs also suffered to design a conceptual class diagram of their application and to define what features had to adapt what application classes. This could be explained by the fact that we did not instruct and guide them for this step in our design methodology. Even if only a few confirmed they were comfortable in designing and modelling object-oriented applications, we assumed they had enough skills to design small and simple object-oriented applications since this was part of their cursus at the university. In addition to the timing constraints, we did not really explore this part in the design methodology since we thought that object-oriented modelling was sufficient to do this step. Since it was not the case, we will have to extend this part in our design methodology in the future. For that we could imagine we explain that this part of the design relies on object-oriented modelling and provide an example of how an application can be modelled with object-oriented modelling and with FBCOP. Thus we would emphasise that the modelling mindset is the same but the state and behaviour of each application class should be separated into features to ensure a good separation of concerns. As soon as an application class is split into features, these features should adapt these application classes since they contain a fragment of the original application class.

Finally, the complexity to use our development methodology could also come from the intrinsic complexity to create such applications as already explained previously. So based on this discussion, we believe that we can state that our development methodology, in spite of its complexity and of the improvements we could make, was still interesting for conceiving FBCOP applications.

Programming framework Similar to the previous full user study, our participants found the expressiveness of how they had to declare the different models (context, feature and mapping model) was good. However two pairs found that such a declaration source code become rapidly very long. This

could lead to reduce the readability of such a code. To facilitate its readability, we would have to more insist to split the declaration of each model in methods.

Again the implementation difficulty resided in the implementation of the feature definitions. One possible explanation for this is the source code had to be split into several features. This complexifies probably the global overview of how this application is coded for new programmers. In addition, even if we explained the *proceed* mechanism in the lecture and in hands-on sessions, some pairs found this concept complex. For future user studies, we could probably illustrate this construct with more examples and show it with the help of our visualisation tools, and in particular with the FEATURE VISUALISER tool. Some groups also raised a misunderstanding of how the variables are used through the different features and application classes. For that we should be probably better describe how the features adapt the application classes and explain that the variables are finally simply used like object-oriented applications, as for example the usage of an instance variable in an application class can be used normally through its different features that adapt it.

For the file structure we got an interesting positive tendency even if we gathered the same feedback about the many files we could have in the folder containing the feature definitions for larger applications. However, during the user study, one pair asked us if they could reorganise this feature definitions folder into sub-folders to get a better readability in this folder and enhance its code. We answered that they could make such modifications if they considered it was an improvement for them.

Despite this positive tendency, they agreed to say that the overall complexity of our approach was not simple. As for the previous full user study, they explained that this complexity could come from the steep learning curve, missing documentation, lack of concrete examples, error messages that could be more explicit and the *Ruby* programming language. About *Ruby*, one pair said: *"I do not know if it helps tackling the complexity of building such an application, but it being in a language we did not get used to before hand definitely did not help"*. This could explain why some groups who were not comfortable with *Ruby* had more difficulties to implement FBCOP applications with our programming framework. In spite of our effort to give lectures and practical sessions, we can thus infer our approach is *intrinsically* complex. Nevertheless, in spite of the complexity of our programming framework they agreed to say that our approach and programming framework could help them to build FBCOP applications. Finally we received an interesting comment from a pair of participants who thought that our FBCOP approach could be interesting for implementing other kinds of applications, such as *Internet of Things* (IoT) applications: *"seeing that this framework is trying to emulate what object-oriented programming did (eliminating 'if' conditions to check for a 'type' change) and is*

trying to eliminate conditions checking every time we change contexts, it opened my mind to the possibilities of the language for IoT applications”.

User interface adaptation The mitigated results could be explained by the fact that our version of the UIA library was still a prototype. In fact, as they experienced it was not yet usable for building complex user interfaces. We also did not provide enough concrete examples on its usage, and an elaborated documentation of our application programming interface was missing. Nevertheless, even if we did not provide a full documentation of our library, it relies on a well-documented GUI library to build user interface widgets. And some pairs considered this as a strength of our design choice. Another reason could be we provided this UIA library at the end of the user study and they probably did not have enough time to better explore it to build more interesting user interfaces. But adding this complexity at the beginning could be a real barrier to learn correctly the programming approach since it was already complex for our participants. In general, when we learn a new programming approach, we have the tendency to first study the basics of it. And once we have some expertise, we deeply study more aspects as for example how we can build the user interfaces.

About the incompleteness of our API, they were right in the sense that more behaviour should be added to have a more interesting API for building user interfaces. Examples of such missing methods were: a method adding a user interface widget on top of all the other widgets, and methods to move a user interface to another place.

Despite some of this negative feedback, we can still observe that our first prototype for this UIA library can be promising since it allowed them to create simple user interfaces. Furthermore we can also see that our way to build and compose the user interfaces fits well in the context-oriented programming paradigm, and specifically with the *proceed* mechanism to build user interfaces.

Supporting tools By analysing our visualisation tools, we can observe that our CONTEXT AND FEATURE MODEL VISUALISER tool was perceived more useful than our FEATURE VISUALISER tool. This could be explained by the fact that the former has a similar visualisation to how we present our system architecture and to how they had to design their application. Surprisingly, a few pairs mentioned that our CONTEXT AND FEATURE MODEL VISUALISER tool was also useful to visualise the activation order of the different entities. In fact, even if it was not the main goal of this visualisation, the activation order of the different entities can be inspected step-by-step. However, once they are activated, this information is lost as opposed to the FEATURE VISUALISER tool. In addition one pair also said that some documentation was missing for the FEA-

TURE VISUALISER tool. Despite our introduction and demonstration of both visualisation tools, we will have to put more effort to explain the interest of our FEATURE VISUALISER tool with examples in which it could be interesting since it seemed less useful according to our participants.

In addition to our proposed visualisation tools, they also mentioned other tools that they would have liked during the experience. A first one is a tool helping to generate the source code of the model declarations to save time and avoid easy bugs. We could imagine a kind of tool to generate the different context, feature and mapping model from a visual representation of them to save time. Going further, we could also create the skeleton of the feature definition for each feature declaration, as well as an empty structure of each application class used in the different features. They would have also liked a script that allowed to launch the server and different tools simultaneously to avoid to launch separately the server and then the supporting tools.

Conclusion

For this user study, we focussed on assessing some aspects we evaluated less with real designers and programmers in the previous user studies. There were 36 participants grouped in pairs to answer the questionnaire.

We again followed the *demonstration* and *usage* evaluation strategies as proposed in this user study [Led+18].

Despite their limited knowledge of designing and modelling and their good skills in object-oriented programming, we can observe that our full approach remains relatively complex for new designers and programmers when they had to conceive FBCOP applications. Despite its intrinsic complexity they appreciated our design methodology. For our programming framework and user interface adaptation, we got mixed results due to the learning curve to understand how they had to implement such applications. Finally they confirmed our visualisation tools were useful for their development and debugging tasks, and preferred the CONTEXT AND FEATURE MODEL VISUALISER in particular.

Part V

Epilogue

The previous part presented the validation of our approach. We first assessed FBCOP's expressiveness through five case studies we or other persons have designed and implemented. From this first validation we can say that FBCOP's expressiveness is enough to design and implement context-oriented applications, even if some points could be improved. Then we evaluated FBCOP's design to demonstrate that our implementation of the programming framework is maintainable, extensible, adaptable and readable, but not yet scalable. We also assessed the usability of our programming framework based on the cognitive dimensions of notations framework. Finally we carried out four user studies with real users, master-level students who were asked to play the role of designers and programmers using FBCOP to conceive context-oriented applications. Through the different user studies, we noticed that one of the main problems is the steep learning curve causing a more complex understanding on how they had to implement such applications with FBCOP. However, to overcome this problem partly, our visualisation tools were nicely appreciated in their development and debugging tasks.

The fifth and last part of this dissertation is the epilogue. We provide first some potential future work (Chapter 14). Finally we conclude this dissertation with a global overview of our contributions and a small discussion on the industrial viability of our FBCOP approach (Chapter 15).

CHAPTER

14

FUTURE WORK

In this chapter we provide a potential list of future work we may carry out to improve and complete the FBCOP approach. We argue how we can enhance FBCOP's expressiveness, how we can add a sensory layer in order to create real applications, and how we can improve visualisation tools to provide a better help to our programmers. We discuss how performance and scalability may be potentially improved for some parts of our implementation. Finally we will briefly cite some other concerns we did not tackle or simply mentioned upon passing in this dissertation since they are not the main part of this work. The future work is not exclusive to these potential areas, however.

14.1 Improving FBCOP's expressiveness

We saw that FBCOP's expressiveness is sufficient to design and implement FBCOP applications. However it has some limitations and future work that we discuss in this section.

Context data Our contexts do not have information. The only information we have is that contexts are active or inactive. This design choice implies that application programmers must discretise the contexts in their context model. Such an example is illustrated with the context (user) *Age* and its child contexts *Child*, *Adult* and *Senior* (see the context-feature model of the *smart messaging system* depicted in Figure 11.5). However this design is

still limited to model some contexts, such as the localisation of the user. Indeed, to model the position of the user, we must model each possible location. This could be very restrictive. Discretisation of many different situations also increases the number of contexts. So, by extending contexts with data, application programmers could model some contexts easier and naturally reduce the number of contexts. For example, they could model the exact position with a single context *Localisation* by adding the latitude and longitude as attributes so that the user language could be adapted according to the position, for example. Therefore, adding data in contexts could improve the expressiveness to model the contexts. Nevertheless such future work also means that the mapping model must be extended to perform some computations to treat the different data or data ranges. Also, mechanisms should be devised so that features could be access the data stored in the contexts that trigger them. Finally, this raises many issues regarding persistence, accessibility, and synchronisation of that data.

Propositional logic for the mapping We voluntarily implemented a simple mapping model from contexts to features (see Section 4.6). However this mapping has some limitations in its expressiveness. For example, when the activation or deactivation of two different sets of contexts triggers the activation or deactivation of the same set of features, application programmers must define a mapping relation for each set of context. This increases the mapping model and programmers can lose in readability.

A potential improvement is to extend the existing mapping model's expressiveness by using propositional logic [Ach+09]. As we have already implemented the *and* operator, by adding the operators *or* and *not*, we could enhance its expressivity significantly. Indeed, application programmers could then merge two mapping relations that (de)activate the same set of features. By writing a new mapping relation with an *or* operator between the contexts to express that at least one of both sets of contexts must be activated to trigger these features. Furthermore, the *not* operator would also allow to express mapping relations dedicated to a deactivation.

Keywords in the programming framework We could add additional keywords in the programming framework to ease the readability of FBCOP applications when implementing. For example, when programmers develop the features definitions (*i.e.*, the features and feature parts), currently they must use the keyword *module* for both.

To improve the readability, we could add the keywords *feature* and *part* in the programming framework, as illustrated in Listing 14.1.

```

1 feature ListChats
2   part Model
3     # More code
4   end
5   part View
6     # More code
7   end
8 end

```

Listing 14.1: Code snippet of the structure of the feature *ListChats* of the *smart messaging system* with new keywords in the language. (The original code is depicted in Listing 11.8.)

Other dependencies in the context and feature model In addition to the constraints and dependencies we have, we could also add additional dependencies as proposed by Cardozo et al. [Car+15] such as the causality or the implication.

Usage of the UIA library We should improve the API of the UIA library we proposed by adding methods, such as methods to move a UI object to another place or in relation to another one, to replace a UI object by another one and much more.

We could also improve the domain-specific language of our UIA library to facilitate the implementation of user interfaces in the different feature definitions.

14.2 Adding sensory layer and context definitions

In our previous work, Mens et al. [MCD16] propose a more complete approach in which the sensory layer was integrated. As the work of this dissertation is strongly inspired by this context-oriented software architecture, the next step is to include the sensory layer. With this layer, application programmers could develop real applications that are able to adapt their behaviour depending the contexts that are (de)activated according to sensed information from the real world.

To implement the sensory layer, we could implement a new tooling support that catches all messages sent to the server from different sensors. Then the application programmers should have to implement context definitions that will reify the contextual raw data into context objects to finally send them to the CONTEXT ACTIVATION component in order to propagate the changes into the system behaviour. Nonetheless the context definitions should also

contain some filters to prevent information flooding that could lead to many useless computations or adaptations.

14.3 Improving and adding tools

We should enhance first our existing visualisation tools to include error messages in the visualisation tools to better inform programmers when an error is raised in the application or programming framework.

We could also add interactions on the context model displayed in the `CONTEXT AND FEATURE MODEL VISUALISER` in order to (de)activate contexts through the visualisation tool. For this tool, we could also envisage to implement the ‘back’ button that is missing for now.

Furthermore other tools can also be developed, such as for example a debugging tool to better help application programmers to find errors when executing their FBCOP application.

Other kind of tools can also be conceived to enrich our current toolbox, such as tools dedicated to test FBCOP modelling or implementation or simulators to run FBCOP application in particular situations. Testing tools also include static analysis tools that allows to detect anomalies [Mau21] in a context-feature model. In the perspective to enrich the toolbox, Martou et al. [Mar+21] already proposed a generation tool to create relevant test scenarios of FBCOP applications. This tool generates test scenarios from the context-feature model that allow to test the design of the context-feature model before implementing it. Duhoux [Duh16] also proposed a visual context simulator that allow to test FBCOP applications in a controlled surrounding environment.

14.4 Performance and scalability issues

For this dissertation, we mainly focussed our effort on the usefulness, usability and understanding of the FBCOP approach to designers and programmers. We did not address performance or scalability issues per se. These aspects are thus another area for improvement. In this section, we discuss these issues.

Performance issue An improvement could be made to the satisfiability algorithm (see Section 8.2). Indeed, optimisations could be integrated or other algorithms could be used to be more efficient than our depth-first search algorithm. Other algorithms, such as SAT(satisfiability) solvers [Bat05] or constraint programming techniques [BTR05], could be used to replace our satisfiability strategy. Without benchmarking the different suggestions on different types of models, we cannot infer which one is the most efficient. This will depend

on the kind of models, complexity of the context and feature models (*i.e.*, the number of contexts, features, and mapping relations) and the contexts and features that are/must be activated or deactivated.

Scalability issue An example of a scalability issue is our dynamic adaptation mechanism. Indeed, we can easily observe an overhead with our *un-binding and binding methods* mechanism when *proceed* calls are executed (see Section 9.3). For each *proceed* call, we must undeploy the last adaptation of the method, then deploy its previous adaptation, uninstall it after execution, and finally redeploy the most recent adaptation in the system. Therefore this dynamic adaptation mechanism has a cost when it must execute a *proceed* call.

To solve this, we could imagine a mechanism to deploy all adaptations of a method in the application class and then use pointers to access directly the right adaptation. During a master thesis [Mar21a], a master-level student explored this solution and showed that the overhead drastically decreased the execution time of the *proceed* calls.

14.5 Other concerns

Other concerns were not addressed or only mentioned during this dissertation in the FBCOP approach such as the data layer, testing, context-interaction problem, multimodality and feedback for end-users.

Data layer To create applications for real users, designers and programmers would like to design and implement such applications with a core logic, user interfaces and a data persistence layer to save the data. Nonetheless we only focussed our effort on the concerns that address the core logic and user interfaces. However the data concern may also be contextualised in order to adapt the data manipulation and querying. The research question to be investigated could be how we can integrate the data layer in the FBCOP approach.

Testing We have already mentioned Martou et al.'s work [Mar+21] in which we generate relevant test scenarios to find design errors or inconsistencies in context-feature models. This work tackles partially the modelling field. However the implementation field was not addressed in detail yet. How can we automate testing of FBCOP applications to ensure their behaviour is the expected one in a particular situation? Is unit/integration testing sufficient? Should we explore other approaches like mutation testing or concolic testing?

Context- and feature-interaction problem Adapting software systems can raise some issues at runtime, such as for example the context-interaction problem [MDC17]. This problem comes from unforeseen interactions between contexts and features, that leads to some conflicts when two incompatible features must adapt the behaviour. To illustrate such a problem, consider a home automation system able to dynamically detect fires and water leaks and adapt its behaviour to halt the emergency. In the case of a fire, the system must trigger the sprinklers and send an alarm to the fire brigade. In the case of a water leak, it must turn off the main water supply. When only one event occurs, the system is aware of which action it must undertake. But how should the system react when these two events occur simultaneously? Should it turn off the main water supply and let the house burn down? Or should it extinguish the fire but flood the house? In order to help designers and programmers think about such issues, we already proposed a classification and design space of conflict resolution techniques [MDC17], but we did not implement yet them in the FBCOP approach.

Multimodality We explained that two master-level students explored the integration of multimodality in a FBCOP application (see Section 11.4). This is a first proof of concept that the development of FBCOP applications including multimodality is feasible. Going further, we could analyse other types of applications where multimodality can be of interest (either as inputs or outputs) so that it can ease accessibility of certain functionalities to people with disabilities.

Feedback for users Systems that dynamically adapt their behaviour to the environment without informing or asking for confirmation from end-users can confuse them. Although we need to assess this statement, it seems to make sense because it means that end-users lose control over their systems.

Therefore, another avenue to explore is how to help end-users better understand and accept that their applications adapt their behaviour at runtime.

CHAPTER

15

CONCLUSION

In this dissertation, we proposed a feature-based context-oriented software development (FBCOP) approach to conceive context-oriented systems, that unifies context-oriented programming [HCN08], feature modelling [Kan+90] and dynamic software product lines [HT08; Ach+09; COH14; Men+17] into a single software development approach. This approach consists of a modelling technique, an architecture, a programming framework, a user interface adaptation (UIA) library, a supporting development methodology and visualisation tools to help designers and programmers when designing and implementing systems that must adapt their behaviour depending on the contexts in which they run.

The contributions are the following:

- We proposed a clear and explicit separation of contexts and features since they are different by nature. While contexts represent particular situations in which the system runs, features compose the functionalities of the system behaviour. This separation strengthens the maintainability of our approach when designing and implementing context-oriented applications, as well as the reusability through different projects that fit some part of the requirements.
- Despite this separation, a common modelling notation allows to design contexts and features in a similar way. With feature modelling we designed contexts and features to have a context and feature model. This

single notation eases the modelling of contexts and features for designers.

- We also created an architecture that abstracts the full mechanism to implement context-oriented systems, while keeping this clear separation between contexts and features. This lets programmers develop their FBCOP applications without having to consider the complexity to treat the context (de)activation, feature (un)selection, feature (de)activation and dynamic adaptation to (un)deploy the features in order to adapt or refine the behaviour. Again, this increases the maintainability since the intrinsic complexity of such a programming approach is separated from the application code. In addition, it promotes the modularity by explicitly separating the different concerns (core logic and user interface concern). Furthermore programmers can also create a set of features that can be reused inside a same project or within other projects, and so increase the reusability of their applications.
- We also implemented a FBCOP programming framework that provides building blocks and language constructs to help programmers in their implementation. Even if this programming framework is developed on top of the *Ruby* programming language, *Ruby* only served as a proof of concept since this architecture and programming framework could be implemented in any object-oriented programming language that provides a sufficiently powerful reflective API.
- We also provided a UIA library to help programmers to create the user interfaces in the FBCOP approach. Since the user interfaces must be implemented as fine-grained features just like the core logic components, they are adaptive by definition.
- We also provided a supporting development methodology and two visualisation tools, the `CONTEXT AND FEATURE MODEL VISUALISER` and the `FEATURE VISUALISER`, in the perspective to help designers and programmers for their different tasks. The development methodology suggested an incremental methodology to guide the stakeholders in the conception of systems with our approach. The two visualisation tools aimed to help programmers in their development and debugging tasks.

In addition to these contributions to design, create and implement a FBCOP approach to help designers and programmers conceive FBCOP applications that have a high dynamic nature, we also validated our approach as follows.

- We assessed FBCOP's expressiveness with the help of five case studies: two variants of a *smart messaging system*, a *smart risk information*

system, a *smart meetings system* and a *smart city guide*. Through this validation, we observed that our approach is sufficiently expressive to conceive such systems, from a modelling or implementation perspective. An interesting point was that other persons used our approach to design and/or implement their context-oriented systems. This proved its usefulness and usability.

- We also evaluated FBCOP's design by discussing its design qualities. We pointed out its maintainability, extensibility, adaptability and readability. However, much work still needs to be done to have a scalable programming framework.

We have also evaluated its usability by arguing our own point of view on its usability based on a cognitive dimensions of notations framework.

- We also conducted four user studies to evaluate the usability, usefulness and understanding of the FBCOP approach. For each user study, we carried out a study with a group of 25-40 master-level students whom we asked to play the role of designers and/or programmers of context-oriented systems. Through these user studies, we observed that the FBCOP approach seemed interesting to them to design and implement such dynamic applications, and the visualisation tools seemed really useful and usable. However the programming framework was more complex to use due to the steep learning curve. This result seems coherent since we cannot expect someone to become an expert in a new programming paradigm in only 25-30 hours of lectures and practical sessions.

To conclude this dissertation, we will discuss whether our approach is viable in the industry. At this stage, this approach still needs much work before industrial adoption because specific concerns or aspects are not yet addressed in detail, such as the sensory layer, the data layer or the testing, for example. But the results are promising and encouraging, which should be a motivation to continue to develop it. However, may the intrinsic complexity underlying our approach be an obstacle for industry? We do not believe that the intrinsic complexity of FBCOP can be an obstacle. With the advent of the Internet of Things and (self-)adaptive systems, industry will probably encounter the same problems that we have encountered and for which we have provided answers. It might therefore be more interesting to spend time learning a new existing approach than to fully investigate a new solution, as the latter option would probably take longer. Although the learning curve will be steep, it is not impossible because for every new challenge for which the industry is unfamiliar, a new technology must be learned. So whatever the problem, e.g.

the creation of new web applications, dynamic applications or other kinds of applications, the learning curve will generally always be difficult and takes time.

BIBLIOGRAPHY

- [AB11] S. Apel and D. Beyer. “Feature Cohesion in Software Product Lines: An Exploratory Study”. In: *Proceedings of 33rd International Conference on Software Engineering*. ICSE ’11. ACM, 2011, pp. 421–430. ISBN: 978-1-4503-0445-0.
- [Abo+99] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. “Towards a Better Understanding of Context and Context-Awareness”. In: *Handheld and Ubiquitous Computing*. Springer, 1999, pp. 304–307. ISBN: 978-3-540-48157-7.
- [Abr+21] S. Abrahão, E. Insfran, A. Sluÿters, and J. Vanderdonckt. “Model-based intelligent user interface adaptation: challenges and future directions”. In: *Software and Systems Modeling* 20.5 (2021), pp. 1335–1349.
- [Ach+09] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, and J.-P. Rigault. “Modeling Context and Dynamic Adaptations with Feature Models”. In: *4th International Workshop Modelsrun.time at Models 2009 (MRT’09)*. Oct. 2009, p. 10.
- [AHR08] M. Appeltauer, R. Hirschfeld, and T. Rho. “Dedicated Programming Support for Context-Aware Ubiquitous Applications”. In: *The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE, 2008, pp. 38–43.

- [AK09] S. Apel and C. Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8 (July 2009), pp. 49–84.
- [AKM11] T. Aotani, T. Kamina, and H. Masuhara. “Featherweight EventCJ: A Core Calculus for a Context-oriented Language with Event-based Per-instance Layer Transition”. In: *Proceedings of the 3rd International Workshop on Context-Oriented Programming*. COP ’11. ACM, 2011, 1:1–1:7. ISBN: 978-1-4503-0891-5.
- [App+11] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. “ContextJ: Context-oriented Programming with Java”. In: *Journal of Information Processing* 6 (2011), pp. 399–419.
- [Bal+04] L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary. “CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces”. In: *Ambient Intelligence*. Springer, 2004, pp. 291–302. ISBN: 978-3-540-30473-9.
- [Bat05] D. Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *Software Product Lines*. Springer, 2005, pp. 7–20. ISBN: 978-3-540-32064-7.
- [BDR07] M. Baldauf, S. Dustdar, and F. Rosenberg. “A Survey on Context-Aware Systems”. In: *International Journal of Ad Hoc and Ubiquitous Computing* 2.4 (June 2007), pp. 263–277. ISSN: 1743-8225.
- [Ben+08a] N. Bencomo, G. S. Blair, C. A. Flores-Cortés, and P. Sawyer. “Reflective Component-based Technologies to Support Dynamic Variability.” In: *VaMoS*. 2008, pp. 141–150.
- [Ben+08b] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. “Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems.” In: *DSPL ’2008*. Jan. 2008, pp. 23–32.
- [Bet+10] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. “A survey of context modelling and reasoning techniques”. In: *Pervasive and Mobile Computing* 6.2 (2010), pp. 161–180. ISSN: 1574-1192.
- [BKM09] A. Bangor, P. Kortum, and J. Miller. “Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale”. In: *Journal of Usability Studies* 4.3 (May 2009), pp. 114–123. ISSN: 1931-3357.

- [Bla+01] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. “Cognitive Dimensions of Notations: Design Tools for Cognitive Technology”. In: *Cognitive Technology: Instruments of Mind*. Springer, 2001, pp. 325–341. ISBN: 978-3-540-44617-0.
- [BLH10] N. Bencomo, J. Lee, and S. Hallsteinsen. “How dynamic is your dynamic software product line?” In: *Proceedings of the 14th International Software Product Line Conference*. SPLC ’10. Lancaster University, 2010, pp. 61–67.
- [Blo+11] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jézéquel. “Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation”. In: *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS ’11. ACM, 2011, pp. 85–94. ISBN: 9781450306706.
- [BNK14] J. S. Bauer, M. W. Newman, and J. A. Kientz. “What Designers Talk About When They Talk About Context”. In: *Human-Computer Interaction* 29.5-6 (2014), pp. 420–450.
- [Bob+15] S. Bobek, S. Dziadzio, P. Jaciów, M. Slazynski, and G. J. Nalepa. “Understanding Context with ContextViewer - Tool for Visualization and Initial Preprocessing of Mobile Sensors Data”. In: *Modeling and Using Context - Proceedings of 9th International and Interdisciplinary Conference*. Vol. 9405. Lecture Notes in Computer Science. Springer, 2015, pp. 77–90.
- [Bol80] R. 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*. SIGGRAPH ’80. ACM, 1980, pp. 262–270. ISBN: 0897910214.
- [Bou+17] S. Bouzit, G. Calvary, J. Coutaz, D. Chêne, E. Petit, and J. Vanderdonckt. “Design space exploration of adaptive user interfaces”. In: *11th Int. Conference on Research Challenges in Information Science RCIS*. IEEE, 2017.
- [BPH12] Q. Boucher, G. Perrouin, and P. Heymans. “Deriving Configuration Interfaces from Feature Models: A Vision Paper”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS ’12. ACM, 2012, pp. 37–44. ISBN: 9781450310581.

- [BQ15] L. Baresi and C. Quinton. “Dynamically Evolving the Structural Variability of Dynamic Software Product Lines”. In: *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’15. IEEE, 2015, pp. 57–63.
- [Bro96] J. Brooke. “SUS: a quick and dirty usability”. In: *Usability evaluation in industry* 189 (1996).
- [BTR05] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. “Automated Reasoning on Feature Models”. In: *Advanced Information Systems Engineering*. Springer, 2005, pp. 491–503. ISBN: 978-3-540-32127-9.
- [Cal+03a] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paternò, C. Santoro, N. Souchon, D. Thevenin, and J. Vanderdonckt. “The CAMELEON reference framework, deliverable 1.1, version V1. 1, CAMELEON project (2002)”. In: *Proceedings of XML Europe*. 2003.
- [Cal+03b] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. “A Unifying Reference Framework for multi-target user interfaces”. In: *Interacting with Computers* 15.3 (June 2003), pp. 289–308. ISSN: 0953-5438.
- [Cal+05] G. Calvary, J. Coutaz, O. Dâassi, L. Balme, and A. Demeure. “Towards a New Generation of Widgets for Supporting Software Plasticity: The “Comet””. In: *Engineering Human Computer Interaction and Interactive Systems*. Springer, 2005, pp. 306–324. ISBN: 978-3-540-31961-0.
- [Cap+14] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. “An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry”. In: *Journal of Systems and Software* 91 (2014), pp. 3–23. ISSN: 0164-1212.
- [Car+11] N. Cardozo, S. Günther, T. D’Hondt, and K. Mens. “Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations”. In: *Proceedings of the International Conference on Software Engineering Advances*. ICSEA ’11. IARIA, 2011, pp. 130–135.
- [Car+14] N. Cardozo, K. Mens, S. González, P.-Y. Orban, and W. De Meuter. “Features on Demand”. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’14. ACM, 2014. ISBN: 9781450325561.

- [Car+15] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, J. Vallejos, and T. D'Hondt. "Semantics for consistent activation in context-oriented systems". In: *Information and Software Technology* 58 (2015), pp. 71–94. ISSN: 0950-5849.
- [Car13] N. Cardozo. "Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems". PhD thesis. 2013, p. 313.
- [CCT01a] G. Calvary, J. Coutaz, and D. Thevenin. "A Unifying Reference Framework for the Development of Plastic User Interfaces". In: *Engineering for Human-Computer Interaction. ECHI '01*. Springer, 2001, pp. 173–192. ISBN: 978-3-540-45348-2.
- [CCT01b] G. Calvary, J. Coutaz, and D. Thevenin. "Supporting Context Changes for Plastic User Interfaces: A Process and a Mechanism". In: *People and Computers XV—Interaction without Frontiers*. Springer, 2001, pp. 349–363. ISBN: 978-1-4471-0353-0.
- [CD08] P. Costanza and T. D'Hondt. "Feature Descriptions for Context-oriented Programming". In: *Proceedings of 12th International Software Product Lines Conference. Second Volume (Workshops)*. SPLC '08. Lero Int. Science Centre, 2008, pp. 9–14.
- [CDD19] A. Clarinval, B. Duhoux, and B. Dumas. "Supporting Citizen Participation with Adaptive Public Displays: A Process Model Proposal". In: *Proceedings of the 31st Conference on l'Interaction Homme-Machine: Adjunct. IHM '19*. ACM, 2019. ISBN: 978-1-4503-7027-1.
- [CGM09] A. Cádiz, S. González, and K. Mens. "Orchestrating Context-aware Systems: A Design Perspective". In: *Proceedings of the First International Workshop on Context-aware Software Technology and Applications*. CASTA '09. ACM, 2009, pp. 5–8. ISBN: 978-1-60558-707-3.
- [CH05] P. Costanza and R. Hirschfeld. "Language Constructs for Context-Oriented Programming: An Overview of ContextL". In: *Proceedings of the 2005 Symposium on Dynamic Languages*. DLS '05. ACM, 2005, pp. 1–10. ISBN: 9781450378161.
- [CHD15] R. Capilla, M. Hinchey, and F. J. Díaz. "Collaborative Context Features for Critical Systems". In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '15. ACM, 2015, pp. 43–50. ISBN: 9781450332736.

- [CLC05a] T. Clerckx, K. Luyten, and K. Coninx. “DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development”. In: *Engineering Human Computer Interaction and Interactive Systems*. Springer, 2005, pp. 77–95. ISBN: 978-3-540-31961-0.
- [CLC05b] T. Clerckx, K. Luyten, and K. Coninx. “Generating Context-Sensitive Multiple Device Interfaces from Design”. In: *Computer-Aided Design of User Interfaces IV*. Springer, 2005, pp. 283–296. ISBN: 978-1-4020-3304-9.
- [Cle+07] T. Clerckx, C. Vandervelpen, K. Luyten, and K. Coninx. “A Prototype-Driven Development Process for Context-Aware User Interfaces”. In: *Task Models and Diagrams for Users Interface Design*. Springer, 2007, pp. 339–354. ISBN: 978-3-540-70816-2.
- [CM22] N. Cardozo and K. Mens. “Programming language implementations for context-oriented self-adaptive systems”. In: *Information and Software Technology* 143 (2022), p. 106789. ISSN: 0950-5849.
- [COH14] R. Capilla, Ó. Ortiz, and M. Hinchey. “Context Variability for Context-Aware Systems”. In: *Computer* 47.2 (Feb. 2014), pp. 85–87. ISSN: 0018-9162.
- [Con+03] K. Coninx, K. Luyten, C. Vandervelpen, J. Van den Bergh, and B. Creemers. “Dygames: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems”. In: *Human-Computer Interaction with Mobile Devices and Services*. Springer, 2003, pp. 256–270. ISBN: 978-3-540-45233-1.
- [Cou+05] J. Coutaz, J. L. Crowley, S. Dobson, and D. Garlan. “Context is Key”. In: *Communications of the ACM* 48.3 (Mar. 2005), pp. 49–53. ISSN: 0001-0782.
- [CVC08] B. Collignon, J. Vanderdonckt, and G. Calvary. “Model-Driven Engineering of Multi-target Plastic User Interfaces”. In: *Proceedings of Fourth International Conference on Autonomic and Autonomous Systems*. ICAS ’2008. IEEE, 2008, pp. 7–14.
- [DAS01] A. K. Dey, G. D. Abowd, and D. Salber. “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications”. In: *Human-Computer Interaction* 16.2-4 (2001), pp. 97–166.

- [DCC08] A. Demeure, G. Calvary, and K. Coninx. “COMET(s), A Software Architecture Style and an Interactors Toolkit for Plastic User Interfaces”. In: *Interactive Systems. Design, Specification, and Verification*. Springer, 2008, pp. 225–237. ISBN: 978-3-540-70569-7.
- [Dem+07] A. Demeure, G. Calvary, J. Coutaz, and J. Vanderdonckt. “The Comets Inspector: Towards Run Time Plasticity Control Based on a Semantic Network”. In: *Task Models and Diagrams for Users Interface Design*. Springer, 2007, pp. 324–338. ISBN: 978-3-540-70816-2.
- [Des+07] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D’Hondt. “Context-Oriented Domain Analysis”. In: *Modeling and Using Context: Proceedings of the 6th International and Interdisciplinary Conference*. CONTEXT ’07. Springer, 2007, pp. 178–191. ISBN: 978-3-540-74255-5.
- [DMD18] B. Duhoux, K. Mens, and B. 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*. COP ’18. ACM, 2018, pp. 15–22. ISBN: 9781450357227.
- [DMD19] B. Duhoux, K. Mens, and B. Dumas. “Implementation of a Feature-Based Context-Oriented Programming Language”. In: *Proceedings of the Workshop on Context-Oriented Programming*. COP ’19. ACM, 2019, pp. 9–16. ISBN: 9781450368636.
- [DT22] E. Delhove and H. Y. Tsang. “Adaptive applications with multimodal user interfaces”. MA thesis. Université catholique de Louvain, 2022.
- [Duh+19a] B. Duhoux, B. Dumas, K. Mens, and H. Leung. “A context and feature visualisation tool for a feature-based context-oriented programming language”. In: *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution*. SATToSE ’19. CEUR-WS, July 2019.
- [Duh+19b] B. Duhoux, B. Dumas, H. S. Leung, and K. Mens. “Dynamic Visualisation of Features and Contexts for Context-Oriented Programmers”. In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS ’19. ACM, 2019. ISBN: 9781450367455.
- [Duh16] B. Duhoux. “L’intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte”. MA thesis. Université catholique de Louvain, 2016.

- [EVP00] J. Eisenstein, J. Vanderdonckt, and A. R. Puerta. "Adapting to mobile contexts with user-interface modeling". In: *Proceedings of 3rd Workshop on Mobile Computing Systems and Applications*. WMCSA '00). IEEE, 2000, p. 83.
- [Fis12] G. Fischer. "Context-Aware Systems: The 'right' Information, at the 'Right' Time, in the 'Right' Place, in the 'Right' Way, to the 'Right' Person". In: *Proceedings of the International Working Conference on Advanced Visual Interfaces*. AVI '12. ACM, 2012, pp. 287–294. ISBN: 9781450312875.
- [FWM08] P. Fernandes, C. M. L. Werner, and L. G. P. Murta. "Feature Modeling for Context-Aware Software Product Lines". In: *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*. SEKE '08. Knowledge Systems Institute Graduate School, 2008, pp. 758–763.
- [FWT11] P. Fernandes, C. M. L. Werner, and E. Teixeira. "An Approach for Feature Modeling of Context-Aware Software Product Line". In: *Journal of Universal Computer Science* 17.5 (2011), pp. 807–829.
- [Gab+11] Y. Gabillon, M. Petit, G. Calvary, and H. Fiorino. "Automated Planning for User Interface Composition". In: *Proceedings of the 2nd International Workshop on Semantic Models for Adaptive Interactive Systems of the 2011 International Conference on Intelligent User Interfaces*. SEMAIS '11. Feb. 2011.
- [Giu+19] T. Giuffrida, S. Dupuy-Chessa, J. Poli, and É. Céret. "Fuzzy4U: A fuzzy logic system for user interfaces adaptation". In: *Proceedings of the 13th International Conference on Research Challenges in Information Science*. RCIS '19. IEEE, 2019, pp. 1–12.
- [GMC08] S. González, K. Mens, and A. Cádiz. "Context-Oriented Programming with the Ambient Object System". In: *Journal of Universal Computer Science* 14.20 (Nov. 28, 2008), pp. 3307–3332.
- [GMH07] S. González, K. Mens, and P. Heymans. "Highly Dynamic Behaviour Adaptability through Prototypes with Subjective Multimethods". In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS '07. ACM, 2007, pp. 77–88. ISBN: 9781595938688.
- [God08] S. Goderis. "On the Separation of User Interface Concerns - A Programmer's Perspective on the Modularisation of User Interface Code". PhD thesis. Vrije Universiteit Brussel, 2008.

- [Gon+11] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. “Subjective-C”. In: *Proceedings of the 3rd International Conference on Software Language Engineering*. SLE '10. Springer, 2011, pp. 246–265. ISBN: 978-3-642-19440-5.
- [Gon+13] S. González, K. Mens, M. Colacioiu, and W. Cazzola. “Context Traits: Dynamic Behaviour Adaptation Through Run-time Trait Recomposition”. In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*. AOSD '13. ACM, 2013, pp. 209–220. ISBN: 978-1-4503-1766-5.
- [GP96] T. Green and M. Petre. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework”. In: *Journal of Visual Languages and Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X.
- [GPS10] C. Ghezzi, M. Pradella, and G. Salvaneschi. “Programming Language Support to Context-aware Adaptation: A Case-study with Erlang”. In: *Proceedings of 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, 2010, pp. 59–68. ISBN: 978-1-60558-971-8.
- [Hal+06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. “Using Product Line Techniques to Build Adaptive Systems”. In: *Proceedings of the 10th International on Software Product Line Conference*. SPLC '06. IEEE, 2006, pp. 141–150. ISBN: 0769525997.
- [HCH08] R. Hirschfeld, P. Costanza, and M. Haupt. “An Introduction to Context-Oriented Programming with ContextS”. In: *Generative and Transformational Techniques in Software Engineering II*. GTTSE '07. Springer, 2008, pp. 396–407. ISBN: 978-3-540-88643-3.
- [HCN08] R. Hirschfeld, P. Costanza, and O. Nierstrasz. “Context-Oriented Programming”. In: *Journal of Object Technology* (2008), pp. 125–151.
- [Her07] S. Herrmann. “A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java”. In: *Applied Ontology* 2.2 (Apr. 2007), pp. 181–207. ISSN: 1570-5838.
- [Her21] F. Hermans. *The Programmer’s Brain: What every programmer needs to know about cognition*. Simon and Schuster, 2021. ISBN: 9781617298677.

- [HI06] K. Henriksen and J. Indulska. “Developing context-aware pervasive computing applications: Models and approach”. In: *Pervasive and Mobile Computing* 2.1 (2006), pp. 37–64. ISSN: 1574-1192.
- [HIM11] R. Hirschfeld, A. Igarashi, and H. Masuhara. “ContextFJ: A Minimal Core Calculus for Context-Oriented Programming”. In: *Proceedings of the 10th International Workshop on Foundations of Aspect-Oriented Languages*. FOAL ’11. ACM, 2011, pp. 19–23. ISBN: 9781450306447.
- [HO93] W. Harrison and H. Ossher. “Subject-Oriented Programming: A Critique of Pure Objects”. In: *Proceedings of the Eight Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’93. ACM, 1993, pp. 411–428. ISBN: 0897915879.
- [HT08] H. Hartmann and T. Trew. “Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains”. In: *Proceedings of 12th International Software Product Line Conference*. SPLC ’08. IEEE, 2008, pp. 12–21. ISBN: 978-0-7695-3303-2.
- [Igl22] J. Iglesias Garcia. “Sensor detection and simulation in feature-based context-oriented programming”. MA thesis. Université catholique de Louvain, 2022.
- [ILE16] S. Illescas, R. E. Lopez-Herrejon, and A. Egyed. “Towards Visualization of Feature Interactions in Software Product Lines”. In: *IEEE Working Conference on Software Visualization*. VISSOFT ’16. IEEE, Oct. 2016, pp. 46–50.
- [JLS10] Z. Jaroucheh, X. Liu, and S. Smith. “Mapping Features to Context Information: Supporting Context Variability for Context-Aware Pervasive Applications”. In: *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Vol. 1. WI ’10. IEEE, Aug. 2010, pp. 611–614.
- [Joh+19] V. Johnston, M. Black, J. Wallace, M. Mulvenna, and R. Bond. “A Framework for the Development of a Dynamic Adaptive Intelligent User Interface to Enhance the User Experience”. In: *Proceedings of the 31st European Conference on Cognitive Ergonomics*. ECCE ’19. ACM, 2019, pp. 32–35. ISBN: 9781450371667.
- [KAM11] T. Kamina, T. Aotani, and H. Masuhara. “EventCJ: A Context-Oriented Programming Language with Declarative Event-Based Context Transition”. In: *Proceedings of the Tenth International*

- Conference on Aspect-Oriented Software Development. AOSD '11. ACM, 2011, pp. 253–264. ISBN: 9781450306058.*
- [Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Carnegie-Mellon University Software Engineering Institute, Nov. 1990.
- [Kas+09] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. “FeatureIDE: A Tool Framework for Feature-oriented Software Development”. In: *Proceedings of the 31st International Conference on Software Engineering. ICSE '09. IEEE, 2009, pp. 611–614. ISBN: 978-1-4244-3453-4.*
- [KPG04] B. Kurz, I. Popescu, and S. Gallacher. “FACADE - a framework for context-aware content adaptation and delivery”. In: *Proceedings of the Second Annual Conference on Communication Networks and Services Research. CNSR '04. IEEE, 2004, pp. 46–55.*
- [KR03] R. Keays and A. Rakotonirainy. “Context-Oriented Programming”. In: *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access. MobiDE '03. ACM, 2003, pp. 9–16. ISBN: 1581137672.*
- [Kru+15] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. “A Survey on Engineering Approaches for Self-Adaptive Systems”. In: *Pervasive and Mobile Computing 17* (Feb. 2015), pp. 184–206. ISSN: 1574-1192.
- [KS15] E. Karuzaki and A. Savidis. “Yeti: Yet Another Automatic Interface Composer”. In: *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems. EICS '15. ACM, 2015, pp. 12–21. ISBN: 9781450336468.*
- [Küh+14] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. “A Metamodel Family for Role-Based Modeling and Programming Languages”. In: *Proceedings of the 7th International Conference on Software Language Engineering. SLE '14. Springer, 2014, pp. 141–160. ISBN: 978-3-319-11245-9.*
- [Küh17] A. Kühn. “Reconciling Context-Oriented Programming and Feature Modeling”. MA thesis. Université catholique de Louvain, 2017.
- [LDN07] M. von Löwis, M. Denker, and O. Nierstrasz. “Context-Oriented Programming: Beyond Layers”. In: *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with*

- the 15th International Smalltalk Joint Conference 2007*. ICDL '07. ACM, 2007, pp. 143–156. ISBN: 9781605580845.
- [Led+18] D. Ledo, S. Houben, J. Vermeulen, N. Marquardt, L. Oehlberg, and S. Greenberg. “Evaluation Strategies for HCI Toolkit Research”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. ACM, 2018, pp. 1–17. ISBN: 9781450356206.
- [Leu19] H. S. Leung. “Visualisation of Contexts and Features in Context-Oriented Programming”. MA thesis. Université catholique de Louvain, 2019.
- [Lim+05] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. “USIXML: A Language Supporting Multi-path Development of User Interfaces”. In: *Engineering Human Computer Interaction and Interactive Systems*. Springer, 2005, pp. 200–220. ISBN: 978-3-540-31961-0.
- [Lin+11] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. “An open implementation for context-oriented layer composition in ContextJS”. In: *Science of Computer Programming* (2011), pp. 1194–1209. ISSN: 0167-6423.
- [Lóp+08] V. López-Jaquero, J. Vanderdonckt, F. Montero, and P. González. “Towards an Extended Model of User Interface Adaptation: The Isatine Framework”. In: *Engineering Interactive Systems*. Springer, 2008, pp. 374–392. ISBN: 978-3-540-92698-6.
- [Luy+08] K. Luyten, J. Meskens, J. Vermeulen, and K. Coninx. “Meta-Gui-Builders: Generating Domain-Specific Interface Builders for Multi-Device User Interface Creation”. In: *CHI '08 Extended Abstracts on Human Factors in Computing Systems*. CHIEA '08. ACM, 2008, pp. 3189–3194. ISBN: 9781605580128.
- [Lyl08] J. Lyle. *FXRuby: create lean and mean GUIs with Ruby*. Raleigh, 2008.
- [Mal+10] D. Malandrino, F. Mazzoni, D. Riboni, C. Bettini, M. Colajanni, and V. Scarano. “MIMOSA: context-aware adaptation for ubiquitous web access”. In: *Personal and ubiquitous computing 14.4* (2010), pp. 301–320.
- [Mar+17] J. Martinez, J.-S. Sottet, A. G. Frey, T. Ziadi, T. Bissyandé, J. Vanderdonckt, J. Klein, and Y. Le Traon. “Variability Management and Assessment for User Interface Design”. In: *Human Centered Software Product Lines*. Springer, 2017, pp. 81–106. ISBN: 978-3-319-60947-8.

- [Mar+21] P. Martou, K. Mens, B. Duhoux, and A. Legay. “Test Scenario Generation for Context-Oriented Programs”. In: *Computing Research Repository* abs/2109.11950 (2021).
- [Mar+22] P. Martou, K. Mens, B. Duhoux, and A. Legay. “Generating Virtual Scenarios for Cyber Ranges from Feature-Based Context-Oriented Models: A Case Study”. In: *Proceedings of the International Workshop on Context-Oriented Programming and Advanced Modularity*. COP ’22. ACM, 2022.
- [Mar21a] P.-O. Martin. “Context-Specific Composition of Features in Context-Oriented Programming”. MA thesis. Université catholique de Louvain, 2021.
- [Mar21b] P. Martou. “Towards a Testing Approach for Feature-Based Context-Oriented Programming Systems”. MA thesis. Université catholique de Louvain, 2021.
- [Mau+16] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. “Context Aware Reconfiguration in Software Product Lines”. In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’16. ACM, 2016, pp. 41–48. ISBN: 978-1-4503-4019-9.
- [Mau+18] J. Mauro, M. Nieke, C. Seidl, and I. Chieh Yu. “Context-aware reconfiguration in evolving software product lines”. In: *Science of Computer Programming* 163 (2018), pp. 139–159. ISSN: 0167-6423.
- [Mau21] J. Mauro. “Anomaly Detection in Context-Aware Feature Models”. In: *15th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS’21. ACM, 2021. ISBN: 9781450388245.
- [MBC09] M. Mendonca, M. Branco, and D. Cowan. “S.P.L.O.T.: Software Product Lines Online Tools”. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. ACM, 2009, pp. 761–762. ISBN: 978-1-60558-768-4.
- [MCD16] K. Mens, N. Cardozo, and B. Duhoux. “A Context-Oriented Software Architecture”. In: *Proceedings of the 8th International Workshop on Context-Oriented Programming*. COP ’16. ACM, 2016, pp. 7–12. ISBN: 978-1-4503-4440-1.

- [MDC17] K. Mens, B. Duhoux, and N. Cardozo. “Managing the Context Interaction Problem: A Classification and Design Space of Conflict Resolution Techniques in Dynamically Adaptive Software Systems”. In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. Programming ’17. ACM, 2017. ISBN: 9781450348362.
- [Men+17] K. Mens, R. Capilla, H. Hartmann, and T. Kropf. “Modeling and Managing Context-Aware Systems’ Variability”. In: *IEEE Software* 34.6 (Nov. 2017), pp. 58–63. ISSN: 0740-7459.
- [Mor+08] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. “An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability”. In: *In Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*. Vol. 5301. MoDELS ’08. Springer, 2008, pp. 782–796.
- [Mou22] H. Mouligneaux. “Context-Oriented State”. MA thesis. Université catholique de Louvain, 2022.
- [MPS03] G. Mori, F. Paternò, and C. Santoro. “Tool Support for Designing Nomadic Applications”. In: *Proceedings of the 8th International Conference on Intelligent User Interfaces*. IUI ’03. ACM, 2003, pp. 141–148. ISBN: 1581135866.
- [MPV11] G. Meixner, F. Paternò, and J. Vanderdonckt. “Past, Present, and Future of Model-Based User Interface Development”. In: *i-com* 10.3 (2011), pp. 2–11.
- [Mur+14] A. Murguzur, R. Capilla, S. Trujillo, Ó. Ortiz, and R. E. Lopez-Herrejon. “Context Variability Modeling for Runtime Configuration of Service-based Dynamic Software Product Lines”. In: *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*. SPLC ’14. ACM, 2014, pp. 2–9. ISBN: 978-1-4503-2739-8.
- [MV13] V. G. Motti and J. Vanderdonckt. “A computational framework for context-aware adaptation of user interfaces”. In: *IEEE 7th International Conference on Research Challenges in Information Science*. RCIS ’13. IEEE, 2013, pp. 1–12.
- [MVA08] F. Martinez-Ruiz, J. Vanderdonckt, and J. Arteaga. “Context-Aware Generation of User Interface Containers for Mobile Devices”. In: *2008 Mexican International Conference on Computer Science*. ENC ’08. IEEE, 2008, pp. 63–72.

- [NES17] M. Nieke, G. Engel, and C. Seidl. “DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines”. In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems*. VAMOS ’17. ACM, 2017, pp. 92–99. ISBN: 9781450348119.
- [Pat04] F. Paternò. “ConcurTaskTrees: an engineered notation for task models”. In: *The handbook of task analysis for human-computer interaction* (2004), pp. 483–503.
- [PBD09] C. Parra, X. Blanc, and L. Duchien. “Context awareness for dynamic service-oriented product lines”. In: *Proceedings of the 13th International Software Product Line Conference*. Vol. 446. SPLC ’09. ACM, 2009, pp. 131–140.
- [PBV05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Vol. 1. Springer, 2005. ISBN: 978-3540243724.
- [PS02] F. Paternò and C. Santoro. “One Model, Many Interfaces”. In: *Computer-Aided Design of User Interfaces III: Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces*. Springer, 2002, pp. 143–154. ISBN: 978-94-010-0421-3.
- [PSS09] F. Paternò, C. Santoro, and L. D. Spano. “MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments”. In: *ACM Transactions on Computer-Human Interaction* 16.4 (Nov. 2009). ISSN: 1073-0516.
- [PV12] T. Poncelet and L. Vigneron. “The Phenomenal Gem: Putting Features as a Service on Rails”. MA thesis. Université catholique de Louvain, 2012.
- [PVM09] S. Pietschmann, M. Voigt, and K. Meißner. “Dynamic Composition of Service-Oriented Web User Interfaces”. In: *2009 Fourth International Conference on Internet and Web Applications and Services*. ICIW ’09. IEEE, 2009, pp. 217–222.
- [RGR18] P. Rosenberger, D. Gerhard, and P. Rosenberger. “Context-Aware System Analysis: Introduction of a Process Model for Industrial Applications.” In: *Proceedings of the 20th International Conference on Enterprise Information Systems*. Vol. 2. ICEIS ’18. Science and Technology Publications, 2018, pp. 368–375.
- [Sau11] J. Sauro. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. Measuring Usability LLC, Apr. 2011.

- [SGP11] G. Salvaneschi, C. Ghezzi, and M. Pradella. “JavaCtx: Seamless Toolchain Integration for Context-oriented Programming”. In: *Proceedings of 3rd International Workshop on Context-Oriented Programming*. COP ’11. ACM, 2011, 4:1–4:6. ISBN: 978-1-4503-0891-5.
- [SGP12a] G. Salvaneschi, C. Ghezzi, and M. Pradella. “Context-oriented programming: A software engineering perspective”. In: *Journal of Systems and Software* 85.8 (2012), pp. 1801–1817. ISSN: 0164-1212.
- [SGP12b] G. Salvaneschi, C. Ghezzi, and M. Pradella. “ContextErlang: Introducing Context-oriented Programming in the Actor Model”. In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*. AOSD ’12. ACM, 2012, pp. 191–202. ISBN: 978-1-4503-1092-5.
- [SHT06] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. “Feature Diagrams: A Survey and a Formal Semantics”. In: *Proceedings of 14th IEEE International Requirements Engineering Conference*. RE ’06. IEEE, 2006, pp. 139–148.
- [SLR13] K. Saller, M. Lochau, and I. Reimund. “Context-Aware DSPLs: Model-Based Runtime Adaptation for Resource-Constrained Systems”. In: *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops*. SPLC ’13. ACM, 2013, pp. 106–113. ISBN: 9781450323253.
- [SMH17] M. Springer, H. Masuhara, and R. Hirschfeld. “A Layer-based Approach to Hierarchical Dynamically-scoped Open Classes”. In: *Journal of Information Processing* 25 (2017), pp. 296–307.
- [Sot+07] J.-S. Sottet, V. Ganneau, G. Calvary, J. Coutaz, A. Demeure, J.-M. Favre, and R. Demumieux. “Model-Driven Adaptation for Plastic User Interfaces”. In: *Human-Computer Interaction – INTERACT 2007*. Springer, 2007, pp. 397–410. ISBN: 978-3-540-74796-3.
- [Sou+17] I. de Sousa Santos, M. L. de Jesus Souza, M. L. Luciano Carvalho, T. Alves Oliveira, E. S. de Almeida, and R. M. de Castro Andrade. “Dynamically Adaptable Software Is All about Modeling Contextual Variability and Avoiding Failures”. In: *IEEE Software* 34.6 (2017), pp. 72–77.
- [ST09] M. Salehie and L. Tahvildari. “Self-Adaptive Software: Landscape and Research Challenges”. In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (May 2009). ISSN: 1556-4665.

- [TC99] D. Thevenin and J. Coutaz. “Plasticity of User Interfaces: Framework and Research Agenda.” In: *Human-Computer Interaction INTERACT '99: IFIP TC13*. Vol. 99. IOS Press, 1999, pp. 110–117.
- [Url+15] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser. “A visual support for decomposing complex feature models”. In: *Proceedings of the 3rd Working Conference on Software Visualization*. VISSOFT '15. IEEE. Sept. 2015, pp. 76–85.
- [Van+05] J. Vanderdonckt, D. Grolaux, P. Van Roy, Q. Limbourg, B. Macq, and B. Michel. “A Design Space for Context-Sensitive User Interfaces.” In: *Proceedings of the 14th International Conference on Intelligent and Adaptive Systems and Software Engineering*. ISCA '14. ISCA, 2005, pp. 207–214.
- [Van20] A. Van den Bogaert. “Consistency Management of Contexts and Features in Context-Oriented Programming Language with SAT Solving”. MA thesis. Université catholique de Louvain, 2020.
- [VLC03] J. Van den Bergh, K. Luyten, and K. Coninx. “A run-time system for context-aware multi-device user interfaces”. In: *Human-Computer Interaction: Theory and Practice 2* (2003), p. 308.
- [Was+10] B. H. Wasty, A. Semmo, M. Appeltauer, B. Steinert, and R. Hirschfeld. “ContextLua: Dynamic Behavioral Variations in Computer Games”. In: *Proceedings of the 2nd International Workshop on Context-Oriented Programming*. COP '10. ACM, 2010. ISBN: 978-1-4503-0531-0.
- [Yig+19] E. Yigitbas, K. Josifovska, I. Jovanovikj, F. Kalinci, A. Anjorin, and G. Engels. “Component-Based Development of Adaptive User Interfaces”. In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '19. ACM, 2019. ISBN: 9781450367455.
- [Yig+20] E. Yigitbas, I. Jovanovikj, K. Biermeier, S. Sauer, and G. Engels. “Integrated model-driven development of self-adaptive user interfaces”. In: *Software and Systems Modeling* 19.5 (2020), pp. 1057–1081.