UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

L'intégration des adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte

Mémoire présenté par
Benoît DUHOUX

en vue de l'obtention du grade de Master
sciences informatiques
Options: ingénierie logicielle et systèmes de programmation & sécurité et réseaux informatiques

Promoteurs
Kim MENS *(UCL)*, Bruno DUMAS *(UNamur)*

Lecteurs
Anthony CLEVE *(UNamur)*, Guillaume MAUDOUX *(UCL)*

Année académique 2015-2016

Résumé

A l'avenir, les *smart cities* se développeront et créeront ainsi des villes dites connectées. Quoi de mieux, que de pouvoir bénéficier de telles informations collectées par les senseurs, pour adapter le comportement des applications des utilisateurs. Cependant, avec les technologies d'aujourd'hui, il est fastidieux de changer le comportement de telles applications à l'exécution.

Ce travail a pour but de développer un framework permettant de créer plus aisément ce type d'applications. Celles-ci devront pouvoir s'adapter en fonction de leur environnement courant, c'est-à-dire que leur comportement changera selon la nouvelle situation qui a été repérée. La détection d'une situation peut venir soit d'un évènement externe, comme la météo, soit d'un évènement interne, tel que le niveau de la batterie du smartphone, mais devra également prendre en compte les tâches de l'utilisateur, comme par exemple, le fait que ce dernier donne ses préférences.

Cette solution devra pouvoir gérer de manière uniformisée les aspects sur les interfaces utilisateur, sur les comportements ainsi que sur les données. Cependant, dans ce mémoire, la perspective étudiée concernera les interfaces utilisateur, à savoir comment les adaptations pourront rendre le meilleur visuel possible à l'utilisateur tout en considérant son environnement global.

Enfin, un outil web sera également développé pour simuler de tels environnements afin de pouvoir créer, tester et mieux comprendre le fonctionnement de ce type d'applications.

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé à réaliser ce travail de fin d'études, et plus particulièrement :

MENS Kim mon promoteur, pour son expertise dans la programmation orientée contexte, ainsi que son aide et ses précieux conseils qui m'ont aiguillé dans ce domaine inconnu pour moi ;

DUMAS Bruno mon second promoteur, pour son expertise dans l'interaction homme-machine, ainsi que pour son intérêt porté à mon mémoire ;

NANDRIN Vincent et ses parents pour leur soutien et encouragements tout au long de mon parcours universitaire ;

DENIS Isabelle pour sa relecture.

Table des matières

Table des figures

Table des codes

Chapitre 1

Introduction

On entend de plus en plus parler de villes dites connectées, c'est-à-dire, des villes intelligentes (*smart cities*). Celles-ci sont équipées de capteurs de tous genres, connectés entre eux. Leur but étant de collecter un maximum d'informations sur leur environnement, et ce pour améliorer la vie de et dans celles-ci.

Dans ces villes, on pourrait imaginer des capteurs pour détecter des catastrophes naturelles, comme les tremblements de terre. Ceux-ci enverraient leurs données aux sismologues afin que ces derniers les étudient. Cependant, dans le cas d'un désastre, il serait bien de pouvoir aussi les utiliser afin de prévenir directement la population de façon à la mettre en sécurité. Des applications de ce style pourraient être créées dans le but de prévenir les personnes du problème, avec une solution générique pour les sauver.

Néanmoins, si on souhaite que l'utilisateur ait la meilleure expérience et ainsi avoir plus de chances de pouvoir se sauver, cette application devra s'adapter en fonction de l'environnement dans lequel elle s'exécute. De plus, il sera rare que deux personnes se retrouvent exactement dans le même environnement. Effectivement, il se peut qu'une personne n'ait plus beaucoup de batterie, une autre n'ait plus de connectivité, une tierce personne n'ait plus de puce GPS fonctionnelle, ...

Par conséquent, créer de telles applications, étant capables de s'adapter toutes seules, en se basant uniquement sur l'environnement l'entourant, n'est pas chose aisée avec les architectures actuelles de développement (cf. le chapitre 4 montrant pourquoi cela n'est pas si simple). C'est pourquoi, ce mémoire a pour but de développer un tel framework pouvant aider à prendre en compte la situation courante et ainsi adapter l'application.

Dans la suite de ce mémoire, je commencerai par faire un tour d'horizon sur les travaux apparentés (cf. le chapitre 2), en parlant de la programmation orientée contexte et des adaptations des interfaces utilisateur. Ensuite, afin de mieux comprendre ce genre d'applications, j'expliquerai l'étude de cas (cf. le chapitre 3), une application sur les *emergency response system*, et l'approche utilisée (cf. la chapitre 4), où, entre autres, la motivation y sera décrite. Viendront par la suite, l'architecture (cf. la chapitre 5) et l'implémentation du framework (cf. la chapitre 6), dans lesquelles j'exemplifierai le processus interne d'une application orientée contexte avec cette étude de cas. Par après, je présenterai un autre outil, le simulateur (cf. la chapitre 7). Il servira, non seulement, à créer et tester des applications orientées contexte,

mais également, à valider le framework. Avant de conclure, j'exposerai les limitations de ce dernier et des travaux futurs pouvant être réalisés (cf. la chapitre 8).

Les codes des outils développés dans ce mémoire se trouvent sur le bitbucket suivant : https://benoitduhoux@bitbucket.org/benoitduhoux/context-oriented-programming-unified-approach.git.

Chapitre 2

Les travaux apparentés

Dans ce chapitre, je commencerai par définir le mot `contexte`, pour ensuite expliquer les applications conscientes du contexte (les *context-aware systems*). Après, une vision de la programmation orientée contexte sera exemplifiée par le langage de programmation *Subjective-C* [25]. Après avoir donné un aperçu de ces notions, j'expliquerai certains concepts dans les adaptations des interfaces utilisateur, tels que la plasticité dans les systèmes conscients du contexte, ainsi que l'intérêt des transitions dans de tels systèmes. Enfin, je résumerai ce qu'on entend par le *feature modeling*.

Je tiens à préciser que les définitions données dans ce chapitre sont restées en anglais, afin de garder tout le sens donné par l'auteur original.

2.1 Un contexte, qu'est-ce ?

Baldauf, Dustdar et Rosenberg [11] centralisent plusieurs définitions du mot `contexte`, vu par plusieurs auteurs à travers les années. Les exposer toutes n'aurait que très peu d'intérêt mais certaines d'entre elles se retrouvent ci-dessous.

En 1994, Schilit, Adams et Want [2] ont défini trois aspects importants dans la définition d'un contexte :

— *Where you are*, représentant les emplacements telle la location de l'utilisateur ;

— *Who you are with*, représentant les personnes qui entourent l'utilisateur, c'est-à-dire la situation sociale de l'utilisateur. En fonction des personnes aux alentours, l'application agira différemment ;

— et *What resources are nearby*, caractérisant les senseurs externes (les capteurs de lumière, de bruit, ...) ainsi que l'état du système (état de la mémoire, état de la connectivité, ...).

En 2001, Dey [7] donnera sa propre définition du terme `contexte` sur base de précédentes définitions : "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*".

En 2005, Coutaz [10] insista sur ce mot avec la phrase suivante : "*Context is Key.*". Comme elle le fait remarquer, le contexte n'est pas seulement un environnement. Il est considéré comme partie intégrante du processus, car

sans lui, les applications ne pourraient pas changer de comportement afin d'améliorer l'expérience utilisateur.

2.2 Les systèmes conscients du contexte

Plus connu sous son nom anglophone *context-aware systems*, c'est un domaine qui est traité depuis plusieurs années.

En 2001, Dey [7] introduira ces systèmes en faisant une analogie avec les conversations entre humains. Il explique que les humains utilisent leur environnement pour enrichir leurs conversations, comme par exemple, utiliser leurs mains pour montrer un évènement au loin, ... Du coup, en intégrant ces contextes dans les communications hommes-machines, il pensait que de meilleurs services informatiques pourraient être fournis. Sur base de cette pensée et de cette analogie, Dey définira les systèmes conscients du système comme suit : "*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.*".

Beaucoup d'autres définitions sont similaires à la précédente comme le montrent les suivantes : "*Context-aware systems are computer systems that can provide relevant services and information to users by exploiting context*" [9], ou encore " *context-aware systems is to acquire and utilize information on the context of a device in order to provide services that are appropriate to the particular people, place, time, event, etc.*" [20].

En résumé, il faudra considérer qu'un système est conscient du contexte lorsque celui s'adapte en fonction des contextes/de l'environnement qui l'entourent. La figure 2.1 [25] montre une architecture type pour ces systèmes. Les données des senseurs seront découvertes ainsi que les interactions avec l'utilisateur pour être traitées par le gestionnaire de contextes. Ce dernier mettra à jour ceux activés, ce qui impactera le comportement de l'application. Par exemple, si le module *Context Discovery* découvre que la batterie du smartphone est faible, le module *Context Management* activera le contexte *Low Battery*. Ce changement aura pour effet de modifier le comportement de l'application en supprimant certaines fonctionnalités, comme par exemple, celle du GPS qui affichait une carte navigable dynamique.

2.3 La programmation orientée contexte

Maintenant que j'ai expliqué les systèmes conscients du contexte, je présenterai dans cette section une branche de ces systèmes : les langages de programmation orientée contexte.

La programmation orientée contexte (COP) est depuis peu un nouveau paradigme de programmation comme l'annoncent Gonzáles, Cardozo, Mens, Cádiz, Libbrecht et Goffaux [25]. Ces auteurs précisent l'avantage qu'on peut avoir en utilisant ce paradigme plutôt qu'un autre, comme l'orienté objet, lorsqu'on doit développer des applications conscientes du système. Dans l'orienté objet, si on devait prendre en compte des contextes, cela deviendrait très vite ingérable vu le nombre de déclarations de conditionnelles

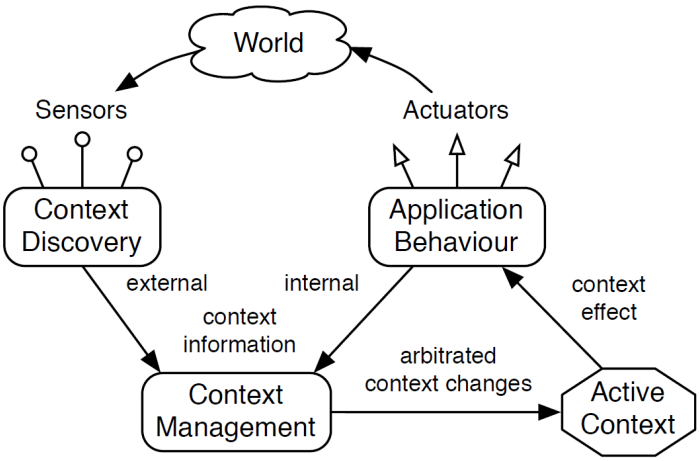


FIGURE 2.1: Vue d'une architecture type dans les systèmes conscients du contexte. Figure extraite de l'article [25]

(*IF-ELSE*) pour gérer la multitude de cas différents correspondant aux divers contextes. De plus, lors de la maintenance, il serait très difficile d'en étendre le comportement. Tandis qu'en utilisant ce nouveau paradigme, il serait bien plus facile de prendre en compte l'environnement courant, et d'adapter le comportement de l'application, vu qu'il a été prévu à cette effet. Par la même occasion, la maintenance en serait d'autant plus agréable.

Plusieurs langages ont leurs propres versions, voire extensions, pour gérer ce paradigme, comme en Erlang [22, 29], en SmallTalk [18], en JavaScript [26, 31], en Java [14, 23, 27], en Ambience [13, 15, 16], en Objective-C [25], en Ruby [28], ou encore, en Common Lisp Object System (CLOS) [12]. Toutes ces versions ont un même point commun : adapter le comportement de l'application mais pas les interfaces utilisateur, ni l'aspect base de données.

Je ne vous parlerai que du *Subjective-C* dans la section suivante 2.3.1, car c'est principalement sur celui-là que je me baserai pour faire mon travail (cf. les chapitres 5 et 6). De plus, ce langage sert d'exemple représentatif d'une classe importante de langages orientés contexte.

2.3.1 *Subjective-C*

Cette section résume l'article sur le langage orienté contexte *Subjective-C* [25], développé par Gonzáles, Cardozo, Mens, Cádiz, Libbrecht et Goffaux. Ce langage *Subjective-C* est une couche développée au-dessus du langage *Objective-C* en incluant ce nouveau paradigme. Cette couche permet de développer des applications orientées contexte pour les environnements *iOS*. Cette section se divisera en deux sous-sections, l'une pour exposer le *Subjective-C* et l'autre pour montrer certains détails d'implémentation le concernant.

La programmation orientée-contexte en *Subjective-C*

Les auteurs ont voulu donner une définition légèrement différente pour un `contexte` : "*Context is an abstraction of the particular state of affairs or set of circumstances for which specialised application behaviour can be defined*". Celle-ci est plus précise que celles vues à la section 2.1 car cette définition instaure des changements de comportement avec d'autres comportements plus raffinés, et ce, en fonction du contexte découvert.

Comme le montre la figure 2.1, ces données contextuelles seront perçues par un module *Context Discovery* pour les traduire en contextes, déclarés dans le *Context Manager* du système.

Chaque contexte devra garder un état qui permettra de savoir si le contexte est activé ou non et le nombre de fois qu'il a été activé. Ce concept permettra de connaître à tout moment l'environnement courant dans lequel l'application s'exécute, car cet environnement pourra être reconstitué sur base des contextes activés. Dès lors, il est évident que seuls les contextes actifs auront un effet sur le comportement de l'application. En *Subjective-C*, les changements de contexte se feront par les *context switches* du module *Context Manager*, c'est-à-dire qu'à chaque changement de contexte, le système passera d'un état à un autre, où l'état représente des contextes activés.

En plus de savoir si un contexte est actif ou pas, les auteurs ont également prévu des relations entre ceux-ci. C'est-à-dire que l'activation d'un contexte pourra entraîner l'activation d'un autre. Une seconde illustration serait d'en exclure l'activation si ces deux-ci s'excluent mutuellement. Par exemple, si le contexte `Bruxelles` s'active, cela impliquera que le contexte `Belgique` s'activera à son tour. Un autre exemple pourrait être une relation d'exclusion entre les contextes `En mouvement` et `Immobile`. Cette relation entraînera qu'un seul des deux pourra être activé à la fois. Ce principe implique des vérifications au préalable avant de les activer réellement, et ce, pour éviter des incohérences. J'en parlerai plus en détail dans la section 2.3.2.

Mais l'activation en cascade pouvait amener l'activation d'un contexte plusieurs fois. C'est pourquoi, ils ont aussi ajouté un compteur à chaque contexte activé. Ce compteur représente le nombre de fois qu'il a été activé. Lorsqu'il descend à zéro, ce dernier doit être désactivé.

Comme l'ont précisé les auteurs, chaque changement de contexte impactera le comportement de l'application grâce aux adaptations correspondantes. Une adaptation est un comportement par défaut qui a été modifié pour mieux répondre aux besoins de l'utilisateur en fonction de l'environnement détecté. Ces adaptations peuvent atteindre un niveau de granularité très fine, jusqu'au niveau des méthodes.

Comme dans toute bonne conception en génie logicielle, lorsqu'un développeur implémente, il se peut qu'il doive utiliser le principe d'héritage pour éviter les duplications de code. Grâce à la notion du *supercontext*, le comportement spécifique à un contexte particulier peut appeler le comportement d'un contexte moins spécifique, voire même le comportement du contexte par défaut pour ainsi éviter les duplications de code. Cependant, il ne faut

pas confondre ce principe de *supercontext* à celui d'héritage, car le principe du *supercontext* est plus permissif grâce à la sélection du développeur. C'est-à-dire que chaque *supercontext* ne doit pas spécialement appeler son contexte parent, il peut appeler un autre contexte avec lequel il n'a aucun lien.

Implémentation du *Subjective-C*

Mais comment les auteurs ont-ils implémenté cette couche pour gérer ces divers concepts ?

Premièrement, ils ont utilisé le *name mangling*. Le *name mangling* a pour but de rajouter de l'information à une fonction afin de la dissocier des autres. Ce concept a été utile car lors de la définition des adaptations, plusieurs versions d'une même méthode existaient ; il y avait en effet celle par défaut, celle pour le contexte A, celle pour le contexte B, ... chacune avec des annotations représentant les contextes auxquels ces méthodes s'appliquent. Cela permettait de distinguer les méthodes entre-elles. Par conséquent, lorsque le développeur donnait une version d'une méthode pour un contexte particulier, le *Subjective-C* la traduisait en préfixant le nom de la méthode par le mot *Context* suivi du contexte.

Après une traduction des noms de méthodes, il fallait trouver un moyen pour appeler les méthodes correspondant aux contextes activés. Ils utilisèrent le processus appelé *method predispatch*. Ils ont implémenté cette solution en changeant dynamiquement les implémentations des méthodes en *Objective-C*. Pour cela, ils ont dû utiliser la couche de réflexion en *Objective-C* pour changer les implémentations des méthodes. Cependant, cette solution a comme inconvénient qu'à chaque changement de contexte, le *Subjective-C* doit recalculer l'ensemble des adaptations à activer.

Enfin, une macro pré-processeur `SUPERCONTEXT` a été définie pour gérer les super contextes.

2.3.2 Les dépendances entre contextes

Comme je l'ai précisé dans la section précédente, le *Subjective-C* permet de déclarer et de tenir compte des dépendances entre contextes. Toujours dans ce même article, les auteurs ont défini quatre types de dépendances possibles entre les contextes :

exclusion lorsque deux contextes A et B sont exclus, cela signifie que le contexte A (respectivement B) ne pourra pas être activé si le contexte B (respectivement A) est déjà activé ;

requirement lorsqu'un contexte A *requires* un contexte B, cela signifie que le contexte A ne pourra jamais être activé si le contexte B n'est pas activé au préalable. Dans l'article, les auteurs ont pris l'exemple suivant pour illustrer cette relation : un contexte `HDVideo` ne pourra être activé que si le contexte `HighBattery` est activé.

weak inclusion il est possible, de temps en temps, qu'un contexte A doive pouvoir activer un contexte B. Les auteurs considèrent cette relation comme une faible inclusion, car, après avoir activé les contextes A

et B, le contexte B peut être désactivé sans impliquer la désactivation du contexte A. En reprenant l'exemple de l'article, un contexte `Cafétéria` inclut faiblement un contexte `Bruit`. Il est vrai qu'une cafétéria inclut souvent du bruit, mais ce n'est pas toujours le cas.

strong inclusion cette inclusion est similaire à la précédente, à l'exception suivante : si le contexte B doit être désactivé, le contexte A devra également être désactivé. Par exemple, le contexte `Bruxelles` est fortement inclus avec le contexte `Belgique` [25]. Ici, il est vrai que lorsque le contexte `Belgique` est désactivé, le contexte `Bruxelles` doit l'être aussi.

Cependant, il s'est avéré que, après la publication de l'article sur le *Subjective-C*, il pouvait y avoir des inconsistances avec certaines de ces dépendances, comme l'a montré Nicolás Álvarez Cardozo lorsqu'il a essayé de les formaliser [30]. Les problèmes de consistances étaient liés à la gestion des compteurs des contextes. Pour démontrer les inconsistantes dans les applications conscientes du système, il s'est basé sur le formalisme des petri nets [1], qu'il appelera *Context Petri nets*.

Dans ce *context petri nets*, il y a deux types de places : les places représentant les contextes et les places temporaires permettant de "préparer" l'activation/la désactivation, c'est-à-dire passer dans un état actif afin de vérifier si l'activation pourra avoir lieu définitivement. Deux types de transitions sont également présents : les transitions internes, représentant les (dés)activations des contextes, et les transitions externes, représentant les demandes d'activation/de désactivation des contextes. Les tokens représentent le nombre de fois que les contextes sont activés.

Pour résoudre les problèmes d'inconsistances liés aux compteurs, Nicolás Álvarez Cardozo a changé la sémantique des compteurs pour la *strong inclusion*. En prenant les deux contextes A et B, si le compteur du contexte de B tombe à zéro, celui de A devra aussi tomber à zéro, même si le compteur du contexte A est supérieur à 1 de par les autres dépendances.

Afin de ne pas confondre les diverses versions des dépendances et d'améliorer la compréhension, il a également changé les noms de deux dépendances : la dépendance *weak inclusion* sera renommée en une dépendance *causality*, tandis que la *strong inclusion* deviendra une dépendance de *implication*.

La sémantique de chaque dépendance sera donnée dans le chapitre 6 sur l'implémentation.

1. Un petri net est un formalisme se basant sur des places et des transitions afin de faire passer des tokens d'une place à une autre, grâce à des transitions. Un petri net a pour but de montrer l'état courant d'un système à chaque transition [30].

2.4 Les adaptations interfaces utilisateur

2.4.1 La plasticité

La plasticité, expliquée par Thevenin et Coutaz [5], est la capacité d'une interface utilisateur à s'adapter aux changements physiques et environnementaux. Cela signifie qu'une interface utilisateur peut s'adapter quel que soit le terminal à condition d'avoir prévu ce dernier. Les avantages sont de minimiser le développement ainsi que les coûts de maintenance, car il suffira d'un design et d'une implémentation pour gérer les diverses contraintes physiques, telles que la résolution de l'écran, ... Les auteurs présentent dans cet article un framework inspiré d'une approche d'ingénierie dirigée par des modèles.

Afin de mieux comprendre la plasticité, il faut commencer par comprendre le terme *adaptation*. Celui-ci est constitué de deux propriétés :

l'adaptabilité représentant la capacité d'un système à pouvoir être personnalisable, comme par exemple, sur base des préférences de l'utilisateur ;

et l'adaptativité désignant la capacité d'un système à s'adapter automatiquement, sans aide de l'utilisateur, comme par exemple, lorsque la batterie devient faible.

Pour gérer la plasticité, leur framework se base sur des générateurs d'interfaces utilisateur pour y ajouter des valeurs. Représentée dans la figure 2.2, il se divise en sept composantes. Le *user task model* constituera l'ensemble des tâches de l'utilisateur. Il décrira comment les tâches pourraient être achevées avec son introduction dans le système grâce au *system task model*. L'*abstract user interface* servira de vision abstraite pour le *rendering* du *system task model*. L' *interactors model* jouera le rôle de *listeners* car il produira des évènements et traitera des évènements de l'utilisateur ou du système. Un exemple d'*interactor* serait les widgets. Le *platform model* représentera les aspects physiques de chaque plate-forme, tandis que l'*environment model* spécifiera les contextes d'utilisation. Enfin, la *physical user interface* résultera de ces quatre derniers composantes (l'*abstract user interface*, l' *interactors model*, le *platform model* et l'*environment model*).

Souvent, pour gérer des adaptations au niveau des interfaces utilisateur, les chercheurs se dirigeaient dans une approche d'ingénierie dirigée par des modèles [5]. Cependant, cette méthode n'est pas la meilleure dans mon approche, car ces chercheurs se focalisaient seulement sur les interfaces utilisateur dans le domaine des systèmes conscients du contexte. Pour ce travail, je devais avoir une vision plus large car je devais également penser à gérer les autres aspects tels que les aspects comportementaux et données. Mon approche sera exposée dans les chapitres 5 et 6.

2.4.2 Les transitions

L'utilisateur peut être déstabilisé lorsque son application change son interface utilisateur sans que ce soit une action venant de lui. Ce phénomène est appelé la déstabilisation cognitive [1]. Pour lui, c'est comme s'il perdait le contrôle de l'application, en plus d'essayer de trouver les liens

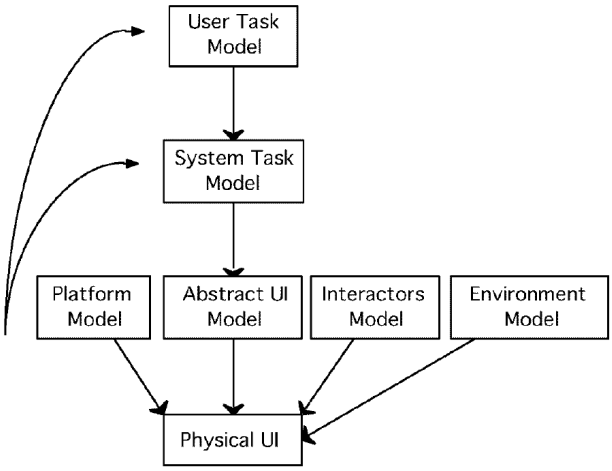


FIGURE 2.2: Les composantes gérant la plasticité. Figure extraite de l'article [5]

entre l'ancienne et la nouvelle interface. Pour pallier à ce problème, chaque développeur devrait inclure des transitions afin de montrer les changements à l'utilisateur, dans le but de l'accompagner dans cette adaptation. En ajoutant des transitions, l'utilisateur discernera l'adaptation et acceptera plus facilement ce changement.

Pour López-Jaquero, Vanderdonckt, Montero et González [19] et Dessart, Motti et Vanderdonckt [24], les transitions animées ont pour but d'informer l'utilisateur lors d'une adaptation afin d'attirer son attention sur ce changement. Par exemple, dans le cas où l'utilisateur changeait de localité et qu'une composante devait dorénavant afficher le nombre de kilomètres en miles, on pourrait utiliser une animation de type *Cartoons-inspired visual effects* afin de montrer visuellement que le changement de localisation a provoqué une adaptation sur son écran de bord. Un exemple de ce type d'animation (*Cartoons-inspired visual effects*) pourrait être de grandir et rétrécir la composante en explicitant ces évolutions. Il est vrai, qu'outre les *fade in/out*, les *swipe*, ..., les effets visuels inspirés des *Cartoons* peuvent être par moment plus réalistes [3]. Comme le précise l'article [24], l'utilisation des transitions animées attirera toujours plus l'attention d'un utilisateur que de simples images mises à jour.

De plus, les transitions doivent être équilibrées correctement, c'est-à-dire pas trop longues, histoire d'éviter des latences, mais pas trop courtes, sinon l'utilisateur n'aura pas le temps d'être attiré par cette transition. Dans l'article [24], ils précisent que les transitions peuvent aller de 300 milli-secondes à quelques secondes, en fonction de la complexité de la transition.

Les transitions doivent être différentes en fonction des adaptations et de ce que l'application souhaite mettre en évidence. De plus, certaines transitions sont plus adaptées que d'autres en fonction de la circonstance. Par exemple, une transition d'iris qui s'ouvre afficherait plus de détails concernant l'élément sélectionné, ou encore un déplacement horizontal de la droite afficherait les éléments suivants.

2.5 Feature modeling

Le *feature modeling* est un concept clé dans l'ingénierie des lignes de produits [17]. Il se constitue entre autres d'un diagramme de features [6], qui est composé de deux ensembles d'éléments : un ensemble de nœuds, représentant les sous-features et un ensemble de liens entre ces derniers. Ces liens peuvent être de plusieurs types et sont répertoriés dans le livre de Czarnecki et Eisenecker [6], ainsi que dans l'article de Van Gurp, Bosch et Svahnberg [8].

Un de ces types est la *variant feature*. Il implique qu'il a au moins un enfant direct et que ce dernier spécifie le comportement de la feature plus générique. Par exemple, si une feature permet d'écrire un email, la *variant feature* permet d'avoir un éditeur plus perfectionné pour envoyer un email [8]. Ce type de feature sera utilisé dans ce travail afin de pouvoir créer des features plus spécifiques, qui éviteront les duplications de code.

Hartmann et Trew [17] créeront une nouvelle modélisation pour les lignes de produits logiciels. Cette modélisation, figurant en 2.3, utilisera deux branches : une pour le *context variability model* et l'autre pour le *feature modeling*. L'utilisation du *context variability model* modélisera plusieurs lignes de produits en supportant plusieurs dimensions dans l'espace de contexte. Ce modèle se compose de classificateurs de contextes dans lesquels le produit pourra être utilisé. Un exemple d'un classificateur pourrait être une région géographique. Par conséquent, si des features doivent exister en Europe, toutes les features devront être activées si le produit est destiné à l'Europe.

Afin de pouvoir lier ces deux branches, ces auteurs proposeront des dépendances entre la branche du *context variability model* et celle du *feature model*. Ces dépendances seront également ajoutées à notre architecture (cf. le chapitre 5) afin de garder l'idée générale de Hartmann et Trew [17].

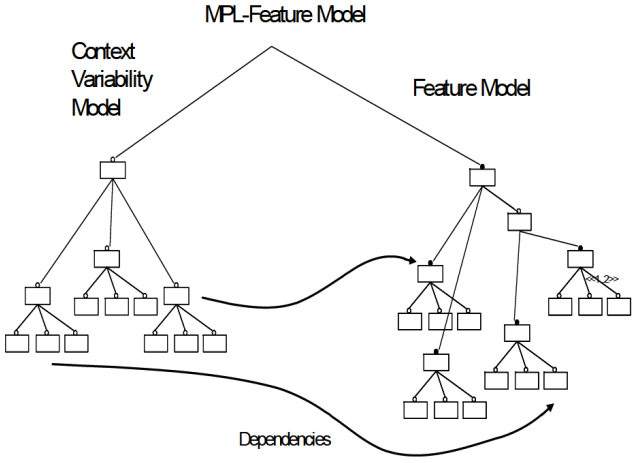


FIGURE 2.3: La modélisation d'Hartmann et Trew. Figure extraite de l'article [17]

Chapitre 3

L'étude de cas

L'étude de cas sera expliquée dans ce chapitre afin de mieux comprendre l'intuition qui se cache derrière les applications orientées contexte.
Tout au long de ce travail, les exemples seront basés sur cette étude de cas.

Celle-ci est une application pour un *emergency response system*. Ce système aidera les victimes d'une catastrophe naturelle, à savoir les tremblements de terre et les inondations, à trouver leur chemin pour rallier le point de sécurité, grâce à une carte ou aux instructions de route.

Premièrement, j'exposerai les fonctionnalités de l'application. Ensuite, je schématiserai les graphes des contextes et des features. Ces concepts seront vus dans le chapitre 6, ainsi que des bribes de l'implémentation de cette application. Enfin, je montrerai un scénario d'une telle application sensible à son environnement.

3.1 Les fonctionnalités

Les fonctionnalités sont peu nombreuses, car ce qui est intéressant dans cette étude de cas, c'est d'observer si, lors d'un changement d'environnement, son comportement s'adapte correctement. Ces fonctionnalités sont décrites comme suit :

— afficher un message par défaut. Ce dernier précise les instructions générales à suivre lorsqu'une victime vit une catastrophe naturelle (un tremblement de terre ou une inondation), et qu'elle n'a pas encore été détectée ;

— spécialiser les instructions par défaut, lorsqu'une catastrophe naturelle est détectée, c'est-à-dire, que le message par défaut sera adapté en fonction du désastre ;

— montrer sur une carte navigable le chemin à suivre pour rejoindre un point sécurisé ;

— indiquer les instructions de route pour atteindre le lieu sécurisé ;

— traduire l'application en fonction de la localisation.

La carte navigable sera affichée lorsqu'il y aura au moins une connexion réseau (Wi-Fi ou 3G) et si le niveau de batterie est acceptable (> 20%). Si le niveau de batterie est trop faible, les instructions de l'itinéraire remplaceront la carte.

3.2 Les contextes et features

Sur base des fonctionnalités décrites dans la section 3.1, les graphes des contextes (à savoir l'ensemble des situations dans lesquelles l'application peut s'exécuter) et des features (autrement dit l'ensemble des adaptations (cf. la section 2.3.1)) peuvent être construits de plusieurs manières. Une d'elles se trouve dans cette section.

Pour les diagrammes suivants, les ellipses blanches représentent les entités abstraites, tandis que les bleues symbolisent les non-abstraites. Les flèches pleines sont les liens de parenté, et les flèches pointillées sont les dépendances, excepté la dépendance *xor* qui se trouve sur les liens de parenté.

3.2.1 Le graphe des contextes

La figure 3.1 représente le graphe des contextes de l'application. Pour améliorer sa lisibilité, certains éléments, comme les conditions et les origines (à savoir d'où doivent provenir les données), ont été retirés.

Les contextes `English` et `French` sont abstraits car ces deux-là n'impliqueront pas l'ajout d'une fonctionnalité. En effet, la spécification pour la traduction sera toujours présente dans l'application. Même si elle impose une traduction, elle ne sera pas considérée comme une fonctionnalité pouvant changer l'architecture de l'application.

De plus, il est possible de définir une langue par défaut grâce à la dépendance *causality*. Cette dernière est expliquée dans la section 2.3.2 et sa sémantique sera définie dans la section 6.1.1. Pour cette étude de cas, la langue par défaut est l'anglais.

3.2.2 Le graphe des features

La figure 3.2 expose le graphe des features de l'application. Ce graphe reprend l'ensemble des fonctionnalités, excepté celle de la traduction pour la raison expliquée précédemment.

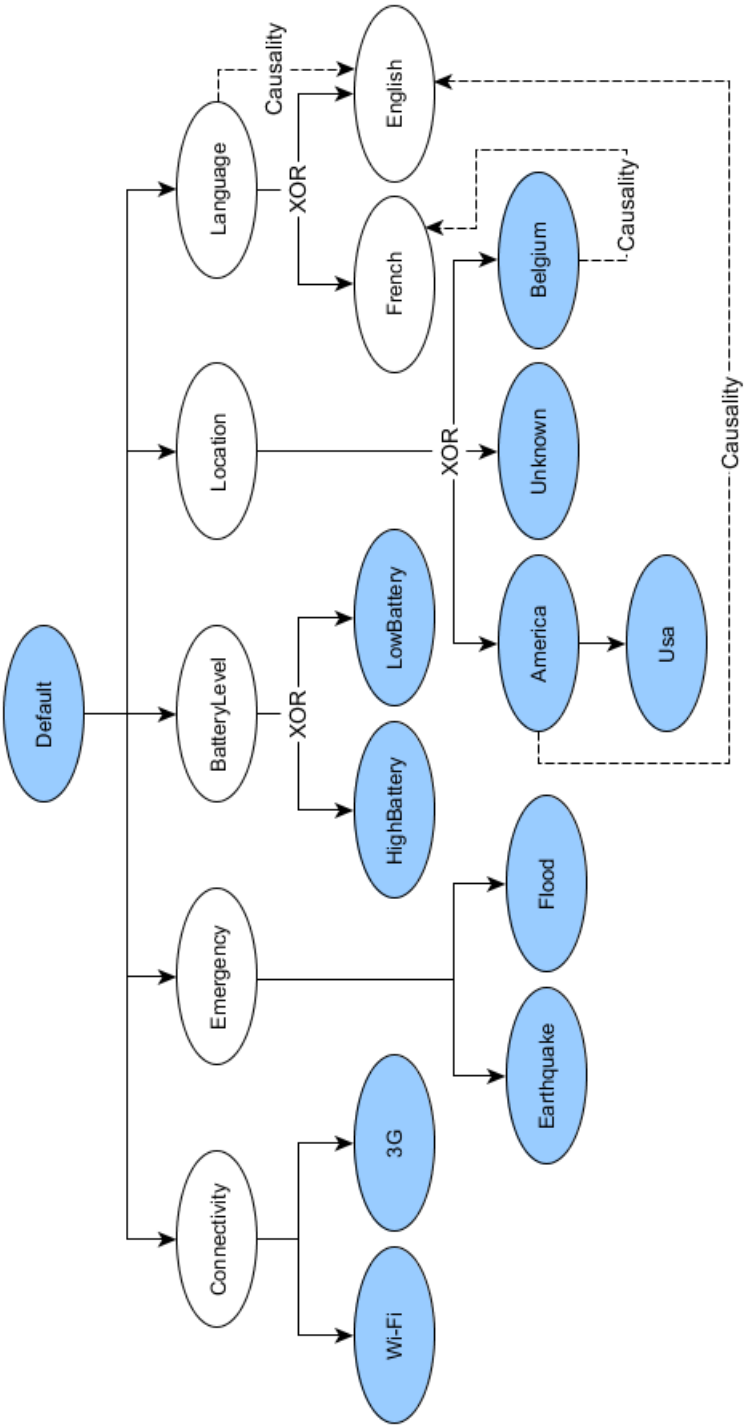


FIGURE 3.1: Le graphe des contextes pour l'étude de cas

3.3 Le scénario

La figure 3.3 montre l'exécution du scénario expliqué ci-dessous. Lorsqu'une nouvelle feature est ajoutée, elle sera associée à une transition de type *fade in*, c'est-à-dire que chaque nouvelle feature apparaîtra de manière progressive.

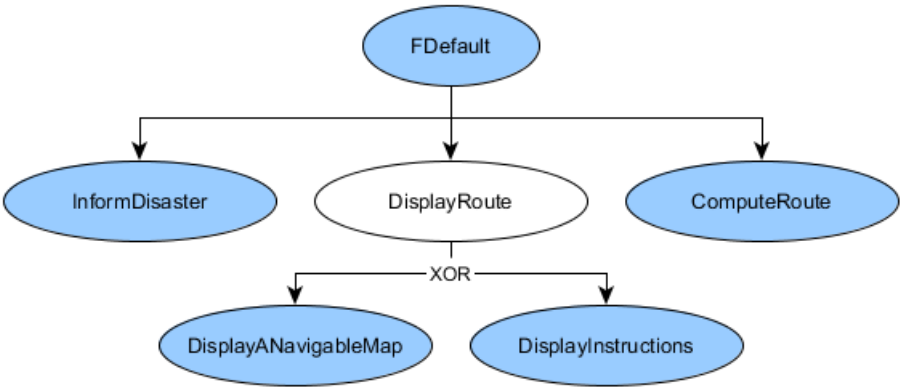


FIGURE 3.2: Le graphe des features pour l'étude de cas

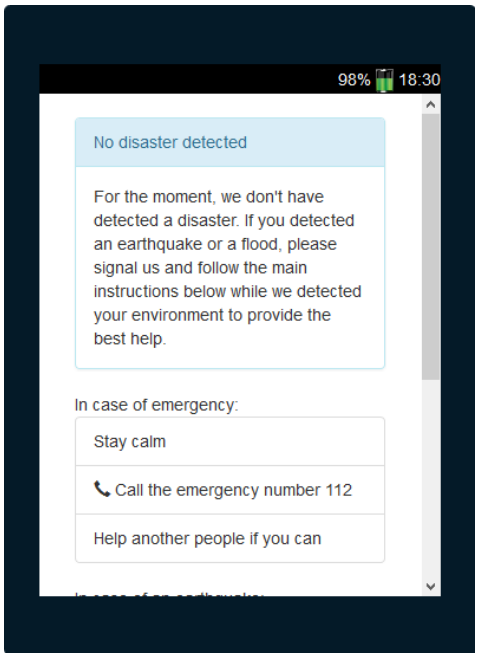
Le scénario est le suivant :

(A) au lancement de l'application, comme aucun environnement n'est connu, le contexte par défaut est activé. Cela entraînera la sélection du comportement par défaut (cf. la figure 3.3a) ;

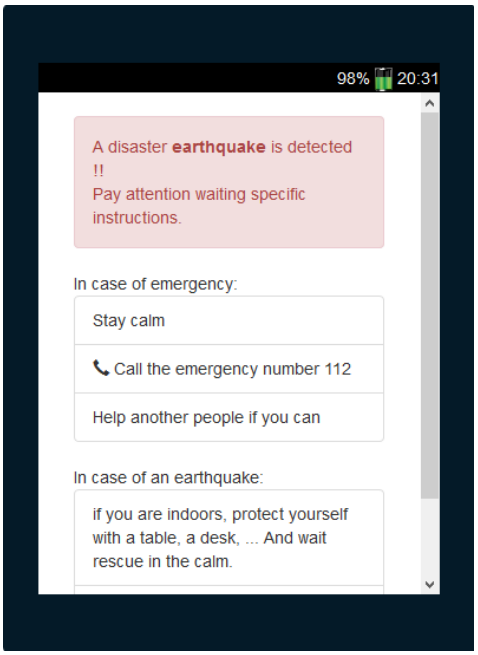
(B) peu de temps après, par la réception d'un signal externe, un tremblement de terre est détecté. Afin de prévenir la victime, il affichera qu'un phénomène sismique est découvert, ainsi que les instructions à suivre dans ce cas-là (cf. la figure 3.3b) ;

(C) après détection de ce tremblement de terre, un signal GPS est reçu pour indiquer sa position (New-York), ainsi qu'un signal de connexion 3G. Une fois ces deux environnements repérés, l'application affichera une carte navigable avec le chemin vers le lieu sécurisé (cf. la figure 3.3c) ;

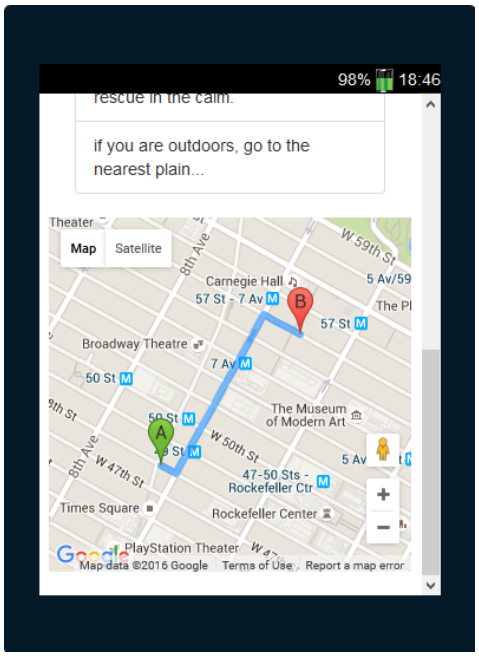
(D) enfin, après vingt minutes, la batterie étant faible (19%), l'application remplacera la carte navigable par la suite d'instructions indiquant l'itinéraire à suivre (cf. la figure 3.3d).



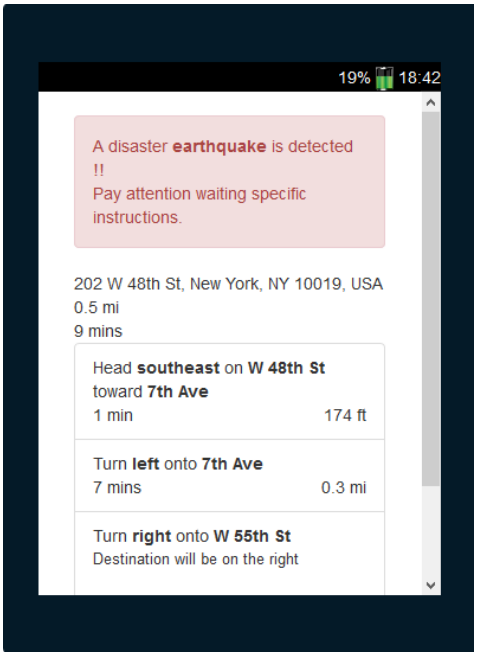
(A) Comportement par défaut



(B) Lors de la détection d'un désastre



(C) Lors de la détection de la position et de la 3G



(D) Lors de la détection du niveau faible de batterie

FIGURE 3.3: Résultats de l'application *emergency response system*

Chapitre 4

L'approche

Ce chapitre exposera la motivation de créer un tel outil. Par la suite, il expliquera succinctement la solution proposée, ainsi que les technologies et outils utilisés lors de ce travail.

4.1 La motivation

Souvent, lors de développements d'applications en orientée objet, leurs architectures sont divisées en couches. Ces couches suivent les architectures *MVC* (Modèle-Vue-Contrôleur) comme le montre la figure 4.1.

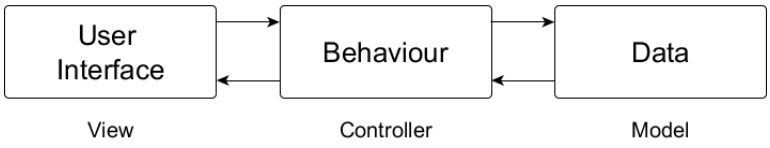


FIGURE 4.1: L'architecture modèle-vue-contrôleur

L'avantage de ce type d'architecture est le découplage. Par exemple, la couche *View* ne communiquera qu'avec la couche *Controller*. Si elle a besoin de communiquer avec la couche *Data*, elle passera par la couche *Controller*, qui servira d'intermédiaire.

Cependant, si on souhaite ajouter la gestion de contextes dans de telles architectures, cela devient difficile, car il faudrait ajouter toutes des conditions *IF/ELSE* ou des *SWITCH/CASE* dans toutes les couches. Un exemple pour la couche *View* se trouve dans le code 4.1, où ce dernier affichera une carte GPS navigable si une connectivité est existante, sinon elle exposera une carte fixe, et selon le niveau de la batterie, elle affichera cette carte ou indiquera les instructions de route. Cet exemple est simple mais montre déjà le nombre de conditionnelles qu'il faut pour l'implémenter. Du coup, pour gérer une application complète, cette solution deviendrait très vite ingérable à la création, mais aussi à la maintenance de l'application.

```
1 if ['WiFi', '3G'].include? (connectivity)
2   if battery_level < 20
3     # Display instructions
4   else
5     # Display a map
6   end
```

```
else
  if battery_level < 20
    # Display instructions
  else
    # Display a fixed map
  end
end
```

CODE 4.1: Exemple de code pour une application orientée contexte avec des conditionnelles

Pour résoudre ces problèmes de conditionnelles, on pourrait utiliser les technologies développées dans le chapitre 2. Néanmoins, cette solution impliquerait d'utiliser, pour chaque couche, des technologies différentes. Cela complexifierait nettement le développement et la maintenance.

Par conséquent, ce mémoire a pour objectif d'apporter une solution acceptable à ce type de développement, à savoir, une solution qui apportera une aisance dans le développement complet, c'est-à-dire avec les trois aspects (interfaces utilisateur, comportemental et données), d'une application orientée contexte.

4.2 La solution

La solution est de créer un framework unifié, autrement dit qu'il gère les contextes et les features d'une manière centralisée pour tous les aspects. Ce framework découvrira, sélectionnera et activera les contextes. En fonction de ces activations, il créera/modifiera l'application en y ajoutant les features correspondantes pour chaque couche. La figure 4.2 montre l'intuition de cette solution.

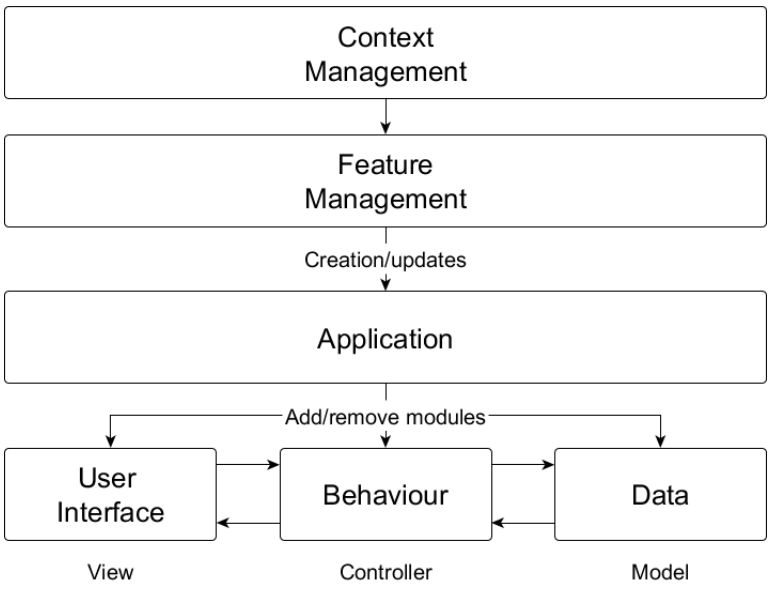


FIGURE 4.2: L'intuition du framework

Pour réaliser cette solution, mes contributions sont les suivantes :

le framework qui servira d'aide au développement d'applications orientées contexte (cf. les chapitres 5 et 6) ;

l'intégration des adaptations des interfaces utilisateur qui intégrera les concepts des interfaces utilisateur, entre autres les transitions, dans l'application créée par le framework (cf. les chapitres 5 et 6) ;

le simulateur qui permettra de tester et valider le framework par des applications orientées contexte, ainsi que leurs créations (cf. le chapitre 7).

En plus de ces contributions, une petite application orientée contexte *web-based* sera développée dans le but d'évaluer mes contributions. Elle a déjà été présentée dans le chapitre 3, tandis que son implémentation se retrouvera dans les chapitres 6 et 7. Une discussion sur ces limitations aura lieu dans le chapitre 8.

4.3 Les technologies utilisées

4.3.1 Les technologies du framework

Le framework a été implémenté en *Ruby*. Une des principales raisons d'utiliser ce langage est sa couche de méta-programmation, entre autres pour sa partie intercession [1]. La seconde raison est la faisabilité de développer un framework/langage pour la programmation orientée contexte avec *Ruby*. En effet, *Phenomenal Gem* [28] a été implémenté avec ce dernier. Cependant, il n'a été conçu que pour l'aspect comportemental, tandis que ma solution est conçue pour gérer tous les aspects (interfaces utilisateur, comportemental et données). De plus, deux autres points positifs sont la documentation complète et la communauté derrière ce langage.

Pour créer les fichiers de configurations, c'est-à-dire les fichiers de déclarations de contextes, ..., j'ai décidé d'utiliser le *JSON* par rapport au *XML* grâce à la légèreté des fichiers *JSON* par rapport aux fichiers *XML* [21].

4.3.2 Les technologies du simulateur

Le simulateur étant un site web, j'ai dû choisir quel langage et quel framework utiliser. Comme je venais d'apprendre le *Ruby* pour ce mémoire, j'ai décidé d'utiliser le même langage. Pour le framework web, j'ai pris le *Ruby on Rails*. Ce dernier est le framework le plus connu pour développer des applications web en *Ruby*. De plus, une documentation exhaustive et une communauté active sont présentes sur Internet.

Afin de faire les liaisons entre le site web, le simulateur, et le framework, j'ai décidé d'utiliser l'*AJAX* avec *JQuery*, qui est un framework *JavaScript* très connu. Le choix s'est porté automatiquement sur l'*AJAX*, car c'est la technologie utilisée s'il suffit de changer une partie de la page sans devoir recharger la page entière.

1. L'intercession est la capacité d'un programme à se modifier lors de son exécution.

Pour son design graphique, j'ai utilisé le framework *CSS Twitter Bootstrap*, tout simplement car je le connaissais au préalable.

4.4 Les autres outils utilisés

Afin de développer le framework et le simulateur, j'ai utilisé : *Sublime Text*. Ce dernier est un éditeur de texte plus performant qu'un simple bloc-notes car il permet, entre autres, la complétion de code.

En ce qui concerne la gestion des versions du code, j'ai utilisé *Git*. Mon code est hébergé sur *Bitbucket*.

Pour les schémas se trouvant dans ce mémoire, j'ai utilisé *yEd Graph Editor*. Cet outil permet de schématiser plusieurs types de diagrammes, dont les diagrammes *UML*.

Enfin, l'application *Trello* a été utilisée pour planifier mes tâches.

Chapitre 5

L'architecture

Dans ce chapitre, l'architecture sera expliquée en deux temps. Dans un premier temps, une vision globale de l'architecture sera exposée afin de mieux appréhender les diverses couches. Ensuite, chaque couche sera vue plus en détail.

Les schémas suivants se baseront sur les notations suivantes : les couches seront représentées par des rectangles blancs, les regroupements conceptuels de composantes par des rectangles gris et les composantes, donc les briques de base implémentant l'architecture, par des rectangles bleus. Les flux de données seront représentés par des flèches discontinues, tandis que les flux de contrôle seront interprétés par des flèches continues.

5.1 Vue générale

La figure 5.1 présente une vue générale de l'architecture.

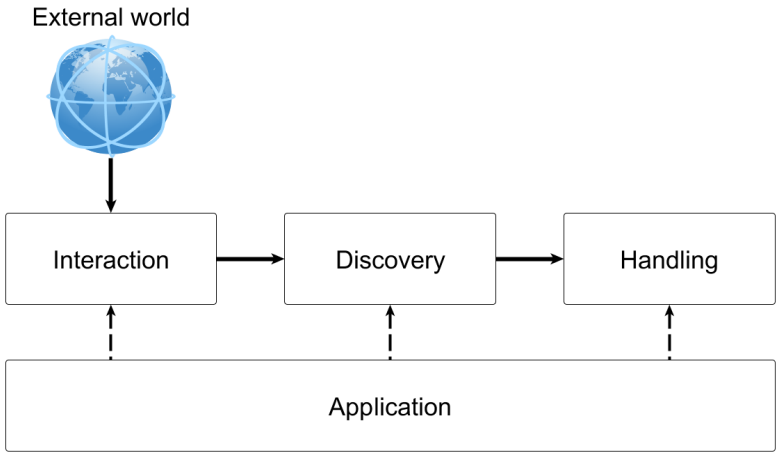


FIGURE 5.1: La vue globale de l'architecture

Deux parties constituent notre architecture :

la partie framework contenant tout le comportement interne des applications orientées contexte, c'est-à-dire les couches INTERACTION, DISCOVERY et HANDLING ;

la partie application fournissant toutes les données propres à l'application (la couche APPLICATION), c'est-à-dire, les différentes déclarations de celle-ci, ses features, ses transitions, ...

La couche INTERACTION sera l'intermédiaire entre le monde externe, autrement dit l'environnement physique et l'utilisateur, et l'application. L'utilisateur et les senseurs généreront des données contextuelles, à savoir des données qui permettront de découvrir le nouveau contexte pour l'application en cours. Elles seront interceptées par des *listeners* de la couche APPLICATION (cf. la section 5.5) et transférées à la couche DISCOVERY.

Cette dernière traduira en objets les données contextuelles perçues dans la couche précédente et les filtrera. Ceux-ci seront exploités afin d'en retirer les contextes caractérisant l'environnement courant, en se basant sur les déclarations des contextes dans la couche APPLICATION.

Lorsque les contextes auront été sélectionnés, la couche HANDLING activera les contextes à l'aide de la couche APPLICATION. Après leurs activations, la couche HANDLING sélectionnera, activera et exécutera les features de l'application grâce à la couche APPLICATION.

5.2 La couche INTERACTION

La couche INTERACTION, comme montré dans la figure 5.2, liera le monde externe à l'application.

Ce monde externe, générant des données contextuelles, est constitué de plusieurs éléments :

— l'utilisateur, par ces tâches, comme par exemple la tâche lui permettant d'envoyer des messages signalant un problème lors d'un tremblement de terre, ...

— l'environnement physique, détecté par les différents senseurs, tels que des capteurs de localisation, de température, de lumière, ...

— la plate-forme hardware et software du smartphone (*Computing platform*), comme l'état de la batterie, de la mémoire.

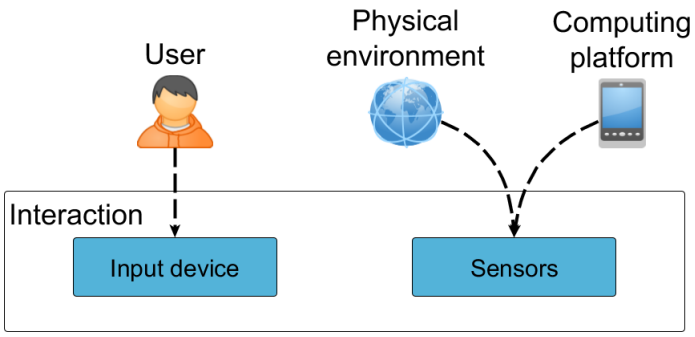


FIGURE 5.2: La couche INTERACTION dans l'architecture

5.3 La couche DISCOVERY

La couche DISCOVERY interprétera et raisonnera sur les données interceptées, afin de trouver les contextes caractérisant les nouveaux environnements détectés.

Comme le montre la figure 5.3, cette couche comporte deux composantes :

Composante *Interpretation* Elle créera les objets contenant ces données et filtrera les données reçues. Ces objets modéliseront les données contextuelles. Par exemple, pour des données venant d'un capteur de localisation, cette composante créera un objet *Location* avec les données perçues, à savoir les coordonnées géographiques de la position où se trouve l'utilisateur.

Composante *Reasoning* Celle-ci recevra les objets créés et recherchera les contextes admissibles. Un contexte admissible est un contexte qui caractérise un élément du monde extérieur mais qui n'est pas encore activé dans l'application. Par exemple, sur base du graphe des contextes de la section 3.2.1, si l'utilisateur se trouve à Bruxelles, les contextes admissibles seront `Location` et `Belgium`[1]. L'activation de ces ceux-ci sera traitée dans la section suivante 5.4.

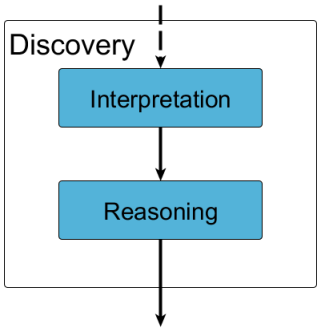


FIGURE 5.3: La couche DISCOVERY dans l'architecture

5.4 La couche HANDLING

La couche HANDLING vérifiera les dépendances sur les contextes, les activera en fonction de ces dernières, sélectionnera et activera les features pour finalement les exécuter.

Cette couche est subdivisée en deux grands sous-modules. Le premier module achèvera le traitement des contextes, tandis que le deuxième traitera entièrement les features, comme l'expose la figure 5.4.

1. Le contexte `Default` ne sera plus sélectionné, car c'est un cas spécifique lorsqu'aucun contexte n'est détecté.

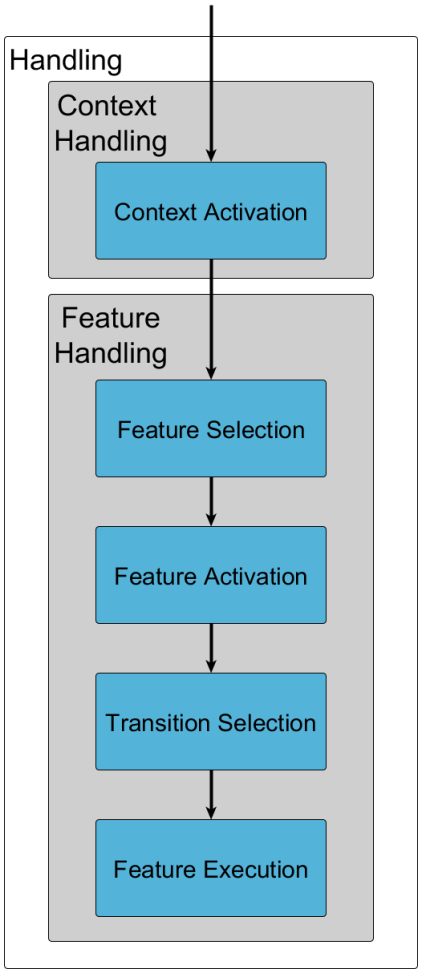


FIGURE 5.4: La couche HANDLING dans l'architecture

Context Handling

Sur base des contextes admissibles, la composante *Context Activation* vérifiera quel contexte devra être activé. Lors de cette vérification, il est possible que certaines dépendances imposent l'activation/la désactivation d'autres contextes. Un bon exemple se baserait sur les contextes de localisation qui incluent faiblement le choix de la langue utilisée dans l'application. En reprenant le graphe des contextes de la section 3.2.1, l'activation du contexte `Belgium` forcerait l'activation du contexte `French`.

Feature Handling

En fonction des contextes activés, la composante *Feature Selection* sélectionnera les features correspondantes. Par exemple, lorsque le contexte `HighBattery` sera activé, cette composante prendra la feature qui affichera les données sur une carte GPS plutôt que de les communiquer sous format texte (cf. l'étude de cas dans le chapitre 3).

Comme pour les contextes, les features devront être (dés)activées, grâce à la composante *Feature Activation*. Cependant, cette dernière devra également prendre en compte les dépendances entre les contextes et les features. Ce concept est inspiré du travail de Hartmann et Trew [17] comme expliqué dans la section 2.5.

Après l'activation/la désactivation des features, la composante *Transition Selection* sélectionnera les transitions correspondantes. En fonction de la feature à activer, les transitions seront différentes, afin de permettre à l'utilisateur de percevoir le changement d'une feature (cf. la section 2.4.2). Par exemple, si la carte GPS doit disparaître pour des raisons de batterie faible (cf. la chapitre 3), le développeur devra utiliser une transition de type *fade out* pour bien montrer que l'élément est supprimé.

Enfin, après les sélections et les (dés)activations des features, la composante *Feature Execution* modifiera le comportement de l'application à l'exécution en y ajoutant les nouvelles features, ainsi qu'en supprimant les features devenues inutiles.

5.5 La couche APPLICATION

Finalement, la couche APPLICATION servira entièrement au développeur. Il y définira tout le contenu d'une application orientée contexte.

De manière à visualiser plus facilement les diverses interactions entre la couche APPLICATION et les autres couches, la figure 5.5 reprend l'architecture complète.

Dans la suite de cette section, j'exposerai premièrement les différentes composantes de la couche APPLICATION succinctement, pour ensuite expliquer ses interactions avec les autres couches (INTERACTION, DISCOVERY). Enfin, je développerai les composantes de la couche APPLICATION et ses interactions avec la couche HANDLING, composante par composante, basée sur cette dernière couche afin d'améliorer sa lecture. Chacune d'elles sera reprise dans le chapitre 6 afin de montrer comment elles ont été implémentées.

5.5.1 L'interaction entre la couche APPLICATION et la couche INTERACTION

Comme le montre la figure 5.6, deux composantes sont présentes dans la couche APPLICATION :

Sensor listeners Ils récupéreront les données contextuelles générées par les capteurs.

Input listeners Ils écouteront l'input de l'utilisateur. Deux types d'input existent pour l'utilisateur : l'input exécutant une nouvelle tâche, à savoir que cet input sera considéré comme un nouvel environnement détecté, et l'input interne à une tâche, c'est-à-dire qu'il servira à compléter cette tâche. Ce dernier input fera partie intégrante de la couche APPLICATION. Par exemple, lorsque l'utilisateur souhaite envoyer un message, il exécutera une tâche pour activer la feature lui permettant

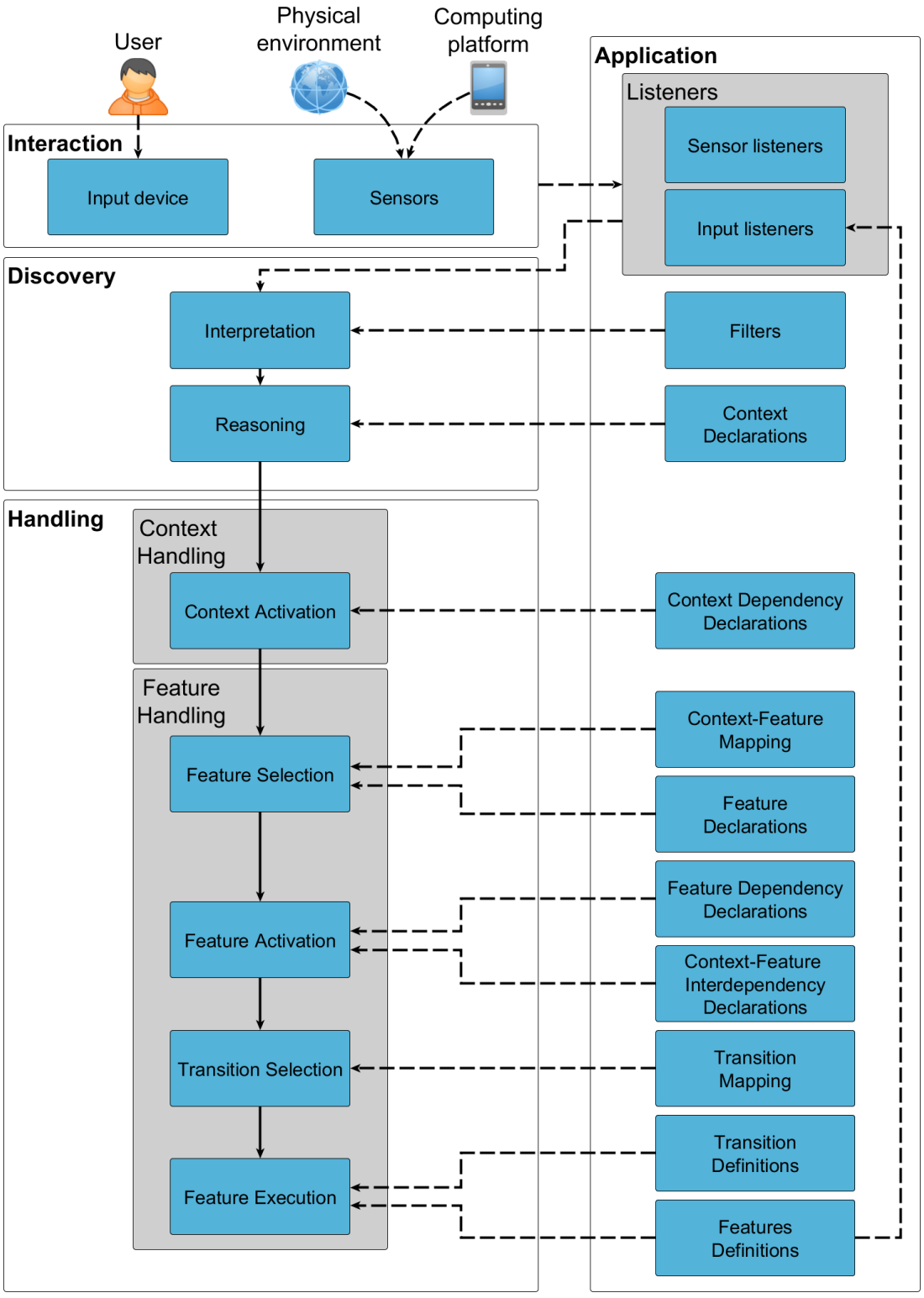


FIGURE 5.5: La vue complète de l'architecture

de créer et envoyer ce message. Cet input sera évalué comme un nouveau contexte. Tandis que l'input du formulaire, lors de l'envoi, sera envisagé comme des données réelles appartenant à cette feature.

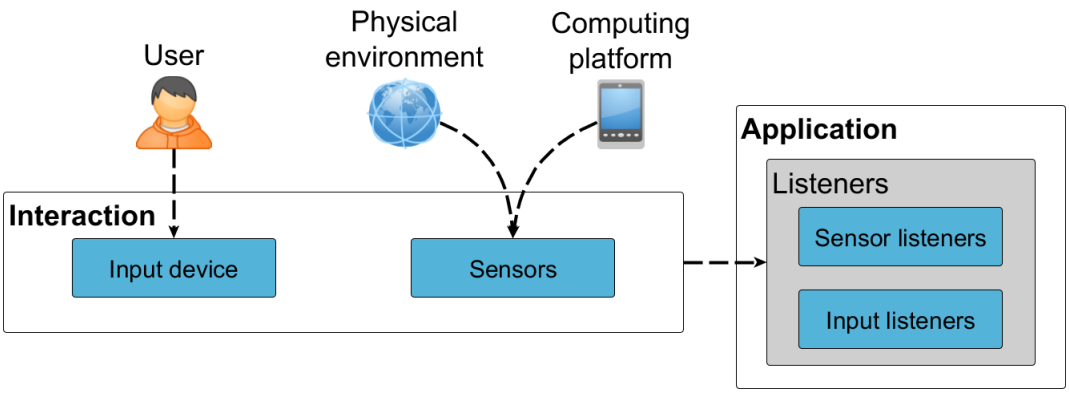


FIGURE 5.6: La couche INTERACTION dans l'architecture et ses interactions avec la couche APPLICATION

Ces données externes générées seront interceptées par des composantes *listeners* en fonction de leurs sources. C'est-à-dire que les interactions de l'utilisateur seront captées par l'*input listener*, tandis que celles des senseurs et du *Computing platform* par les *sensor listeners*.

5.5.2 L'interaction entre la couche APPLICATION et la couche DISCOVERY

En plus des *listeners*, la figure 5.7 montre deux nouvelles composantes dans la couche APPLICATION :

Filters Ils filtreront les données contextuelles reçues des *listeners*.

Context Declarations Ces déclarations contiendront le graphe des contextes. Le chapitre 6 donnera de plus amples informations sur celui-ci.

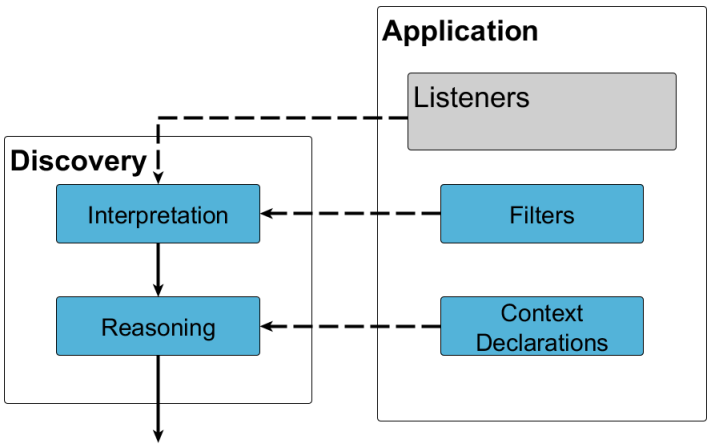


FIGURE 5.7: La couche DISCOVERY dans l'architecture et ses interactions avec la couche APPLICATION

Lors de la réception des données contextuelles, la composante *Interpretation* les parsera en objets (modélisant ces données) et les filtrera. Les filtres ont une part importante dans notre architecture. En effet, il se peut que, durant l'exécution de l'application, des senseurs reçoivent trop souvent des données contextuelles inutiles. Un exemple d'un tel capteur serait par exemple un senseur recensant le niveau de la batterie du smartphone. En admettant qu'il intercepte des données toutes les minutes, cela signifie que, pour chaque donnée interceptée, l'application les interprétera et les analysera. Cependant, en fonction de l'application, les traiter toutes les minutes ne serait pas toujours utile, et provoquerait des surcharges de traitements pour très peu de changement de contexte car ils sont définis de manière *coarse-grained* (`HighBattery` et `LowBattery`). Comme adapter le code de ces senseurs pour résoudre ce problème n'était pas possible, l'idée était de bloquer les interprétations par des filtres. Ainsi, le développeur pourra ajuster les filtres en fonction de son application. Dans ce cas, il faudrait un filtre sur le temps. Plusieurs types de filtres sont possibles, comme par exemple des filtres qui n'acceptent que les données satisfaisant une condition, ... Pour l'exemple du senseur récupérant le niveau de batterie, il faudrait créer un filtre de temps qui ne recenserait son niveau que toutes les 10 minutes.

Une fois les objets créés et filtrés, la composante *Reasoning* recherchera les contextes admissibles sur base du *Context Declarations*.

5.5.3 L'interaction entre la couche APPLICATION et la couche HANDLING

La figure 5.8 représente les diverses composantes de la couche APPLICATION utilisée par la couche HANDLING.

Les *Context Dependency Declarations* contiendront les dépendances entre les contextes. Ces dépendances peuvent être de type *xor*, *exclusion*, *requirement*, *implication* ou *causality*. Ces déclarations seront utilisées par la composante *Context Activation*, qui, sur base des contextes admissibles, vérifiera quels contextes devront être activés.

Une fois les contextes activés, la composante *Feature Selection* sélectionnera les features correspondantes grâce au *Context-Feature Mapping* et au *Feature Declarations*. Ce *Context-Feature Mapping* liera chaque contexte à une feature. Tandis que ce *Feature Declarations* contiendra le graphe des features (cf. la section 2.5 sur le feature modeling et le chapitre 6).

Comme pour les contextes, les features sélectionnées devront être activées. Elles le seront par la composante *Feature Activation* en fonction des dépendances définies dans les *Feature Dependency Declarations*. En plus de ces dernières, il peut y avoir des dépendances entre les contextes et les features grâce aux *Context-Feature Interdependency Declarations* (cf. Hartmann et Trew [17] dans la section 2.5). Elles aussi devront être vérifiées et activées par la même composante (*Feature Activation*).

Après l'activation/la désactivation des features, la composante *Transition Selection* sélectionnera les transitions correspondant aux features (dés)activées à l'aide du *Transition Mapping*. Ce mapping contiendra les

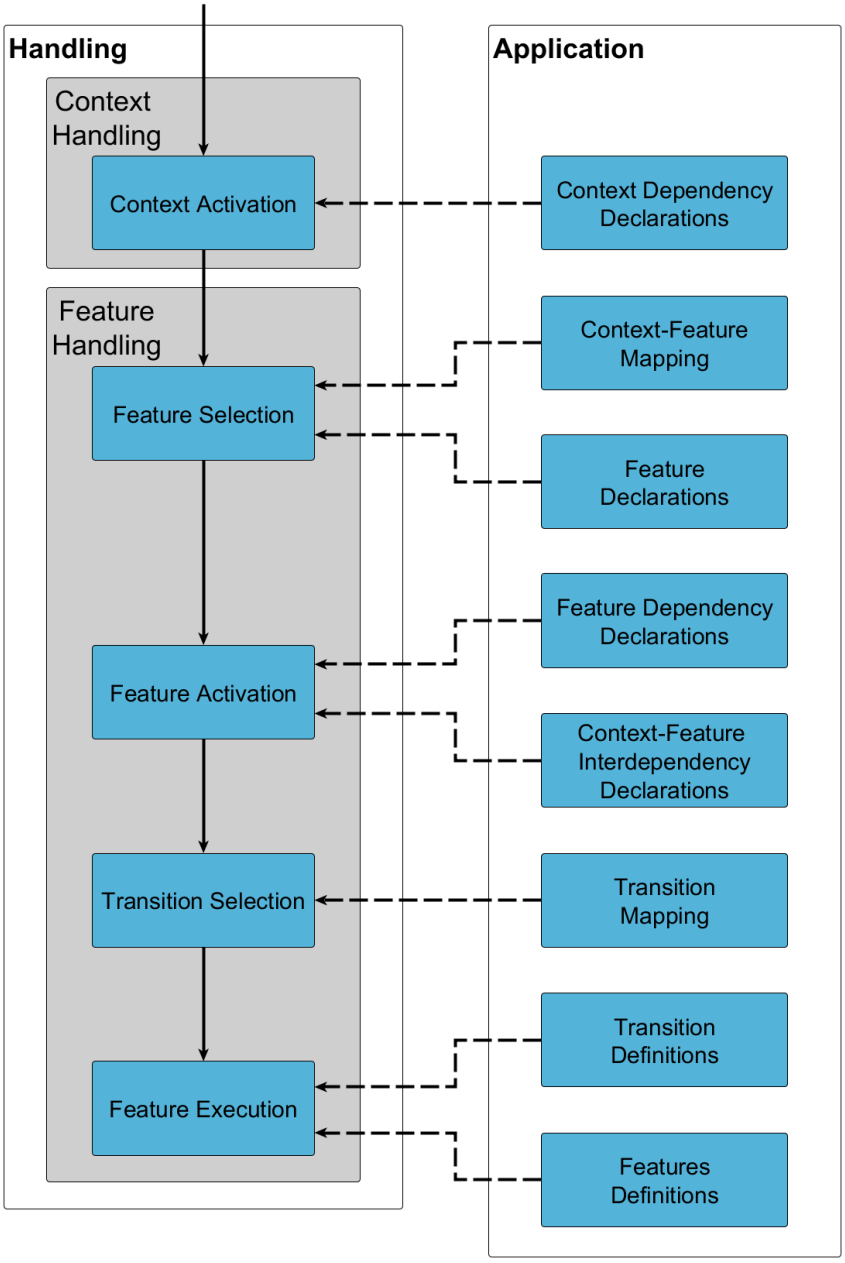


FIGURE 5.8: La couche HANDLING dans l'architecture et ses interactions avec la couche APPLICATION

transitions pour l'activation/la désactivation de chaque feature. Comme précisé dans la section 2.4.2, il est important que le développeur utilise des transitions afin de faire accepter les changements plus facilement à l'utilisateur.

Enfin, après toutes les sélections et les (dés)activations, la composante *Feature Execution* mettra à jour le code de l'application grâce aux *Transition Definitions* et aux *Feature Definitions*. Les *Transition Definitions* contiendront le code des transitions. Elles pourront être de type *swipe*, *fade in*, *fade out*,

Cartoons-inspired visual effects [3, 24] alors que les *Feature Definitions* contiendront le code de l'application.

5.6 En résumé

Pour résumer, l'architecture est divisée en quatre couches : INTERACTION, DISCOVERY, HANDLING et APPLICATION.

Les couches INTERACTION, DISCOVERY et HANDLING contiendront tout le comportement interne d'une application orientée contexte. Elles reprendront tout le traitement de la réception des données contextuelles à l'exécution des features correspondantes, en passant par la découverte des contextes de l'environnement courant. Ces trois couches joueront le rôle d'un framework applicatif [4] qui appellera les composantes de la couche APPLICATION au moment voulu.

Tandis que la dernière couche (APPLICATION) sera définie par le développeur pour décrire l'application.

Chapitre 6

L'implémentation

Dans ce chapitre, j'expliquerai premièrement la notion du graphe des contextes utilisé dans le framework. Ensuite, je parcourrai le flux de contrôle du framework où j'exposerai les manières dont ce dernier est implémenté. Enfin, je développerai l'architecture des applications créées par le framework, en m'axant sur les interfaces utilisateur. Tout au long de ce chapitre, je donnerai également des exemples de codes de l'application du chapitre 3 pour illustrer mes propos.

Mais avant de commencer à développer ce chapitre, je tiens à préciser que ce mémoire fait partie d'un sujet plus générique telle que la création d'un nouveau langage de développement pour la programmation orientée contexte. Comme dit précédemment, je me suis occupé de l'aspect des interfaces utilisateur, mais d'autres perspectives doivent être également prises en compte comme celle sur le comportement et celle des données. Un autre étudiant, David Sarkozi, a pris l'aspect comportemental.

Par conséquent, notre travail a été scindé en deux parties. Premièrement, nous avons dû collaborer pour créer ledit framework. Ensuite, nous devions intégrer notre aspect à ce framework.

Dans le framework, j'ai contribué à tout le processus interne, de la composante *Sensors* à l'activation des contextes, en passant par la création du graphe des contextes, ainsi que la composante *Transition Selection*. Afin de pouvoir intégrer l'aspect sur les interfaces utilisateur, j'ai également dû créer les autres composantes de manière simplifiée afin de pouvoir intégrer mes features propres aux interfaces utilisateur. La gestion des logs, implémentée par David, est la seule composante que j'utilise.

6.1 Le graphe des contextes

Dans notre architecture, deux types d'entités existent :

— les contextes,

— et les features, c'est-à-dire les adaptations.

L'ensemble des contextes (resp. features) forment un graphe des contextes (resp. features). Ces deux graphes étant assez similaires, je ne présenterai que le graphe des contextes.

Dans la suite de cette section, j'introduirai la notion de contexte, avec son arbre de conditions et ses dépendances. Et pour terminer cette section, je donnerai un exemple d'un graphe des contextes avec sa configuration.

La figure 6.1 montre une version simplifiée de la modélisation des objets utilisés dans le framework, où, entre autres, les différences entre un contexte et une feature peuvent être déceler.

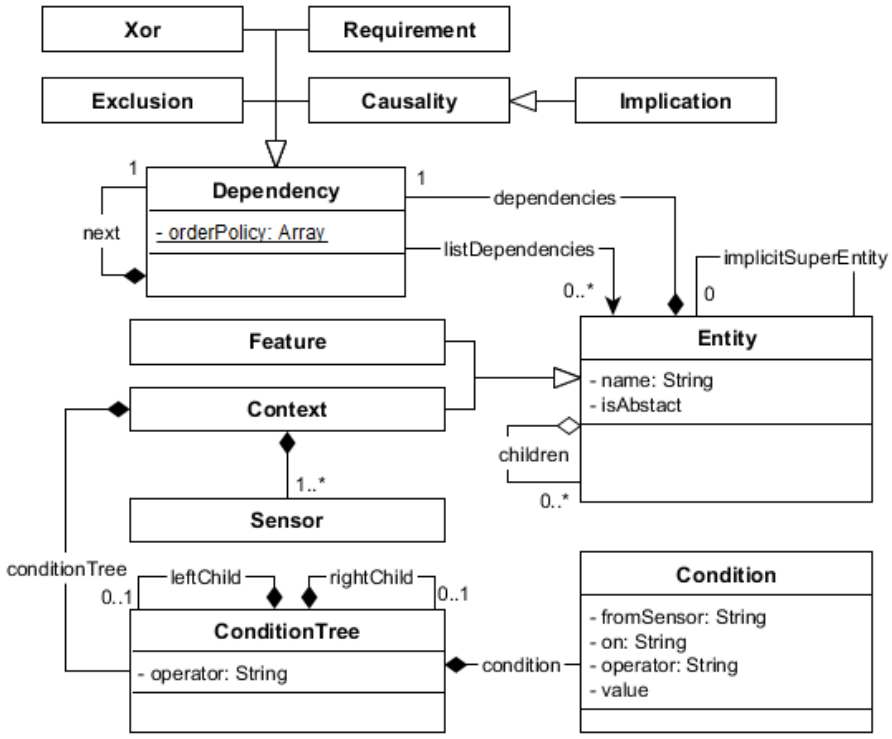


FIGURE 6.1: Le diagramme de classes d'une partie des objets

6.1.1 Un contexte

Un contexte est une entité représentant une situation actuelle dans laquelle l'application s'exécute. Son état, figurant en 6.1, est composé des attributs suivants :

un nom unique identifiant le contexte ;

un booléen *isAbstract* signifiant si un contexte est abstrait. S'il l'est, cela symbolisera qu'il n'impliquera pas de feature. Par exemple, un contexte abstrait serait `BatteryLevel`. En effet, ce dernier est utile dans la bonne conception du graphe des contextes, car il permet de mieux l'organiser. Mais il n'aura pas d'impact dans l'application, car savoir qu'il est activé sans connaître le niveau restant de la batterie n'a pas beaucoup de sens ;

une liste d'enfants étant des contextes plus précis ;

son parent implicite contenant une référence vers son parent ;

ses senseurs exposant l'origine des capteurs. Par exemple, un contexte *English* pourrait être détecté sur base de plusieurs sources, soit en fonction des préférences de l'utilisateur, soit sur base de la position de l'utilisateur ;

une condition devant être satisfaite pour que le contexte soit sélectionné ;

une liste de dépendances créant les dépendances entre lui et les autres contextes. Une dépendance est une relation entre des contextes, comme expliqué dans la section 2.3.1. Leur sémantique sera donnée plus loin.

Les conditions

Afin de pouvoir détecter une nouvelle situation, il faudra trouver les contextes la représentant. Pour cela, chacun d'eux aura une expression logique évaluant cette situation pour ainsi les découvrir. Par exemple, l'expression pour le contexte `LowBattery` serait la suivante :

$$(BatteryLevelSensor.level <= 20)$$

Afin de savoir l'évaluer facilement, cette condition sera modélisée sous la forme d'un arbre binaire (*ConditionTree*), car cette structure de données permet de traiter aisément la précédence des opérateurs logiques. Comme le montre la figure 6.1, l'arbre est représenté par la classe *ConditionTree*. Cette dernière contient plusieurs attributs dont les références vers ses deux enfants. Dans le cas où un arbre n'a aucun enfant, il sera considéré comme une feuille. Du coup, il aura un objet *Condition*. Cet objet représente l'attribut (*on*) sur lequel on évaluera la situation avec la valeur (*value*) sur base de l'opérateur (*operator*). Cependant, si au moins un enfant existe, l'arbre ne pourra contenir un tel objet *Condition*. Il devra avoir un opérateur logique *operator* (le *not*, le *and* ou encore le *or*). Ce dernier liera les parties de l'expression dont chaque enfant est responsable.

Les dépendances

Comme le présente la figure 6.1, plusieurs types de dépendances existent :

Xor La dépendance *xor* assure qu'un seul contexte enfant sera activé à la fois. Dès qu'un nouveau contexte enfant est activé, l'autre contexte enfant sera désactivé. Il peut être utilisé comme une sorte d'*exclusion* entre tous les contextes enfants, excepté que le *xor* impose la désactivation de l'ancien contexte.

Par exemple, pour un contexte général `BatteryLevel` et ses sous-contextes `HighBattery` et `LowBattery`, on sait que ceux-ci ne peuvent être activés en même temps. Par conséquent, le contexte `BatteryLevel` devra avoir la dépendance *xor*.

Sa sémantique est la suivante, où le contexte A est le contexte parent et les contextes A1, A2 sont les contextes enfants :

— si aucun contexte enfant n'est activé, le contexte A1 ou A2 peut être activé ;

- si le contexte A1 (resp. A2) est activé, et qu'on souhaite activer le contexte A2 (resp. A1), la dépendance entraînera la désactivation du contexte A1 (resp. A2) et l'activation du contexte A2 (resp. A1) ;
- un contexte A1 ou A2 peut être désactivé.

Exclusion La dépendance *exclusion* exclut que deux contextes soient activés en même temps. Un exemple a été donné dans la section 2.3.2 sur les dépendances.

Sa sémantique est décrite comme suit, sachant que les contextes A et B ont une dépendance d'*exclusion* entre eux :

- un contexte A (resp. B) peut être activé si et seulement si le contexte B (resp. A) n'est pas activé ;
- un contexte A (resp. B) peut être désactivé.

Requirement La dépendance *requirement* implique qu'un contexte doit déjà être activé. A nouveau, un exemple a été avancé dans la section 2.3.2.

Sa sémantique est détaillée ci-dessous, où un contexte A requiert un contexte B :

- le contexte A peut être activé si et seulement si B est activé ;
- le contexte B peut être activé ;
- le contexte A peut être désactivé ;
- le contexte B peut être désactivé mais, par conséquent, le contexte A devra l'être aussi.

Causality La dépendance *causality* est une faible inclusion comme expliquée dans la section 2.3.2.

Sa sémantique, où un contexte A inclut faiblement un contexte B, est la suivante :

- l'activation (resp. la désactivation) du contexte A entraînera celle du contexte B ;
- le contexte B peut être activé ou désactivé indépendamment de A.

Implication La dépendance *implication* est une implication au sens mathématique. Elle a déjà été définie dans la section 2.3.2.

Sa sémantique est quasi similaire à la dépendance *causality*, excepté que si le contexte B devient inactif, le contexte A sera désactivé entièrement (c'est-à-dire que son compteur deviendra égal à 0).

Il est évident qu'un contexte peut dépendre de plusieurs contextes, et ce, de manière différente pour chaque contexte si nécessaire. Pour gérer ces dépendances, une liste chainée a été choisie. Elle contiendra l'ensemble des dépendances existantes pour son contexte. Par exemple, un contexte de position `Usa` pourrait hériter d'une dépendance *xor* de son parent (`America`), ainsi qu'avoir une causalité avec `English`. Par conséquent, sa liste contiendrait les deux nœuds dont le premier serait un objet de la classe *Xor* et le second, un objet de la classe *Causality*. L'avantage d'utiliser une liste est de pouvoir distinguer chaque nœud de manière indépendante, et ainsi améliorer la compréhension des dépendances. De plus, chaque nœud est utilisé pour gérer sa propre dépendance, c'est-à-dire son activation ou sa désactivation. Par conséquent, il sera facile d'ajouter d'autres types de dépendances.

Cette liste chainée est également ordonnée en suivant l'ordre d'importance des dépendances, comme l'affiche la figure 6.2. Cet ordre permet d'éviter des (dés)activations inutiles. En effet, admettons que la causalité soit placée avant l'exclusion, cela signifierait que, pour un contexte A, ayant une dépendance de causalité avec C et une exclusion avec B, l'algorithme essayera d'activer C et engendrerait peut-être d'autres activations en cascade, avant de revenir à la vérification de A, pour se rendre compte que tout le processus peut être annulé à cause de l'exclusion de B. Grâce à cet ordre, l'exclusion de B sera remarquée directement et évitera ainsi les activations inutiles.

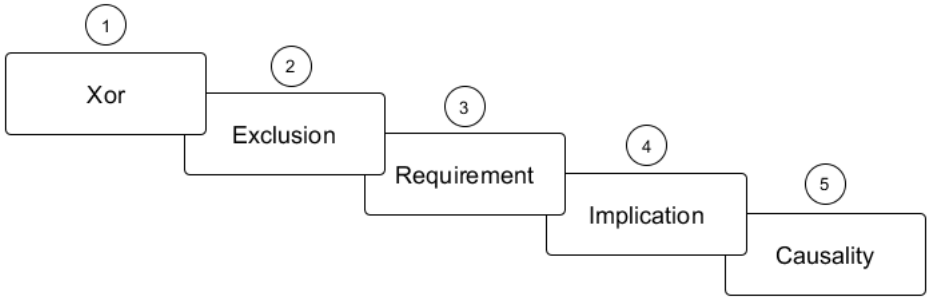


FIGURE 6.2: L'ordre des dépendances

Le contexte `Default`

Dans le graphe des contextes, un contexte sera toujours présent : `Default`. Il est considéré comme la racine du graphe et sera seulement activé au lancement de l'application, lorsqu'aucun contexte n'est encore détecté. Il a la particularité d'être non-abstrait, car il impliquera l'activation de la feature par défaut (*FDefault*), pour ainsi montrer à l'utilisateur que son application fonctionne correctement mais qu'elle n'a pas encore détecté une situation interne ou externe.

Un exemple

Afin de mieux comprendre les notions détaillées dans cette section, voici un exemple d'un graphe des contextes. Ce graphe représente différents contextes pour découvrir le niveau de la batterie. Un exemple de configuration (*Context Declarations*) pourrait être le code 6.1.

```
{
  "contexts": {
    "Default": {
      "subentities": {
        "BatteryLevel": {
          "isAbstract": true,
          "sensors": ["BatteryLevelSensor"],
          "subentities": {
            "HighBattery": {
              "condition": "(BatteryLevelSensor.level > 20)"
            },
            "LowBattery": {
              "condition": "(BatteryLevelSensor.level <= 20)
                  "
            }
          }
        }
      }
    }
  },
  "dependencies": {
    "BatteryLevel": {
      "xor": []
    }
  }
}
```

CODE 6.1: Exemple de déclarations de contextes

La figure 6.3 modélise cette configuration. Dans ce schéma, la couleur blanche symbolise les contextes abstraits, et la couleur bleue les contextes non-abstraits. Afin d'améliorer la lisibilité de ce schéma, j'ai volontairement enlevé leur origine (à savoir d'où proviennent les données).

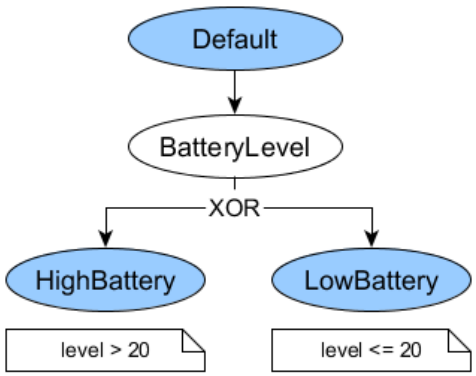


FIGURE 6.3: Un exemple d'un graphe des contextes

6.2 L'implémentation de la couche INTERACTION avec ses dépendances de la couche APPLICATION

La figure 6.4 représente les interactions entre les couches INTERACTION et APPLICATION et est expliquée dans la section 5.5.1.

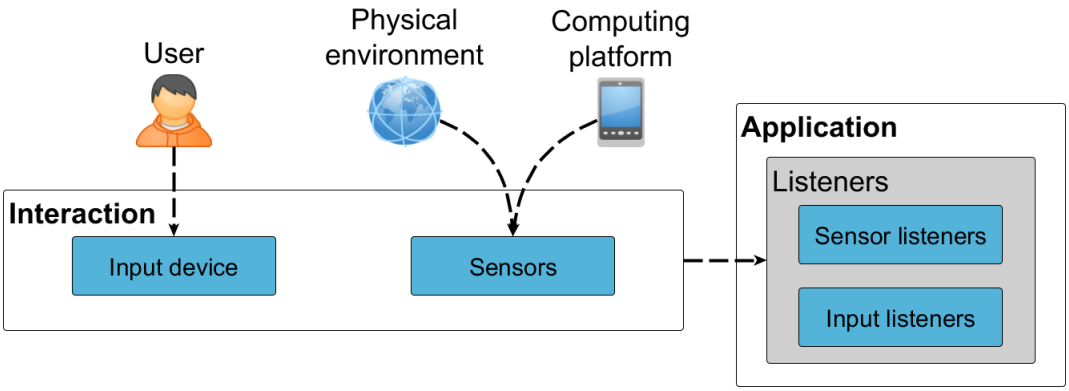


FIGURE 6.4: Les interactions de la couche INTERACTION avec la couche APPLICATION

6.2.1 Les *sensors* et leurs *listeners*

En fonction de l'application, des données seront produites selon les différents capteurs. Afin d'éviter les duplications de code, chaque senseur devra hériter du comportement de *ApplicationSensor*. Ce comportement définira une création générique des données grâce au patron de conception *Factory Method*. Le code 6.2 explicite cette création.

```
1 class ApplicationSensor
2   include Observable
3   def self.factory(data)
4     object = self.new(data)
5     app = object.notify
6     return object, app
7   end
8 end
```

CODE 6.2: Factory Method de la classe *ApplicationSensor*

Par exemple, le code 6.3 représente un senseur pour connaître le type de catastrophe naturelle.

```
1 module Application
2   module AppERS
3     class EmergencySensor < ApplicationSensor
4       attr_reader :disaster
5       def initialize(data)
6         update_state(data)
7       end
8       def update_state(data)
```

```
9           @disaster = data["disaster"]
10        end
11      end
12    end
13  end
```

CODE 6.3: Code du capteur d'une catastrophe naturelle dans la couche APPLICATION

Comme on peut le constater dans le code source 6.2, chaque senseur inclura le module *Observable*. Ce dernier notifiera la couche DISCOVERY lors d'une découverte d'un nouvel environnement, comme montré dans le code 6.4.

```
1 def set_data(data)
2   self.update_state(data)
3   notify
4 end
5
6 def notify
7   Discovery::Interpretation.instance.interpret(self.to_json)
8 end
```

CODE 6.4: Code pour notifier la couche *Discovery*

Ces données sont envoyées à la couche suivante sous format *JSON* afin de garder le type de chaque attribut.

6.3 L'implémentation de la couche DISCOVERY avec ses dépendances de la couche APPLICATION

Une fois les données reçues du capteur, la figure 6.5 montre qu'elles seront interprétées, pour ensuite raisonner dessus afin de trouver les contextes de l'environnement courant.

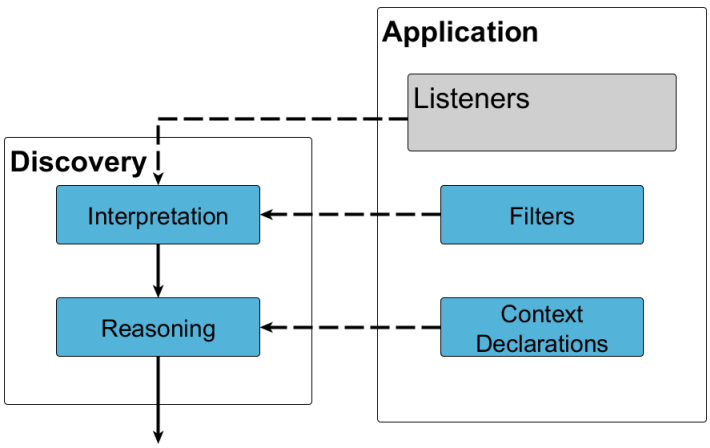


FIGURE 6.5: Les interactions de la couche DISCOVERY avec la couche APPLICATION

6.3.1 La composante *Interpretation*

Comme l'expose le code 6.5, la composante *Interpretation* parsera les données pour ensuite les filtrer [1] grâce aux filtres de l'application. Chaque senseur devra avoir son propre filtre qui héritera la classe *Filter*, car cette classe imposera la redéfinition de la méthode *filter*.

```
def interpret(json)
  hash_data = JSON.parse(json)
  class_name = hash_data['class_name']
  object = parser(hash_data)
  if filter(class_name, object)
    Reasoning.instance.reason(object)
  else
    # Raise exception
  end
end
```

CODE 6.5: Code pour interpréter les données des capteurs

Par exemple, le code 6.6 implémente le filtre pour le capteur *EmergencySensor* du code 6.3. Toutes les deux minutes, ce filtre peut accepter les données reçues pour décrire une catastrophe naturelle. En effet, il se peut qu'un tremblement de terre provoque un raz-de-marée et donc des inondations violentes.

```
class EmergencyFilter < Application::TimeFilter
  def initialize
    super
    @new_wait_to_filter = 60 * 2 # 2 minutes
  end
end
```

CODE 6.6: Code du filtre pour le capteur d'une catastrophe naturelle dans la couche APPLICATION

Comme le montre le code 6.6, le développeur peut créer des filtres plus génériques. Par exemple, il pourra créer un filtre sur le temps, où il faudra attendre un certain temps avant de récupérer à nouveau des données. Le code 6.7 montre ce filtre générique sur le temps.

```
class TimeFilter < Filter
    def initialize
        @wait_to_filter = 0
        @new_wait_to_filter = 0
        @time_now = Time.now
    end
    def filter
        if Time.now >= @time_now + @wait_to_filter
            @wait_to_filter = @new_wait_to_filter
            @time_now = Time.now
            return true
        end
        return false
```

1. La raison a déjà été évoquée dans la section 5.5.2

```
14      end
15  end
```

CODE 6.7: Code d'un filtre générique sur le temps

6.3.2 La composante *Reasoning*

Après avoir récupéré le graphe des contextes, la composante *Reasoning* cherchera les contextes admissibles en parcourant tout le graphe. Pour chaque élément du graphe, on vérifiera si le contexte est valable en fonction de sa condition. Si la condition est satisfaite, le contexte testé sera ajouté à la liste des contextes admissibles. Le code 6.8 détaille l'algorithme pour trouver ces contextes.

```
1  def getAdmissibleContexts(contextsGraph, o)
2    admissibleContexts = []
3    queue = [contextsGraph]
4    while !(queue.empty?)
5      contextsGraph = queue.shift
6      isOk = contextsGraph.applyConditions(o)
7      admissibleContexts << contextsGraph if isOk
8      queue = contextsGraph.children + queue
9    end
10   admissibleContexts
11 end
```

CODE 6.8: Code pour récupérer les contextes admissibles

En se basant sur le graphe des contextes de la figure 6.3, si un objet *BatteryLevel* est créé avec son attribut *level* valant 19, les contextes admissibles trouvés seront : `BatteryLevel` et `LowBattery`.

6.4 L'implémentation de la couche HANDLING avec ses dépendances de la couche APPLICATION

La figure 6.6 présente les interactions entre les couches HANDLING et APPLICATION et est expliquée dans la section 5.5.3.

6.4.1 L'activation des contextes

Les composantes *Context Activation* et *Feature Activation* sont très similaires car chacune (dés)activera les nouvelles entités découvertes. Par conséquent, je ne parlerai que de la composante *Context Activation*.

Le code 6.9 montre l'algorithme du *Context Activation*. Cette composante récupérera la liste des contextes activés. Il la clonera de façon à pouvoir la modifier et ainsi tester si les nouveaux contextes peuvent être activés. S'il n'y a pas d'erreur lors de l'activation, la liste des contextes activés sera changée avec la nouvelle. Les erreurs sont dues à l'essai d'activation d'un contexte alors qu'il requiert un contexte non-activé, ou alors qu'il s'exclut avec un contexte déjà activé, comme expliqué plus haut (cf. la section 6.1.1).

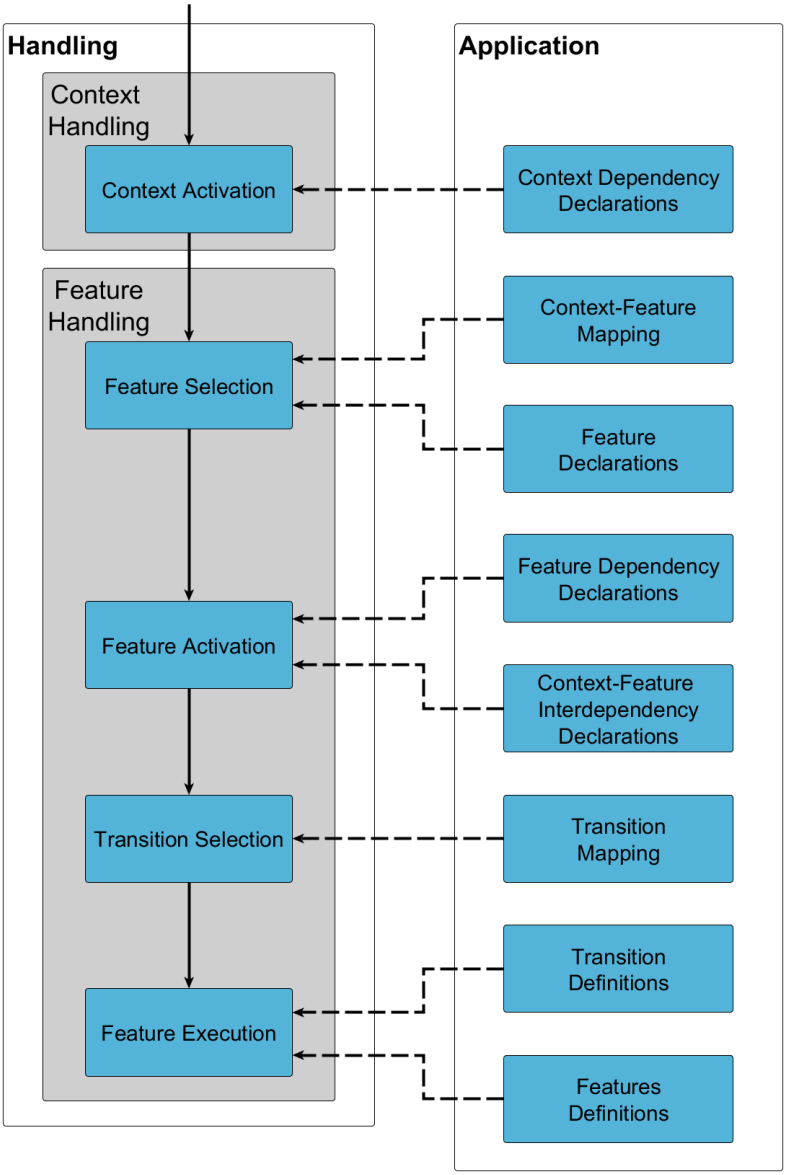


FIGURE 6.6: Les interactions de la couche HANDLING avec la couche APPLICATION

```
1  def activate(o, contexts)
2    actividContextsCounters = Application::ContextDefinition.
       instance.actividContextsCounters
3    actividContexts = actividContextsCounters.clone
4    managedContexts = {}
5    begin
6      contexts.each {
7        |context|
8        context.canActivate(actividContexts, managedContexts)
9      }
10     Application::ContextDefinition.instance.
       setActividContextCounters(actividContexts)
11   rescue Exceptions::DependencyException => e
```

```
12      # Manage exception
13    end
14    actividContextsCounters = Application::ContextDefinition.
        instance.actividContextsCounters
15    FeatureSelection.instance.select(o, managedContexts)
16  end
```

CODE 6.9: Code de la composante *Context Activation*

Le processus d'activation sera délégué au contexte-même comme le montre le code 6.9. Ce contexte délèguera, à son tour, l'activation à ses propres dépendances.

En plus du principe d'activation et de désactivation, comme annoncé dans le chapitre 2 et expliqué dans les travaux [15], [25] et [28], j'ai également géré un compteur pour chaque contexte activé. Pour rappel, un contexte activé est un contexte pour lequel son compteur est plus grand que zéro. S'il est équivalent à zéro, le contexte sera considéré comme désactivé. Cette notion sera implémentée grâce à une classe *ActivedEntitiesCounters* qui contient une *HashMap* dont les clés seront les objets *Entity* avec comme valeur, leur compteur.

6.4.2 La sélection des features

La composante *Feature Selection* sélectionnera les features correspondantes aux contextes activés et désactivés.

Premièrement, elle sélectionnera les contextes les plus spécifiques. En effet, seuls les contextes spécifiques impliqueront une recherche sur les features, car en général, les contextes plus généraux seront abstraits.
Ensuite, elle sélectionnera les features correspondantes grâce au *Context-Feature Mapping* (cf. la figure 6.6). Ce mapping est une simple *HashMap*, dont les clés sont les identifiants des contextes. Tandis que les valeurs associées sont les identifiants de la feature s'y rapportant.
Enfin, après avoir sélectionné les features, elle récupérera les objets features dans le graphe des features, similaire au graphe des contextes. Ce graphe sera créé grâce à la composante *Feature Declarations* (cf. la figure 6.6).

Le code 6.10 pourrait être un exemple simplifié pour la configuration de la composante *Context-Feature Mapping* dans notre étude de cas (cf. le chapitre 3). Si le contexte activé est `LowBattery`, cette sélection récupérera le feature *DisplayInstructions*.

```
1 {
2   "Default": "FDefault",
3   "Earthquake": "InformDisaster",
4   "LowBattery": "DisplayInstructions"
5 }
```

CODE 6.10: Exemple de déclarations pour la composante *Context-Feature Mapping*

6.4.3 La sélection des transitions

Comme expliqué dans la section 2.4.2, les transitions sont d'une grande importance pour l'utilisateur. Par conséquent, chaque feature sera associée à une transition et celle-ci pourra changer si la feature doit être activée ou désactivée.

Cependant, définir la même transition pour chaque feature (ou un ensemble de features ayant le même parent) peut s'avérer fastidieux pour le développeur. Du coup, il a la possibilité de définir une transition pour une feature parent et que cette animation soit héritée par ses features enfants. Mais dès qu'une nouvelle transition est déclarée pour une feature plus spécifique, elle remplacera celle héritée. De plus, cette version évitera les redondances du code *JSON* dans le *Transition Mapping*.

Un exemple de cette configuration pour l'étude de cas (cf. le chapitre 3) est montré dans le code 6.11. Cette configuration appliquera une transition de *fade in* avec une vitesse *slow* pour toutes les features lors de leur activation.

```
{
  "activated": {
    "FDefault": {
      "FadeIn": "slow"
    }
  }
}
```

CODE 6.11: Exemple de configuration pour la composante *Transition Mapping*

Le code 6.12 détaille l'implémentation de la récupération de la transition la plus spécifique. Sur base de la configuration déclarée dans le code 6.11, il retournera toujours la transition *fade in*.

```
def getMostSpecificTransitions(feature, transitionMapping)
  if !transitionMapping
    return nil
  end
  queue = [feature]
  while !(queue.empty?)
    f = queue.shift
    if transitionMapping.keys.include?(f.name)
      return transitionMapping[f.name]
    end
    queue += f.getAllSuperEntities
  end
  return nil
end
```

CODE 6.12: Code pour récupérer la transition la plus spécifique

6.4.4 L'exécution des features

La composante *Feature Execution* créera ou modifiera l'application en y ajoutant et/ou supprimant les définitions des transitions (*Transition Definitions*) et des features (*Feature Definitions*). Ce traitement se fera avec le code 6.13.

```
1  def execute(o, featuresTransitionsToActivate,
       featuresTransitionsToDeactivate)
2    currentApp = Application::Config.instance.currentApp
3    if currentApp
4      currentApp.removeFeatureTransitions(
       featuresTransitionsToDeactivate)
5      currentApp.addFeaturesTransitions(o,
       featuresTransitionsToActivate)
6      Application::Config.instance.setApp(currentApp)
7      currentApp
8    else
9      # App doesn't exist.
10     appModule = Application.const_get(Application::Config.
       instance.appName)
11     app = appModule.const_get('Main').new()
12     app.addFeaturesTransitions(o,
       featuresTransitionsToActivate)
13     Application::Config.instance.setApp(app)
14     app
15   end
16 end
```

CODE 6.13: Code pour créer/modifier l'application sur base des features

Au lancement de l'application, lorsqu'aucun contexte n'est détecté, la composante *Feature Execution* créera une application avec sa feature par défaut. L'intérêt de supporter un tel contexte et une telle feature apporte une meilleure homogénéité dans le framework, car cela permet de gérer tous les cas de la même manière pour l'utilisateur du framework, à savoir le développeur.

Ensuite, lorsqu'un nouvel environnement est décelé, cette composante modifiera l'application courante, retenue dans sa configuration.

Par exemple, en prenant en compte les deux premières parties du scénario de l'étude de cas (cf. la section 3.3), au lancement de l'application, elle contiendra seulement la feature *FDefault*. Après la détection du tremblement de terre, la feature *FDefault* sera supprimée et la feature *InformDisaster* sera ajoutée.

Les définitions des features et des transitions seront vues respectivement dans les sections 6.5.3 et 6.5.4.

6.5 L'application en tant que telle

L'application est gérée par la dernière composante *Feature Execution* et suit l'architecture *MVC* (modèle-vue-contrôleur). Le schéma 6.7 présente cette architecture d'un point de vue global. Certains éléments ont été retirés pour faire ressortir les aspects importants. Dans cette figure, les modules sont des rectangles arrondis et les classes sont des rectangles classiques. Lorsque les noms des modules ou des classes dépendent de l'application, j'ai précisé cela par *Name of application/feature/transition*. Seules les classes en bleu seront présentées, car pour rappel, mon mémoire concerne l'aspect des interfaces utilisateur.

6.5.1 La classe *Main*

La classe *Main* sert de point d'entrée pour l'application. Comme cette application est construite au fur et à mesure, cette classe contiendra une référence vers chaque grande couche du *MVC*. Comme le montre la figure 6.7, cette classe contient une référence vers la classe *Gui*.

Cette classe *Main* comporte trois méthodes importantes :

addFeaturesTransitions appelée depuis la composante *Feature Execution*, elle ajoutera les features dans chaque couche ainsi que les transitions pour la couche vue ;

removeFeatureTransitions également utilisée dans la même composante, elle supprimera les features dans chaque couche et montrera ces changements grâce à leurs transitions ;

run exécutant le programme.

6.5.2 Le module *Gui* et sa classe

Le module *Gui* représente la couche vue dans le *MVC* et contient une classe *Gui*. Cette classe centralise toutes les features et transitions de l'application courante, c'est-à-dire l'application avec les fonctionnalités correspondant à l'environnement dans lequel elle s'exécute.

Elle contiendra les attributs suivants :

features représentant une *HashMap* dont les clés sont les noms des features et les valeurs sont les objets features. L'utilisation de cette map permet des accès rapides lors de la récupération et la suppression des features ;

transitions modélisant une *HashMap* dont les clés sont les noms des features, afin de récupérer facilement la transition pour chaque feature, et les valeurs étant les objets transitions. Cet objet transition sera vu dans la section 6.5.4 ;

orderGui contenant une liste de features afin de toujours garder la même structure dans la création des interfaces utilisateur. En effet, il est important de conserver le même rendu pour l'utilisateur, afin qu'il ne soit pas perdu à chaque adaptation. Par exemple, pour les features *InformDisaster*, affichant un message prévenant du type de désastre, et *DisplayInstructions*, indiquant les instructions de route, l'ordre pourrait

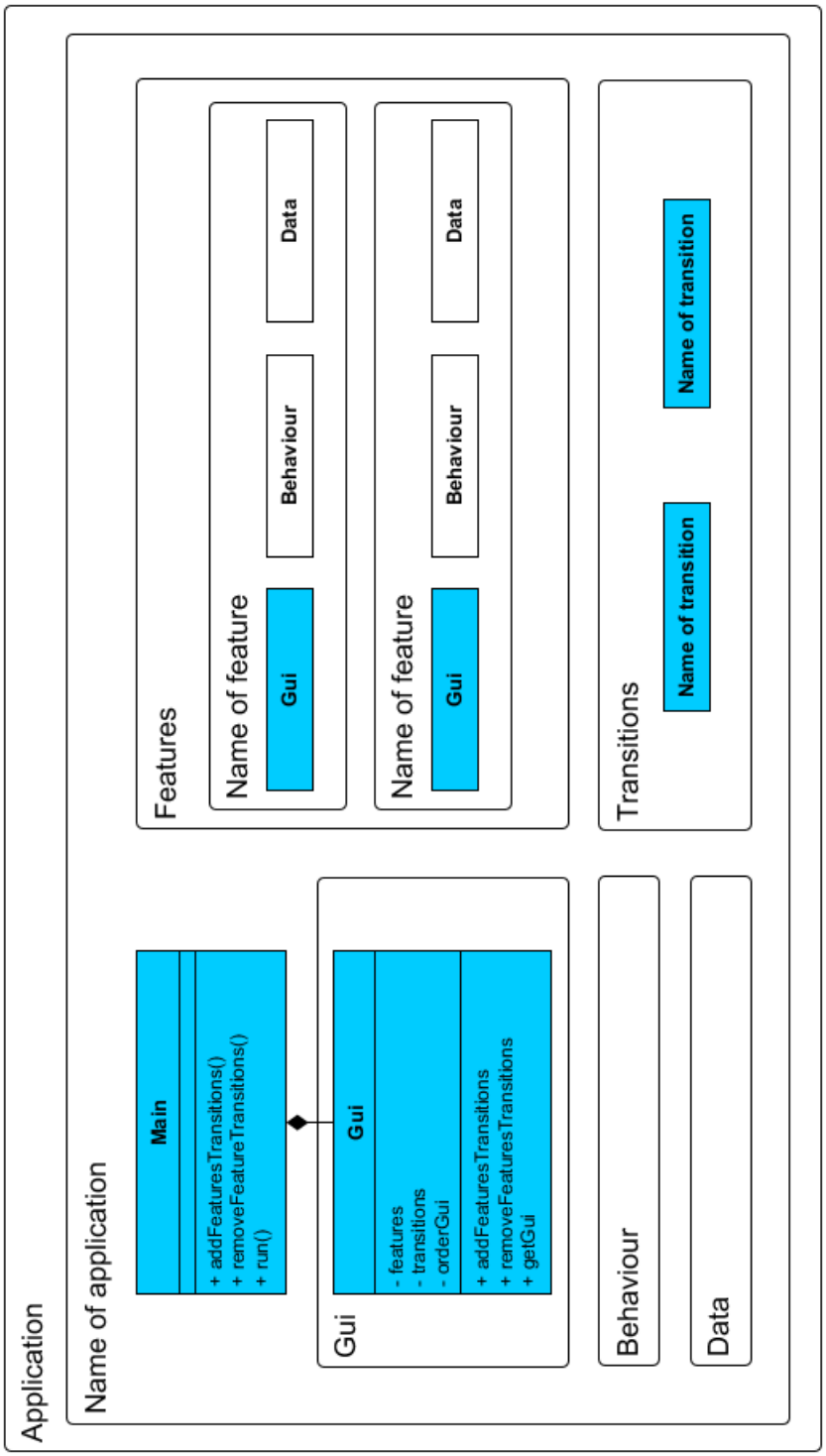


FIGURE 6.7: L'architecture de l'application

être le suivant : *InformDisaster* et *DisplayInstructions*. Cet ordre imposera que le message apparaisse toujours avant les instructions pour trouver son chemin, si les deux features sont activées en même temps.

Comme l'expose la figure 6.7, cette classe *Gui* contient trois principales méthodes. Outre les méthodes *addFeaturesTransitions* et *removeFeaturesTransitions* dont le comportement a déjà été expliqué dans le point précédent, il y a la méthode *getGui*.

Pour notre étude de cas, cette méthode créera l'entièreté de l'interface utilisateur pour l'application web, comme l'expose le code 6.14. Cet algorithme parcourra son *orderGui* afin de construire l'interface petit à petit, en récupérant pour chaque feature, le code de l'interface utilisateur ainsi que le code de sa transition correspondante. Une fois la transition utilisée, elle sera supprimée afin de ne porter l'attention de l'utilisateur que sur les changements courants.

```
def getGui
  # Code to manage the default case
  gui = ""
  js = '<script type="text/javascript">'
  transitions = ""
  @orderGui.each {
    |orderedFeature|
    allSubFeatures = [orderedFeature] + orderedFeature.
   getAllOwnChildren
    allSubFeaturesNames = allSubFeatures.map { |feature|
   feature.name }
    intersection = allSubFeaturesNames & @features.keys
    intersection.each {
      |featureName|
      gui << @features[featureName].getUI
      js << @features[featureName].getJS
      if @transitions[featureName]
        transitions << @transitions[featureName].
   getTransition
        @transitions.delete(featureName)
      end
    }
  }
  # Some code to include a JavaScript API, like Google Maps
  js << '</script>'
  appUI = gui << js << transitions
  return appUI
end
```

CODE 6.14: Code pour créer les interfaces utilisateur

6.5.3 Le module *Features*

Ce module contiendra toutes les features de l'application. Chaque feature correspondra à un module qui contiendra les trois classes du modèle *MVC*. Chaque classe apportera sa pierre à l'édifice dans la construction de de l'application.

En ce qui concerne les classes *Gui* dans chaque feature, elles contiendront la partie de l'interface pour la feature en question. En suivant le cas d'une application web, elle contiendra deux méthodes : *getUI* et *getJS*. La première

s'occupera de la gestion du code statique, à savoir le code *HTML*. Tandis que la deuxième (*getJS*) inclura le code dynamique, donc les scripts *JavaScript*.

Par exemple, le code 6.15 représente la définition de la feature *InformDisaster* avec sa classe *Gui*.

```
module Application
  module AppERS
    module Features
      module InformDisaster
        class Gui
          include Application::Features::Proceed
          def initialize(o)
            @disaster = o.disaster
          end
          def getUI
            ui = '<div id="InformDisaster" style="margin: 20
    px 30px">
              <div class="alert alert-danger" role="alert">
                A disaster <b>' << @disaster << '</b> is
    detected !! <br/>
                Pay attention waiting specific instructions.
              </div>
            </div>'
            ui << proceedUI
          end
          def getJS
            '$("#NoDetected").remove();
             $("#SpecificCases > :not(#' << @disaster << ')
    ").remove();'
          end
        end
      end
    end
  end
end
```

CODE 6.15: Code *Ruby* pour la feature *InformDisaster*

Afin d'éviter la redondance de code dans les features, il est également possible de combiner le code de plusieurs features pour construire une nouvelle interface utilisateur. Ce concept vient de l'article sur le *Subjective-C* [25] avec le `SUPERCONTEXT`, comme expliqué dans la section 2.3.1. Il est implémenté dans le framework avec le module *Proceed*.

Comme le propose le *Subjective-C*, la méthode *supercontext* n'est pas restrictive grâce à son sélecteur. Du coup, dans mon implémentation, si la méthode *proceedUI* est appelée sans paramètre, elle ajoutera le code statique de la feature parent, sinon elle cherchera le code source statique de la feature demandée. Le code 6.16 montre son implémentation pour la partie statique du code (à savoir la partie *HTML*) tandis que le code 6.15 expose son utilisation.

```
def proceedUI(selector=nil)
  featureGraph = nil
  if !selector
    featureName = self.class.name.split("::")[-2]
    currentFeature = FeatureDefinition.instance.
   getEntitiesGraph(featureName)
    featureGraph = currentFeature.implicitSuperEntity
  else
    featureGraph = FeatureDefinition.instance.
   getEntitiesGraph(selector)
  end
  appName = Application::Config.instance.appName
  moduleObject = Application.const_get(appName).const_get("
    Features").const_get(featureGraph.name)
  o = moduleObject.const_get("Gui").new()
  return o.getUI
end
```

CODE 6.16: Code de la méthode *proceed*

6.5.4 Le module *Transitions*

Enfin, le module *Transitions* contiendra toutes les classes des transitions définies pour l'application. Ainsi, chaque classe sera dédiée à une transition spécifique. Le code 6.17 montre l'implémentation de la transition *fade in* pour l'étude de cas du chapitre 3.

```
module Application
  module AppERS
    module Transitions
      class FadeIn
        def initialize(featureName, speed)
          @featureName = featureName
          @speed = speed
        end
        def getTransition
          '<script type="text/javascript">
           $(function(){
            $("#' << @featureName << '").fadeIn("' << @speed
    << '");
           });
           </script>'
        end
      end
    end
  end
end
```

CODE 6.17: Exemple de code pour la transition *fade in*

Comme l'expose le code 6.17, le développeur devra faire attention à bien entourer son code *HTML* d'un élément *DIV* dans sa feature afin de lancer la transition sur l'entièreté de la feature.

Chapitre 7

Le simulateur

Le simulateur est une application web créée pour réaliser deux objectifs.

Premièrement, il permet de simuler un environnement de test pour les applications orientées contexte suivant l'architecture développée dans le chapitre 5. L'utilisateur pourra reproduire des environnements dans lesquels l'application s'exécutera. Il devra lui-même changer la situation courante afin de simuler le changement de contexte. Ce dernier provoquera l'adaptation de l'application. Dans ce cas, le simulateur est considéré comme un *debugger*.

Son deuxième objectif est de valider l'architecture (cf. la chapitre 5) et l'implémentation (cf. le chapitre 6) du framework. Effectivement, par la création des applications orientées contexte, il est plus aisé pour le développeur du framework de constater quel concept est bien défini, ou quelle notion peut manquer, ...

Dans ce chapitre, je présenterai le fonctionnement du simulateur, ainsi qu'une partie de son implémentation. J'exposerai également des exemples de code se référant à l'étude de cas du chapitre 3. Ceci afin de montrer comment on peut la tester.

7.1 Son fonctionnement

La figure 7.1 représente le fonctionnement du simulateur lorsque l'utilisateur simule un changement de contexte.

Son fonctionnement est le suivant :

1. lorsque l'utilisateur simulera un changement de contexte, le navigateur enverra une requête vers le serveur. Cette requête contiendra les données contextuelles du capteur ;
2. à la réception de cette dernière, il notifiera le framework via les divers senseurs de l'application. Pour rappel, ceux-ci sont à la base du traitement d'un nouveau contexte ;
3. une fois que le framework a achevé son processus, il retournera au serveur une application modifiée contenant l'ensemble des features activées ainsi que les transitions nécessaires ;
4. le serveur exécutera l'application. Cette exécution construira son application entière, entre autres son interface utilisateur ;

5. l'application retournera son code source. Pour ce mémoire, elle enverra le code de son module concernant les interfaces utilisateur, c'est-à-dire les codes *HTML* et *JavaScript* ;
6. finalement, le serveur répondra au client en lui envoyant le résultat de l'exécution de l'application.

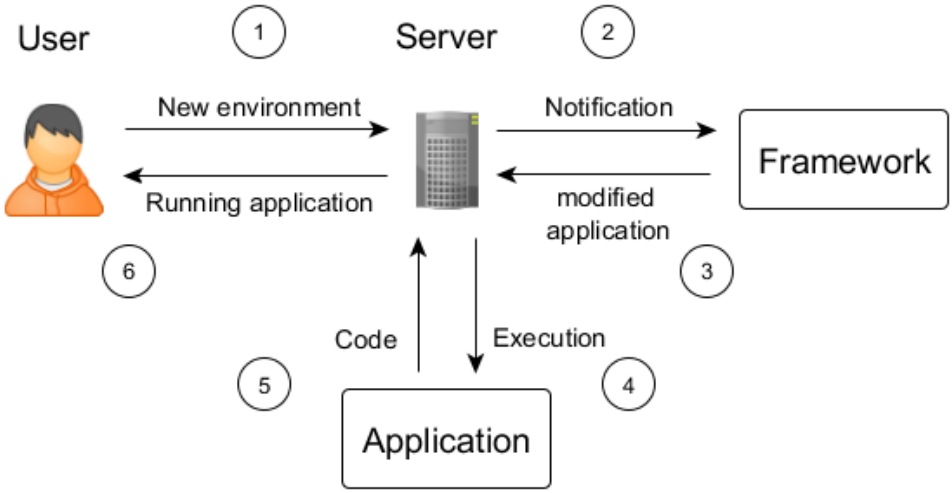


FIGURE 7.1: Le fonctionnement du simulateur lors d'un changement de contexte

7.2 Son implémentation

Comme précédemment, cet outil est une application web, mais comment a-t-il été implémenté du côté client et du côté serveur ? Comment le serveur fera-t-il sa liaison avec le framework orienté contexte ?

7.2.1 Côté client

Lors de la simulation d'un changement de contexte de la part de l'utilisateur, un évènement *JavaScript* sera lancé. Ce dernier récupérera le capteur qui a produit ces données et ces données contextuelles. Il appellera ensuite la méthode *changeContext*. Cette dernière créera certains paramètres de la requête et lancera celle-ci via la méthode *launchRequest*. La méthode de cette requête sera une méthode *POST*, car son but est de modifier le contenu de l'application.

En fonction du résultat de la requête, l'interface utilisateur de l'application sera affichée. Pour cela, le contenu de l'élément *HTML DIV#smartphone-content* sera remplacé par le nouveau. Sinon une erreur sera ajoutée dans la console web intégrée à la page web.

Le code 7.1 montre cette implémentation générique.

```
function changeContext(sensor, data) {
  var params = 'appName=' + $("#appName").text() +'&sensor='
      + sensor + '&' + data;
  var url = 'changeContextAjax';
  launchRequest(params, url);
}

function launchRequest(params, url) {
  $.ajax({
    method: 'post',
    url: url,
    data: params,
    dataType: 'html',
    success: function(res, status) {
      getActivationInfoAjax();
      $("#smartphone-content").empty();
      $("#smartphone-content").html(res);
    },
    error: function(res, status, error) {
      $("#logging").append(resError);
    }
  });
}
```

CODE 7.1: Code pour envoyer les requêtes en *AJAX*

Les codes 7.2 et 7.3 développent un exemple de capteur permettant à l'utilisateur de simuler la détection d'une catastrophe naturelle. Son résultat est montré dans la figure 7.2.

```
<div id="EmergencySensor">
  <div class="panel panel-default">
    <div class="panel-heading">EmergencySensor</div>
    <div class="panel-body">
      <button type="button" data-disaster="earthquake"
          class="btn btn-danger col-md-12 margin-bottom"
          style="padding:10px 0px">
        Earthquake detected
      </button>
      <button type="button" data-disaster="flood" class="btn
          btn-primary col-md-12" style="padding:10px 0px">
        Flood detected
      </button>
    </div>
  </div>
</div>
```

CODE 7.2: Code *HTML* représentant l'environnement d'un désastre

```
1 $(function(){
2   $("#EmergencySensor button").click(function() {
3     var disaster = $(this).attr('data-disaster');
4     var data = 'disaster=' + disaster;
5     changeContext('EmergencySensor', data);
6   });
7 });
```

CODE 7.3: Code *JavaScript* interceptant une catastrophe naturelle

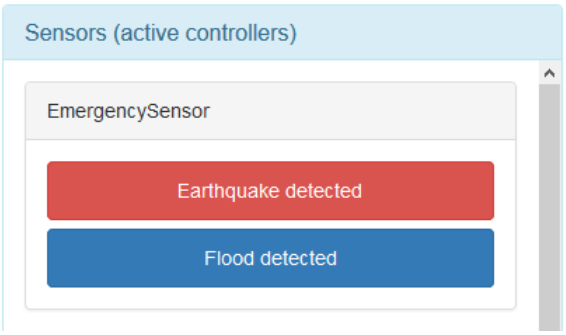


FIGURE 7.2: Le capteur web simulant une catastrophe naturelle

7.2.2 Côté serveur

D'un point de vue serveur, une application orientée contexte sera modélisée sous la forme d'un *board*. Son état est constitué du nom de l'application, d'une description de l'application facultative et d'un chemin absolu vers l'application (depuis la racine de l'ordinateur). Ce dernier attribut sera utilisé lors de la création des senseurs actifs et passifs pour le simulateur. Un capteur actif représentera la situation externe au smartphone, comme par exemple la météo. Tandis qu'un senseur passif découvrira l'environnement interne au smartphone, tel que l'état de sa batterie.

Après le changement de situation simulé par l'utilisateur, le contrôleur web recevra la requête contenant toutes les informations. Il créera le capteur correspondant au framework orienté contexte avec les données reçues grâce à la méthode *factory(data)*. Pour rappel, cette création lancera le flux de contrôle du framework comme expliqué dans le chapitre 5. Une fois la fin de ce processus, le serveur exécutera l'application reçue afin de construire son interface utilisateur. Le code 7.4 expose cette implémentation.

```
def changeContextAjax
  # split data in request to get appName, sensor and data

  begin
    temp, app = Application.const_get(appName).const_get(
    sensor).factory(data)
    render html: app.run.html_safe if app
  rescue Exceptions::FrameworkException => e
    error = "#{e.class}" + ' -> ' + "#{e}"
    render json: error, :status => 500
  end
end
```

CODE 7.4: Code serveur lors de la réception d'une requête

7.3 Le résultat

La figure 7.3 montre le simulateur avec comme application en cours l'étude de cas du chapitre 3. Cette application contient seulement le comportement par défaut activé.

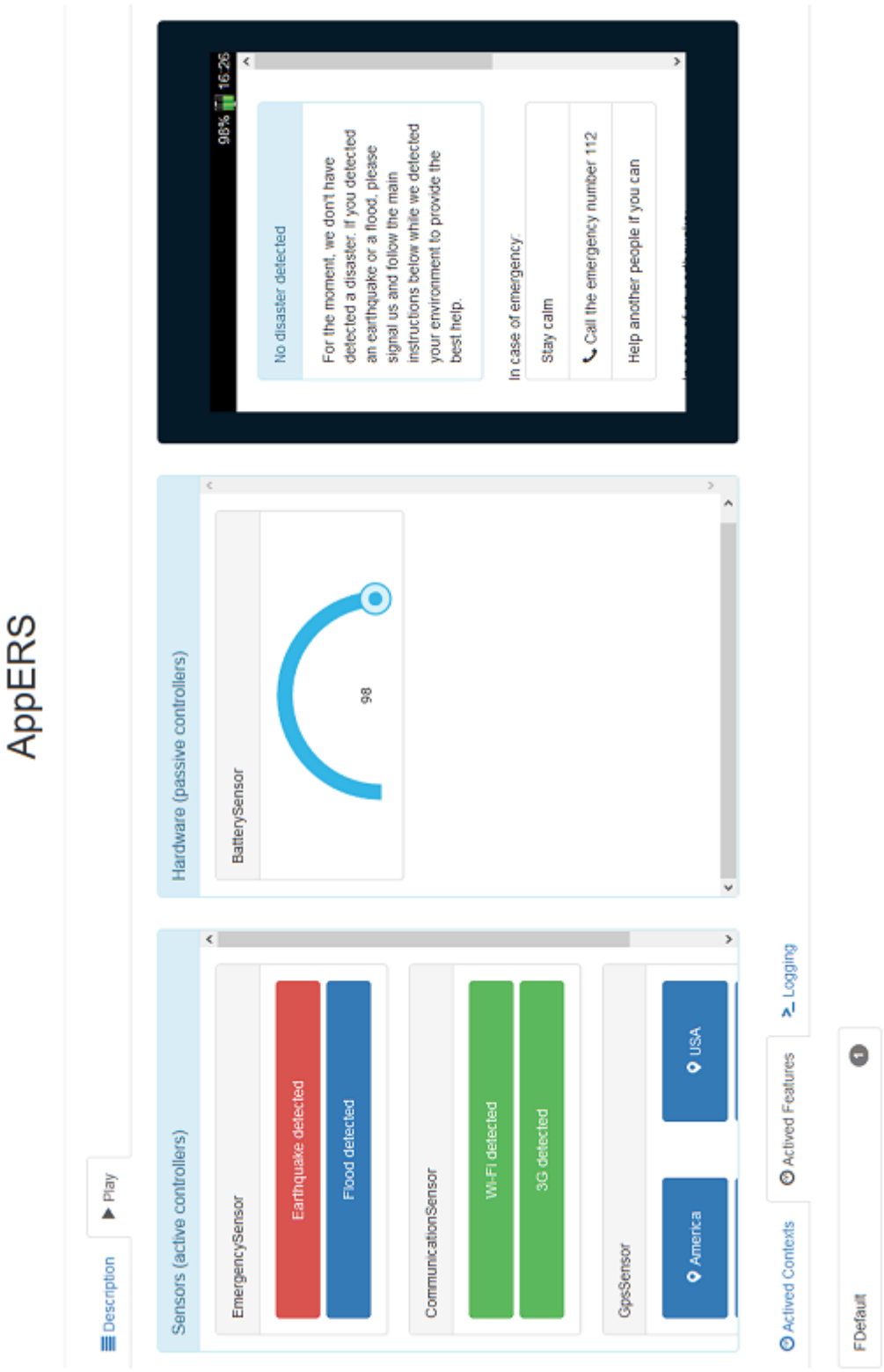


FIGURE 7.3: Le simulateur avec l'application *emergency response system*

Chapitre 8

La discussion

Tandis que le résultat de l'application, en suivant un scénario, a déjà été illustré dans la section 3.3, je présenterai, dans ce chapitre, les limitations du framework (cf. les chapitres 5 et 6) détectées lors de la création de cette application. Enfin, je terminerai ce chapitre par les travaux futurs.

8.1 Les limitations

Une première limitation dans le framework est l'utilisation d'une seule origine, à savoir d'où proviennent les données. Dans notre étude de cas (cf. le chapitre 3), la langue de l'application dépendra de la localisation de l'utilisateur. Par exemple, si la position de l'utilisateur est à New-York, la langue de l'application sera l'anglais. Mais cela reste assez restrictif, car la langue pourrait dépendre de plusieurs origines, comme par exemple, la localisation ainsi que la préférence de l'utilisateur. Sur base du graphe des contextes de l'application (cf. la section 3.2.1), supposons que l'utilisateur ait déjà rentré ses préférences, avec comme langue par défaut le français, cela signifiera que le contexte `French` est déjà actif dans l'application. Imaginons maintenant qu'une position à New-York a été détectée. De ce fait, l'application agira en activant `NewYork`. Par la sémantique des dépendances (cf. 6.1.1), lors de son activation, il essayera d'activer `English`. Du coup, `French` sera désactivé pour être remplacé par `English`. Mais ces changements impliqueront que l'application sera traduite en anglais alors que le choix de l'utilisateur est de garder l'application en français.

Une solution possible serait de donner une précédence sur les origines des contextes lors de l'activation. C'est-à-dire donner une priorité plus importante à la préférence de l'utilisateur plutôt qu'à la langue dépendant de la localité. Cependant, cette solution reste encore limitée, car elle ne résoudra qu'une seule partie du problème.

En effet, en suivant la sémantique (cf. la section 6.1.1), le contexte `NewYork` ne pourra pas s'activer vu la dépendance de causalité. Donc, en plus d'ajouter des priorités sur les origines, il faudrait aussi créer une dépendance. Celle-ci devrait suivre la sémantique suivante, en connaissant les contextes A, A1, A2, B, B1 et B2, où A et B sont respectivement les parents de A1 et A2, B1 et B2, et que A a une dépendance de ce type vers B :

— un enfant de A activera un enfant de B s'il n'est pas encore activé ;

— un enfant de A pourra être activé si un enfant de B est déjà activé, et par conséquent, cela ne changera pas l'enfant de B.
— B, B1 ou B2 pourra être (dés)activé indépendamment de A, A1 ou A2.

Une autre limitation est de n'avoir qu'un seul parent par contexte. Il se pourrait, dans des cas très spécifiques, qu'un contexte ait besoin de plusieurs parents. Le fait de prévoir une telle notion dans le framework augmenterait son expressivité. Cependant, jusqu'à ce jour, ce cas ne s'est jamais présenté, c'est pourquoi je ne l'ai pas implémenté.

8.2 Les travaux futurs

Dans un travail de recherche de cette envergure, il me semble évident que tout ne peut être comblé dans ce mémoire, quelques exemples sont donnés dans cette section.

Un premier travail futur pourrait être de résoudre les limitations du framework dans la section précédente (cf. 8.1) afin d'augmenter son expressivité.

Deuxièmement, il faudrait développer la dernière composante *Context-Feature Interdependency Declarations* qui n'a pas pu être implémentée dans le framework. Pour rappel, elle permettait de créer des dépendances entre les contextes et les features comme le suggèrent Hartmann et Trew [17]. Cela permettra d'avoir plus d'expressivité pour le développeur lors de la création des applications orientées contexte.

De plus, le framework ne gère pas les aspects concurrents lors de la réception des données contextuelles. En effet, dans le monde réel, les senseurs ne se synchroniseront pas entre eux pour l'envoi de leurs données. Cette synchronisation devra se faire à la réception des données dans notre framework.

Enfin, un travail futur serait d'améliorer les interactions utilisateur avec, entre autres, les *user acceptance*. Ceci afin de toujours laisser le contrôle à l'utilisateur. Il est essentiel que ce dernier se sente maître de son application. Par exemple, lors d'une détection d'une batterie faible, il faudrait laisser le choix à l'utilisateur de changer la carte GPS en instructions de route. S'il refuse ce changement, il sera averti qu'une carte GPS consomme beaucoup plus que du texte. Sinon, l'adaptation aura lieu.

Chapitre 9

Conclusion

Ce mémoire avait pour but d'intégrer les adaptations interfaces utilisateur dans une approche de développement logiciel orientée contexte. Elle devait également permettre d'inclure aisément les autres aspects, tels que celui sur les comportements et celui sur les données.

Lors de ce travail, trois contributions ont été réalisées :

le framework est une technologie facilitant les développements des applications sensibles aux contextes. Son architecture est divisée en quatre couches : INTERACTION, DISCOVERY, HANDLING et APPLICATION. Cette couche INTERACTION interceptera les données du monde externe et les enverra à la couche DISCOVERY, qui les interprétera et raisonnera dessus pour trouver les contextes caractérisant la situation courante. Ensuite, la couche HANDLING les activera. De plus, elle sélectionnera et activera les features correspondantes afin d'adapter le comportement de l'application.

Ces trois couches utiliseront la dernière couche APPLICATION qui contiendra les déclarations et les définitions de l'application ;

l'intégration des adaptations interfaces utilisateur concerne principalement la couche APPLICATION. En effet, pour l'application, une architecture suivant le modèle *MVC* (Modèle-Vue-Contrôleur) a également été définie.

En outre, la gestion des transitions est venue enrichir le framework ;

le simulateur est un site web réalisant deux objectifs. D'une part, il permet de créer et tester des applications orientées contexte dans un environnement simulé par l'utilisateur. D'autre part, il valide les concepts du framework.

De plus, une étude de cas, concernant un *emergency response system*, a été implémentée. Cette application, servant à aider les victimes lors d'une catastrophe naturelle, a permis de valider ces contributions.

Cependant, il est toujours possible d'améliorer un tel framework comme le montrent les limitations et les travaux futurs, notamment pour les *user acceptance*.

Finalement, ce mémoire a fait l'objet d'un article scientifique [32] pour le *COP'16 workshop @ ECOOP 2016*. Ecrit par le professeur Kim Mens, Nicolás Cardozo et moi-même, il sera publié dans l'*ACM Digital Library*.

Bibliographie

[1] Margaret M GARDINER et Bruce CHRISTIE. *Applying cognitive psychology to user-interface design*. Wiley Chichester, 1987.

[2] Bill SCHILIT, Norman ADAMS et Roy WANT. « Context-aware computing applications ». Dans : *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*. IEEE. 1994, p. 85–90.

[3] Bay-Wei CHANG et David UNGAR. « Animation : from cartoons to the user interface ». Dans : (1995).

[4] Mohamed FAYAD et Douglas C SCHMIDT. « Object-oriented application frameworks ». Dans : *Communications of the ACM* 40.10 (1997), p. 32–38.

[5] David THEVENIN et Joëlle COUTAZ. « Plasticity of user interfaces : Framework and research agenda ». Dans : *Proceedings of INTERACT*. T. 99. 1999, p. 110–117.

[6] Krzysztof CZARNECKI et Ulrich W. EISENECKER. *Generative programming : methods, tools, and applications*. Addison Wesley, 2000.

[7] Anind K DEY. « Understanding and using context ». Dans : *Personal and ubiquitous computing* 5.1 (2001), p. 4–7.

[8] Jilles VAN GURP, Jan BOSCH et Mikael SVAHNBERG. « On the notion of variability in software product lines ». Dans : *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE. 2001, p. 45–54.

[9] Harry CHEN, Tim FININ et Anupam JOSHI. « An ontology for context-aware pervasive computing environments ». Dans : *The Knowledge Engineering Review* 18.03 (2003), p. 197–207.

[10] Joëlle COUTAZ et al. « Context is key ». Dans : *Communications of the ACM* 48.3 (2005), p. 49–53.

[11] Matthias BALDAUF, Schahram DUSTDAR et Florian ROSENBERG. « A survey on context-aware systems ». Dans : *International Journal of Ad Hoc and Ubiquitous Computing* 2.4 (2007), p. 263–277.

[12] Pascal COSTANZA et Robert HIRSCHFELD. « Reflective layer activation in ContextL ». Dans : *Proceedings of the 2007 ACM symposium on Applied computing*. ACM. 2007, p. 1280–1285.

[13] Sebastián GONZÁLEZ, Kim MENS et Patrick HEYMANS. « Highly dynamic behaviour adaptability through prototypes with subjective multimethods ». Dans : *Proceedings of the 2007 symposium on Dynamic languages*. ACM. 2007, p. 77–88.

[14] Malte APPELTAUER, Robert HIRSCHFELD et Tobias RHO. « Dedicated programming support for context-aware ubiquitous applications ». Dans : *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on*. IEEE. 2008, p. 38–43.

[15] Sebastián GONZÁLEZ, Kim MENS et Alfredo CÁDIZ. « Context-Oriented Programming with the Ambient Object System. » Dans : *J. UCS* 14.20 (2008), p. 3307–3332.

[16] Sebastian Andres GONZALEZ MONTESINOS et al. « Programming in ambience : gearing up for dynamic adaptation to context ». Thèse de doct. UCL., 2008.

[17] Herman HARTMANN et Tim TREW. « Using feature diagrams with context variability to model multiple product lines for software supply chains ». Dans : *Software Product Line Conference, 2008. SPLC'08. 12th International*. IEEE. 2008, p. 12–21.

[18] Robert HIRSCHFELD, Pascal COSTANZA et Michael HAUPT. « An introduction to context-oriented programming with ContextS ». Dans : *Generative and Transformational Techniques in Software Engineering II*. Springer, 2008, p. 396–407.

[19] Víctor LÓPEZ-JAQUERO et al. « Towards an extended model of user interface adaptation : the ISATINE framework ». Dans : *Engineering Interactive Systems*. Springer, 2008, p. 374–392.

[20] Jong-yi HONG, Eui-ho SUH et Sung-Jin KIM. « Context-aware systems : A literature review and classification ». Dans : *Expert Systems with Applications* 36.4 (2009), p. 8509–8522.

[21] Nurzhan NURSEITOV et al. « Comparison of JSON and XML Data Interchange Formats : A Case Study. » Dans : *Caine* 2009 (2009), p. 157–162.

[22] Carlo GHEZZI, Matteo PRADELLA et Guido SALVANESCHI. « Programming language support to context-aware adaptation : a case-study with Erlang ». Dans : *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM. 2010, p. 59–68.

[23] Tomoyuki AOTANI, Tetsuo KAMINA et Hidehiko MASUHARA. « Featherweight EventCJ : a core calculus for a context-oriented language with event-based per-instance layer transition ». Dans : *Proceedings of the 3rd International Workshop on Context-Oriented Programming*. ACM. 2011, p. 1.

[24] Charles-Eric DESSART, Vivian GENARO MOTTI et Jean VANDERDONCKT. « Showing user interface adaptivity by animated transitions ». Dans : *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM. 2011, p. 95–104.

[25] Sebastian Andres GONZALEZ MONTESINOS et al. « Subjective-C : Bringing Context to Mobile Platform Programming ». Dans : *International software language engineering conference*. 2011.

[26] Jens LINCKE et al. « An open implementation for context-oriented layer composition in ContextJS ». Dans : *Science of Computer Programming* 76.12 (2011), p. 1194–1209.

[27] Guido SALVANESCHI, Carlo GHEZZI et Matteo PRADELLA. « JavaCtx : seamless toolchain integration for context-oriented programming ». Dans : *Proceedings of the 3rd International Workshop on Context-Oriented Programming*. ACM. 2011, p. 4.

[28] Thibault PONCELET et Loıc VIGNERON. « The Phenomenal gem : Putting features as a service on rails ». Mém.de mast. Université catholique de Louvain, 2012.

[29] Guido SALVANESCHI, Carlo GHEZZI et Matteo PRADELLA. « ContextErlang : introducing context-oriented programming in the actor model ». Dans : *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM. 2012, p. 191–202.

[30] Nicolás CARDOZO ÁLVAREZ et al. « Identification and management of inconsistencies in dynamically adaptive software systems ». Thèse de doct. UCL, 2013.

[31] Sebastián GONZÁLEZ et al. « Context traits : dynamic behaviour adaptation through run-time trait recomposition ». Dans : *Proceedings of the 12th annual international conference on Aspect-oriented software development*. ACM. 2013, p. 209–220.

[32] Kim MENS, Nicolás CARDOZO ÁLVAREZ et Benoît DUHOUX. « A Context-Oriented Software Architecture ». Accepté pour le *COP'16 @ ECOOP 2016*, il sera publié dans l'*ACM Digital Library*. 2016.

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