MASTER EN SCIENCES INFORMATIQUES

Définition d'un DSML destiné à modéliser des applications mobiles multi-plateformes

Trapletti, Lisa

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016–2017

Définition d'un DSML destiné à modéliser des applications mobiles multi-plateformes

Lisa Trapletti

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent Englebert

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Depuis plusieurs années, le marché du mobile ne cesse de prendre de l'importance, entraînant ainsi une augmentation des applications smartphones sur les plateformes comme Google Play ou l'App Store. Lorsqu'une personne veut réaliser sa propre app, 2 possibilités s'offrent à elle. La première est de coder son application dans différents langages afin de couvrir les plateformes mobiles de son souhait. La deuxième possibilité est de se tourner vers le développement multi-plateforme. De nombreux outils existent sur le marché mais sont essentiellement destinés aux développeurs car trop techniques pour un public plus large. Pourtant, d'autres solution sont envisageables.
Dans ce mémoire, nous étudierons une solution capable de modéliser des applications mobiles multi-plateformes. Cette solution sera implémentée sous forme d'un DSML grâce à un outil de méta-modélisation appelé MetaDONE. Ce DSML sera chargé de reprendre tous les concepts d'une application mobile, afin de pouvoir en modéliser une le plus fidèlement possible. A la fin de ce mémoire, une critique sur le travail réalisé et sur MetaDONE sera proposée afin de cibler les points forts et les points faibles de la solution globale par rapport à la concurrence.

Abstract

For several years, the mobile market continues to grow up and more and more mobile applications are emerging. When someone wants to create his own app, he has 2 possibilities. The first one is to code his app with different programmatic languages in order to cover mobile plateformes of his wish. The second one is to turn to the mobile cross-platforms development. Many cross-platforms tools exist on the market but this ones are essentially for developers because they are too difficult to use for non-developers people. However, other solutions can be envisaged.
In this thesis, we will discover a solution able to modelize mobile cross-platforms apps, created with a meta-modelling tool called MetaDONE. This solution will be presented as a DSML in charge of representing all concepts of a mobile app. At the end of this thesis, a review of the implemented work and of MetaDONE will be done in order to focus on strengths ad weaknesses of the global solution comparing to the concurrence.

Remerciements

Je tiens à remercier mon promoteur, Monsieur Vincent Englebert, sans qui ce projet n'aurait pas eu lieu. Je le remercie pour le temps qu'il m'a consacré tout au long de ce travail ainsi que pour ses nombreux conseils dans la réalisation et la rédaction de ce mémoire.
Je remercie le corps professoral de l'Université de Namur pour m'avoir transmis leur savoir. Beaucoup m'ont inspiré dans ce mémoire et m'ont enseigné des disciplines que je mets en application quotidiennement dans ma vie professionnelle.
Enfin, je remercie toutes les personnes qui m'ont soutenue et épaulée tout au long de la rédaction de ce mémoire. Je remercie particulièrement ma famille proche, toujours présente pour me soutenir et m'encourager à donner chaque jour le meilleur de moi-même et à dépasser mes limites.
Merci à tous.

"Pour réussir il ne suffit pas de continuer, il faut toujours se dépasser"
Madeleine Ferron

Table des matières

Chapitre 1

Introduction

Depuis plusieurs années, le marché du mobile prend de plus en plus d'ampleur : la vente d'appareils mobiles ne cesse d'accroître d'années en années, devenant toujours de plus en plus importante [4, 19]. Petit à petit, le smartphone s'est imposé à nous et a littéralement changé notre manière de vivre, comme les ordinateurs ou encore internet ont pu le faire. Pour tout propriétaire d'un de ces bijoux de technologie devenu commun, il n'est pas pensable d'aller au travail sans son smartphone. De partir en vacances sans son smartphone. Bref, il n'est pas pensable de vivre sans son smartphone. Au fil des ans, cet appareil est devenu pour la plupart d'entre nous une nécessité : que ce soit pour lire ses mails, pour attendre le coup de fil d'un proche, pour prendre des photos, pour mesurer ses performances sportives, pour tuer le temps en jouant à « Candy Crush »... nous prenons toujours soin de garder notre smartphone près de nous. De part cette nouvelle manière de vivre, c'est donc sans surprise que le marché du mobile continue de prendre de l'importance.

Sur ce marché toujours en pleine croissance, 3 acteurs principaux se distinguent puisqu'ils couvrent la quasi totalité de celui-ci. Il s'agit d'Apple, de Google et de Microsoft. Chacune de ces entreprises possède sur le marché des appareils mobiles utilisant leur propre plateforme, leur système d'exploitation qui sont respectivement iOS, Android et Windows Phone. Malheureusement pour les développeurs, ces plateformes utilisent des langages complètement différents et nécessitent donc chacune un développement spécifique pour qu'une application couvre l'ensemble du marché. Pourtant, développer une version par plateforme en utilisant un langage différent est une option très coûteuse pour une entreprise aussi bien en terme d'argent que de temps ou de ressources. C'est pour répondre à ce besoin que les solutions de développement multi-plateforme ont fait leur apparition. Le but de celles-ci est de réaliser un développement unique de l'application et de couvrir un maximum de plateformes. Pour atteindre cet objectif, plusieurs types d'approches existent.

La première consiste à écrire l'application mobile dans un langage spécifique, tandis que l'outil choisi se chargera de « traduire » l'app dans le langage des autres plateformes mobiles. Cette approche est utilisée par des logiciels comme Xamarin ou encore PhoneGap, que l'on retrouve en majorité sur le marché du développement mobile multi-plateforme. Ce type de solutions est surtout réservé aux développeurs, puisque la connaissance d'un langage de programmation est nécessaire.

Une autre approche consisterait à modéliser l'application mobile de manière visuelle et de construire le code qui découle de cette modélisation de manière transparente. L'avantage est que la représentation visuelle ouvre la création d'applications mobiles à un public plus large, ne devant pas obligatoirement posséder des connaissances en développement, ce qui était le cas avec la première gamme d'outils. Le

but de ce travail sera donc de présenter une solution capable de modéliser visuellement une application mobile et d'en générer le code qui en découle.

1.1 Objectifs du mémoire

Pour atteindre notre objectif, nous utiliserons un MetaCASE tool [1] appelé MetaDONE, qui nous permettra de mettre en place une solution capable de modéliser une application mobile multi-plateforme sous la forme d'un DSML.

Le but principal de ce mémoire sera donc d'exploiter les caractéristiques de MetaDONE sur l'état de l'art afin d'apporter une contribution originale dans la conception d'un DSML, destiné à modéliser des applications mobiles. Cette contribution permettra par la suite de tirer des enseignements sur les futurs besoins à implémenter dans l'outil à cet égard.

Comme expliqué au début de ce chapitre, l'implémentation d'une solution capable de créer des applications mobiles multi-plateformes sera conçue sous la forme d'un DSML. Pour cela, il sera tout d'abord nécessaire de comprendre les exigences du développement multi-plateforme afin de cerner les aspects importants à ne pas manquer lors de l'implémentation. Il faudra également définir quelles sont les fonctionnalités offertes par MetaDONE dans le but de cerner la faisabilité d'une solution. Les langages existants et déjà implémentés dans l'outil (comme le réseau de Petri) permettront de découvrir les possibilités de MetaDONE. Après avoir tenu compte des exigences multi-plateformes et avoir implémenté une solution, le but sera de modéliser une application mobile compatible Android et iOS sur un smartphone.

Afin de critiquer la solution implémentée lors de ce travail, plusieurs outils concurrents à MetaDONE et à cette solution seront présentés. Le but de la critique sera donc double : analyser les forces et faiblesses de MetaDONE par rapport à d'autres logiciels de méta-modélisation mais également comparer le résultat du travail obtenu avec des outils multi-plateformes existants.

Enfin, les perspectives d'évolution émises à la fin de ce mémoire seront présentées comme des pistes d'amélioration de la solution construite avec MetaDONE. Ces pistes se baseront sur les obstacles et difficultés rencontrées lors du travail.

1.2 Méthodologie

A l'instant, nous avons introduit MetaDONE [1], le logiciel qui sera utilisé pour implémenter la solution. Celui-ci a été créé par Vincent Englebert à l'Université de Namur et permet aux ingénieurs software de créer leurs propres méta-modèles sous forme d'un DSML. Ce DSML (Domain Specific Modelling Language) est un langage de modélisation créé par une entreprise, un développeur... qui répond exclusivement à ses besoins et exigences. Il peut être utilisé pour modéliser tout ce que l'on souhaite : une interface utilisateur, un produit business, un système de contrôle... En clair, cela signifie que les ingénieurs peuvent, grâce à MetaDONE, créer leur propre langage chargé de modéliser un domaine ou encore un contexte qui leur est propre. Durant ce travail, le logiciel va permettre de créer un DSML capable de modéliser des applications mobiles multi-plateformes.

L'implémentation de cette solution sous la forme d'un DSML comportera plusieurs étapes. La première consistera à définir un méta-modèle capable de gérer les aspects les plus importants d'une

[1] Pour plus d'informations sur MetaDONE, veuillez consulter le chapitre 3 entièrement consacré à la structure du logiciel et à son fonctionnement

application mobile [2] et de l'intégrer dans MetaDONE qui pour rappel, est un MetaCASE tool. D'après la référence *Meta-CASE Technology* [1], « les MetaCASE tools sont utilisés pour générer des CASE tools ». Le terme CASE (*Computer Aided Software Engineering*) quant à lui, a fait son apparition dans les années 90 [11]. Les CASE tools sont des outils permettant de créer des modèles, en utilisant un ensemble de concepts, de règles et de notations capables de modéliser ce que les ingénieurs souhaitent. Cet ensemble de contenu représente ce qu'on appelle une méthode. Celle-ci est souvent basée sur l'UML et permet de définir comment les modèles peuvent être construits, vérifiés et/ou analysés mais également de définir quel en serait le code généré. Cependant, un problème majeur est présent dans cette catégorie d'outil : la méthode contenant tous les concepts, notations... disponibles est hard-codée. Les fonctionnalités proposées sont fixes et uniquement modifiables par le vendeur du CASE tool. Il ne faut pas oublier que chaque projet est différent et chaque organisation est différente, ce qui implique un ensemble de concepts spécifiques à leur contexte. Par exemple, une application web ne peut pas être modélisée de la même manière qu'une application mobile. C'est donc là la faiblesse du CASE tool : le même ensemble de concepts devait être utilisé pour construire différents projets menés dans différents contextes, ce qui ne correspondait pas toujours aux besoins des ingénieurs software.

C'est donc pour répondre à ces problèmes que les MetaCASE tools, comme MetaDONE, sont apparus. Ces outils sont en mesure de prendre en considération les exigences spécifiques des ingénieurs software afin de construire un méta-modèle qui leur est propre et ensuite, de générer le CASE tool qui en découle automatiquement. Nous revenons donc à notre définition de départ : le MetaCASE tool a pour rôle de générer un CASE tool mais adapté à un contexte précis. Si les spécifications du système sous développement évoluent, le méta-modèle ainsi que le CASE tool seront mis à jour. Cela signifie que cet outil n'impose pas de limites aux ingénieurs software dans la création de leurs solutions puisqu'ils fournissent eux-même à l'outil les concepts dont ils auront besoin. Le MetaCASE tool rend cette « customisation » possible car, contrairement au CASE tool, le méta-modèle n'est pas hard-codé mais stocké comme un ensemble de données pouvant être modifiées. La partie hard-codée présente dans l'outil s'appelle le langage de méta-modélisation : c'est avec celui-ci que le développeur va pouvoir créer, adapter et corriger son méta-modèle. Pour revenir à notre contexte, nous utiliserons le langage de méta-modélisation de MetaDONE pour fournir à l'outil notre méta-modèle reprenant les concepts nécessaires pour créer une application mobile.

Une fois le méta-modèle défini, la deuxième étape de l'implémentation consistera à représenter visuellement tous les concepts que le méta-modèle contient, toujours grâce aux possibilités offertes par MetaDONE.

L'implémentation de la solution terminée, le logiciel sera en mesure de créer des modèles d'applications mobiles en utilisant le DSML créé durant ce travail. Les possibilités fournies par le DSML seront alors illustrées en créant le modèle d'une application mobile de référence dont les spécifications sont fournies ci-après.

1.3 Problématique de référence

Afin d'établir un contrat de départ, cette application mobile de base doit reprendre la majorité des points que devra satisfaire le DSML.

Nous baptiserons cette application « Smart Namur ». Celle-ci sera chargée de présenter l'ensemble des commerces, restaurants, événements et lieux culturels de la ville de Namur. Par le biais d'un menu, l'utilisateur pourra accéder aux différentes catégories des références namuroises. Celles-ci seront

[2] Ces aspects seront définis sur base de recherches menées dans l'état de l'art

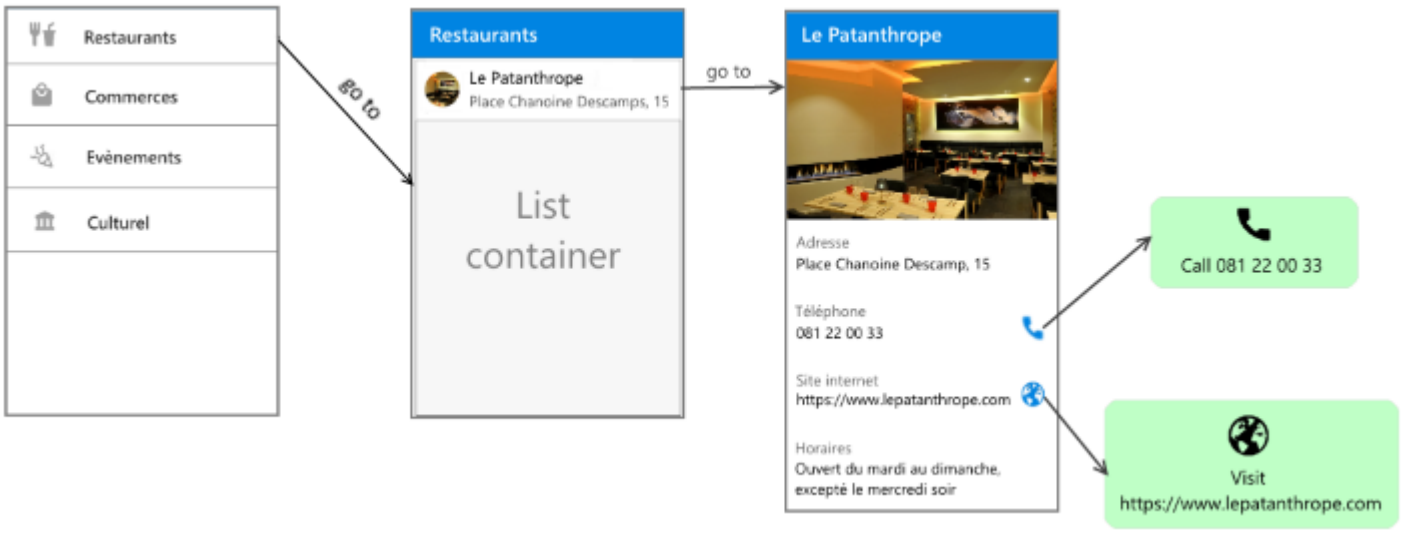

FIGURE 1.1 : Application mobile de référence

présentées sous forme de listes, reprenant tous les éléments de cette catégorie. L'utilisateur pourra alors, s'il le souhaite, voir le détail de chaque référence en cliquant dessus et en accédant à un nouvel écran.

Dès que l'utilisateur accède au détail d'un élément, il peut visualiser les informations importantes de ce dernier (Numéro de téléphone, site internet, heures/jours d'ouverture...). Si le lieu possède un site internet, l'utilisateur pourra accéder à celui-ci par le biais de l'application native du smartphone. Si le lieu possède un numéro de téléphone (comme les restaurants), l'utilisateur aura la possibilité d'utiliser directement l'application Téléphone native au smartphone.

Toutes les données de l'application seront téléchargées d'une source extérieure et stockées dans une base de données locale. Si le smartphone ne possède pas de connexion réseau, l'application affichera un message d'erreur. Sinon, il mettra à jour les données via internet.

Le prototype de la figure 1.1 représente le visuel que notre DSML devra être capable de reproduire dans une première phase. Dans une seconde phase, qui ne sera pas abordée dans ce mémoire, il devra être en mesure de générer le code qui en résulte.

Chapitre 2

Etat de l'art

Dans ce chapitre, nous allons découvrir les différents types de développement multi-plateforme qui existent à l'heure actuelle. Nous nous pencherons ensuite sur certaines de ces solutions, pour lesquelles l'approche peut s'avérer intéressante pour la suite du travail. Enfin, nous terminerons en tirant certaines conclusions de ces recherches, qui seront à prendre en compte lors de la création du DSML avec MetaDONE.

2.1 Les différents types d'applications mobiles

A l'heure actuelle, on distingue 3 types d'applications mobiles : les applications natives, les applications hybrides et les applications web.

Les **applications natives** sont codées dans le langage propre à chaque plateforme. Elles sont plus performantes que les applications web ou hybrides et permettent d'accéder facilement aux ressources de l'appareil (GPS, microphone...). Cependant, le développement de ces applications doit se faire pour chaque plateforme, ce qui rend le développement beaucoup plus lent et complexe. En effet, puisque l'environnement de chaque plateforme est complètement différent, l'équipe de développement doit être composée de profils très différents. De plus, cette hétérogénéité implique une mise à jour et une maintenance de l'application non pas pour une, mais pour plusieurs plateformes. Autrement dit, même si les applications natives représentent un avantage certain en terme de performance et d'authenticité, elles restent très coûteuses par rapport à d'autres solutions comme les applications web ou hybrides dans le cadre du développement multi-plateforme.

Les **applications web** sont des applications multi-plateformes codées en HTML5, CSS3 et JavaScript. Elles ont pour plus grand avantage leur aspect multi-plateforme qui permet de développer l'application de manière unique, contrairement aux applications natives. Ces applications web ne sont pas soumises sur les stores (Play Store, App Store...) et sont accessibles via un navigateur, comme n'importe quel site. Celles-ci ont néanmoins 2 défauts majeurs. Le premier est que l'application exige une connexion internet permanente. En effet, puisque celle-ci est en réalité un site internet, la connexion est obligatoire. Cela entraîne une diminution de la performance par rapport aux applications natives. Le second problème majeur des applications web est qu'elles n'ont pas la possibilité d'accéder aux ressources de l'appareil, ce qui limite grandement leurs actions. De plus, puisque ces applications sont multi-plateformes, leur ergonomie ne peut pas satisfaire les contraintes propres à chaque environnement

[1].

Enfin, les **applications hybrides** sont dérivées des applications web : celles-ci utilisent majoritairement les technologies HTML5, CSS3 et JavaScript. Cependant, elles utilisent également une partie native afin de pouvoir accéder aux ressources du téléphone. Le problème de la performance et de l'ergonomie restent tout de même toujours présents.

2.2 Les différentes approches multi-plateformes

Dans ce chapitre, nous allons découvrir plusieurs solutions répondant à la demande des applications multi-plateformes. En premier lieu, nous nous attarderons sur le méta-langage proposé par MetaEdit+. MetaEdit+ est un metaCASE [2], comme MetaDONE, capable de générer des DSML suivant les concepts spécifiques d'un domaine. Ensuite, nous découvrirons USIXML. Il s'agit d'un langage de description d'interfaces utilisateurs, capable de définir une vue unique pour différents contextes d'utilisation. Nous poursuivrons avec IFML et l'approche multi-plateforme de 3 chercheurs de la Polytechnique de Milan, basée sur ce standard. Nous verrons par après Xamarin, qui est l'outil le plus utilisé mais est également la solution la plus aboutie dans le développement multi-plateforme d'applications natives sur le marché professionnel. C'est pourquoi nous avons choisi de l'étudier dans cet état de l'art. Ensuite, nous découvrirons MD2, une solution model-driven, utilisant un DSML (*Domain Specific Modelling Language*) pour créer des applications Android et iOS. Nous découvrirons également Native-2-Native, une solution générant du code multi-plateforme en s'inspirant d'informations trouvées sur le web. Enfin, nous terminerons avec XIS-Mobile, permettant de créer le modèle d'une application sous forme d'UML. Toutes ces approches sont reprises dans l'état de l'art car elles apportent toutes une perspective différente sur le problème du développement multi-plateforme, ce qui permet non seulement de repérer les éléments communs entre ces approches mais également de tirer le meilleur de chacune d'entre elles.

2.2.1 MetaEdit+

MetaEdit+ ([18, 25]) est un outil capable de générer un DSM répondant à des règles et des concepts définis par l'utilisateur. De plus, cet outil est capable de générer le code à la place de l'utilisateur en reliant un générateur au DSM construit. L'utilisateur devra donc simplement définir le mapping entre les propriétés/concepts rencontrés dans le langage et le code à générer. Pour arriver à une telle solution, MetaEdit+ a créé son propre méta-langage, reprenant les différents concepts possibles à utiliser dans l'outil. De la sorte, le logiciel propose à l'utilisateur une gamme d'objets (des symboles, des propriétés, des règles...) qui lui permettront de créer son propre DSM.

La solution multi-plateformes mobile proposée par MetaEdit+

Par défaut, MetaEdit+ propose un ensemble de DSML que l'utilisateur peut manipuler pour créer sa propre solution. Parmi ceux-ci se trouvent un DSML spécialisé dans la conception d'interfaces utilisateurs mobiles ([26, 15]), plus particulièrement pour les téléphones S60. Le langage proposé par

[1] Il faut savoir que chaque plateforme mobile possède ses propres conventions ergonomiques. Il est aisé de constater que les applications sous Android 5.0 sont complètement différentes des applications iOS. Les conventions d'iOS sont disponibles à l'url suivante : `https://developer.apple.com/ios/human-interface-guidelines/` et celles d'Android à cet endroit :`https://material.io/guidelines/` pour les conventions d'Android

[2] Voir le chapitre 1.2 pour plus d'informations sur les metaCASE

MetaEdit+ est plutôt orienté sur la logique et la navigation entre les écrans plutôt que sur le design de leur interface. Plusieurs catégories d'objets ont été créées :

- Les *start & stop*, qui symbolisent le début et la fin d'une application. A l'heure actuelle, si une application doit obligatoirement posséder un écran de démarrage, elle n'a pas de fin à proprement parler. Si l'utilisateur veut quitter l'application, il doit utiliser les boutons de l'appareil (Bouton « Home », ou bouton d'arrêt).

- Les *UI Controls* (formulaires, listes...), qui s'affichent sur l'ensemble de l'écran et possèdent leur propre structure ainsi que leur propre comportement.

- Les *widgets*, qui représentent différentes fenêtres disponibles dans l'application.

- L'objet *condition*, qui permet d'ajouter de la logique au niveau des liens de navigation entre écrans.

- Les *phone services*, qui représentent les différentes ressources et services de l'appareil. Cela inclut l'accès aux fichiers stockés sur l'appareil, le service SMS....

- Les objets *navigation flow*, qui définissent les différents types de relation qu'il peut exister entre les écrans.

La figure 2.1 [3] liste l'ensemble des objets disponibles dans le DSML de MetaEdit+. La figure 2.2, quant à elle, est un exemple d'application mobile réalisée avec MetaEdit+. Cette application permet de s'enregistrer à une conférence et de visualiser le programme de celle-ci. La référence [26] fournit plus d'informations sur cet exemple et sur les possibilités de MetaEdit+.

2.2.2 USIXML - User Interface eXtensible Markup Language

USIXML est un langage de description d'interfaces utilisateurs (UIDL) [29], permettant de définir une vue pour plusieurs contextes d'utilisation. Les vues générées par cet outil peuvent être indépendantes du type d'appareil (TV, smartphone...), du type de plateforme (Desktop, web...), ou encore du type de modalité (Interaction graphique, vocale...). USIXML est principalement destiné à un public de personnes qui ne développent pas (Designers, managers...) afin de séparer la réflexion du design de l'application de son implémentation. Dans cette section, nous découvrirons le framework Cameleon, sur lequel USIXML se base pour créer ses applications multi-contextes. Nous verrons ensuite comment est structuré USIXML et quel est le rôle de chaque élément dans cette structure. Enfin, nous listerons les différents outils qui composent USIXML.

Le framework Cameleon

La plupart des développements proposant la création d'interfaces utilisateurs multi-contextes reposent sur le framework Cameleon. Ce framework fournit 4 étapes de développement à respecter lors de la conception d'une application multi-contexte. Ces étapes sont :

[3] Les illustrations des figures 2.1 et 2.2 sont toutes deux tirées du rapport technique fourni par MetaEdit+ (Voir [26])

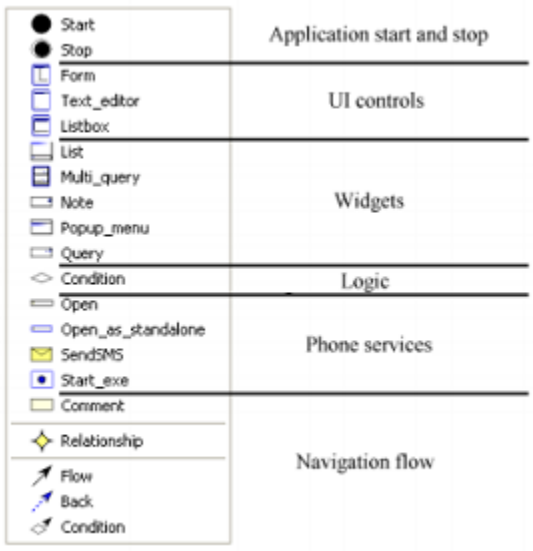

FIGURE 2.1 : Concepts repris dans le DSML de MetaEdit+

- L'*interface utilisateur finale (FUI)*, chargée de représenter une interface graphique utilisée dans un contexte particulier. Chaque contexte d'utilisation (Plateforme, appareil...) sera représenté par une FUI.
- L'*interface utilisateur concrète (CUI)* dont le rôle est d'abstraire une FUI, afin de fournir une interface qui soit indépendante de toute plateforme. La CUI utilise des *objets d'interaction concrets (CIO)*. Ces objets ont pour but de concrétiser une interface utilisateur abstraite (AUI) en symbolisant le layout et la navigation de l'interface dans un contexte d'utilisation spécifique.
- L'*interface utilisateur abstraite (AUI)*, qui définit des espaces d'interaction mais aussi le schéma de navigation entre ces espaces. Elle sélectionne des objets d'interaction abstraits (AIO) afin de représenter chaque concept sans se lier à un contexte particulier. L'AUI abstrait une CUI, afin de la rendre indépendante de toute forme de modalité d'interaction.
- *Les tâches et les concepts*, qui est un ensemble reprenant toutes les tâches devant être exécutées dans l'application mais aussi tous les modèles, les objets du domaine nécessaires dans l'exécution de ces tâches. L'ensemble de ces objets représente tous les concepts pouvant être manipulés par l'application.

Le framework Cameleon utilise 3 types de transformations, afin de transiter d'une étape à une autre. Ces 3 transformations sont :

- L'*abstraction*, permettant de passer à un niveau supérieur d'abstraction ;
- La *réification*, permettant de passer à un niveau supérieur de concrétisation des éléments d'une interface utilisateur ;

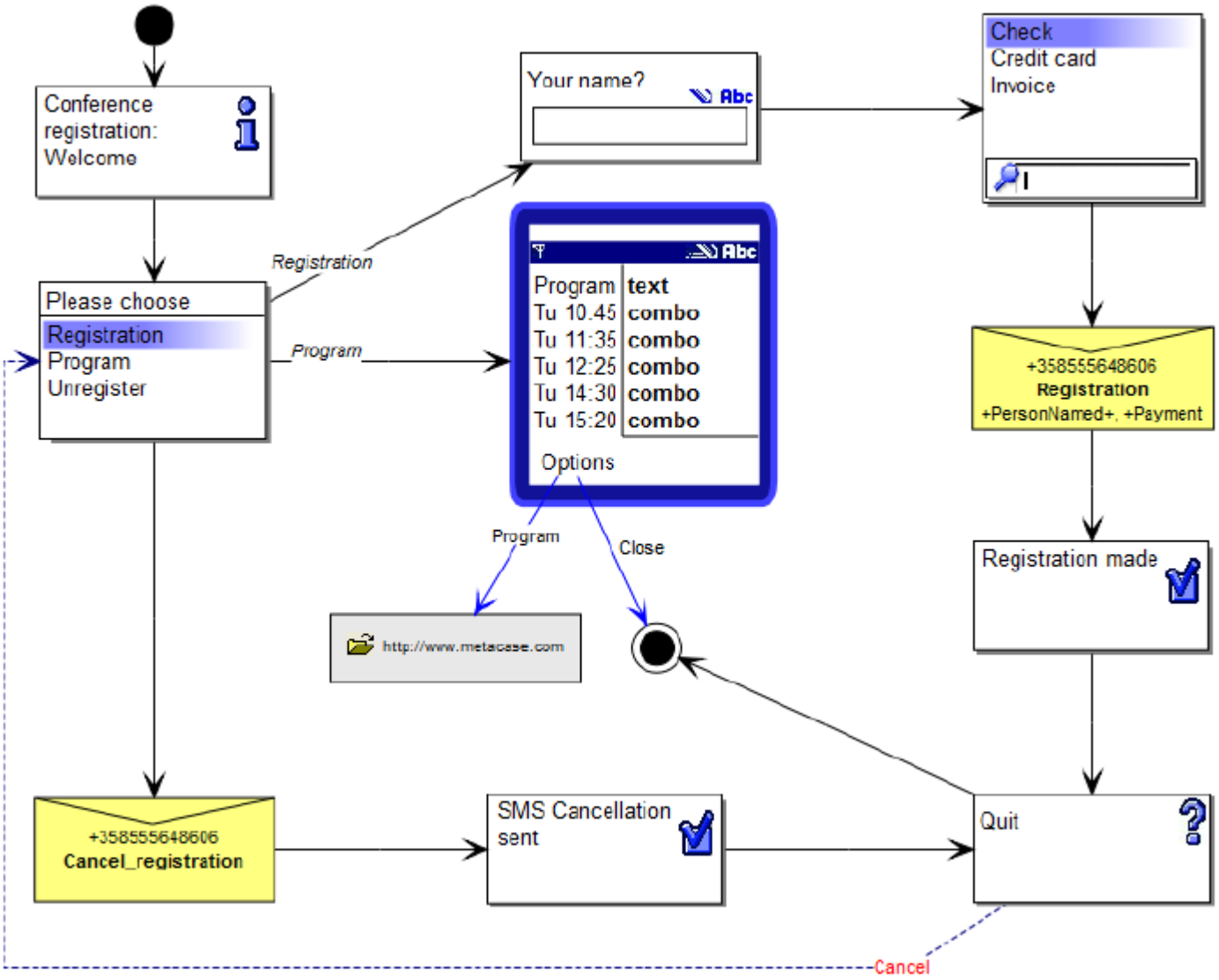

FIGURE 2.2 : Modélisation d'une application mobile avec MetaEdit+

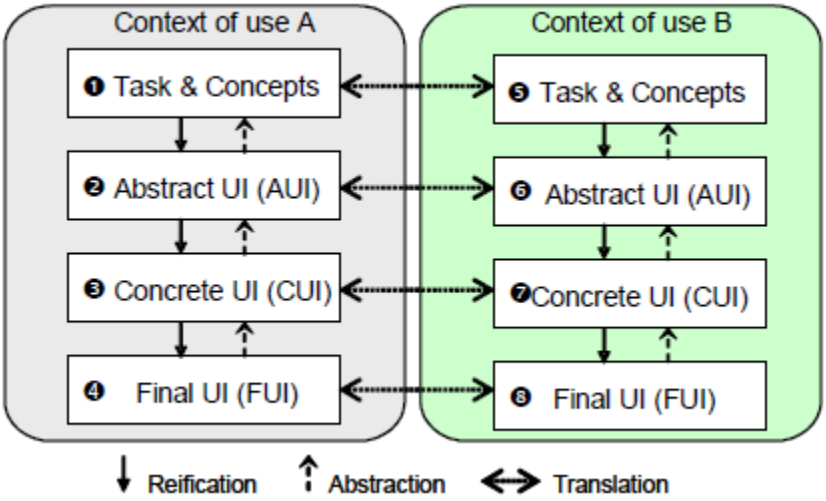

FIGURE 2.3 : Illustration de la mise en application du framework Cameleon

- La *traduction*, permettant de « traduire » l'interface utilisateur d'un contexte d'utilisation en une autre interface représentant un contexte différent.

La figure 2.3 [4] illustre le framework Caméléon utilisé dans 2 contextes d'utilisation.

Le contenu d'USIXML

Afin de créer des applications multi-contextes, USIXML possède un ensemble de modèles basiques. 2 d'entre eux, le modèle AUI et le modèle CUI correspondent à l'AUI et au CUI du framework Cameleon. Pour les autres modèles, USIXML les définit comme suit :

- Le *modèle de tâches* reprend l'ensemble des tâches réalisées et/ou perçues par l'utilisateur final. Plus précisément, ce modèle représente le découpage des tâches en sous-tâches et relient celles-ci entre elles. Chaque tâche possède un nom, une fréquence mais également un type d'action permettant d'affiner le rôle de la tâche.

- Le *modèle de domaine* reprend tous les concepts du monde réel qui seraient manipulés par l'utilisateur, ainsi que leurs interactions. Il est en tout point similaire à un diagramme de classe.

- Le *modèle de mapping* reprend un ensemble de relations permettant de lier des modèles ou les éléments de modèles entre eux. Ces relations sont de différents types :

  * *Manipulates* : lie une tâche à un concept du modèle de domaine. Ce lien démontre qu'un concept est lié à l'exécution d'une tâche.

[4] Cette illustration est tirée d'un article dédié à UsiXML (Voir [13])

  * *Is rendered by* : lie un concept du domaine à un élément graphique. Ce lien spécifie si le concept du domaine doit être « influencé » par une entrée de l'utilisateur ou si il est tout simplement affiché à l'écran.
  * *Is Executed by* : lie une tâche à un AUI ou à un CUI. Cette relation montre qu'une tâche est exécutée par un AUI ou un CUI.
  * *Is Abstract into/Is reified to* : définit une relation d'abstraction/de réification entre un AUI et un CUI.
  * *Has context* : lie un élément du modèle à un contexte d'utilisation.
  * *Corresponds to* : lie une relation temporelle d'une tâche à une relation de navigation.

- Le *modèle contextuel* décrit tous les éléments pouvant influencer le déroulement d'une tâche et est composé de 3 modèles. Premièrement, il possède un modèle utilisateur, découpant les utilisateurs en différentes catégories, suivant des « stéréotypes ». Ces catégories d'utilisateurs réaliseraient la tâche demandée différemment. Deuxièmement, il doit définir un modèle de plateformes, décrivant les différentes plateformes sur lesquelles la tâche pourrait être réalisée. Ces plateformes peuvent concerner des tailles d'écrans différentes, des résolutions d'écran.... Enfin, le modèle contextuel doit être composé d'un modèle d'environnement, décrivant les différentes caractéristiques environnementales dans lesquelles l'utilisateur pourrait utiliser l'application (Lumière, bureau...).

- Le *modèle d'interface utilisateur* correspond à la superclasse des modèles énoncés ci-dessus. Le modèle UI est chargé de contenir l'ensemble des caractéristiques utilisées par tous les modèles.

Les outils d'USIXML

En plus d'une série de modèles, USIXML met plusieurs outils à disposition, afin de gérer plus intuitivement les applications. Un de ces outils se nomme *ReversiXML* et permet de transformer une page HTML existante en langage USIXML. De cette manière, il est possible de travailler avec USIXML sur une interface existante. Ensuite, il existe *GrafiXML*, qui permet de gérer de manière graphique les CIO et leurs propriétés. Grâce à GrafiXML, le designer peut directement déplacer ses éléments dans leur interface et éditer leurs propriétés dans un panneau d'informations. Enfin, il y a *TransformiXML*, basé sur un outil de grammaires graphiques attribuées (AGG). TransformiXML est chargé de symboliser l'ensemble des transformations entre les modèles d'USIXML, sous forme d'un graphe.

2.2.3 L'approche multi-plateforme sur base d'IFML

Cette approche multi-plateforme a été réalisée par 3 chercheurs de la Polytechnique de Milan : Eric Umuhoza, Andrea Mauri et Marco Brambilla. Cette solution a été réalisée en étendant les capacités du standard IFML (*Interaction Flow Modelling Language*). Pour comprendre ce qu'ont apporté les chercheurs au langage IFML existant, nous allons tout d'abord décrire les capacités d'IFML, qui s'adresse plutôt à des applications web ou desktop. Ensuite, nous découvrirons comment les 3 chercheurs ont adapté IFML au contexte particulier des applications mobiles.

IFML

IFML est un langage standard de l'OMG (Object Management Group), chargé de modéliser les interfaces utilisateurs d'applications front-end, indépendamment du support utilisé (Ordinateur, smartphone...). Cependant, IFML ne se charge pas de représenter le style ou le look&feel de l'application : il représente les différents éléments graphiques et les flux d'information qui composent l'application de manière à couvrir la description de l'ensemble des plateformes. Un diagramme IFML (illustré dans la figure 2.4 [5]) est capable de créer différentes perspectives design pour une même application :

- La *spécification de la structure de la vue*, qui consiste à créer des containers (view containers dans IFML) et à les relier entre eux ;
- La *spécification du contenu de la vue*, qui consiste à créer des composants (view components dans IFML) représentant des données et à les placer dans les containers ;
- La *spécification des évènements*, chargée de définir les évènements pouvant influencer les interfaces utilisateurs. Ces évènements peuvent être déclenchés par l'utilisateur ou par des éléments extérieurs ;
- La *spécification des transitions d'évènements*, qui décrit les conséquences d'un évènement sur les interfaces utilisateurs ;
- La *spécification des paramètres*, qui consiste à définir les paramètres entrée-sortie existants entre des composants ou entre des composants et des évènements ;
- La *spécification d'actions déclenchées par les évènements de l'utilisateur*. L'effet d'un évènement est illustré sous forme d'un flux d'interaction entre l'évènement et le container ou le composant affecté. Ce flux représente un changement d'état de l'interface utilisateur.

Pour représenter une application front-end, IFML dispose donc des éléments suivants :

- Des *containers ou view containers*, chargés de contenir les pages web ou l'interface d'une fenêtre ;
- Des *composants visuels ou view components*, qui représentent des éléments graphiques chargés de contenir des données (formulaires, listes...) ;
- Des *évènements*, qui relient les composants visuels et les containers entre eux ;
- Des *paramètres entrée-sortie*, passés entre les containers ou les composants visuels.

Le travail des chercheurs de la Polytechnique de Milan

Dans leur étude [2], Eric Umuhoza et ses congénères expliquent que les interfaces utilisateurs des applications mobiles sont beaucoup plus complexes en terme d'interactions puisqu'elles doivent « compacter » leur interface dans un espace plus petit. Pour arriver à cela, elles doivent généralement utiliser une barre de navigation, un menu contextuel.... De plus, ces applications sont susceptibles d'utiliser les fonctionnalités de l'appareil comme le GPS ou l'appareil photo. Et ces aspects, spécifiques au mobile, ne peuvent pas être représentés dans les diagrammes IFML. Les 3 chercheurs ont donc décidé d'étendre

[5] Les illustrations des figures 2.4 et 2.5 sont toutes deux tirées de l'article d'Eric Umuhoza (Voir [2])

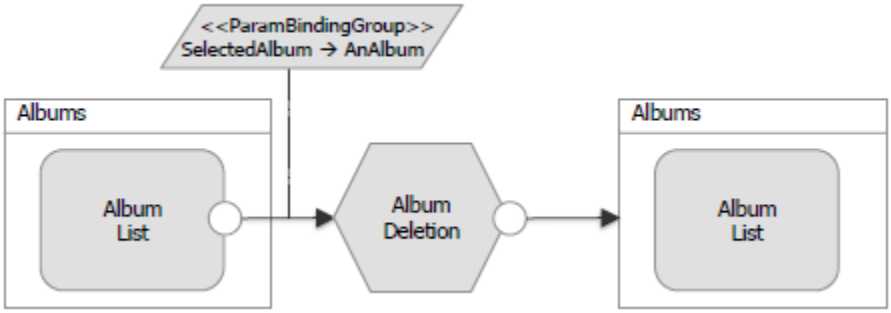

FIGURE 2.4 : Exemple d'un diagramme IFML

les capacités d'IFML, afin de fournir une solution spécifique au contexte du mobile. Pour cela, ils ont défini de nouveaux objets implémentant les composants existants. Tout d'abord, 3 nouveaux types d'évènements ont été créés :

- Les évènements déclenchés par les gestes de l'utilisateur comme un long clic, un glissement de gauche à droite... ;
- Les évènements déclenchés par l'appareil, comme le niveau de batterie faible, la perte de connexion... ;
- Les évènements déclenchés par l'utilisateur en utilisant les fonctionnalités de l'appareil comme prendre une photo...

En plus de ces nouveaux évènements, 3 nouvelles classes étendant le comportement du container ont été créées :

- L'écran (*Screen*) qui est le container principal de l'application mobile ;
- La barre de navigation (*Toolbar*) qui est un sous-container utilisé dans la majorité des applications, capable de stocker une liste d'évènements à déclencher ;
- Le système mobile (*MobileSystem*) représentant un container utilisé par le système d'exploitation de l'appareil et mis à jour par l'application (Ecran de notifications...)

A côté de ces containers, les composants visuels ont également été adaptés pour le contexte mobile : les éléments graphiques utilisés dans les applications mobiles comme les boutons, les images, les listes... possèdent chacun leur propre composant visuel, étendant celui d'IFML. Ces composants peuvent déclencher des évènements utilisateur.

Dans la figure 2.5, les chercheurs de la Polytechnique de Milan ont réalisé le modèle d'une application permettant de rechercher un produit particulier dans une liste, soit en utilisant une recherche classique, soit en utilisant la recherche vocale de l'appareil.

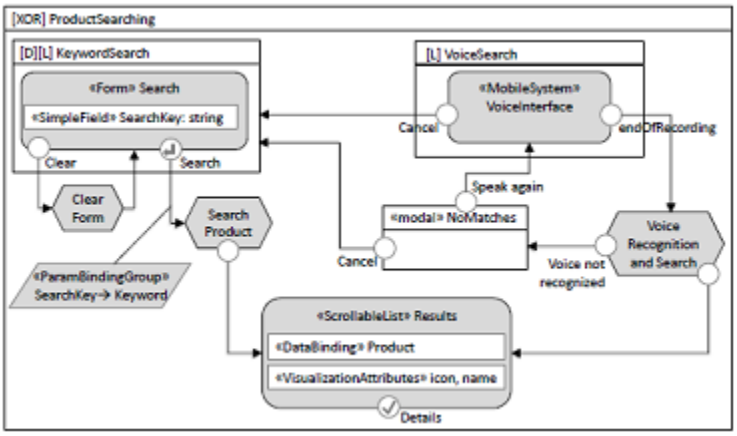

FIGURE 2.5 : Schéma illustrant la recherche d'un produit réalisé par la Polytechnique de Milan

Pour avoir une solution multi-plateforme aboutie, Eric Umuhoza et ses collègues ont également créé un générateur de code. Celui-ci est capable de transformer un schéma entités-associations afin de connaître les objets utilisés par l'application et de convertir un diagramme IFML en une application HTML5, CSS3 et JavaScript, basée sur l'architecture d'Apache Cordova[6], ou anciennement PhoneGap. Le générateur de code produit un dossier compressé qu'il faut simplement soumettre sur la plateforme en ligne d'Apache Cordova afin d'obtenir l'apk de l'application Android et l'ipa de l'application iOS.

2.2.4 Xamarin

Xamarin est un environnement de développement permettant de créer des applications mobiles compatibles Android, iOS et Windows Phone, en utilisant le langage C# ou le langage Java. Les applications créées avec cet environnement sont natives et donc retranscrites en Java pour Android, en Objective-C pour iOS et en C# pour Windows Phone.

Comme expliqué précédemment, les applications natives restent les plus intéressantes en terme de performance et d'ergonomie. Elles sont néanmoins très longues à développer. Les applications web multi-plateformes quant à elles, sont plus rapides à développer mais ne permettent pas l'accès aux ressources de l'appareil mobile et ne respectent pas l'ergonomie de chaque plateforme. Quand nous nous penchons sur la création d'une application mobile, nous pouvons très vite nous apercevoir que certains aspects de l'application (indépendamment du langage de programmation) sont communs à chaque plateforme. Par exemple, les entités manipulées, les choix de création ou non d'une DB locale, la communication avec un web service.... Afin de respecter l'unicité de l'application, tous ces choix doivent être communs pour chaque plateforme. Il est donc aisé de constater que chaque application possède une base commune, même si l'ergonomie de la vue et les caractéristiques spécifiques aux plateformes (Ressources de l'appareil, moteur de cartographie...) diffèrent selon les environnements. C'est sur cette base que Xamarin a décidé de faire reposer sa solution multi-plateforme. Comme il est illustré dans la figure 2.6 [7], le code partagé de l'application se trouve dans une librairie partagée entre

[6]https ://cordova.apache.org/

[7]Illustration tirée de la documentation de Xamarin : `https://developer.xamarin.com/guides/cross-platform/`

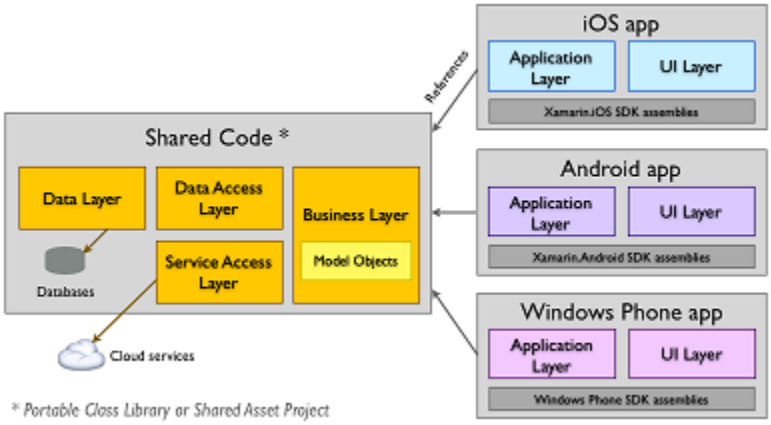

FIGURE 2.6 : Représentation d'un application multi-plateforme avec Xamarin

les 3 différentes plateformes. Cette librairie regroupe :

- la couche Données, qui s'occupe de créer la DB locale ;
- la couche d'accès aux données, qui permet d'accéder et d'interagir avec la DB ;
- la couche d'accès aux services, chargée d'accéder aux ressources distantes comme les web services ;
- la couche Business, regroupant l'ensemble des classes « modèles ».

Chaque plateforme utilise ensuite cette librairie commune et définit non seulement sa propre vue mais utilise aussi ses propres spécificités (Couche application). Chaque vue est réalisée de manière « native » : our Android, celle-ci est construite à base de fichiers XML ; iOS utilise des storyboards ou des fichiers XIB et enfin, Windows Phone utilise le framework XAML. Avec cette structure, il est évidemment préférable d'utiliser un modèle en couches, afin de reprendre un maximum d'aspects dans le code partagé.

2.2.5 MD2 Développement mobile orienté modèle (model-driven)

MD2 est un framework développé par le département des systèmes d'information de l'université de Münster. Il se présente comme un plug-in d'Eclipse et permet de créer des applications business natives pour Android et pour iOS. Afin de modéliser des application, MD2 utilise un DSL (*Domain Specific Language*). Contrairement au DSML, le DSL ne fournit pas une représentation visuelle : il se présente sous la forme d'un nouveau langage à utiliser, tel que le Java. Cependant, ce langage est, comme les DSML, spécifique aux besoins et exigences de la personne qui l'a construit. Dans le cas de MD2, le DSL a donc pour rôle de représenter une application mobile. Cette solution a donc abstrait chaque concept spécifique du développement mobile sur Android et sur iOS et permet aux générateurs de construire des applications natives sur base de celle écrite avec le DSL.

`application_fundamentals/shared_projects/`

Le but du projet MD2 était d'arriver à une solution multi-plateforme capable de créer des applications business. Pour cela, les concepteurs de MD2 ont travaillé en collaboration avec des partenaires de l'industrie, des entreprises afin de cibler leurs besoins les plus importants lors de la conception d'une application mobile. Premièrement, les entreprises et leurs clients finaux attendaient des applications en accord avec l'ergonomie de la plateforme. Cette exigence a donc exclu les applications web et hybrides, pour se pencher vers la conception d'applications natives. Deuxièmement, le monde business, en ce temps, avait au minimum besoin d'applications Android et iOS. MD2 s'est donc concentré sur la conception d'applications mobiles couvrant ces 2 plateformes. Enfin, certaines opérations à réaliser dans les applications étaient jugées nécessaires par les partenaires de l'industrie. Il s'agissait de :

- Définir des types de données et d'accéder à ces dernières aussi bien sur une DB locale que sur un serveur distant, et de réaliser des opérations CRUD (Create - Read - Update - Delete) sur celles-ci ;
- D'implémenter des interfaces utilisateurs avec différents layouts et plus particulièrement des vues composées de tableaux et de listes ;
- Contrôler la séquence des vues de l'interface utilisateur ;
- Utiliser les fonctionnalités de l'appareil (GPS, Microphone...) ;
- Réagir à des évènements et à des changements d'état (Clic sur bouton, perte de connexion...) ;
- Définir des validations d'entrées.

Au niveau du fonctionnement général de MD2, la création d'une application native se déroule en trois phases. La première consiste à définir le modèle de l'application suivant le DSML de MD2. La deuxième phase représente le moment où les générateurs de code de chaque plateforme transforment le modèle créé en code source. Cette conversion est réalisée à chaque fois que le développeur sauve son modèle. Ces générateurs vont donc générer les applications en Java (et XML pour les interfaces graphiques) en Android et en Objective-C pour iOS, mais généreront également les fichiers nécessaires à la compilation de l'application dans Eclipse[8] et dans XCode[9]. La troisième phase consiste à compiler l'application générée dans l'environnement de développement de chaque plateforme. Cette étape est nécessaire afin de créer un fichier binaire exécutable nommé APK via Eclipse pour Android, et afin de valider l'application auprès d'ITunes Connect en passant par XCode.

Enfin, il est utile de préciser que les applications créées par le framework sont structurées selon le pattern MVC (*Model - View - Controller*), ce qui permet de découper le code généré correctement et d'assurer sa maintenabilité. De plus, cela permet de réutiliser certains composants en différents endroits de l'application.

2.2.6 Native-2-Native

Native-2-Native [3] est une solution développée par le département des sciences informatiques de la Virginia Tech, sous la forme d'un plug-in à intégrer dans l'IDE Eclipse. Cette approche est capable de « traduire » automatiquement du code provenant d'une application Android, écrite en Java, en

[8]Environnement de développement de Java et anciennement d'Android. A noter qu'à présent, l'environnement de développement d'Android est Android Studio, basé sur l'environnement IntelliJ

[9]Environnement de développement d'iOS et des applications MAC

code compatible iOS, c'est-à-dire en Swift [10]. Native-2-Native se concentre sur la génération de code utilisant les sensors et services fournis par l'appareil (Caméra, localisation...) et pas sur la génération complète d'une application mobile. Le but principal des chercheurs de la Virginia Tech était de faciliter les tâches les plus communes dans le développement mobile.

Pour synthétiser l'application mobile, la solution va se baser sur les ressources disponibles au travers du web. D'après les chercheurs, les développeurs utilisent, de nos jours, énormément le web au cours de leur développement, que ce soit pour rechercher un bug, implémenter une nouvelle technologie... De plus, certains forums comme StackOverflow permettent de coter les réponses fournies par les membres, afin de définir si celles-ci sont fiables ou non. Native-2-Native va mettre à profit ces informations fournies par le web afin de trouver un équivalent Swift fiable au code Android à traduire.

Le fonctionnement de Native-2-Native

Tout d'abord, le développeur va ajouter une nouvelle fonctionnalité, utilisant les ressources de l'appareil mobile, dans son code Java. Il surligne ensuite le code à traduire et appuie sur le bouton de génération de code Swift, fourni par Native-2-Native. La solution va en premier lieu déterminer la meilleure sémantique pour sa recherche web, en ne gardant que les mots-clés repérés dans le code Java en entrée. Pour cela, Native-2-Native va supprimer les déclarations de méthodes, les commentaires, les noms de variables, les sous-chaînes de caractères d'autres mots-clés.... Les 3 mots-clés repris le plus souvent dans le code filtré formeront la requête web qui sera utilisée.

Une fois la syntaxe définie, Native-2-Native va exécuter la recherche sur le web et rapatrier les résultats. Afin de pouvoir faire un premier tri, la solution va rechercher dans les réponses fournies par le web des objets méta-données qui permettraient d'évaluer la fiabilité de la réponse. Pour certains types de réponses comme les réponses StackOverflow, Native-2-Native ajuste son ensemble d'objets méta-données afin de cibler les meilleurs résultats possibles. Par exemple, dans StackOverflow, Native-2-Native va comparer les réponses des utilisateurs en sélectionnant celles notées comme « réponse acceptée ».

Native-2-native est ensuite chargé de filtrer les résultats obtenus pour ne garder que le meilleur d'entre eux. Pour cela, un algorithme de recherche ainsi qu'un algorithme de classement seront utilisés. L'algorithme de recherche prend en entrée les mots-clés utilisés pour lancer la recherche web, ainsi que l'ensemble des résultats fournis par celle-ci. Il fournira au final la liste de résultats réécrite de manière standardisée, afin de faciliter la comparaison entre ces derniers. L'algorithme de classement détermine quant à lui le niveau de similitude entre le code Android à traduire et l'ensemble des résultats fournis par l'algorithme de recherche. Il fournira en sortie les x meilleurs résultats, avec x précisé par l'utilisateur.

Une fois les résultats affichés, le développeur n'a plus qu'à sélectionner celui de son choix, qui sera directement intégré dans la copie Swift de son application. Pour chaque suggestion fournie au développeur, Native-2-Native précise également l'URL d'où l'information a été tirée. Dans la figure 2.7 [11], on retrouve l'approche détaillée de Native-2-Native.

2.2.7 XIS-Mobile

XIS-Mobile [22] est un framework basé sur le développement model-driven (MDD), capable de créer le squelette d'une application mobile Android et Windows Phone. L'outil permet de créer un modèle

[10] Swift est le nouveau langage utilisé par Apple, destiné à remplacer l'Objective-C
[11] Illustration tirée d'un article dédié à Native-2-Native (Voir [3])

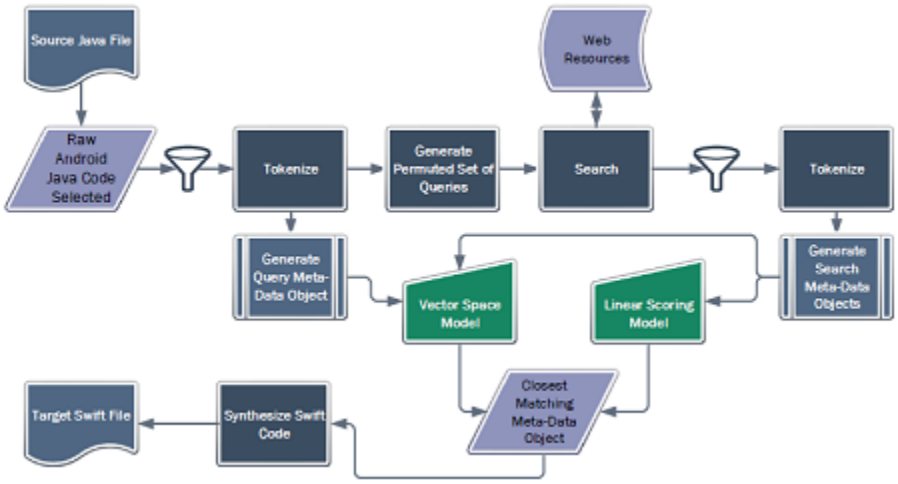

FIGURE 2.7 : Approche détaillée du développement avec Native-2-Native

sous forme d'UML, afin de complètement isoler l'application mobile de toute plateforme en utilisant des composants spécifiques au domaine du mobile. Comme MD2, XIS-Mobile possède son propre DSL. Pour créer ce langage, les auteurs de XIS-Mobile ont d'abord dû mener une analyse du domaine mobile actuel. Les points principaux qui furent ciblés étaient :

- La gestion de la connexion internet ;
- L'adaptation de l'interface utilisateur suivant le terminal (Smartphone, tablette, portrait, paysage...) ;
- La détection des actions réalisées par l'utilisateur (clic, glissement...).

Le framework divise son DSL en 3 grandes classes de composants : les entités, les cas d'utilisations et les interfaces utilisateurs. Les entités représentent tous les objets manipulés par l'application. Ces entités équivalent plus ou moins au contenu de la couche Modèle du pattern MVC. Les cas d'utilisations représentent les actions que l'utilisateur peut réaliser en interagissant avec l'application. Les actions CRUD ou encore l'option de recherche peuvent être reprises dans ces cas d'utilisation. Les interfaces utilisateurs représentent les différents espaces et composants graphiques avec lesquels l'utilisateur peut interagir (La barre de navigation, l'écran principal...). Enfin, une dernière classe de composants a été définie spécifiquement pour les applications mobiles : il s'agit des composants architecturaux. Ceux-ci sont chargés de symboliser les interactions entre l'application mobile et des évènements extérieurs comme la perte de connexion, l'état de la batterie...

La technologie XIS-Mobile se concentre surtout sur la conception de systèmes software interactifs indifféremment de la plateforme utilisée. Cet outil est donc plutôt utilisé pour créer des applications web ou de simples applications Desktop. Lorsque XIS-Mobile est utilisé pour réaliser une application mobile, il montre rapidement ses limites face aux aspects plus spécifiques au mobile comme les évènements générés par le téléphone, par l'utilisateur...

2.3 Les aspects les plus importants d'une application mobile

Afin de cerner les points cruciaux dans le développement d'une application mobile, 5 applications vont être analysées dans le tableau ci-dessous suivant plusieurs critères. Ces applications ont été choisies car considérées comme les plus utilisées en 2016 aux Etats-Unis [12] (Voir [14] pour plus d'informations sur ce classement). Quant aux critères choisis, ceux-ci peuvent être soit des fonctionnalités souvent retrouvées dans une application, soit des aspects importants du développement mobile. Ces derniers ont été sélectionnés suivant les droits les plus demandés sous Android (Voir [20]) mais également suivant une expérience professionnelle acquise dans le développement mobile.

Critères	Facebook	YouTube	Messenger	Google Maps	Google Search
Utilisation de la position du smartphone	O	-	O	O	O
Adaptation de l'app. sur smartphone ET tablette	O	O	O	O	O
Adaptation sur d'autres terminaux (TV, montre...)	-	O	O	O	-
Stockage de fichiers sur l'appareil	O	O	O	O	O
Utilisation d'une base de données sur l'appareil	O	O	O	O	O
Utilisation des fonctions natives (SMS, Contacts...)	O	O	O	O	O
Accès aux connexions réseau	O	O	O	O	O
Communication avec une source de données extérieure	O	O	O	O	O
Réception de notifications	O	O	O	O	-

O = pris en charge | - = non pris en charge

Dans ce tableau, il est aisé de constater que les 5 applications répondent positivement à certains critères de manière unanime. Ces critères, en plus des composants graphiques, devront donc être repris dans notre DSML.

2.4 Comparaison des différentes approches multi-plateformes

Dans cette section, les solutions multi-plateformes citées précédemment sont comparées suivant plusieurs critères. Cela va permettre de définir les aspects les plus prometteurs du développement multi-plateforme ainsi que les manques actuels de celui-ci. Certains de ces critères ont été trouvés durant la phase de recherche sur le développement multi-plateforme, mais la plupart d'entre eux proviennent d'un article écrit par l'université Ain Shams, basée en Egypte [5]. Le premier tableau concerne les solutions spécialement dédiées au développement mobile multi-plateforme. Le deuxième concerne les solutions traitant également d'autres sujets.

[12] Attention : l'application « Google Play Store » faisait partie du top 5 des apps les plus utilisées. Puisque celle-ci n'est pas disponible pour iOS, elle n'a pas été reprise dans notre échantillon

Critères	Xamarin	MD2	Native-2-Native
Objectif principal	Codage app.	Codage app.	Traduction code
Public principal	Développeurs	Développeurs	Développeurs
Utilisation des fonctionnalités natives	D	O	D
Création d'une application complète (vue, business...)	D	D	-
Conversion de A..Z entre les plateformes	D	O	O
Code généré suivant modélisation	-	-	-
Vue - Utilisation des ergonomies natives	D	O	-
Vue - Gestion des différentes tailles d'écran	D	D	-
Code - Code architecturé	D	D	-
Code - Réutilisation facile du code	D	D	D
Code - Transformation en langage natif	D	O	O

O = pris en charge | - = non pris en charge | D = possible, mais à charge du développeur

Critères	MetaEdit+	UsiXML	IFML	XIS-Mob
Objectif principal	Mod. des flux	Mod. vue	Mod. business	Mod. UML
Public principal	Analystes	Non développeurs	Analystes	Analystes
Utilisation des fonctionnalités natives	O	-	-	-
Création d'une app. complète (vue, business...)	-	-	-	O
Conversion de A..Z entre les plateformes	na	-	-	-
Code généré suivant modélisation	O	-	-	-
Vue - Utilisation des ergonomies natives	-	-	-	-
Vue - Gestion des différentes tailles d'écran	-	O	-	O
Code - Code architecturé	?	na	na	na
Code - Réutilisation facile du code	?	na	na	na
Code - Transformation en langage natif	O	na	na	na

O = pris en charge | - = non pris en charge | na = Non applicable

Le public « Non développeurs » qualifié dans la colonne d'UsiXML reprend les analystes, designers, managers... qui voudraient modéliser une application grâce à l'outil.

2.5 Conclusions

Il est indéniable que la modélisation d'applications multi-plateformes est toujours en pleine recherche. Lorsque nous analysons les deux précédents tableaux (Voir section 2.5), 2 catégories d'outils se distinguent. Nous avons tout d'abord les solutions multi-plateformes comme Xamarin ou encore MD2, qui nécessitent des connaissances poussées en développement et donc qui sont exclusivement réservées aux développeurs. Ensuite, nous avons des outils tels qu'IFML, qui se concentrent sur une conception beaucoup plus visuelle de l'application, ce qui permet à un public plus large d'utiliser ces solutions. Pourtant, un problème subsiste : ces solutions ne font que modéliser l'application, et ne fournissent pas le code permettant de créer ce qui a été modélisé. Ce qui nous ramène au même point, le codage d'une application mobile reste réservé aux développeurs. C'est pourquoi une solution telle que MetaEdit+ devient très intéressante : non seulement MetaEdit+ permet de concevoir visuellement une application, mais en plus, elle permet de générer le code pour créer cette dernière. L'outil n'est donc

pas exclusivement dédié aux développeurs et permet de créer l'application de la modélisation jusqu'au code. C'est dans cette optique que sera utilisé MetaDONE dans ce travail.

La base du DSML qui sera implémenté dans ce travail se fondera sur la solution proposée par MetaEdit+, en l'adaptant aux applications mobiles actuelles. De plus, afin de créer le DSML le plus complet possible, plusieurs éléments, déterminés grâce à cet état de l'art, devront être pris en compte.

En tout premier lieu, le DSML devrait laisser la possibilité à l'utilisateur de créer soit une application web, soit une application native. Chacune d'elle possède ses propres avantages et inconvénients et seul le créateur de l'application peut déterminer quel type correspond le mieux à ses besoins. Cependant, il est clair que le mieux serait de débuter par la création d'applications natives, puisque celles-ci permettent d'utiliser les fonctionnalités de l'appareil et ont de meilleures performances. Deuxièmement, d'après le tableau analysé dans la section 2.3, le DSML devrait créer plusieurs types de composants à savoir :

- Des éléments graphiques comme des boutons, des fenêtres... ;
- Des évènements générés par l'appareil, par l'utilisateur... ;
- Des composants « business » comme une base de données, l'accès à des web services... ainsi que les interactions avec ceux-ci ;
- Des composants représentant les applications natives de l'appareil (SMS, Contacts...) ;
- Des accès aux fonctionnalités natives de l'appareil comme l'accès aux connections réseaux ou à la position ;
- Des composants modèles afin de décider de la structure de données utilisée.

Un point important serait également de respecter les normes ergonomiques des différentes plateformes supportées : peu de solutions permettent de visualiser facilement ce que deviendrait une application si elle était créée pour une autre plateforme. Pourtant, l'ergonomie et l'expérience utilisateur sont fortement impactées. Par exemple, une personne utilisant un smartphone Android perd rapidement ses repères sur un iPhone et vice-versa. L'application créée doit donc s'adapter aux différents publics.

Enfin, le code généré pour créer l'application devrait rester clair et lisible. Pour cela, la création d'une architecture en couche comme pour la solution MD2 semble être la meilleure optique à adopter.

Dans le cadre de ce travail, nous nous concentrerons sur la création d'applications natives dans un premier temps (Android et iOS), puisque celles-ci permettent d'utiliser les fonctionnalités de l'appareil et sont de meilleure performance. Nous nous concentrerons plutôt sur la vue smartphone qui reste à l'heure actuelle la plus utilisée au niveau des appareils mobiles (Pour plus d'informations, voir la ressource [19]). Enfin, ce mémoire sera chargé de la définition d'un DSML capable de modéliser efficacement une application mobile mais pas de la génération du code de celle-ci. Cette partie demande trop de temps pour pouvoir être abordée das ce travail.

Chapitre 3

MetaDONE

Comme indiqué dans l'introduction de ce mémoire, MetaDONE est un MetaCASE tool créé à l'Université de Namur et développé en Java. Cet outil est celui utilisé dans ce travail afin de réaliser une solution mobile multi-plateforme sous la forme d'un DSML. Pour intégrer un nouveau DSML à MetaDONE, plusieurs étapes sont nécessaires. En premier lieu, il faut définir le méta-modèle du futur DSML. Afin que MetaDONE puisse comprendre le méta-modèle, celui-ci doit être défini en utilisant le langage de méta-modélisation du MetaCASE tool. Ce langage a pour nom MetaL2. Une fois le méta-modèle défini, il faut l'introduire dans le logiciel. Pour cela, MetaDONE offre la possibilité aux développeurs d'intégrer leur méta-modèle dans un plug-in OSGi et ainsi, de l'ajouter à chaud dans l'outil. Pour finir, il est nécessaire de fournir au logiciel une « représentation visuelle » des concepts qui composent le méta-modèle. Un langage appelé GraSyla doit donc être utilisé afin d'écrire un script à fournir à MetaDONE. Ce script a pour rôle d'associer un visuel à chaque élément du méta-modèle.

Dans ce chapitre, MetaL2 sera présenté afin de cibler tout le potentiel du langage de méta-modélisation de MetaDONE. Nous découvrirons ensuite GraSyla et comment ce langage est capable de fournir un visuel au méta-modèle. Enfin, la structure globale de MetaDONE sera présentée afin de visualiser comment ajouter un nouveau DSML sous la forme d'un plug-in OSGi dans le logiciel.

3.1 MetaL2

Comme dit précédemment, MetaDONE contient son propre langage de méta-modélisation permettant au développeur de créer un méta-modèle adapté à ses besoins et répondant à ses spécifications. Ce langage est appelé MetaL2 et est composé d'un ensemble d'éléments stockés dans la base de données de MetaDONE. Cela signifie que les éléments du langage de méta-modélisation pourraient facilement évoluer suivant les futurs besoins du logiciel. Il est composé de 4 concepts majeurs qui sont : le *MetaModel*, le *MetaObject*, le *MetaRole* et la *MetaProperty*.

Le **MetaObject** est l'élément-clé de MetaL2 : chacun des concepts énoncés ci-dessus hérite de MetaObject. Celui-ci est identifié par un nom et est chargé de représenter un élément du futur méta-modèle. Par exemple, dans la future solution multi-plateforme, *Screen* sera un MetaObject représentant un écran de l'application.

Le **MetaModel** est l'élément représentant le futur méta-modèle. Cet objet est chargé de contenir l'ensemble des MetaObject qui formeront le méta-modèle.

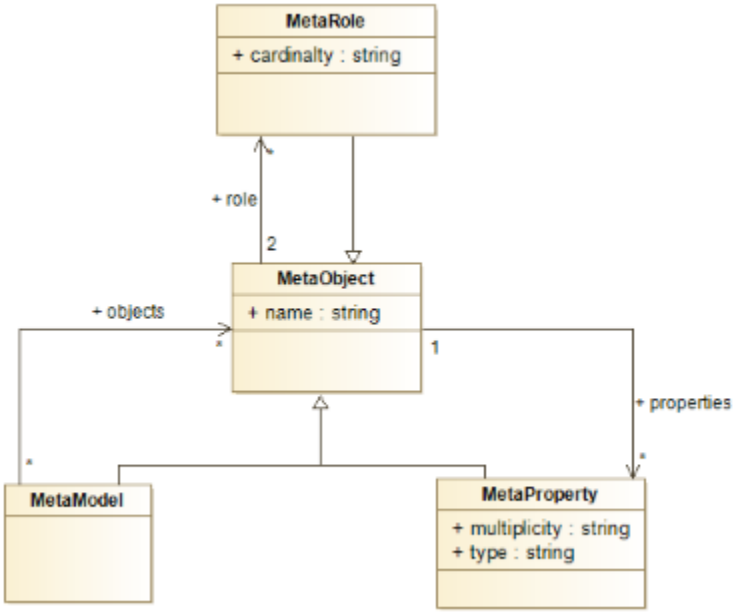

FIGURE 3.1 : MetaL2 - Diagramme de classe

Le **MetaRole** est chargé de symboliser un lien entre un MetaObject (référencé comme le « domain ») et un autre MetaObject (référencé comme le « range »). Dans notre solution, un MetaRole entre un *Screen* et un *ScreenState* représente un lien entre un écran et les différents états dans lequel il peut se retrouver (Créé, démarré, en pause...). Un MetaRole est également composé d'une cardinalité, qui définira de manière plus précise la nature du lien entre les 2 objets.

Une **MetaProperty** est une propriété qui peut être liée à n'importe quel MetaObject. Autrement dit, un MetaModel, un MetaObject, un MetaRole et même une MetaProperty peuvent être décrits par des MetaProperty. Cet objet est composé d'une cardinalité et d'un type (Integer, Float, String ou Boolean). Par exemple, le MetaObject *Screen* pourrait posséder une MetaProperty de type String représentant le nom de cet écran.

La figure 3.1 montre comment les composants de MetaL2 sont reliés entre eux et reprend visuellement les informations décrites ci-dessus. Par simplicité, nous avons choisi l'UML afin de représenter les concepts de MetaL2. Toutefois, l'UML ne peut exprimer visuellement tout le potentiel du langage de méta-modélisation. Par exemple, une MetaProperty peut être le domain d'un MetaRole dont le range serait un MetaModel ou une MetaProperty.

3.2 GraSyla

MetaL2 va permettre au développeur de définir les concepts de son méta-modèle. Dans notre cas, le méta-modèle reprendra l'ensemble des objets nécessaires à la construction d'une application mobile (Exemple : Un écran, un widget, un évènement clic de l'utilisateur...). Cependant, le travail ne s'arrête

pas là : une fois que le méta-modèle est défini, il faut être capable de manipuler les éléments de celui-ci pour créer un modèle et le visualiser. C'est ce que va permettre GraSyla.

GraSyla (Graphical Symbolic Language) est lui-même un DSML basé sur un méta-modèle construit avec MetaL2 et a pour rôle de définir la syntaxe concrète des DSML conçus avec MetaL2. Pour ce faire, le script GraSyla créé par le développeur devra contenir un ensemble d'équations devant suivre la syntaxe grammaticale imposée par le langage. Chacune d'elle sera chargée d'associer un MetaObject du méta-modèle utilisé par le modèle à une représentation visuelle. De manière simplifiée, GraSyla va donc nous permettre d'associer une « Vue » à un modèle. Pour pouvoir associer une représentation visuelle à un concept du méta-modèle, une équation GraSyla a besoin de plusieurs informations :

- Un foncteur, précisant à GraSyla dans quel contexte cette équation doit être utilisée ;
- Une multiplicité, indiquant combien de concepts l'équation doit représenter (no, one ou many) ;
- Une méta-classe de MetaL2, afin de définir la nature de l'objet concerné par l'équation (MetaObject, ...) ;
- L'objet à représenter, appartenant au méta-modèle créé avec MetaL2. Le nom indiqué doit correspondre à celui identifiant l'objet ; et enfin
- Une définition déclarative de la représentation visuelle de l'objet, définie à l'aide d'éléments graphiques mis à disposition par GraSyla.

Chaque équation est composée d'un côté gauche et d'un côté droit, séparés par un « = ». La partie gauche de l'équation précise « ce qui est affiché » tandis que la partie droite se charge d'expliciter « comment il est affiché ». Pour cette partie, le langage permet de créer des formes simples et complexes telles que des formes géométriques (rectangles, cylindres, flèches...), des images (Bitmap ou SVG) mais il permet également de garnir ces formes à l'aide de propriétés telles que des bordures, des couleurs... Des composants Swing tels que les textfields, les checkboxs... sont également disponibles. Un rapport technique sur GraSyla [7] est également disponible afin de visualiser l'ensemble de ses possibilités et composants graphiques.

3.3 Architecture du logiciel

Afin de pouvoir facilement ajouter de nouveaux méta-modèles dans le logiciel, MetaDONE a été construit comme une architecture de plug-ins, en suivant le framework OSGi. De manière très rapide, OSGi permet de créer une plateforme dynamique construite comme un ensemble de services pouvant être ajoutés, supprimés, mis à jour ou encore désinstallés sans que cela nécessite le redémarrage du framework OSGi. Les composants de cette architecture sont ce qu'on appelle des bundles OSGi et permettent donc d'ajouter des services à la plateforme sans devoir redémarrer celle-ci. Chaque plug-in est donc en réalité un bundle OSGi pouvant être ajouté dans MetaDONE, même si celui-ci est en marche. Ainsi, le développeur pourra créer son propre bundle dans le but d'ajouter son méta-modèle dans l'architecture de MetaDONE. Un bundle est réalisé en Java et devra contenir l'ensemble des MetaObject, MetaRole et MetaProperty composant le MetaModel. Si le développeur doit réaliser une modification dans celui-ci, il lui suffira de recharger le bundle à chaud dans MetaDONE comme dans la figure 3.2.

Comme expliqué précédemment, le script GraSyla va permettre au développeur de lier une représentation visuelle aux modèles créés grâce à son méta-modèle. Une fois que le développeur a son script,

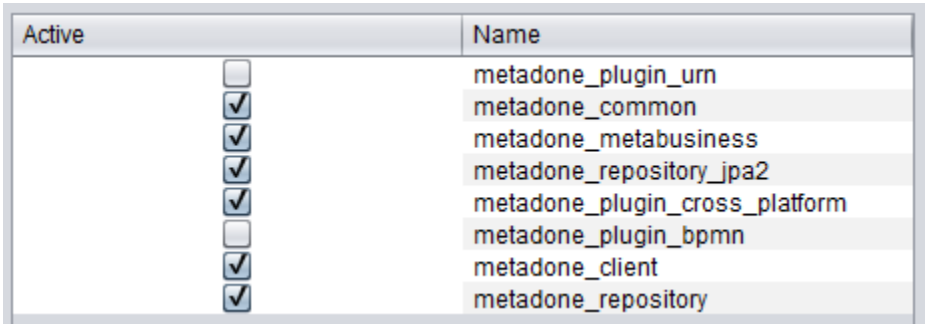

FIGURE 3.2 : Ensemble des bundles chargés dans MetaDONE

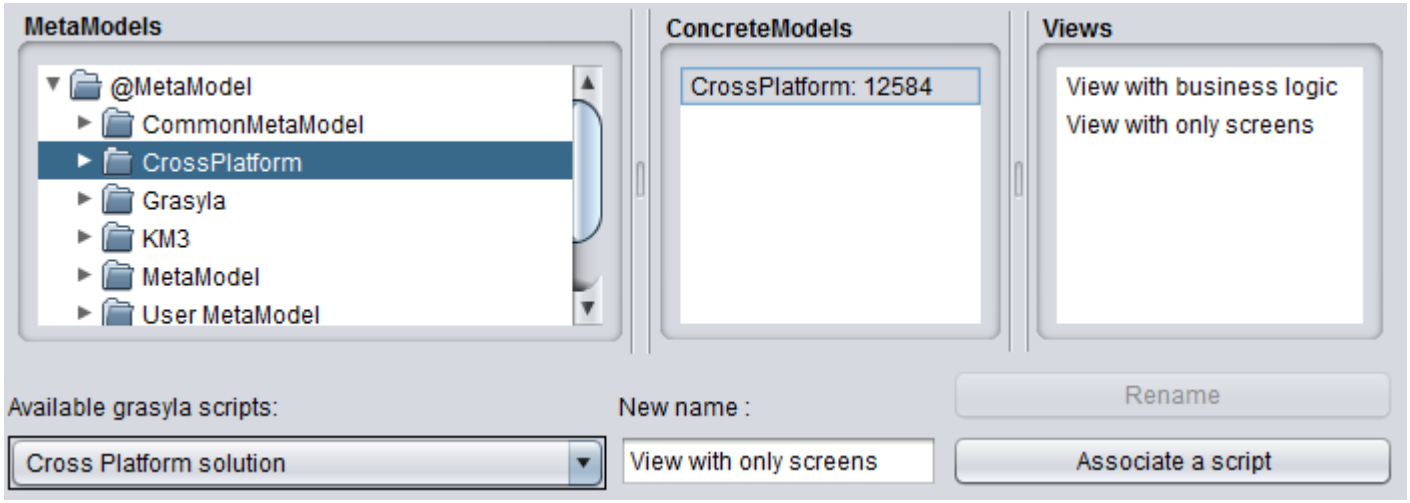

FIGURE 3.3 : Association d'un script GraSyla à un modèle

il lui est possible de l'assigner à un modèle directement dans MetaDONE comme dans la figure 3.3. Si le développeur en a besoin, le logiciel lui laisse également la possibilité de visualiser en même temps un modèle en lui assignant plusieurs scripts GraSyla différents. Pour cela, MetaDONE s'est inspiré du design pattern « Observer » : une modification sur la première vue impactera donc aussi l'autre vue.

Chapitre 4

Analyse

L'état de l'art mené précédemment dans ce travail a permis de découvrir de nouveaux éléments à prendre en compte avant l'implémentation de la solution. Le but de ce chapitre sera donc d'exploiter toutes les informations acquises sur les applications mobiles multi-plateformes afin d'améliorer la problématique de référence présente au début de ce mémoire. Pour cela, ce chapitre débutera en listant l'ensemble des éléments minimum qui devront composer la future solution, en reprenant les éléments de MetaEdit+ et en y appliquant les informations contenues dans les conclusions de l'état de l'art. Ensuite, l'analyse des 2 plateformes mobiles Android et iOS sera présentée. Celle-ci a pour rôle de comprendre au mieux les composants des 2 plateformes et de pouvoir trouver des similitudes entre tous ces éléments. De plus, cette analyse est nécessaire afin de représenter le plus fidèlement possible les composants de ces 2 plateformes dans un unique méta-modèle. Enfin, une nouvelle version de la problématique de référence sera présentée, en mettant à profit tous les concepts mis en avant durant cette analyse.

4.1 Contenu d'une application

Comme expliqué dans les conclusions de l'état de l'art, la solution à implémenter durant ce travail se basera sur celle de MetaEdit+ et devra adapter cette dernière aux mobiles d'aujourd'hui. Pour rappel, le DSML de MetaEdit+ dédié aux applications mobiles concerne les téléphones S60 et est composé des éléments suivants :

- Des *start & stop*, qui symbolisent le début et la fin d'une application ;
- Des *UI Controls* qui s'apparentent à des formulaires, des listes ... ;
- Des *widgets*, qui représentent différentes fenêtres disponibles dans l'application ;
- Un objet *condition*, qui permet d'ajouter de la logique au niveau des liens de navigation entre écrans ;
- Des *phone services*, qui représentent les différentes ressources et services de l'appareil ;
- Des objets *navigation flows*, qui définissent les différents types de relation qu'il peut exister entre les écrans.

Certains de ces éléments doivent être adaptés aux applications mobiles d'aujourd'hui. Tout d'abord, les objets *start et stop* ne doivent pas être pris en compte tels quels. A l'heure actuelle, une application mobile n'a pas de « stop » a proprement parlé : si l'utilisateur veut quitter l'app, il lui suffit d'appuyer sur le bouton « Home » de son smartphone. L'objet stop ne doit donc pas être repris dans le DSML. Par contre, le *start* devra être mentionné mais sous une autre forme : lorsqu'une application est créée, un écran de « départ » doit être précisé par le développeur. Cet écran représente le premier que l'utilisateur verra lorsque l'app démarrera. Le *start* pourra donc être repris dans le DSML tout en étant lié à un écran.

Concernant les UI controls, les applications mobiles actuelles possèdent des éléments similaires mais plus évolués. Dans MetaEdit+, une liste reste une liste : aucune nuance n'est apportée dans la représentation de celle-ci. La future solution devra être capable de représenter le contenu d'une liste en séparant celui-ci du container de la liste. Cela permettra de mieux distinguer la représentation de la liste dans un écran et celle de son contenu.

Comme exposé dans la figure 2.1 reprenant l'ensemble des éléments du DSML de MetaEdit+, les *navigation flows* sont des liens très sommaires. Ils peuvent être de 3 types : de simples « liens » entre écrans, des liens de retour ou des liens liés à une condition. Dans le DSML, les « liens » entre écrans ne seront pas représentés tels quels simplement car ces liens découlent toujours d'une action ou d'un évènement arrivé dans l'app. Le lien entre 2 écrans sera donc présent mais représenté comme le fruit d'une série d'actions et non d'une interaction directe. Le DSML devra par contre, reprendre des catégories d'objets similaires aux *widgets*, aux *conditions* et aux *phone services*. Ces catégories devront cependant être complétées avec des objets relatifs aux applications d'aujourd'hui.

Les éléments définis à l'aide du DSML de MetaEdit+ ne représentent que la base du DSML. Les recherches menées durant l'état de l'art ont permis de cibler une liste d'éléments à intégrer dans la future solution, pour répondre aux exigences des applications mobiles actuelles. Pour rappel, ces éléments étaient :

- Des éléments graphiques comme des boutons, des fenêtres... ;
- Des évènements générés par l'appareil, par l'utilisateur... ;
- Des composants « business » comme une base de données, l'accès à des web services... ainsi que les interactions avec ceux-ci ;
- Des composants représentant les applications natives de l'appareil (SMS, Contacts...) ;
- Des accès aux fonctionnalités natives de l'appareil comme l'accès aux connections réseaux ou à la position ;
- Des composants modèles afin de décider de la structure de données utilisée.

Tous ces éléments devront donc être repris dans le DSML, afin de représenter une application mobile de manière la plus complète possible.

4.2 Analyse des plateformes Android et iOS

Les chapitres 1 et 2 exposent le fait que les plateformes mobiles sont différentes les unes des autres tant au niveau de leur langage qu'au niveau de leur ergonomie ou de leur expérience utilisateur. Pourtant,

une solution mobile multi-plateformes exige de représenter de manière unique les concepts de ces plateformes si différentes. Cette section a donc pour objectif de reprendre les objets qui composent les plateformes Android et iOS et de leur trouver des similitudes. Une fois ces similitudes définies, nous serons en mesure de les intégrer dans le DSML de manière à représenter au mieux les 2 plateformes. [1]

4.2.1 Le visuel d'une application

Pour permettre au développeur de créer le visuel de son application, iOS prend en charge des « storyboards ». Ces storyboards pourraient s'apparenter à des « scénarios visuels » de l'application, reprenant l'ensemble des écrans et vues de celle-ci. Typiquement, une application iOS possède un unique storyboard. En complément, la plateforme permet au développeur de créer des fichiers .xib, capables de contenir une seule vue. Ces fichiers .xib peuvent être repris dans les storyboards et sont généralement chargés de définir un élément visuel présent à plusieurs endroits de l'application. Android a, quant à lui, opté pour un mécanisme plus simple : les différents écrans composant une application doivent être construits sous forme de fichiers xml. Le développeur n'a donc jamais un aperçu de l'ensemble des écrans et de leurs enchaînements comme dans un storyboard iOS.

Même si les mécanismes mis à disposition du développeur pour construire la vue sont très différents, les composants visuels de chaque plateforme se ressemblent fortement. On peut distinguer sur celles-ci 2 catégories d'objets : les « containers » comme les listes et les « widgets » comme les boutons. Un container a pour rôle de contenir d'autres éléments graphiques tandis que le widget est plutôt chargé de représenter une valeur spécifique. Il est à noter toutefois qu'en Android comme en iOS, les containers sont représentés séparément de leur contenu : cela permet au développeur de bien distinguer comment doit être représenté le container dans son environnement et de quoi il doit être composé. Cette distinction sera reprise dans le DSML afin d'être le plus aligné possible aux pratiques des plateformes. Cela devrait faciliter l'étape de génération de code de l'application qui pour rappel, ne sera pas abordée dans ce mémoire. Cette distinction est également nécessaire afin de faire la séparation entre les évènements utilisateurs qui concernent un « container » (onItemClick, onScrollDown...) et ceux qui concernent un « widget » (onClick, onLongClick,...).

4.2.2 Le cycle de vie d'un écran

Dans une application mobile, chaque écran peut se retrouver dans différents « états ». Si l'utilisateur démarre un nouvel écran, celui-ci est « en cours de démarrage », l'écran précédent est « stoppé »... Tous ces états arrivent dans un ordre précis et forment ainsi le cycle de vie d'un écran. Ce cycle de vie existe aussi bien en Android qu'en iOS, même si la dénomination des états n'est pas tout à fait pareille. En effet, chaque plateforme possède sa propre classe représentant un écran (Android parle alors d'*Activity* et iOS de *ViewController*) et chacune de ces classes possède ses propres méthodes qui ont trait aux différents états du cycle. C'est d'ailleurs en redéfinissant ces méthodes que le développeur va pouvoir définir le comportement de l'écran dans un état précis. Le tableau 4.1 reprend les méthodes disponibles dans les classes *Activity* et *ViewController* et définit dans quel « état » se trouve alors l'écran. L'ensemble de ces états devront être repris dans le DSML. Cela permettra à l'utilisateur de prendre le contrôle du cycle de vie de chacun de ses écrans mais également lors de la génération de code, de pouvoir définir de manière certaine le comportement d'un écran.

[1] L'ensemble des concepts mobiles exposés dans cette section peuvent être approfondis en consultant les sites de développement d'Apple : https ://developer.apple.com/ et d'Android : https ://developer.android.com

Android	iOS	Etat de l'écran
onCreate	viewDidLoad	L'écran est créé pour la première fois par l'application mobile.
onStart	viewWillAppear	L'application va passer l'écran au premier plan et ainsi, le rendre visible aux yeux de de l'utilisateur.
onResume	viewDidAppear	L'écran est au premier plan, l'utilisateur peut interagir avec celui-ci.
onPause	viewWillDisappear	L'utilisateur est en train de quitter l'écran (Passe à la suite de l'application, quitte l'application, affichage d'une fenêtre de dialogue temporaire...).
onStop	viewDidDisappear	L'écran est stoppé et n'est plus au premier plan. Il devra être redémarré lors de sa prochaine apparition
onDestroy	didReceiveMemoryWarning	L'application détruit l'écran. Il devra être recréé lors de sa prochaine apparition

TABLE 4.1 : Les différents états d'un écran selon Android et iOS

4.2.3 Les sources de données

Dans une application mobile, quelle que soit sa plateforme, il est possible de gérer des données au travers des sources de données suivantes :

- Dans une base de données,
- au travers d'un web service,
- dans le stockage interne du smartphone,
- dans les « préférences » de l'application. [2]

Toutes ces sources de données sont disponibles aussi bien en Android qu'en iOS. Les interactions avec celles-ci devront donc être reprises dans le DSML, afin de permettre à l'utilisateur de gérer les données de son application comme il le souhaite.

4.2.4 Les notifications

Dans une application mobile, 2 types de notifications existent. Tout d'abord, il existe les notifications push envoyées à l'application depuis l'extérieur de celles-ci. Lorsqu'une app mobile veut être capable de recevoir des notifications push, elle doit s'enregistrer dans un service natif fourni par la plateforme, capable de gérer les notifications. En Android, ce service s'appelle *GCM (Google Cloud Messaging)* tandis qu'en iOS, il s'agit de l'*APNs (Apple Push Notification service)*. Une fois enregistrée et capable d'écouter les évènements venant de l'extérieur [3], l'application est en mesure de recevoir des notifications push de la part d'un service tiers et de définir des actions à prendre en fonction de celles-ci.

[2] Les « Préférences » représentent un espace de stockage unique à l'application, présent à chaque instance de celle-ci. Ces préférences sont stockées sous forme d'un ensemble de paires clé-valeur et sont chargées de contenir des informations basiques

[3] Pour écouter les notifications push, les applications Android et iOS demandent un minimum de configuration en ajoutant du code spécifique. Ce code ne sera pas présenté dans ce mémoire

En Android comme en iOS, il existe aussi un mécanisme de notifications internes. Contrairement aux notifications push où l'application n'a aucun contrôle sur leur envoi, les notifications internes sont déclenchées et reçues au sein même de l'app. Ce procédé est utilisé afin de définir un évènement déclenché à un point précis de l'application qui doit engendrer une action à un autre endroit de celle-ci. En Android, ce mécanisme se présente sous forme de *BroadcastReceivers* tandis qu'en iOS, il s'agit du concept de *UserNotificationCenter*.

Ces 2 catégories de notifications ainsi que leurs interactions avec l'application seront reprises dans le DSML.

4.3 Nouvelle version de Smart Namur

Au vu des nouvelles informations recueillies durant cette analyse, la problématique de référence exposée au début de ce mémoire (Voir section 1.3) doit être adaptée. En effet, l'application Smart Namur exposait surtout des concepts visuels à respecter pour représenter une application. Cependant, il manque beaucoup d'éléments plus orientés sur le comportement de l'application plutôt que sur son visuel. En clair, il manque la logique d'arrière-plan de l'app : comment elle va gérer ses données, quels évènements vont être déclenchés... Cette nouvelle version, illustrée à la figure 4.1, va donc prendre en compte les éléments suivants :

- Les services natifs à l'application comme la gestion de la connectivité ;
- La prise en charge d'un objet « Condition » ;
- La distinction, au niveau visuel, entre les « containers » et leur contenu ;
- La distinction entre les évènements utilisateurs qui traitent d'un « container » ou d'un « widget » ;
- Les différents états du cycle de vie d'un écran ;
- Les interactions avec les différentes sources de données ;
- Les notifications push et leurs interactions avec l'app ;
- Les notifications internes et leurs interactions avec l'app.

Il est évident que le DSML devra également tenir compte de ces éléments.

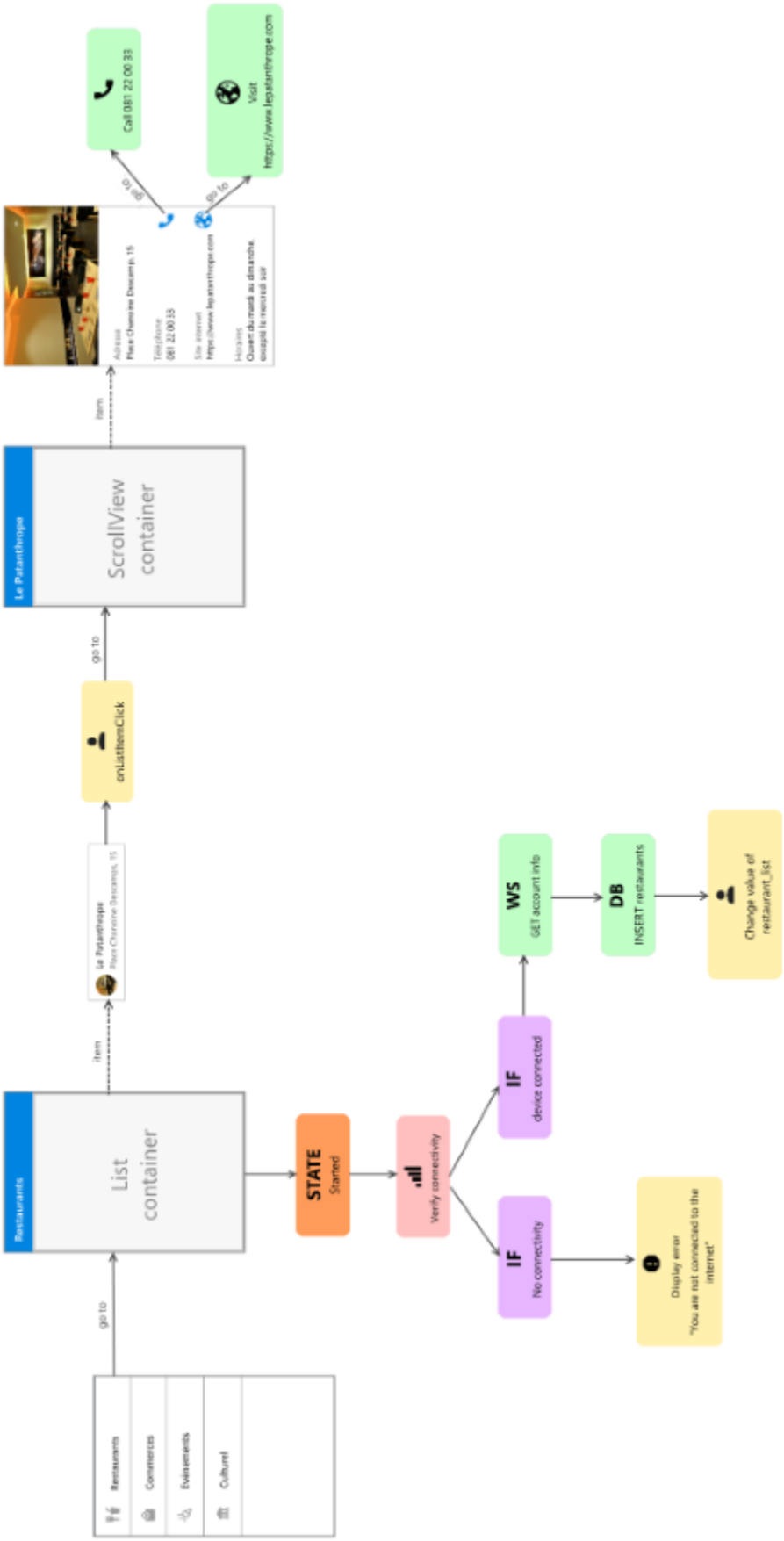

FIGURE 4.1 : Nouvelle version de Smart Namur

Chapitre 5

Implémentation

Dans ce chapitre, nous allons découvrir le rendu final de la solution « Smart Namur » proposée dans l'introduction de ce mémoire (Voir figure 1.1 & 4.1). Pour cela, nous exposerons le méta-modèle et décrirons les éléments les plus importants qui le composent. Ensuite, nous analyserons le script Grasyla créé pour représenter les modèles construits sur base du méta-modèle créé.

5.1 Le méta-modèle

5.1.1 Contenu

La figure 5.1 présente une version simplifiée du contenu du méta-modèle. Le but de cette section est de décrire la composition de celui-ci et de comprendre l'intérêt de ces objets en son sein.

Plusieurs catégories d'objets existent dans le méta-modèle : des objets orientés vers le visuel, d'autres vers la logique et le business.... Les objets les plus importants et repris dans la version simplifiée (Cf. figure 5.1) vont être décrits et leur rôle éclairci.

Commençons par l'objet *Application*, qui, comme son nom l'indique, représente l'application mobile en elle-même. Cet objet est chargé de contenir les informations importantes relatives à l'application comme son nom, sa version et son identifiant. Cet objet permettra de faire évoluer l'application dans les différents stores si cela s'avère nécessaire en mettant à jour les informations requises.

L'application peut être composée d'un *AppMenu*, qui représente un menu global dans l'application. En Android, ce menu se retrouve majoritairement sous la forme d'un menu coulissant sur la gauche tandis qu'en iOS, on le retrouve sous la forme d'une tab bar. L'AppMenu est composé d'*AppMenuItems* qui constituent les entrées du menu.

L'application est également composée d'*AppWindows*, c'est-à-dire l'ensemble des fenêtres disponibles dans l'app. Elles sont de 2 types : les *Dialogs* et les *Screens*. La *Dialog* est une fenêtre de dialogue, généralement composée de très peu d'éléments et sensée attendre une action de la part de l'utilisateur. Généralement, elles possèdent un rôle bien précis [1] mais cela n'a pas été imposé dans notre méta-modèle. Le *Screen* représente quant à lui un écran de l'application mobile. Il peut posséder une barre d'actions et contient la quasi totalité des éléments visuels de l'application mobile (quelques-uns

[1] Sur les plateformes Android et iOS, on parle souvent d'*AlertDialog*, de *DatePickerDialog*... contenant un élément graphique unique

se trouvent dans les fenêtres de dialogue). Dans le méta-modèle, l'écran a été différencié de la fenêtre de dialogue pour plusieurs raisons :

- La création de ces 2 éléments se fait de manière différente sur les 2 plateformes mobiles ;
- Un écran doit absolument être identifié (en Android par le biais de l'AndroidManifest, en iOS dans les storyboards) ce qui n'est pas le cas des fenêtres de dialogue ;
- L'écran possède une barre d'action, ce qui n'est pas le cas d'une fenêtre de dialogue ;
- Des actions peuvent survenir suivant l'« état » dans lequel se trouve l'écran (état nommé *ScreenState* dans notre méta-modèle) ;

Les éléments visuels disponibles dans MetaDONE et pouvant se retrouver dans la vue d'une AppWindow sont appelés les *ViewComponents* et sont de 2 types : les *Widgets* et les *ViewGroups*. Les widgets sont des éléments graphiques simples tels que les labels, les boutons... Les viewgroups quant à eux représentent des objets plus volumineux. Les *ScrollableViewGroups* sont une sous-catégorie de viewgroup, servant surtout de « containers » : au visuel, ces objets sont susceptibles de regrouper d'autres éléments visuels. Ces ScrollableViewGroups reprennent notamment les listes, les ScrollViews ou les TabViews.

Nous savons donc que chaque AppWindow peut contenir d'autres éléments visuels. Ils sont en sorte des « containers de vues », autrement dit des *MetaDONEContainers*. Ces containers possèdent chacun un identifiant visuel et sont les seuls éléments dans notre méta-modèle capables de contenir d'autres vues : aucun autre type d'élément n'y sera autorisé. Il est légitime de penser que seules les fenêtres de l'application pourront servir de containers dans notre méta-modèle puisque visuellement parlant, tous les éléments graphiques tiennent dans une fenêtre .... Pourtant, ce n'est pas le cas dans notre méta-modèle : le *ViewItemContainer* est également un MetaDONEContainer. Le ViewItemContainer est automatiquement lié à un ScrollableViewGroup et est chargé de reprendre le contenu visuel de celui-ci. Par exemple, pour une liste, le ViewItemContainer qui lui sera associé reprendra les éléments graphiques contenus dans une cellule de celle-ci.

L'ensemble des éléments purement visuels ont été présentés. Passons maintenant aux composants plus orientés business, destinés à enrichir l'interaction entre tous les composants de l'application .

Commençons par les *Events* : il s'agit d'évènements pouvant survenir tout au long de l'application. Certains d'entre eux sont déclenchés par le biais d'un composant interne, comme un *BroadcastReceiver* et sont catégorisés en tant qu'*InternalAppEvents*. D'autres surviennent par le biais d'un composant externe, comme une notification push : ils sont définis comme étant les *ExternalAppEvents*. Mais les évènements les plus nombreux sont ceux déclenchés par une action de l'utilisateur comme un clic, un scroll... Nous les appellons les *UserPhoneInteractionEvents*.

Définir une série d'évènements dans une application est un début, mais cela ne suffit pas. Lorsque nous détectons un clic de l'utilisateur sur un bouton, un scroll sur une liste... nous voulons réaliser quelque chose : afficher un message d'erreur, sauvegarder des données, vérifier la connectivité de l'appareil... Bref, nous voulons réaliser une *Action*. Plusieurs catégories d'actions sont disponibles dans notre méta-modèle :

- Les actions visuelles impactant directement la vue (Cacher un bouton, afficher un message d'erreur...) ;

- Les actions permettant de démarrer un nouvel écran. Cet écran peut faire partie de l'application mais également mener vers une application extérieure (comme le téléphone, l'application web...) ;
- Les actions nécessitant l'utilisation de l'appareil (Vérifier la connectivité, la batterie...) ;
- Les actions chargées de s'occuper des données, disponibles au travers d'une DB, d'un web service... ;

Plusieurs actions peuvent être exécutées à la suite ou de manière parallèle, ce qui laisse à l'utilisateur plus de liberté dans la configuration de son app.

Une action peut également se terminer de différentes manières. Par exemple, vérifier la connectivité de l'appareil peut être intéressant pour ensuite aller dialoguer avec un web service afin de rapatrier des données. Mais comment doit se comporter l'application si l'appareil n'est pas connecté à internet ? C'est dans ce but que l'objet *Condition* a été créé : il permet à l'utilisateur de définir une série de conditions qui exécuteront différentes actions suivant la situation dans laquelle se retrouvera l'app.

Enfin, le méta-modèle reprend également des concepts liés au modèle que pourrait contenir l'application, le but final étant de laisser la possibilité à l'utilisateur de lier un élément du modèle avec un élément graphique. Ainsi, l'app pourra adapter directement le contenu de son interface suivant le modèle défini et les données choisies.

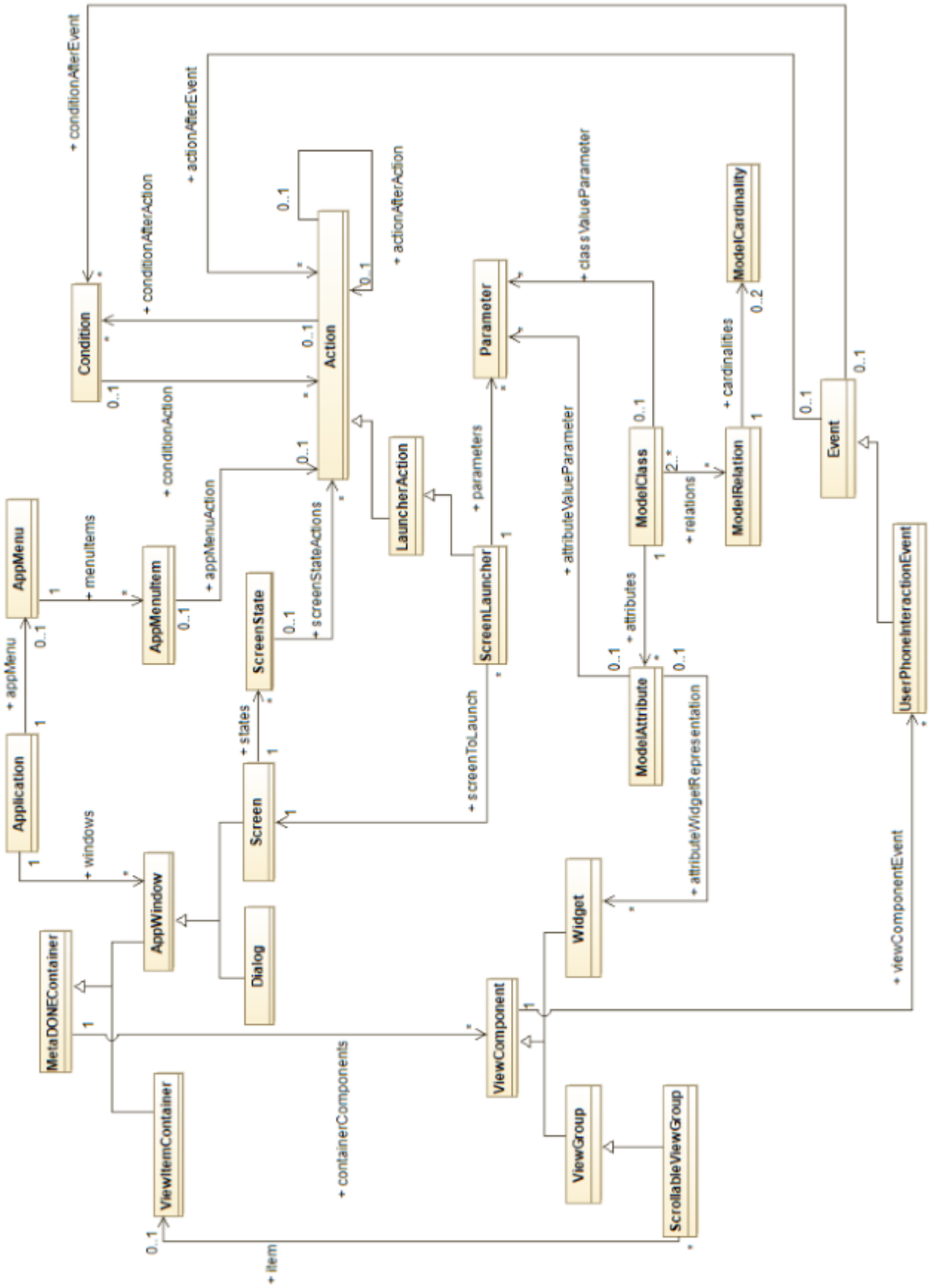

FIGURE 5.1 : Version simplifiée du méta-modèle d'une application mobile multi-plateforme

Comme expliqué précédemment, la figure 5.1 présente une version simplifiée du méta-modèle, reprenant seulement les éléments les plus importants. La version complète de celui-ci est disponible dans l'annexe A de ce mémoire.

5.1.2 Création du plug-in

Une fois que notre méta-modèle est défini, il ne reste plus qu'à l'intégrer dans MetaDONE. Pour cela, celui-ci sera repris dans un plug-in OSGi, ce qui nous permettra (comme expliqué dans la section 3.3) d'ajouter notre méta-modèle à chaud dans le logiciel.

Pour créer un plug-in OSGi reprenant notre méta-modèle, il est obligatoire d'implémenter 2 interfaces : il s'agit de *MetadonePlugin* et *BundleActivator*. *MetadonePlugin* est une interface fournie par MetaDONE et représente un nouveau plug-in. Le développeur doit l'implémenter afin de créer son propre plug-in et préciser son nom, sa composition, son menu principal.... MetaDONE propose une classe abstraite appelée *AbstractMetadonePlugin* implémentant cette interface, permettant de dégrossir le code que le développeur doit fournir. Afin de créer le plug-in contenant le méta-modèle destiné à la construction d'une application mobile multi-plateforme, *AbstractMetadonePlugin* a été réutilisée. Dans ce cas, le but principal de l'implémentation est de fournir la composition du méta-modèle, en décrivant l'ensemble des éléments de celui-ci sous la forme de *MetaObjet*, *MetaProperty* et *MetaRole* (Cf. section 3.1). Voici un extrait de code chargé de la création de ces objets :

```
// imports

public class CrossPlatform extends AbstractMetadonePlugin {
    private String[] swingColors = {"yellow", "blue", "red", "green", "orange", "gray",
        "black", "white", "cyan", "pink", "magenta"};

    @Override
    public void trigger(WorkspaceChanged event) {
        super.trigger(event);
        switch (event.type) {
        case OPENED:
            try {
                initializeMetaModel(event.workspace);
                eventLoad.setManager(getEventManager()).subscribe();
            } catch (BadPreCondition e) {
                e.printStackTrace();
            }
            break;
        }
    }

      private void initializeMetaModel(Workspace ws) throws BadPreCondition {
        if (ws == null) {
            throw new BadPreCondition("No workspace has been given");
        }
        final MetaModel main = ws.getMainMetaModel();

        // CREATE META MODEL
        mm_cross_platform = main.produceMetaModel("CrossPlatform");
```

```
    // CREATE ALL OBJECTS OF THE PLUGIN
    this.initializeMainElements();
    this.initializeUIElements();
    this.initializeViewItemContainers();
    this.initializeViewGroups();
    this.initializeWidgets();
    this.initializeActions();
    this.initializeEvents();
    this.initializeScreenStates();
    this.initializeModelRepresentation();
}

private MetaModel mm_cross_platform;

private MetaObject mo_cp_App;
private MetaPropertyExt<String> mp_cp_App_Name;
private MetaPropertyExt<String> mp_cp_App_Id;
private MetaPropertyExt<String> mp_cp_App_Version;

private MetaObject mo_cp_Metadone_Container;
private MetaPropertyExt<String> mp_cp_Metadone_Container_VisualId;

private MetaObject mo_cp_App_Window;
private MetaPropertyExt<String> mp_cp_App_Window_Name;

private MetaObject mo_cp_Screen;
private MetaPropertyExt<Boolean> mp_cp_Screen_IsMainLauncher;
private MetaPropertyExt<String> mp_cp_Screen_WindowBackgroundColor;
private MetaPropertyExt<String> mp_cp_Screen_ActionBarBackgroundColor;
private MetaPropertyExt<Boolean> mp_cp_Screen_Full_Screen;

private MetaRole mr_cp_App_Screens;

  ...

private void initializeMainElements() throws BadPreCondition {
    mo_cp_App = mm_cross_platform.produceMetaObject("App");
    mp_cp_App_Name = mo_cp_App.produceMetaProperty(String.class, "App.Name", 1,
        mm_cross_platform);
    mp_cp_App_Id = mo_cp_App.produceMetaProperty(String.class, "App.Id", 1,
        mm_cross_platform);
    mp_cp_App_Version = mo_cp_App.produceMetaProperty(String.class, "App.Version", 1,
        mm_cross_platform);

    mo_cp_Metadone_Container = mm_cross_platform.produceMetaObject("MetadoneContainer");
    mp_cp_Metadone_Container_VisualId =
        mo_cp_Metadone_Container.produceMetaProperty(String.class,
        "MetadoneContainer.VisualId", 1, mm_cross_platform);

    mo_cp_App_Window = mm_cross_platform.produceMetaObject("AppWindow",
```

```
            mo_cp_Metadone_Container);
        mp_cp_App_Window_Name = mo_cp_App_Window.produceMetaProperty(String.class,
            "AppWindow.Name", 1, mm_cross_platform);

        mo_cp_Screen = mm_cross_platform.produceMetaObject("Screen", mo_cp_App_Window);
        mp_cp_Screen_IsMainLauncher = mo_cp_Screen.produceMetaProperty(Boolean.class,
            "Screen.IsMainLauncher", 1, mm_cross_platform);
        mp_cp_Screen_WindowBackgroundColor =
            mo_cp_Screen.produceMetaPropertyEnumerated(String.class,
            "Screen.WindowBackgroundColor", 1, mm_cross_platform, true, swingColors);
        mp_cp_Screen_ActionBarBackgroundColor =
            mo_cp_Screen.produceMetaPropertyEnumerated(String.class,
            "Screen.ActionBarBackgroundColor", 1, mm_cross_platform, true, swingColors);
        mp_cp_Screen_Full_Screen = mo_cp_Screen.produceMetaProperty(Boolean.class,
            "Screen.FullScreen", 1, mm_cross_platform);

        mr_cp_App_Screens = mm_cross_platform.produceMetaRole("App_Screens",
            Cardinality.ONE_TO_MANY, mo_cp_App, mo_cp_Screen);

        ...
    }

    ...
}
```

En analysant le code ci-dessus, il est aisé de constater que la création du méta-modèle par le biais des éléments du langage MetaL2 est répétitive et simple à comprendre.

BundleActivator est une interface fournie, quant à elle, par le framework OSGi et a pour rôle de gérer un plug-in OSGi. MetaDONE fournit une classe abstraite implémentant cette interface, demandant au développeur qui l'étend de simplement retourner le *MetadonePlugin* concerné. Le code du *BundleActivator* pour le méta-modèle d'une application mobile multi-plateforme se résume donc à :

```
import org.osgi.framework.BundleContext;
import metadone.client.plugin.AbstractPluginActivator;
import metadone.client.plugin.MetadonePlugin;

public class CrossPlatformActivator extends AbstractPluginActivator {
    @Override
    protected MetadonePlugin initializePlugin(BundleContext context) {
        return new CrossPlatform();
    }
}
```

5.2 Grasyla

Une fois que le méta-modèle fut intégré dans le logiciel, fournir un script GraSyla à MetaDONE était nécessaire afin d'illustrer les modèles construits sur base de notre méta-modèle. La suite de cette section décrit donc le contenu du script Grasyla à associer à notre méta-modèle.

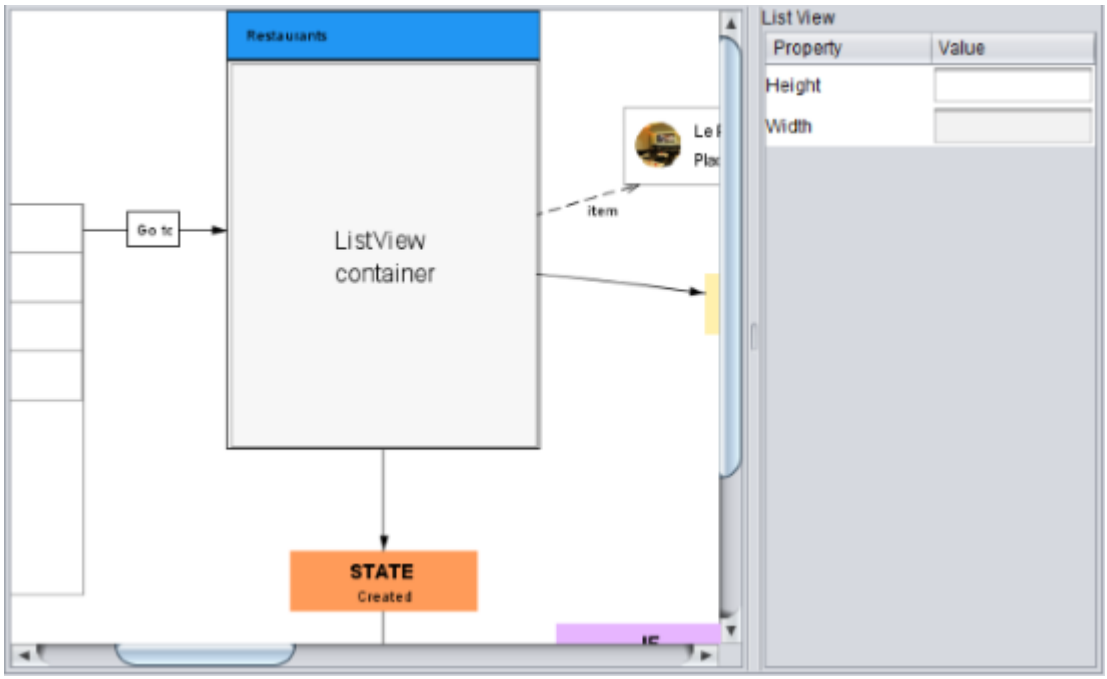

FIGURE 5.2 : Vue principale du modèle dans MetaDONE

Lorsque le développeur écrit son script Grasyla, il lui est possible de déterminer la vue principale de la fenêtre contenant le modèle. Dans notre cas, comme illustré dans la figure 5.2, la vue se compose :

- D'un panneau couvrant 80% de la fenêtre sur la gauche, chargé de contenir la partie visuelle de l'application multi-plateforme ;
- D'un panneau plus étroit sur la droite de la fenêtre, permettant de lister et d'éditer les propriétés des éléments graphiques contenus dans le panneau principal ;

Le code suivant correspond à la représentation visuelle du viewgroup *ListView*. Son contenu sera analysé juste après.

```
$ node one metaobject ListView = rectangle {
   background: "#F7F7F7"
   border: "line" { color: "#808080" stroke: "solid" { width: 1 } }

   space { width:
              if {
                 guard { condition: empty ($"ViewComponent.Width") 245 }
                 guard { value($"ViewComponent.Width") }
              }
           height:
              if {
```

```
                guard { condition: empty ($"ViewComponent.Height") 305 }
                guard { value($"ViewComponent.Height") }
             }
   }

   boxV {
      font : "SansSerif 20"
      "ListView"
      "container"
   }

   bitt: "shape" {
      for: [ $ListView_ListItemContainer.domain $ViewGroup_ViewGroupEvent.domain
           $MetaDoneContainer_ViewComponents.range $ChangeValueAction_ViewComponent.range]
   }

   action: "contextualMenu" {
      menuitem {
         label: "Add a list item"
         action : "click" { grv{s{{{
            def mr = metamodel.getMOByName("ListView_ListItemContainer").narrow2MetaRole()
            def this_list = self.getHead()
            def list_item =
                concretemodel.createObject(metamodel.getMOByName("ListViewItemContainer"))
            concretemodel.createRole(mr,this_list,list_item)
            }}}}
            "consume"
         }
      }
      ...
   }
}
```

Comme expliqué dans la section 3.2, une équation GraSyla est composée des éléments suivants : un foncteur, une multiplicité, une méta-classe de MetaL2, un objet à représenter et une représentation visuelle. L'objet représenté est bien une *ListView* et correspond à un *MetaObject* de notre méta-modèle. Dans cette équation, le foncteur est précisé par le mot *node*. Pour rappel, le foncteur est chargé de préciser à GraSyla dans quel « contexte » représenter l'objet de cette équation. Pour représenter les objets du méta-modèle, le script GraSyla représentant les applications mobiles n'utilise que 2 foncteurs : il s'agit de *node* et *properties*. Le premier précise à GraSyla que cette équation concerne la représentation de l'objet dans le panneau principal de la vue globale. Le foncteur *properties* précise quant à lui que l'équation représentera l'objet dans le panneau de propriétés. La figure 5.3 illustre la hiérarchie des différents foncteurs utilisés par notre script GraSyla.

Dans la suite de l'équation, *one* représente la multiplicité de l'objet représenté. Les multiplicités offertes par GraSyla sont les suivantes : *one*, *many* et *none*. Lorsque l'utilisateur écrit son script, il peut choisir de présenter un objet suivant sa multiplicité c'est-à-dire une représentation lorsque aucun objet de l'équation n'est présent dans le modèle (none), une représentation pour un seul objet (one), et une représentation lorsque le modèle possède plusieurs objets comme celui précisé dans l'équation (many). Dans notre cas, chaque instance d'objet doit être représentée de manière individuelle. C'est

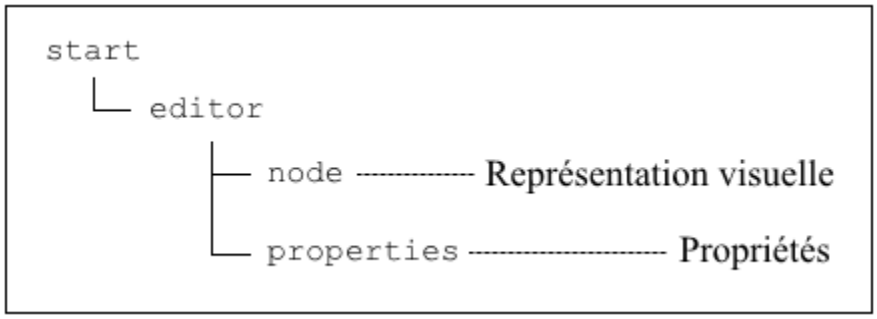

FIGURE 5.3 : Structure des foncteurs

pourquoi les multiplicités de nos équations sont toujours *one*.

A présent, il est possible de décrire ce que l'équation doit représenter. Elle est chargée de représenter une seule instance de *ListView* qui est un *MetaObject* de notre méta-modèle dans le panneau visuel de la vue principale. Le reste de l'équation est donc chargé de définir *comment* sera représentée cette ListView. Cette partie peut être découpée en plusieurs morceaux. Le premier morceau concerne la représentation visuelle a proprement parlé de l'objet. Le deuxième, qui commence par l'expression *bitt* est chargé de définir la place de l'objet dans les différents MetaRole qui le concerne. Autrement dit, la section bitt précise si l'objet est le domain ou le range de MetaRole spécifiques. Le dernier morceau concerne l'ensemble des actions liées à l'objet et commence par l'expression *action*.

Concernant la représentation visuelle de la ListView de l'équation précédente, celle-ci est représentée sous la forme d'un rectangle. L'expression *space* permet de préciser les dimensions de ce rectangle. Puisque l'utilisateur peut éditer la hauteur et la largeur de la liste par le biais de ces propriétés, la liste doit posséder une hauteur et une largeur « par défaut » pour ensuite s'adapter à la valeur de ces propriétés. C'est dans ce but que sont utilisées les expressions *if* et *guard* : Si la propriété de hauteur (ou de largeur) de la liste est vide, on précise une valeur par défaut. Sinon, on fixe la hauteur (ou largeur) de la liste à la valeur contenue dans la propriété. Dans le rectangle, un composant appelé *boxV* est également présent. Cette expression définit un espace invisible aux yeux de l'utilisateur, chargé de contenir d'autres éléments graphiques de manière verticale. Dans la même logique, GraSyla possède un composant appelé *boxH* qui contient d'autres éléments visuels et les affiche horizontalement. Dans le cas de la liste, la boxV contient simplement 2 labels et précise leur typographie. De manière simplifiée, la liste est donc représentée comme un rectangle de taille variable, avec un simple mot « ListView » en son centre.

La section *bitt*, comme expliquée précédemment, définit l'emplacement de la liste dans les différents MetaRole qui la concernent. L'attribut *shape* précise que lorsque la représentation du MetaRole sera créée, elle débutera sur les bords du rectangle. GraSyla permet de définir d'autres attributs comme *center* où la représentation du MetaRole débuterait au centre de l'objet.

Enfin, il y a également la section *action : « contextualMenu »* chargée de définir le menu affiché lorsque l'utilisateur clique droit sur la liste. Il est possible, dans Grasyla, de définir 2 types de menu. Le premier est un menu global, identique pour chacun des objets, relations... contenus dans le modèle. L'entièreté des composants est alors affichée dans ce menu et l'utilisateur est livré à lui-même pour créer et relier ses objets. Le deuxième type de menu est un menu contextuel, adapté à chaque objet du modèle. C'est donc vers cette option que nous nous sommes dirigés. En proposant un menu contextuel pour

chaque objet, nous permettons à l'utilisateur de construire son application de manière plus intuitive. Par exemple, si celui-ci affiche le menu d'une liste, il pourra constater qu'il est impossible d'ajouter directement d'autres éléments graphiques dans le viewGroup : il devra avant tout créer une entrée de liste. Cette entrée sera donc créée dans le modèle ainsi que le lien qui l'unit la liste, ce qui n'est pas le cas avec un menu global puisqu'on ne peut attacher le lien à aucun autre objet.

La représentation précédente est très importante car c'est grâce à elle que l'utilisateur va pouvoir visualiser au mieux le rendu de son application. Cependant, il lui est encore impossible d'éditer les caractéristiques spécifiques d'un composant (sa couleur, sa taille...). C'est dans ce but que le panneau properties a été créé : il va permettre d'éditer les propriétés d'un élément affiché dans le panneau principal. Voici donc l'équation de la ListView décrivant le contenu de l'objet dans le panneau properties :

```
$ properties one metaobject ListView = boxV {
   boxH { "List View" spring }
   table {
      columns: [ "Property" "Value" ]
      tr { "Height" textfield {
           value($"ViewComponent.Height")
           action: "validate" { update{$"ViewComponent.Height"}} }
      }
      tr { "Width" textfield {
           value($"ViewComponent.Width")
           action: "validate" { update{$"ViewComponent.Width"}} }
      }
   }
}
```

Le rendu final du panneau de propriétés équivaut à un tableau reprenant le nom de la propriété ainsi que sa valeur. Les éléments affichant ces valeurs sont des composants « éditables » comme des textfields ou des comboboxs, afin de permettre à l'utilisateur d'éditer facilement les propriétés. La figure 5.4 illustre le panneau de propriétés d'un écran (plus garni que celui de la ListView).

5.3 Rendu de la solution Smart Namur

Grâce à notre méta-modèle et notre script Grasyla, notre DSML pour représenter une application mobile multi-plateforme est terminé. Le logiciel est donc en mesure de modéliser entièrement notre application de référence (illustrée à la figure 1.1). La figure 5.5 montre le résultat auquel nous sommes arrivés. Le prochain chapitre est destiné à critiquer ce résultat, à définir quels sont les points forts et les points faibles de notre solution.

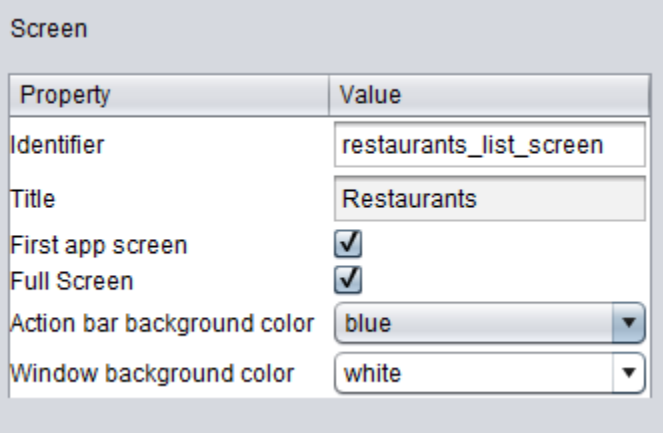

FIGURE 5.4 : Panneau de propriétés d'un écran

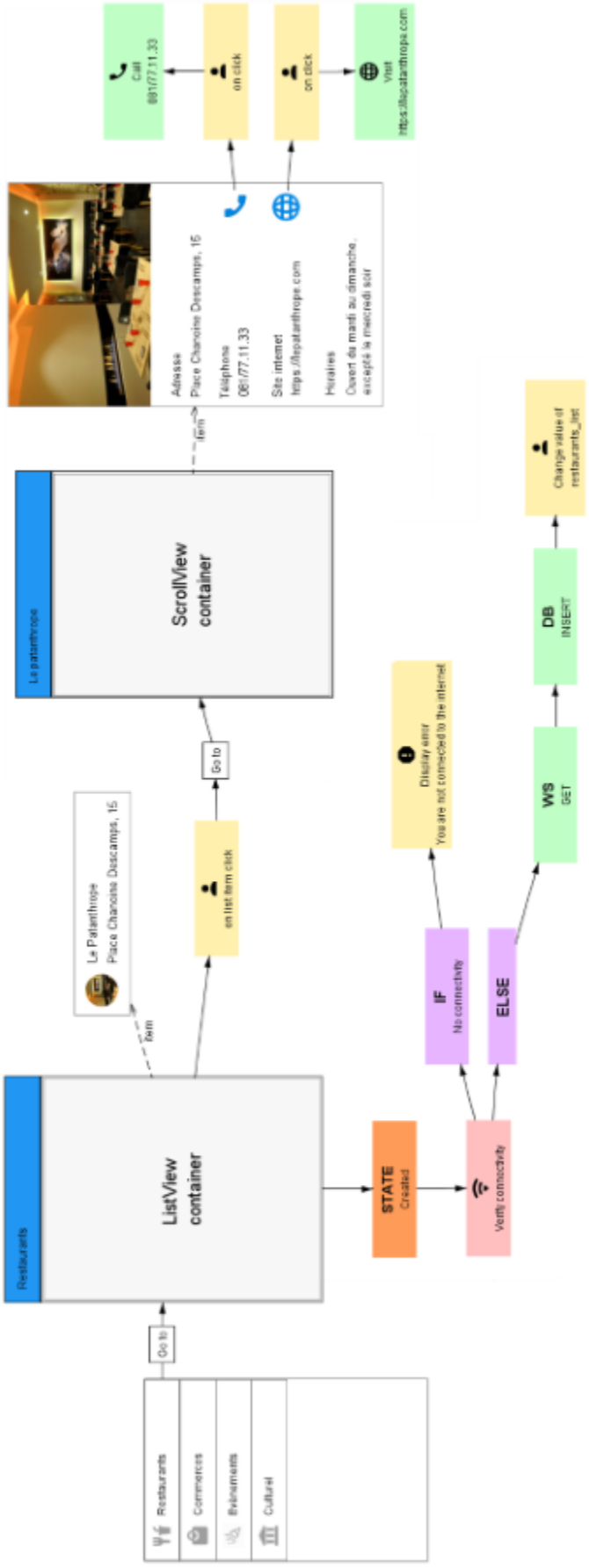

FIGURE 5.5 : Rendu final de Smart Namur

Chapitre 6

Critique de la solution

Dans ce chapitre, le résultat obtenu après l'implémentation de notre solution sera comparé avec l'application de référence présentée dans l'introduction de ce mémoire. Les possibilités de la solution seront ensuite comparées aux autres approches multi-plateformes par le biais de la conclusion tirée à la suite de l'état de l'art. Enfin, nous listerons les points forts et points faibles de MetaDONE dans la mise en place de ce DSML.

6.1 Critique de la solution

Dan cette section, nous comparerons le résultat de notre travail par rapport à la problématique de référence reprise dans l'introduction. Cette application nous a permis de définir les points les plus cruciaux dans la représentation d'une application mobile. Toutefois, ce n'est pas suffisant : après avoir mené des recherches, nous avons tiré plusieurs conclusions. En premier lieu, certains points étaient très importants à implémenter afin de répondre aux exigences de la majorité des applications mobiles. En deuxième lieu, la solution implémentée avec MetaDONE devait enrichir celle fournie par MetaEdit+ (Cf. conclusions du chapitre 2.5) afin d'ouvrir la création d'applications mobiles à un public plus large que celui des développeurs. C'est pourquoi nous analyserons notre solution par rapport aux conclusions tirées de l'état de l'art.

6.1.1 Comparaison avec la problématique de référence

Par rapport à l'application de référence illustrée à la figure 1.1, nous pouvons considérer notre travail comme abouti. Les éléments graphiques tels que le menu, la liste, les labels ou encore les images présents dans celle-ci sont bien représentés dans notre solution. Même si le résultat final modélise plus d'éléments que l'application de départ, il est tout de même aisé de visualiser les liens entre les différents écrans ainsi que de visualiser leur contenu comme présenté au début.

Toutefois, même si les exigences de départ sont satisfaites, nous pourrions reprocher au résultat actuel de ne pas réellement représenter le visuel final d'une application mobile. Cela se justifie par le fait que nous voulions modéliser facilement le contenu (qu'il soit visuel ou business) d'une application quelque soit la plateforme mobile. Cependant, au vu des possibilités offertes par MetaDONE, nous pourrions définir 2 nouveaux scripts Grasyla chargés de représenter le rendu final de l'application : l'un pour Android, l'autre pour iOS.

6.1.2 Comparaison par rapport aux conclusions de l'état de l'art

Le contenu du DSML

Comme nous pouvons le constater, des éléments supplémentaires sont présents dans le méta-modèle créé ainsi que dans le résultat obtenu après avoir illustrer l'app de référence, par rapport à la problématique de départ proposée à la section 1.1. Cela se justifie par rapport aux conclusions tirées de l'état de l'art. Ces conclusions listaient dans un premier temps les éléments devant absolument se trouver dans notre DSML pour représenter la majorité des applications mobiles. Pour rappel, ces éléments étaient :

- Des éléments graphiques comme des boutons, des fenêtres... ;
- Des évènements générés par l'appareil, par l'utilisateur... ;
- Des composants « business » comme une base de données, l'accès à des web services... ainsi que les interactions avec ceux-ci ;
- Des composants représentant les applications natives de l'appareil (SMS, Contacts...) ;
- Des accès aux fonctionnalités natives de l'appareil comme l'accès aux connections réseaux ou à la position ;
- Des composants modèles afin de décider de la structure de données utilisée.

La quasi totalité de ces éléments peuvent être représentés par notre DSML, comme expliqué dans le chapitre dédié à l'implémentation de la solution 5.1.1. Toutefois, 2 points doivent être soulignés.

Le premier concerne les composants business ainsi que les interactions avec ceux-ci. Dans le DSML, l'interaction avec un de ces composants est représenté sous forme d'une *DataStorageAction*. Chaque source de données possède sa série d'actions capables d'interagir avec elle (Cf la version complète du méta-modèle pour plus de détails [**?**]). Cependant, les composants business en eux-mêmes tels que la base de données ou le web service ne sont pas représentés. Nous ne pouvons donc pas qualifier la solution comme complète : si le code de l'application était généré à partir de notre modèle, il serait impossible de savoir sous quel format parser les données transmises par le web service ou par la base de données. Nous pourrions associer ces résultats à nos objets représentant le modèle de l'application, mais dans ce cas, plusieurs informations manquent. Les interactions avec les composants business devraient en plus définir quel élément du modèle serait concerné par l'opération et quelles « instances » seraient concernées. Par exemple, si on réalise une requête de suppression dans une DB, faut-il supprimer tout le contenu d'une table ? Un identifiant spécifique ? Ne garder que les 10 plus récents éléments et supprimer le reste ? Tout ces aspects ne sont pas couverts par le DSML et reviendraient à symboliser la gestion complète d'une source de données, quel qu'elle soit. Et ceci ne faisait pas partie de notre scope. Le DSML se contente donc de représenter l'interaction avec une source de données, en précisant le type de source concerné et le type d'interaction.

Le deuxième point concerne les objets dédiés à la représentation du modèle de l'application. Si ceux-ci sont repris dans le méta-modèle, ils ne sont pour l'instant pas modélisés par le script Grasyla. La représentation du visuel de l'application mobile couplé à celle de son modèle demande beaucoup plus d'efforts que les autres composants, même les plus business. En effet, dans une application, un élément graphique peut être source d'un évènement, un évènement peut être source d'une action... mais même si ces objets sont tous reliés entre eux, il est aisé de constater leurs différences et donc de les symboliser différemment les uns des autres. En ce qui concerne le modèle, ce n'est pas le cas. Pour justifier cela, reprenons notre application de référence, Smart Namur. Nous pouvons déjà cibler

plusieurs potentiels problèmes liés au modèle.

Le premier se passerait au niveau de l'écran représentant la liste des restaurants : Avec la solution actuelle, nous pourrions envisager de créer une classe du modèle que l'on nommerait « Restaurant » et de relier ses attributs aux widgets de la cellule de la liste. Mais lorsque nous analysons le style graphique de l'écran, nous pouvons remarquer que celui-ci est très générique et pourrait tout aussi bien être repris pour représenter la liste des évènements culturels de Namur par exemple. Il suffirait d'adapter le « contenu » de la cellule, à savoir lier les widgets de celle-ci vers les attributs d'une classe « CulturalEvent » et le tour est joué. Le tour est joué un peu vite en fait ... Comment représenter aisément aux yeux de l'utilisateur qu'un même écran peut utiliser 2 sources du modèle différentes ? Comment le prévenir que la liste ne devra plus être triée suivant le nom du restaurant mais par la date de l'évènement ? Comment, en réalité, adapter toute la logique business de l'écran suivant le modèle à représenter ? Une solution facile serait de dupliquer les écrans et ainsi adapter les éléments business suivant le contexte dans lequel nous nous trouvons. Mais si du code était généré en adoptant cette optique, cela engendrait énormément de duplication de code et ne serait pas optimal. Toute cette logique de généricité des écrans doit être creusée de manière beaucoup plus approfondie que ce qui nous a été possible de faire dans le cadre de ce travail.

Un deuxième problème pourrait survenir lorsque nous voulons adapter le visuel suivant le contenu d'une « instance » de modèle. Imaginons que nous voudrions adapter le contenu du label reprenant les horaires du restaurant si celui-ci est en vacances. Nous voudrions alors afficher un message similaire à « Fermé jusqu'au 4 avril compris » plutôt que les horaires d'ouverture normaux. Comment modéliser aux yeux de l'utilisateur cette situation ? Comment lui expliquer facilement que si le restaurant est en vacances (que l'on pourrait avoir sous forme d'un boolean), le texte repris dans le label sera différent et ce, sans alourdir la représentation visuelle ?

L'utilisateur doit également créer son modèle pour pouvoir le relier aux différents éléments de son application. Comment permettre à l'utilisateur, d'une part, de représenter le visuel et d'autre part, de créer son modèle (et adapter la vue en conséquence) ? Toutes ces interrogations demandent énormément de temps et de réflexion. Nous avons préférer nous concentrer sur l'ensemble de l'application plutôt que sur un aspect spécifique de celle-ci.

Au vu de ces remarques, nous ne pouvons pas considérer le travail comme terminé. Cependant, les résultats actuels prouvent que notre travail possède déjà ses avantages par rapport à d'autres outils. Il est possible de visualiser qu'un clic sur une cellule peut nous emmener vers un autre écran. Qu'une action peut en engendrer d'autres, suivant certaines conditions. Par rapport aux approches comme UsiXML ou IFML qui se concentrent sur un aspect unique d'un programme (le visuel pour l'un, l'ensemble des flux pour l'autre), notre solution possède l'avantage de se représenter le visuel de l'application et les interactions à l'intérieur de celle-ci d'un seul coup d'oeil.

Comparaison avec MetaEdit+

L'enrichissement du DSML s'est réalisé grâce aux différentes recherches menées dans l'état de l'art mais également en gardant une idée précise en tête : partir des possibilités offertes par MetaEdit+ et réadapter la solution aux environnements mobiles actuels [1]. Si le langage de MetaEdit+ ne se concentre pas réellement sur le visuel d'une application, il permet de rapidement cibler la logique et la

[1] Pour rappel, la solution fournie par MetaEdit+ était destinée aux téléphones S60. Pour plus d'informations, voir la section 2.2.1 dédiée à MetaEdit

navigation qui existe entre les différents écrans qui la composent. De plus, et c'est là son principal atout, MetaEdit est capable de générer le code correspondant au modèle créé par l'utilisateur. Cela ouvre donc la création d'applications mobiles à un public plus large que celui des développeurs : n'importe qui, moyennant une connaissance de l'outil et du cycle de vie d'une application mobile pourrait créer sa propre app.

Dans ce travail, nous sommes partis du DSML de MetaEdit+ que nous avons enrichi avec toutes les informations que nous avons pu accumuler sur le développement d'applications mobiles. Nous pouvons donc dire, pour la partie modélisation, que notre travail a atteint son objectif : remettre au goût du jour la solution de MetaEdit+. En ce qui concerne la génération du code, cette partie n'a pas été abordée dans ce mémoire mais il faut savoir que MetaDONE offre cette possibilité également. Il serait donc possible dans le futur, sur base de notre DSML, de générer le code d'une application en Java ou en Swift.

6.2 Critique de MetaDONE

Commençons par MetaL2, le langage de méta-modélisation de MetaDONE. MetaL2 est un langage très malléable et offre énormément de possibilités, ce qui permet de créer des méta-modèles complexes et variés. En effet, une solution capable de modéliser des réseaux de Petri a été réalisée avec MetaDONE, une autre encore pour modéliser des processus BPMN. Concrètement, cela signifie que MetaL2 est suffisamment évolué pour prendre en charge des sujets aussi complexes que BPMN (Cf. source [23]). Un autre atout majeur est que même si MetaL2 offre beaucoup de possibilités, il reste assez simple à appréhender et à manipuler.
Dans les points plutôt particuliers que couvre le langage, il permet par exemple de faire de « l'héritage multiple » : un MetaObject pourrait hériter du comportement de plusieurs MetaObject. Cela n'a pas été nécessaire pour notre méta-modèle mais pourrait s'avérer tout de même très utile. MetaL2 permet également de lier une propriété à un rôle, de définir un MetaRole comme le domain d'un autre rôle... Bref, beaucoup de manipulations sont possibles avec MetaL2. Cependant, MetaDONE ne s'arrête pas là : en choisissant d'intégrer les nouveaux méta-modèles dans des plugs-in OSGi, il est plus facile d'ajouter sa propre solution dans le logiciel. Aucun besoin de redémarrer MetaDONE pour intégrer le nouveau méta-modèle, il est possible de l'ajouter directement à chaud dans celui-ci. De plus, MetaDONE met à disposition plusieurs interfaces et classes abstraites pour faciliter la création d'un nouveau plug-in. Cela signifie que le développeur peut écrire son méta-modèle sans pour autant devoir connaître toute la logique interne de MetaDONE.

Passons maintenant à la partie plutôt dédiée à la symbolisation des modèles créés à partir d'un nouveau méta-modèle. L'idée d'assigner plusieurs scripts Grasyla à un même modèle est un réel atout : cela permet de visualiser celui-ci sous divers points de vue. Dans notre cas, puisque notre solution finale ne représente pas le visuel définitif d'une application, nous pourrions imaginer écrire 2 nouveaux scripts Grasyla : l'un pour représenter la version finale d'une application Android, l'autre pour une application iOS. Nous pourrions également créer un script qui se concentrerait plutôt sur les différents flux de l'application. Cette possibilité est un réel avantage par rapport à des logiciels similaires comme MetaEdit+.

Concernant GraSyla lui-même, celui-ci ne s'appréhende pas du tout de la même manière qu'un langage classique comme Java ou C. Il est donc plus difficile de maîtriser la totalité de ses capacités. D'ailleurs, durant ce travail, nous n'avons pas utilisé tout le potentiel de GraSyla, notamment dans l'utilisation des foncteurs où nous nous sommes limité à ceux fournis de base par la solution. Cependant, puisque la structure d'un fichier se présente sous forme d'équations, une fois GraSyla maîtrisé, il est

plutôt simple d'éditer ou de maintenir celui-ci. Puisque la logique d'une équation reste la même, la lecture d'un script devient donc plus aisée. Durant la création de notre script GraSyla, 2 points négatifs sont ressortis. Le premier concerne la personnalisation des éléments graphiques en utilisant GraSyla. Si le langage permet de symboliser beaucoup de composants, la « customisation » de ceux-ci était un peu moins complète ce qui, dans notre cas, nous empêcherait de symboliser fidèlement le rendu final d'une application mobile. Par exemple, il n'est pas possible en utilisant GraSyla de changer la couleur d'un texte. Les « widgets » offerts par Swing ne représentent pas du tout les composants d'une application Android ou iOS et éditer le style d'un de ces widgets demanderait beaucoup de temps. Toutefois, il faut faire remarquer que le but premier de GraSyla était de pouvoir modéliser les éléments d'un méta-modèle, pas de faire du prototypage : il paraît donc logique que les éléments graphiques ne soient pas aussi modulables que ce qui est requis dans notre cas. Le deuxième point que nous avons soulevé se présente dans la lecture du langage en lui-même. Puisque GraSyla n'est pas un langage couramment utilisé, les logiciels comme Eclipse présentent les scripts de manière brute : aucune coloration du code, pas d'auto-complétion... De plus, l'ajout d'une erreur dans le script n'est pas mise en évidence par l'éditeur, ce qui induit que celle-ci ne pourra être découverte qu'à la soumission du script à MetaDONE. Si ce point n'est pas critique, il permettrait un gain de temps considérable pour l'auteur du script.

Pour conclure, si l'aspect graphique offert par GraSyla n'était pas assez poussé pour notre travail (qui, je le rappelle, n'entrait pas en marge avec le but de départ de GraSyla), MetaDONE a déjà montré son potentiel en intégrant d'autres DSML dont un pour représenter les réseaux de Petri ou encore un second pour représenter des schémas BPMN. Il est à remarquer également que même si le visuel ne correspond pas tout à fait à ce que nous voudrions, notre méta-modèle a pu être entièrement symbolisé, que ce soit du point de vue visuel ou fonctionnel.

Chapitre 7

Perspectives et investigations

Dans ce chapitre, les perspectives d'évolution du travail accompli dans ce mémoire seront exposées. Ces pistes concernent la solution dédiée au développement multi-plateforme et donnent une idée de comment celle-ci pourrait évoluer afin de surpasser ses concurrents directs. Dans la mesure du possible, des pistes d'implémentation sont exposées afin de prouver la légitimité des perspectives énoncées ci-après.

Comme expliqué lors de la critique de la solution (Cf. 6.1.1), les éléments les plus importants d'une application mobile [1] sont modélisés dans notre DSML. Cependant, la gestion de la couche modèle n'a pas été assez approfondie dans ce mémoire, faute de temps. Il serait très intéressant de consacrer les ressources nécessaires pour faire évoluer cet aspect du travail. Il serait alors possible dans une unique solution non seulement de visualiser le design de l'application mais également de visualiser la couche « Modèle », reprenant l'ensemble des objets représentant les données de cette application ainsi que les interactions de la vue avec cette couche. Une piste serait par exemple de tirer profit de la flexibilité de GraSyla en définissant une série de foncteurs chacun chargé de modéliser un point de vue différent de l'application. En adaptant la vue globale du modèle, chaque point de vue pourrait alors être exposé à l'utilisateur, en alourdissant le moins possible la version actuelle du travail. Pour l'instant, la vue globale du modèle se découpe en 2 panneaux : le principal reprenant la représentation visuelle des objets et le second listant les propriétés de ces objets (Voir figure 5.2). Le panneau des propriétés est nécessaire puisqu'il faut pouvoir adapter les caractéristiques des objets, quel que soit la perspective dans laquelle l'utilisateur se trouve. Par contre, nous pourrions revoir l'espace laissé à la représentation visuelle de l'application. L'idée serait que cette vue ne soit pas présentée sous forme d'un « panneau » mais sous forme d'un tableau comprenant plusieurs onglets : un premier dédié à la représentation actuelle, un deuxième permettant uniquement à l'utilisateur de gérer le modèle de l'application, un troisième permettant de visualiser les interactions entre la vue et le modèle... Cette piste nécessiterait l'adaptation de l'ossature du modèle, reprise dans le script GraSyla ainsi que de la symbolisation des perspectives dédiées au modèle. Les interactions entre vue et modèle devraient sans doute être plus réfléchies et donc représentées de manière plus précise qu'actuellement dans le méta-modèle.

Une autre piste d'évolution qui pourrait être adoptée serait de converger vers une solution beaucoup plus ergonomique aux différentes plateformes mobiles. Comme expliqué dans le chapitre 1, chaque plateforme possède sa propre ergonomie, sa propre expérience utilisateur... ce qui idéalement pousse les

[1] Pour rappel, ces éléments ont été déterminés suite aux recherches menées durant l'état de l'art et exposés dans ses conclusions 2.5

développeurs à penser leur application de manière différente suivant les environnements. Malheureusement, ce n'est pas toujours possible puisque cela inclut du travail et de la recherche supplémentaire en terme de réflexion UX, de design, d'ergonomie.... Avec les possibilités offertes par MetaDONE, il serait envisageable d'implémenter une solution capable d'adapter le visuel d'une application mobile d'une plateforme à une autre. Une piste serait de créer un script GraSyla par plateforme de telle sorte que les éléments de l'application soient symbolisés suivant l'ergonomie requise. Ainsi, il sera aisé pour l'utilisateur de contempler le rendu de son app pour chaque environnement, sans se soucier des contraintes UX et ergonomiques puisque la solution s'en chargerait pour lui. Il faut cependant rappeler que le but principal de GraSyla n'est pas de faire du prototypage. Cela signifie que dans sa version actuelle, GraSyla ne pourra pas symboliser au pixel près le visuel final de l'application. Cette piste d'évolution nécessiterait un apport d'éléments graphiques dans GraSyla ainsi qu'une customisation (couleurs, taille de texte...) plus poussée de ces éléments.

Enfin, il est impossible de clore cette section sur les perspectives sans évoquer la piste d'une solution capable de générer le code d'une application mobile sur base d'un modèle créé dans MetaDONE. Une telle solution permettrait de « populariser » la création d'applications mobiles en la rendant accessible à un public plus large que celui des développeurs. En effet, il ne serait plus obligatoire de maîtriser un langage de programmation pour créer son application mais seulement de définir son visuel dans un modèle via MetaDONE. L'utilisateur devra cependant prendre connaissance des différents concepts liés aux application mobiles comme leur cycle de vie géré par le biais de *states*, le modèle contenu dans celles-ci, les interactions avec les sources de données... Mais cette évolution resterait marquante dans l'histoire du développement mobile multi-plateforme.

Chapitre 8

Conclusion

Dans ce mémoire, nous avons investigué une solution capable de réaliser une application mobile multi-plateforme. Cette solution a été implémentée grâce à un outil de méta-modélisation appelé MetaDONE. Le but principal de ce travail était donc de mettre à profit les capacités de l'outil MetaDONE dans la mise en place d'un DSML chargé de représenter des applications mobiles et ainsi, de cerner les limites et futures exigences de l'outil.

Afin de fournir une solution la plus complète possible, le but des recherches menées durant l'état de l'art était double. Dans un premier temps, celles-ci étaient chargées de cibler de potentiels concurrents à MetaDONE dans le cadre du développement multi-plateforme. Cela a permis de cerner les possibilités offertes par ces outils et de définir dans quelle optique la solution de MetaDONE serait construite, afin de parer aux manques de ses concurrents. Dans un second temps, les recherches se sont concentrées sur les points les plus importants à implémenter dans une application mobile. Cela a permis de définir un « contrat » de départ, définissant les éléments minimum que devait représenter le DSML.

Après avoir défini les exigences de départ, il était nécessaire d'investiguer sur les différentes fonctionnalités offertes par MetaDONE afin de cerner comment implémenter la solution avec l'outil. D'ailleurs, cette implémentation a abouti à un résultat plutôt encourageant puisqu'une application mobile de base a pu être entièrement modélisée. Cependant, si la partie modélisation fut concluante, la partie génération de code est quant à elle inexistante. Cette phase bien que nécessaire pour considérer le travail comme terminé était beaucoup trop importante en termes de temps et de recherches pour l'inclure dans ce mémoire.

Cependant, nous pouvons constater que la solution possède déjà ses avantages par rapport à ses concurrents mais qu'en plus, ses perspectives d'évolution sont nombreuses et permettraient d'en faire une solution unique sur le marché du développement multi-plateforme. Tout ceci est rendu possible grâce à la flexibilité de l'approche basée sur la méta-modélisation et grâce aux possibilités offertes par MetaDONE.

Bibliographie

[1] Albert Alderson. Meta-case technology. In *Software Development Environments and CASE Technology*, pages 81–91, Janvier 1991.

[2] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. Extending the interaction flow modeling language (IFML) for model driven development of mobile applications front end. *Mobile Web Information Systems - 11th International Conference, MobiWIS 2014*, pages 176–191, 2014.

[3] Antuan Byalik, Sanchit Chadha, and Eli Tilevich. Native-2-native : Automated cross-platform code synthesis from web-based programming resources. *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming : Concepts and Experiences*, pages 99–108, 2015.

[4] Julien Cadot. Evolution du marché des smartphones de 2009 à 2016, Février 2017. `http://www.numerama.com/tech/174972-apple-samsung-et-les-autres-evolution-du-marche-des-smartphones.html`.

[5] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef, and Ayman M. Wahba. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, pages 1–24, Août 2015.

[6] Vincent Englebert and Krzysztof Magusiak. Metadone : Plugin architecture. Technical report, Université de Namur, Faculté des Sciences Informatiques, Centre de recherche PreCISE, 2011.

[7] Vincent Englebert and Krzysztof Magusiak. The grasyla 2 language. Technical report, Université de Namur, Faculté des Sciences Informatiques, Centre de recherche PreCISE, 2013.

[8] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *Lecture Notes in Business Information Processing*, volume 140, pages 120–138, Département des systèmes d'information, Université de Münster, Allemagne, 2013.

[9] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with md2. *28th Annual ACM Symposium on Applied Computing*, pages 1–8, Mars 2013.

[10] Benjamin Hubert. Analyse comparative de logiciels multi-plateforme. Master's thesis, Université catholique de Louvain (UCL), B-1348 Louvain-la-Neuve, Belgique, 2013.

[11] Hosein Isazadeh and David Alex Lamb. Case environments and metacase tools. Technical report, Queen's University, Department of Computing and Information Science - Kingston, Ontario, Février 1997.

[12] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. Usixml : A user interface description language for context-sensitive user interfaces. pages 1–8, 2004.

[13] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. Usixml : a user interface description language for specifying multimodal user interfaces. pages 1–7, 2004.

[14] Nathan McAlone. Most used mobile apps in usa in 2016, Août 2016. `http://www.businessinsider.com/top-apps-of-2016-so-far-2016-8/`.

[15] Janne Merilinna. Domain-specific modelling language for navigation applications on s60 mobile phones. pages 1–5, Janvier 2008.

[16] MetaCASE. Abc to metacase technology. Technical report, MetaCASE Consulting Oy, Ylistönmäentie 31, 40500 Jyväskylä - Finlande, 2004.

[17] The metadone research project. `https://sites.google.com/site/precisemetadoneresearchproject/`.

[18] Metacase - metaedit+. `http://www.metacase.com/`.

[19] Le mobile en belgique en 2016. `http://advertising.sanoma.be/fr/insights/le-mobile-en-belgique-anno-2016`.

[20] Kenneth Olmstead and Michelle Atkinson. An analysis of android apps permissions, Novembre 2015. `http://www.pewinternet.org/2015/11/10/an-analysis-of-android-app-permissions/`.

[21] André Ribeiro and Alberto Rodrigues da Silva. *Development of Mobile Applications using a Model-Driven Software Development Approach.* Instituto Superior Técnico Lisbon, Portugal, 2014.

[22] André Ribeiro and Alberto Rodrigues da Silva. Xis-mobile : a dsl for mobile applications. In *ACM Symposium on Applied Computing*, pages 1316–1323, Instituto Superior Técnico Lisbon, Portugal, 2014.

[23] Arnaud Simon. Implémentation d'un éditeur bpmn au sein d'un outil de méta-modélisation. Master's thesis, Université de Namur, Département Informatique, 2014.

[24] Adrian Stanciulescu, Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, and Francisco Montero. A transformational approach for multimodal web user interfaces based on usixml. *05 Proceedings of the 7th international conference on Multimodal interfaces*, pages 259–266, 2005.

[25] Juha-Pekka Tolvanen. Metaedit+ : Integrated modeling and metamodeling environment for domain-specific languages. pages 690–691, Octobre 2006.

[26] Juha-Pekka Tolvanen. Metaedit+ : The mobile ui example. Technical report, MetaCASE Consulting Oy, Ylistönmäentie 31, 40500 Jyväskylä - Finlande, Août 2014.

[27] Eric Umuhoza. Domain-specific modeling and code generation for cross-platform multi-device mobile apps. pages 1–11, 2015.

[28] Eric Umuhoza, Hamza Ed-Douibi, Marco Brambilla, Jordi Cabot, and Aldo Bongio. Automatic code generation for cross-platform, multi-device mobile apps : some reflections from an industrial experience. pages 1–9, Octobre 2015.

[29] Usixml user interface extended markup language. `http://www.usixml.org/`.

[30] Xamarin. `http://www.xamarin.com/`.

Annexe A

DSML : Méta-modèle

Dans ce mémoire, une version simplifiée du méta-modèle du DSML destiné à représenter des applications mobiles multi-plateformes a été présentée. Cette annexe reprend la version complète de ce méta-modèle, incluant tous les éléments qui n'ont pas été exposés précédemment. Cette version est également disponible à cet endroit : `https://www.dropbox.com/s/jzi16qanhogk0si/complete_mobile_multi_platform_meta_model.png?dl=0`.

FIGURE A.1 : Version complète du méta-modèle présenté dans ce mémoire