MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

Évaluation de la méthode de conception d'usines à logiciels BESPOKE au travers d'une étude de cas

JACOBS, Pierre

Award date:
2023

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

Évaluation de la méthode de conception d'usines à logiciels *BESPOKE* au travers d'une étude de cas

JACOBS, Pierre

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : ENGLEBERT, Vincent

Mémoire présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à finalité spécialisée en Software engineering

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Remerciements

Je tiens à remercier Monsieur Vincent Englebert, professeur en faculté de sciences informatiques à l'Université de Namur et promoteur de ce mémoire, pour sa disponibilité, son aide et ses conseils tout au long de la réalisation de celui-ci. Je tiens également à remercier Madame Maouaheb Belarbi pour ses éclaircissements quant à certaines de mes questions techniques.

Un grand merci à mes parents et mon frère qui m'ont accompagné tout au long de mes études et qui m'ont permis d'arriver là où j'en suis aujourd'hui, tant grâce à leur support émotionnel que financier. Ce travail représente la quintessence des valeurs qu'ils m'ont transmises.

Enfin, merci à Pierre-Igor Vervlimmeren, dont la manière de concevoir les choses diffère de la mienne, pour ses remarques et critiques judicieuses qui m'ont ainsi permis d'aborder certains problèmes depuis un angle nouveau et différent.

Résumé

Les lignes de produits logiciels sont un paradigme de création logicielle qui aspire à générer de manière dirigée des produits partageant un ensemble commun de caractéristiques. Ces produits appartiennent à une même famille de logiciels et ces déclinaisons permettent ainsi d'exprimer la variabilité au sein d'un domaine d'application donné. Un client particulier peut alors exprimer des exigences sur ce domaine. Celles-ci seront prises en compte par la ligne de produits dans l'optique de générer un logiciel sur mesure répondant aux besoins émis. La structure méthodologique *Bespoke* met en place un ensemble de mécanismes agnostiques d'un point de vue technologique qui proposent de diriger intégralement la conception d'une ligne de produits logiciels, aspirant ainsi à faciliter l'adoption du paradigme qu'elle promeut. Ce document vise à mettre en pratique les mécanismes exposés par la méthodologie *Bespoke* afin de valider la méthode et de proposer des améliorations dans le cas contraire. À cette fin, un cas d'application holistique lui est soumis et des remarques constructives sont formulées en symbiose tout au long du document. La ligne de produits logiciels développée au terme des différentes étapes s'est révélée être fonctionnelle, bien que certains mécanismes durent être adaptés en conséquence, à l'instar des tactiques d'implémentation de la variabilité. Somme toute, la structure méthodologique *Bespoke* s'avère convenir à la conception intégrale d'une ligne de produits logiciels mais nécessite toutefois certaines améliorations et considérations supplémentaires, afin de parfaire les mécanismes proposés.

Mots-clés *: Ingénierie dirigée par les modèles, Ligne de produits logiciels, Variabilité, Bespoke, Cas d'application*

Table des matières

Table des remarques

Chapitre 1

Introduction

À l'heure de la transition vers le numérique, un nombre croissant de services tend à se digitaliser. En particulier, l'objectif principal est de tirer parti des avancées technologiques afin de proposer des services au plus grand nombre. La quantité de logiciels demandés et produits s'en retrouve alors démultipliée. En conséquence, certaines entreprises se retrouvent parfois en charge de dizaines de logiciels différents. Celles-ci s'efforcent de répondre au mieux aux demandes de leurs clients et consacrent alors de précieuses ressources humaines et techniques à leur conception et maintenance. Par conséquent, il est fort probable que ces logiciels distincts présentent un ensemble commun de caractéristiques. Peut-être certains d'entre eux utilisent-ils le même service d'hébergement pour leur serveur. Peut-être d'autres sont-ils destinés à une même catégorie d'utilisateurs finaux. Peut-être d'autres encore présentent-ils la même architecture logicielle, indépendamment de leur plateforme, ou partagent-ils certaines exigences fonctionnelles. Il existe ainsi une kyrielle d'exemples similaires. Il pourrait donc être bienvenu de prendre du recul et de chercher à factoriser ces caractéristiques, si d'aventure le domaine d'application s'y prête. En effet, les exploiter judicieusement permettrait dès lors d'économiser de précieuses ressources, humaines et techniques, que les entreprises pourraient mettre à profit dans d'autres domaines.

1.1 Ligne de Produits Logiciels

Le paradigme de l'ingénierie dirigée par les modèles est un paradigme de développement logiciel au sein duquel la manipulation de modèles est placée au centre des processus d'ingénierie. Ces modèles peuvent ainsi être employés afin d'interpréter ou de représenter des exigences propres à un domaine ou d'idéer et de représenter une architecture logicielle particulière. L'ingénierie dirigée par les modèles peut avoir une déclinaison généralisée à l'instar de l'utilisation d'*UML* [100], de *SysML* [95] et d'*ArchiMate* [6], ou une déclinaison davantage spécialisée, nécessitant par conséquent l'élaboration de langages de modélisation spécifiques à des domaines - *DSMLs*. Un usage fréquent d'un tel DSML consiste à appréhender par les modèles les variations propres à des systèmes informatiques appartenant à un même domaine [5, 13]. Le concept qui cristallise ce processus est celui de *ligne de produits logiciels*. Une ligne de produits logiciels représente ainsi une famille de logiciels, appelés *configurations*, qui partagent un ensemble commun de caractéristiques relatives à un domaine particulier. Chaque configuration présente également des spécificités qui lui sont propres. Dès lors, une configuration distincte est générée de manière dirigée et automatique ou semi-automatique conformément aux modèles ainsi définis, afin de répondre aux exigences formulées par un client particulier. Cet aspect génératif permet par conséquent d'accélérer la mise sur le marché d'un produit et d'en faciliter sa maintenance. Somme toute, une ligne de produits logiciels permet de réduire les coûts de développement et de maintenance de systèmes logiciels en tirant parti des similitudes exhibées par les configurations d'une même famille. Elle promet en outre une personnalisation rapide d'un produit afin de répondre aux besoins spécifiques des clients. Dans la suite de ce document, le terme *ligne de produits logiciels* se réfère à la définition formelle suivante [69] :

> *Une ligne de produits logiciels est un ensemble de systèmes logiciels qui partagent un ensemble commun et géré de caractéristiques communes répondant aux besoins spécifiques d'un segment de marché ou d'une mission particulière et qui sont développés à partir d'un ensemble commun d'artefacts de base et d'une manière dirigée.*

Cependant, l'ingénierie dirigée par les modèles peine à se faire une place dans le monde de l'industrie. Bien que de multiples exemples d'utilisation aient été fructueux [34, 43, 16, 2, 18], le manque d'outils complets et dirigés ne facilite pas son adoption. L'étude pratique réalisée en 2019 par J.M. Horcas et son équipe a démontré qu'aucun outil à l'heure actuelle ne permettait de mener à bien à lui seul l'intégralité du processus d'ingénierie par les modèles de manière aisée [47]. Les outils permettant de réaliser le processus complet sont tous limités

selon certains aspects techniques et se spécialisent soit dans l'espace du problème, soit dans celui de la solution. Faute de mieux, il est donc obligatoire de maîtriser plusieurs de ces outils et de les combiner au mieux.

1.2 Méthodologie *Bespoke*

Dans l'optique de résoudre cette problématique, une nouvelle structure méthodologique nommée *Bespoke* est présentée [32]. Son objectif est de proposer une méthodologie agnostique d'une technologie particulière, permettant d'aborder l'espace du problème et de la solution de manière conciliée et ergonomique. Il est toutefois important de noter que cette structure méthodologique est actuellement en cours de développement. La figure 1.1 dépeint la méthodologie proposée. Celle-ci est divisée en deux plans. Le plan *SPL Engineer* développe la méthodologie à suivre pour la réalisation de la ligne de produits logiciels. Le plan *Configuration Engineers* présente quant à lui la méthodologie à suivre pour générer une configuration particulière, en accord avec la ligne de produits logiciels développée.

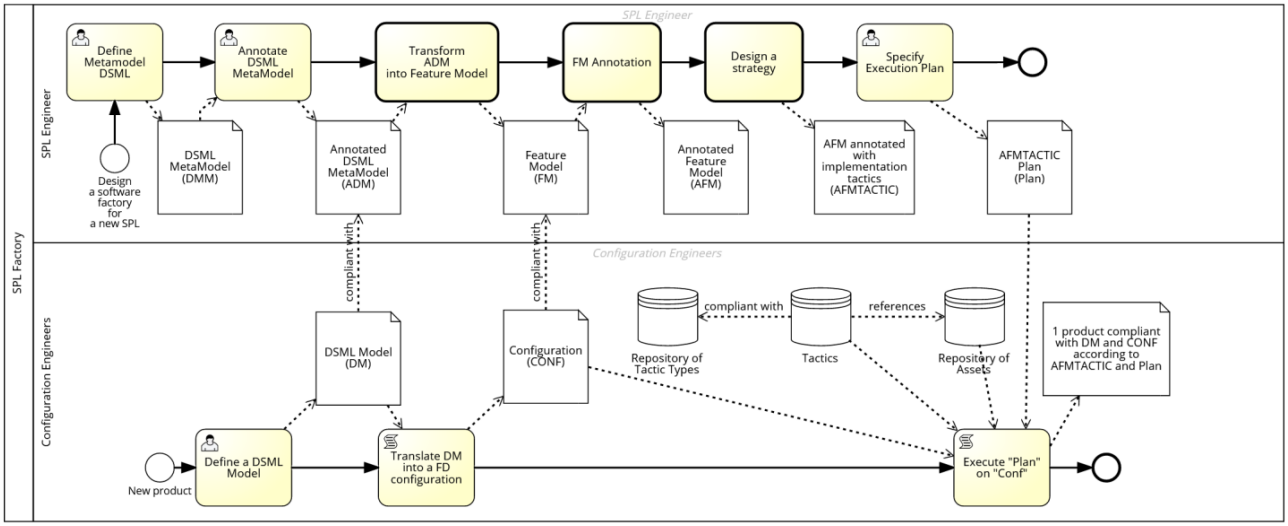


FIGURE 1.1 – Méthodologie proposée par *Bespoke* [32].

Plan *SPL Engineer* La première étape consiste en la définition d'un métamodèle afin de décrire le domaine d'application. Celui-ci est ensuite annoté afin de la transformer de manière automatique en diagramme de caractéristiques. Celui-ci permet d'obtenir une vue arborescente du domaine d'application. Des annotations y sont ensuite ajoutées pour guider la génération de code, en indiquant notamment les temps de liaison ou d'activation des caractéristiques. Sur base du diagramme de caractéristiques annoté sont manuellement dérivées des stratégies d'ingénierie. Il s'agit ici de déterminer quelles tactiques sont les plus appropriées pour implémenter une caractéristique particulière. Enfin, la dernière étape consiste en l'arrangement temporel des tactiques sélectionnées, influencées entre autres par les temps de liaison et d'activation.

Plan *Configuration Engineers* Lorsqu'un nouveau produit est demandé, la première étape consiste en la création d'un modèle conforme au métamodèle précédemment mentionné. Le nouveau modèle est ensuite transformé en une configuration particulière, conforme quant à elle au diagramme de caractéristiques mentionné ci-avant. Enfin, la configuration est implémentée de manière semi-automatique afin de délivrer le produit fini. Par ailleurs, cette dernière étape n'est à l'heure actuelle pas encore prise en charge par la méthodologie.

1.3 Contributions

1.3.1 Objectifs du Mémoire

L'objectif de ce mémoire est de poser un regard critique sur la méthodologie *Bespoke* présentée ci-avant, en mettant en œuvre ses différentes étapes. En particulier, il s'agit de concrétiser la conception et l'implémentation d'une ligne de produits logiciels, visant plusieurs systèmes d'exploitation, en mettant en pratique les mécanismes de stratégies d'implémentation et de plan d'exécution proposés par la structure méthodologique *Bespoke* et d'observer comment ils réagissent, dans l'optique de critiquer et de raffiner la méthode. Ces mécanismes

correspondent respectivement aux étapes *Design Strategy* et *Execution Plan* présentés par la figure 1.1. À ce titre, une ligne de produits logiciels est développée sur un cas d'étude aisé à communiquer. Dès lors, le cas d'application théorique d'une entreprise fictive *BetterSoft* [31] lui est soumis. Celui-ci est par ailleurs représentatif des thèses proposées sur le sujet des lignes de produits logiciels et a la particularité d'être suffisamment holistique pour être représentatif d'un cas d'étude réel. Ce cas d'application sera de fait le premier à traverser intégralement les différentes étapes de la méthodologie. Ainsi, il permettra la mise en avant de résultats plus pertinents que ceux de cas d'utilisation préparés sur mesure à chaque étape.

Remarque(Rem·0) ► Exemple de Remarque

La description de la remarque en lien avec la section courante.

La remarque (Rem·0) présente le formalisme employé pour formuler les remarques émises à propos de la structure méthodologique *Bespoke*. Celles-ci peuvent être positives ou négatives et sont liées à la section dans laquelle elles sont formulées. Le titre de la remarque introduit le concept critiqué tandis que sa description propose une explication plus détaillée de celle-ci.

1.3.2 Portée du Mémoire

L'objectif du mémoire est de questionner la méthodologie *Bespoke* au travers d'un cas d'application. Celle-ci étant imposée et au vu des questions de recherche formulées, établir un état de l'art de la structure méthodologique ne semble par conséquent pas pertinent. Cependant, il est possible de consulter les articles publiés [31, 32] afin d'en prendre connaissance. En outre, afin de ne pas se perdre tant la structure méthodologique est vaste, les hypothèses restrictives suivantes sont formulées :

(H1) L'entreprise fictive *BetterSoft* ne fera pas l'objet d'une étude approfondie en elle-même, car l'objectif de ce document n'est pas d'en réaliser une analyse économique ;

(H2) Ce document ne vise pas à comparer *Bespoke* à ses concurrents les plus aboutis, à l'instar de *Pure::variants* [14], *FeatureIDE* [66] ou *Clafer* [8]. Cet exercice a déjà été réalisé [32, 71] ;

(H3) Bien que la faculté d'automatisation de la méthodologie sera mise à l'épreuve par le biais d'implémentations concrètes, il ne s'agit pas ici de tenter d'automatiser une ou plusieurs étapes en particulier. Cet exercice est laissé comme travail ultérieur. Par ailleurs, certaines étapes de la méthodologie ont déjà fait l'objet de tentatives d'automatisation [15] ;

(H4) Ce document ne s'intéresse pas à la détection et la résolution de liens d'ingérence [31] entre les caractéristiques. Cet exercice est également laissé comme travail ultérieur ;

(H5) Ce document n'aborde ni le test ni l'évolution de la ligne de produits à logiciels, car cela sort du cadre de la recherche. Toutefois, cet exercice pourrait être réalisé ultérieurement ;

(H6) La ligne de produits logiciels ne fait ici pas l'objet d'une implémentation complète. Les fonctionnalités que doivent présenter les produits générés ne seront ainsi pas entièrement rencontrées mais seront émulées par des *mocks*, car il est ici question de s'intéresser aux mécanismes d'agencement des stratégies d'implémentation et non l'implémentation elle-même ;

1.4 Plan du Mémoire

Ce travail a pour but de mettre en œuvre la méthodologie *Bespoke*. Ainsi, le plan de ce mémoire suivra les étapes proposées par celle-ci, en l'articulant autour du cas d'application académique *BetterSoft* [31]. Les étapes liées entre elles des plans *SPL Engineer* et *Configuration Engineers* seront abordées en même temps. Par exemple, lorsque les diagrammes de caractéristiques seront abordés, appartenant donc au plan *SPL Engineer*, des exemples de configurations conformes à ceux-ci seront également présentés, appartenant eux au plan *Configuration Engineers*. De plus, les langages utilisés pour les différentes étapes sont définis dans le chapitre 7, *Méthodologie*, de l'ouvrage syllabus accompagnant la méthodologie *Bespoke* [31].

Afin de formuler les concepts propres aux lignes de produits logiciels, ainsi que de présenter les techniques de modélisation et d'implémentation existantes et compatibles, le chapitre 2 propose une revue succincte des concepts qui aideront à la réalisation de ce mémoire. Le chapitre 3 développe quant à lui la présentation du cas d'application *BetterSoft* et la définition de son domaine, car elles constituent une condition préalable à remplir pour débuter la méthodologie *Bespoke*. Par la suite, une analyse poussée du domaine d'application est effectuée dans le chapitre 4. Les exigences du domaine y sont formalisées sous la forme d'un métamodèle, transformé

ensuite en diagramme de caractéristiques. Ceux-ci, ainsi que leur version annotée, sont présentés. Le diagramme de caractéristiques annoté est par ailleurs exprimé grâce au langage *FeatAll*. En outre, un scénario de commande de logiciel intitulé *JarreDeux* est introduit pour les illustrer. Le chapitre 5 détaille la conception du domaine d'application en explorant la variabilité de la ligne de produits logiciels. Il reprend les tactiques employées pour mettre en œuvre chaque point de variation grâce au langage *TacticType* et explicite la mise en place des stratégies d'utilisation des tactiques grâce au langage *Strategy*. Le chapitre 6 définit le plan d'exécution à mettre en pratique pour implémenter la ligne de produits logiciels et ordonner les tactiques précédemment définies. Il présente ensuite une implémentation fonctionnelle de celle-ci, capable de générer certaines configurations, dont celle du logiciel *JarreDeux*. Enfin, le chapitre 7 propose une rétrospective ainsi qu'une conclusion tirées de l'expérience engrangée suite à l'utilisation de la méthodologie *Bespoke*. Des pistes et perspectives de travail futur y sont également proposées.

Chapitre 2

Expression de la Variabilité

Ce chapitre vise à offrir une vue d'ensemble sur les fondamentaux et concepts liés aux lignes de produits logiciels. Ceux-ci sont nécessaires à la compréhension du travail présenté dans ce document et permettent d'étayer des notions qui aideront à sa réalisation. Il introduit les concepts de points de variation et d'usine à logiciels pour exprimer la variabilité des exigences d'un segment de marché. Certains formalismes de modélisation des exigences sont également introduits, ainsi que des tactiques d'implémentation concrètes de la variabilité au sein de la ligne de produits logiciels. Ces tactiques sont également analysées à la lumière du développement mobile multiplateforme. Tel qu'annoncé par l'hypothèse (H2), ce chapitre ne prétend pas présenter une revue exhaustive des travaux relatifs au domaine des lignes de produits logiciels [46], à l'instar des outils ou structures méthodologiques développées dans ce cadre. En effet, ceux-ci ne sont pas pertinents dans le cadre de ce mémoire. À ce titre, seuls les concepts nécessaires sont présentés. Toutefois, diverses revues de la littérature plus larges existent à ce sujet [15, 47, 10, 64, 93, 21].

2.1 Ligne de Produits Logiciels

Une ligne de produits logiciels vise à minimiser les coûts de production et de maintenance [70] engendrés par l'explosion combinatoire de logiciels s'attelant à répondre aux besoins d'un segment spécifique de marché [69, 96, 70]. Outre cette minimisation des coûts, elle promet une flexibilité accrue de la gestion d'un ensemble d'artefacts, de techniques et d'exigences exprimés au-dessus d'un segment de marché [22]. L'entreprise suédoise *CelsiusTech Systems AB* [20] a développé dans les années 1980 l'une des premières lignes de produits logiciels, en réponse à deux commandes simultanées de logiciels navals [17]. Elle a ainsi posé les fondations de ce paradigme de développement logiciel, qui n'a cessé d'être étendu depuis. Une myriade de nouveaux outils aidant à leur conception [10] et de cas d'application les mettant en place ont ainsi vu le jour [64]. Il existe deux catégories de lignes de produits logiciels [82] : certaines sont *statiques*, tandis que d'autres sont *dynamiques*. La différence réside dans la capacité des lignes de produits logiciels dynamiques à adapter et transformer leur structure interne pendant leur exécution [9], au prix d'une difficulté d'implémentation et d'une instabilité logicielle accrue [107]. Cependant, les principes qui régissent ces deux catégories restent similaires.

2.1.1 Points de Variation

Une ligne de produits logiciels permet d'exprimer les similitudes et les différences entre les exigences des logiciels à produire. Elle présente donc un *ADN* élémentaire qu'il est possible de raffiner en fonction des besoins émis par un client particulier. Ainsi, on appelle [62] :

— **Points de variation** : les caractéristiques qui diffèrent d'un produit à l'autre ;

— **Points communs** : les caractéristiques communes partagées par l'ensemble des produits de la ligne de produits logiciels ;

— **Spécificités propres** : les caractéristiques spécifiques à un produit particulier et qui sont trop particulières pour être prises en compte dans la ligne de produits logiciels.

Les points de variation permettent donc d'exprimer la variabilité au sein de la ligne de produits logiciels. Chaque point de variation peut être satisfait par plusieurs *variantes*, dont le choix dépend des dépendances entre celles-ci. Par exemple, deux variantes peuvent être incompatibles entre elles et ne peuvent donc pas être sélectionnées en même temps. En outre, certaines variantes de certains points de variation peuvent présenter des *contraintes de dépendance*, expriment ainsi des contraintes entre celles-ci. Par exemple, la sélection d'une

variante *V1* d'un point de variation *P1* oblige la sélection d'une variante *V2* d'un point de variation *P2* [62, 69].

Différentes contraintes logiques de sélection de variantes peuvent être exprimées sur un point de variation distinct. Ainsi, un groupe de variantes satisfaisant un point de variation peut être [25, 72, 11, 93] :

— **Optionnel** : une variante peut être choisie ou non ;

— **Obligatoire** : une variante doit être choisie ;

— **Disjonctif** : parmi un groupe de variantes, toute combinaison de sélection est possible : aucune, seulement une, plusieurs, toutes ;

— **Alternatif** : parmi un groupe de variantes, une et une seule variante peut être choisie à la fois.

Enfin, un point de variation possède un état d'ouverture. Celui-ci détermine si le point de variation peut être étendu par de nouvelles variantes ou non une fois la ligne de produits logiciels mise en place. Ainsi, on distingue les points de variations suivants [93, 103] :

— **Ouvert** : le point de variation peut être étendu par la suite ;

— **Fermé** : le point de variation ne fera jamais l'objet d'une extension future.

2.1.2 Phases de Liaison

Chaque point de variation nécessite des artefacts à mettre en place afin de le réaliser. Ceux-ci font par conséquent l'objet d'un moment de liaison avec le point de variation. On distingue ainsi trois grande phases [31, 103]. La figure 2.1 présente les étapes qu'il est possible de sélectionner lors de celles-ci.

— **Phase de sélection** : cette phase désigne le moment où la caractéristique renseignée par un point de variation peut être introduite, afin d'être activée par la suite ;

— **Phase d'activation** : cette phase désigne le moment où une caractéristique peut être activée pour les produits qui seront générés.

— **Phase d'extension** : cette phase désigne le moment où il est possible de venir étendre un point de variation noté comme *ouvert*.

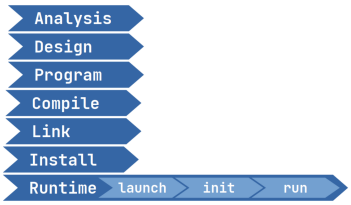


FIGURE 2.1 – Grandes étapes du cycle de vie d'un logiciel [31]

Un temps de liaison est noté comme suit : $[debut–fin]$, où le temps de début survient en premier dans le cycle de vie d'un logiciel. Les temps de liaison des différentes phases doivent respecter un ordre temporel précis. Celui-ci est présenté par la figure 2.2.

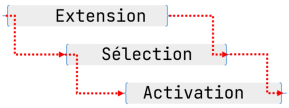


FIGURE 2.2 – Agencement temporel des phases [31]

2.1.3 Usine à Logiciels

Lorsque la variabilité nécessaire pour exprimer un domaine d'application désigné par un segment de marché prend une ampleur sans précédent, trop importante pour être gérée grâce à des moyens traditionnellement mis en place à l'instar des *frameworks*, l'utilisation d'une ligne de produits logiciels se justifie. Cette ligne de produits est mise en œuvre de façon concrète par une *usine à logiciels*, définie comme suit [31] :

> *Une usine à logiciels désigne toute procédure permettant de produire un système logiciel (en tout ou en partie) à partir de descriptions de haut niveaux centrées sur le domaine d'application. Le spectre d'action de cette usine, c-à-d. tous les besoins rencontrés par les systèmes pouvant être générés par cette procédure, désigne le domaine d'application. Cette procédure peut être automatique, semi-automatique ou manuelle.*

Une usine logicielle représente ainsi un logiciel particulier capable de produire d'autres logiciels. Il existe à cette fin deux grandes approches [58]. La première approche est dite *compositionnelle*. Elle vise à combiner au mieux des composants réutilisables, isolés pour offrir une claire séparation des responsabilités. Cette approche est souvent transparente au niveau du code source car la composition n'y est pas explicitée. Cependant, celle-ci présente des désavantages contraignants, à l'instar d'une maintenabilité logicielle plus complexe [55, 56]. La seconde approche est dite *annotationnelle* et propose d'adopter des approches plus traditionnelles. En particulier, il s'agit de diriger la production d'un logiciel grâce à des annotations. Celles-ci peuvent s'exprimer de diverses manières, à l'instar des mesures *préprocesseures* [65] ou de la métaprogrammation à l'aide de traits [78].

Ces deux approches sont distinctes mais restent compatibles [58]. Les combiner judicieusement permet de tirer parti de leurs avantages respectifs et de mitiger leurs contraintes. La section 2.3 s'aventure plus en détail dans des tactiques d'implémentation concrètes, permettant de réaliser la procédure de production des différents logiciels par l'usine.

2.1.4 Cycle de Vie de l'Usine à Logiciels

Le cycle de vie d'une usine à logiciels est composée des cinq étapes successives suivantes [31]. La figure 2.3 les représente de manière graphique.

1. **Définition du domaine** : cette étape consiste en l'étude du segment de marché concerné. Il s'agit en particulier de définir les buts de l'entreprise, le marché concerné, le vocabulaire du domaine, le domaine d'affaires ainsi que le périmètre de l'usine à logiciels. Diverse étapes peuvent être menées pour définir ces préoccupations, à l'instar d'une étude de marché et d'une ingénierie des exigences [81, 38] ;
2. **Analyse du domaine** : cette étape vise à formaliser les exigences en les exprimant sous forme de modèles, en particulier un métamodèle et un diagramme de caractéristiques. La section 2.2 traite en détail de ces modèles ;
3. **Conception du domaine** : cette étape permet de définir l'architecture globale de l'usine à logiciels, en mettant en lumière les différents artefacts et tactiques à employer pour ce faire. Ces tactiques sont explorées dans la section 2.3. Cette étape permet également de définir des stratégies d'implémentation, en utilisant les concepts définis précédemment, afin de générer un logiciel de la manière la plus optimale possible ;
4. **Implémentation du domaine** : cette étape consiste à définir un plan d'exécution dans l'optique d'ordonner les tactiques définies dans l'étape précédente ;
5. **Test du domaine** : cette étape se concentre sur le test des différents artefacts qui seront assemblés pour former un logiciel demandé. En particulier, il s'agit de découvrir si des certains assemblages provoquent des effets indésirables ou imprévus et d'y remédier [72, 80].

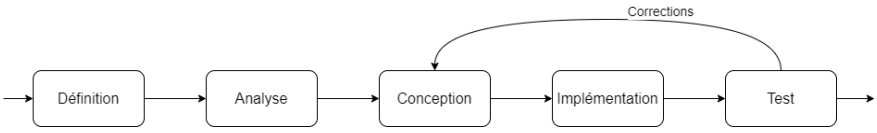


FIGURE 2.3 – Cycle de vie de l'usine à logiciels

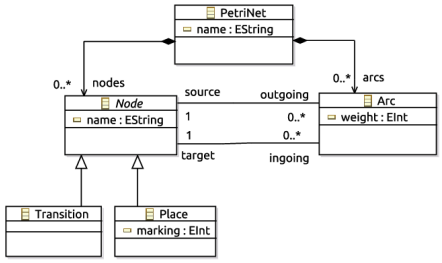
Les étapes une à quatre sont utilisées dans la suite afin de judicieusement baliser ce document, car elles sont implicitement suivies par la structure méthodologique *Bespoke*. Comme énoncé par l'hypothèse (H5), la dernière étape, celle de test, ne sera quant à elle pas abordée.

2.2 Modélisation des Exigences

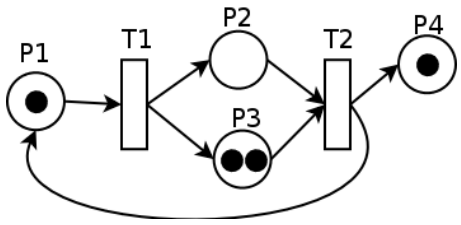
Afin d'exprimer les besoins relatifs au domaine d'application, différents modèles sont employés. De nombreux formalismes distincts ont ainsi vu le jour [88, 67]. Chacun d'eux permettent de mettre en lumière des considérations différentes. Cependant, seuls les formalismes prescrits par la structure méthodologique *Bespoke* [32] sont présentés ici.

2.2.1 Métamodèle

Un métamodèle est un modèle de plus haut niveau permettant d'exprimer des modèles. Dans le cadre du paradigme des lignes de produits logiciels, un métamodèle est la syntaxe abstraite d'un langage de modélisation d'un domaine spécifique - *DSML*. Ce DSML permet de formaliser les exigences d'un domaine d'application [52, 31, 32]. Des langages de modélisation peuvent être employés afin de représenter ce métamodèle. On distingue deux formalismes proéminents : *UML* [110, 88] et *Entité-Association* [31]. Tous deux peuvent être employés pour exprimer un DSML particulier. La figure 2.4a montre le métamodèle d'un réseau de Pétri en UML. Cependant, lorsqu'une personne externe non-initiée à ces formalismes doit exprimer ses besoins, une syntaxe concrète de ce DSML est préférée. Cette syntaxe concrète est le résultat de la transformation du DSML vers un métamodèle dit *d'affichage*. Un DSML peut ainsi avoir de multiples syntaxes concrètes [31, 88]. La figure 2.4b donne un exemple de syntaxe concrète pour un réseau de Petri.



(a) Métamodèle UML [35]

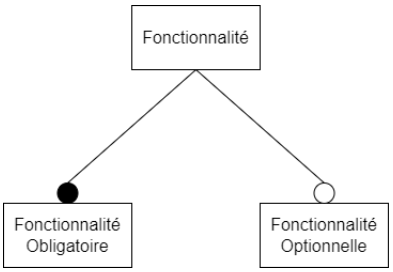


(b) Syntaxe concrète [73]

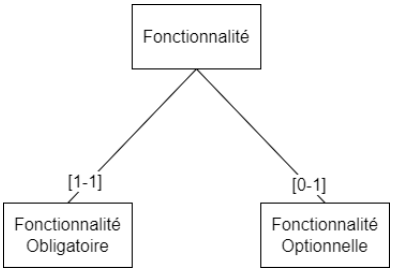
FIGURE 2.4 – Métamodèle, figure 2.4a et exemple de syntaxe concrète, figure 2.4b d'un réseau de Pétri

2.2.2 Diagramme de Caractéristiques

Un diagramme de caractéristiques est un modèle permettant d'exprimer la variabilité du domaine d'application sous forme d'arbre. Il met en évidence les points communs, les points de variation et les variantes associées présentés par le métamodèle et permet de diriger la création des artefacts de l'usine à logiciels [36]. En particulier, un diagramme de caractéristiques se concentre sur l'explicitation des dépendances entre les caractéristiques formulées [60]. Le formalisme *FODA* est le plus répandu. Celui-ci a la particularité de présenter les caractéristiques de manière graphique [54]. Il existe néanmoins une variante qui remplace les notations graphiques par des cardinalités qui permet d'exprimer les mêmes considérations [25, 32]. La figure 2.5 compare les deux notations sur un même exemple. En outre, dans une optique d'automatisation, le diagramme de caractéristiques peut s'exprimer sous une forme textuelle, comme présenté par le langage *FeatAll* [31, 32]. Enfin, un diagramme de caractéristiques peut être utilisé comme pierre angulaire dans la production automatique d'une interface graphique utilisateur, permettant à un client de configurer un produit de manière interactive [89].



(a) Formalisme *FODA*



(b) Formalisme *Cardinalités*

FIGURE 2.5 – Comparaison des notations *FODA*, figure 2.5a et *Cardinalités*, figure 2.5b

2.2.3 Diagramme de Séquence

Un diagramme de séquence est un modèle visant à montrer la séquence temporelle d'interactions entre différents composants d'un système. En particulier, celui-ci permet de modéliser au moyen de constructions qui lui sont propres les messages échangés entre ces composants et ainsi de représenter des cas d'utilisation d'un système. Ce modèle peut être employé pour formaliser et approfondir les fonctionnalités présentées par

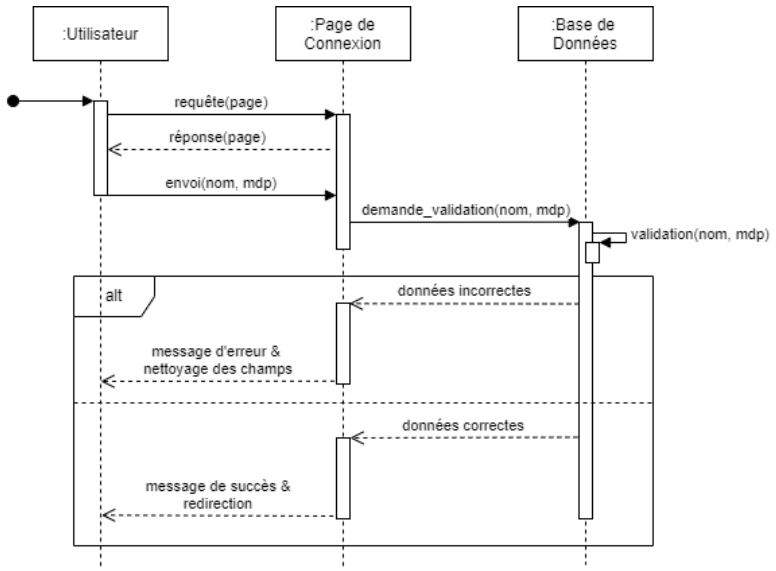


FIGURE 2.6 – Diagramme de séquence pour un système de connexion

le domaine d'application [69] et modélisées par le métamodèle. Au delà de la modélisation de l'interaction, un diagramme de séquence peut également permettre de représenter une ligne de produits logiciels sous un autre formalisme qu'un métamodèle [110]. La figure 2.6 représente un diagramme de séquence qui montre les messages envoyés entre différents composants d'un système dans le cadre d'une demande de connexion.

2.3 Tactiques d'Implémentation

Dans l'optique de concevoir les fondations de l'usine à logiciels, il est impératif de prendre connaissance des techniques et outils permettant de faciliter l'assemblage de l'architecture logicielle recherchée. Cet outillage particulier est appelé *ensemble de tactiques*. Ces tactiques vont permettre de modifier l'ADN commun des différents logiciels que l'usine est capable de produire en venant altérer les artefacts correspondants. Il en existe une multitude et les plus proéminentes sont détaillées dans la suite. Chacune de ces tactiques présente des temps de sélection et d'activation, définis dans la section 2.1.2, qui lui sont propres. Ceux-ci sont documentés dans l'ouvrage syllabus [31]. Il est également important de mentionner que certaines d'entre elles ne peuvent pas être mises en place dans certains langages de programmation. En effet, certaines tactiques nécessitent des caractéristiques intrinsèques qu'ils ne présentent pas.

2.3.1 Architecture Générique

Présenter une architecture générique permet d'offrir des outils capables de répondre à des besoins divers. Ces outils sont plus ou moins dirigistes dans leur manipulation et peuvent pour la plupart être modifiés afin de répondre aux besoins émis par le développeur.

Architecture Hypergénérique

Une architecture hypergénérique est un système qui présente toutes les possibilités liées à celui-ci. Ainsi, lorsqu'une certaine configuration est demandée, les branches inutilisées de l'architecture sont élaguées pour ne laisser que les fonctionnalités demandées. Cependant, cette tactique gère difficilement l'explosion combinatoire des fonctionnalités et le fait que certaines d'entre elles peuvent être contradictoires [31]. Le progiciel de gestion intégré *SAP* [83] est l'illustration parfaite de cette tactique.

Framework

Une structure méthodologique, ou *framework*, est un système qui met en place un squelette extensible et réutilisable. Il embarque en son sein un moteur d'exécution capable d'interpréter les changements apportés à son squelette [33]. Celui-ci peut être modifié de différentes manières [31] :

— **Extension par héritage** : la structure méthodologique met à disposition des composants que le développeur doit étendre lui-même. Les structures méthodologiques développées pour faciliter la conception de sites Internet en sont un exemple ;

— **Extension par *plugins*** : la structure méthodologique permet l'ajout de fonctionnalités externes grâce au support de *plugins*. Ceux-ci doivent respecter les hypothèses posées par la structure, sous peine de ne pas être compatibles. À titre d'exemple, les moteurs de recherche, tels Google Chrome, permettent l'extension de leurs fonctionnalités grâce à cette technique ;

— **Extension par services** : la structure méthodologique permet de créer et d'agréger des services, des composants fonctionnels réutilisables, au sein de services composites. Le développeur manipulant la structure est par conséquent capable de créer des fonctionnalités complexes [62].

Traits

Des traits sont des unités de code dont la préoccupation première est la réutilisabilité [84]. En particulier, un trait adresse les problèmes inhérents à l'héritage et la composition des classes du paradigme de programmation orienté objets [39] en définissant un ensemble de méthodes répondant à un besoin fonctionnel [30, 78]. Plusieurs traits peuvent être combinés ensembles pour offrir un jeu de fonctionnalités étendu [84]. La plupart des langages de programmation multi-paradigmes supportent l'utilisation des traits.

2.3.2 Métaprogrammation

La métaprogrammation est une technique de programmation qui traite des programmes particuliers comme des données de premier ordre. Ainsi, un programme écrit est considéré comme une donnée manipulable par un autre programme. Par conséquent, il est possible de modifier et générer un programme de manière automatique et dirigée [61]. Il en existe différents modèles, présentant chacun une approche de la métaprogrammation différente [61] : les systèmes à base de macros, les systèmes réflexifs, les protocoles métaobjets, la programmation orientée aspects [56], la programmation générative, la programmation multi-étages et la métaprogrammation à l'aide des traits [78]. La métaprogrammation est un style de programmation particulier qui requiert certaines caractéristiques intrinsèques inhabituelles propres à chaque catégorie évoquée ci-avant.

Programmation Orientée Aspects

La programmation orientée aspects est un paradigme de métaprogrammation qui vise à factoriser les préoccupations transversales au système [61, 56]. Un aspect est un artefact logiciel contenant du code source chargé d'étendre les fonctionnalités du système. Ce code additionnel est appelé *greffon* et est tissé par le mécanisme de tissage à l'endroit indiqué par un *point de jonction*. Plusieurs points de jonction peuvent être regroupés pour former une *coupe*, qui désigne ainsi un ensemble d'endroits d'intérêt où venir appliquer un greffon [56, 39]. Il existe par ailleurs plusieurs types de greffons. Les plus communs sont définis ci-après [31] :

— **Avant** : le greffon est appliqué avant le point de jonction ;

— **Après** : le greffon est appliqué après le point de jonction ;

— **Autour** : le greffon, ainsi composé de deux préoccupations, est appliqué autour du point de jonction. La première préoccupation est appliquée avant, tandis que la seconde l'est après. Le greffon peut en outre choisir de supprimer ou non le code balisé par le point de jonction. Un tel mécanisme peut être employé pour vérifier les pré- et postconditions relatives au point de jonction.

Le tissage d'aspects peut être réalisé à deux moments distincts du cycle de vie d'un produit. Le premier est la compilation. Un aspect peut ainsi être tissé lorsque le système est compilé. À l'inverse, un aspect peut également être tissé à l'exécution du système, permettant ainsi d'activer et de désactiver des fonctionnalités *à chaud* [56]. Cependant, peu de langages prennent en charge ce paradigme de programmation et les résultats obtenus lors de son utilisation sur des cas d'application démontrent à l'inverse une certaine inaptitude à être employé en pratique [55].

2.3.3 Programmation Conditionnelle

La programmation conditionnelle désigne la possibilité que possède un programme de choisir entre des flux de compilation ou d'exécution différents. Ce choix s'effectue en fonction de la valeur des variables impliquées dans l'évaluation de conditions booléennes. Ainsi, en fonction du résultat d'une condition particulière, un flux spécifique sera sélectionné.

Instruction Conditionnelle

Une instruction conditionnelle dans un code source permet d'orienter le flux d'exécution d'un programme. La majorité des langages de programmation présente naturellement ce type d'instruction. Celle-ci est la plupart du temps représentée par une syntaxe $If/Then/Else$. La valeur des variables, qui évoluent au cours du cycle de vie du programme, participant à une condition booléenne permet au programme d'emprunter un flux d'exécution particulier [23].

Compilation Conditionnelle

À l'inverse des instructions conditionnelles qui opèrent à l'exécution, la compilation conditionnelle permet d'orienter la compilation d'un programme. Cette orientation s'effectue à l'aide d'instructions *préprocesseures*. Celles-ci sont plus limitées dans leur expressivité que les instructions conditionnelles classiques et peu de langages les supportent. Cependant, elles permettent d'inclure dans le code source l'entièreté des choix à opérer. Ceux-ci sont alors choisis grâce à des jetons spéciaux introduits au moment de la compilation et les blocs d'instructions qui ne sont pas sélectionnés sont retirés du code binaire compilé, le rendant ainsi plus léger [39, 93].

2.3.4 Altération de Fichiers

Il est également possible de gérer la variabilité au niveau des fichiers du code source eux-mêmes. Un programme particulier peut ainsi altérer le contenu d'un fichier dont la version modifiée sera par la suite utilisée dans une configuration. Cette altération peut s'effectuer sur le code source contenu dans un fichier ou sur le fichier lui-même.

Patching

Le mécanisme de *patching* consiste à altérer le contenu d'un fichier en suivant des balises bien précises posées au préalable. Ce mécanisme est extrêmement fragile dans la mesure où il dépend fortement des balises évoquées ci-avant. Le couplage est par conséquent très fort et une modification malheureuse peut compromettre la bonne exécution du mécanisme [31]. Il en existe trois variantes :

— **Patching absolu** : ce mécanisme consiste en la génération d'un fichier qui reprend les différences entre deux répertoires, respectivement un ancien et un nouveau. Il s'agit d'être capable de faire évoluer aisément un répertoire identique à l'ancien, résidant par exemple à un autre endroit physique, vers une nouvelle instance du nouveau répertoire [62]. Un fichier de patching peut s'avérer extrêmement utile pour ajouter une fonctionnalité exclusive à un répertoire de base [31] ;

— **Patching relatif** : ce mécanisme permet d'ajouter des éléments dans un fichier à des endroits précis indiqués par des balises. Ce principe est similaire à la greffe de code de la programmation orientée aspects. Les patchings relatifs peuvent en outre s'accumuler et être appliqués séquentiellement [31] ;

— **Patching par soustraction** : ce mécanisme est l'inverse du patching relatif. En effet, le patching par soustraction vise à soustraire des éléments d'un fichier particulier, en faisant l'hypothèse que ladite soustraction ne produit pas d'effets de bord non désirés sur les autres fonctionnalités [31].

Substitution de Composants

La substitution de composants s'intéresse à l'échange de fichiers présentant une interface identique. Si d'aventure cette condition est respectée, alors deux unités de code sont interchangeables. Dès lors, un programme dépendant d'une telle ressource interchangeable peut présenter des comportements différents en fonction de l'unité de code actuellement chargée [62]. La substitution de composants permet ainsi de lier une fonctionnalité de façon tardive dans le cycle de vie d'un produit, en fonction des limites technologiques des artefacts utilisés [31].

2.3.5 Programmation Orientée Objets

La programmation orientée objets est un paradigme visant à encapsuler les préoccupations relatives à un concept dans un objet. Un objet est une instance d'un plan, appelé *classe*, qui définit les fonctionnalités du concept [77]. Ce paradigme est fortement répandu dans les pratiques de développement car il présente des avantages non négligeables qui facilitent l'écriture de programmes. Certains de ces avantages sont décrits dans la suite.

Patrons de Conception

Un patron de conception est un mécanisme pensé pour optimiser et réduire la complexité d'un système. En particulier, il s'agit de pratiques épurées et éprouvées qui visent à répondre à une problématique spécifique en tirant parti des avantages intrinsèques du paradigme orienté objets. Ils constituent une base réutilisable pour développer des systèmes [40, 39]. Il en existe une multitude et une documentation extensive est disponible à propos de chacun d'eux [41]. Cependant, certains conviennent mieux que d'autres afin d'exprimer la variabilité d'un système logiciel [31, 41] :

- **Héritage** : une classe peut être étendue par une autre, appelée sous-classe. Celle-ci hérite des spécificités de sa classe parent et peut en outre définir de nouvelles fonctionnalités ou modifier celles existantes. La sous-classe doit également respecter le principe de substitution de Liskov [63], qui indique que toute classe peut être remplacée par ses sous-classes sans causer de problèmes au système ;
- **Interpréteur** : ce patron vise à interpréter un langage ad hoc en analysant sa grammaire, dans l'optique de le transformer en une formulation orientée objets. Un tel langage peut être défini à l'aide d'un DSL, abordé dans la section 2.3.6 ;
- **Template** : ce patron consiste à définir un ensemble de grandes étapes et de méthodes d'aide abstraites. Cet ensemble peut alors être concrétisé ou redéfini par une sous-classe. Ce patron est une variante du patron de conception *héritage* ;
- **Factory** : ce patron permet d'adresser le problème de création d'objets sans explicitement spécifier quelle classe sera utilisée à cette fin. Par conséquent, les classes employées par la *factory* doivent également respecter le principe de substitution de Liskov [63] ;
- **Stratégie** : ce patron permet de sélectionner un algorithme particulier au cours de l'exécution du système. Ce mécanisme permet de rendre le code à l'origine de l'appel plus flexible et modulaire, car non lié à un algorithme particulier ;
- **Intercepteur** : ce patron consiste à introduire un composant tierce entre plusieurs composants communiquant ensemble. Ce nouveau composant intercepte les appels et messages échangés de manière automatique et transparente et est capable de réaliser des actions spécifiques en fonction des messages interceptés [85]. Ce patron de conception est typiquement utilisé pour implémenter un *middleware* [68] ;
- **Injection de dépendances** : ce patron permet l'injection d'artefacts dans des fonctions dont celles-ci dépendent. En retirant à ces fonctions la responsabilité d'instancier ces artefacts, cela permet une séparation des préoccupations et une diminution du couplage au niveau du code source [75].

Polymorphisme

Le polymorphisme désigne la pratique selon laquelle une même interface peut être utilisée par des types différents. Une invocation de l'interface par un type particulier résulte en l'exécution d'un fragment de code spécifique. Il existe plusieurs formes de polymorphisme [74, 39] :

- **Ad hoc** : plusieurs fonctions du même nom présentent des paramètres de différents types. Lorsque le nom de ces fonctions est invoqué, la fonction qui sera exécutée dépend des types des paramètres fournis par le code à l'origine de l'appel. Cette opération est également appelée *overloading* ;
- **Paramétrique** : une même fonction définit un type générique qui n'influence pas son exécution. Cela permet d'uniformiser le comportement d'une fonction en restant indépendant des types des paramètres associés ;
- **Par sous-typage** : une fonction tire parti du mécanisme d'héritage de classes. Elle autorise comme type des paramètres une classe qui peut être étendue par des sous-classes. Ce mécanisme doit par conséquent respecter le principe de substitution de Liskov [63].

2.3.6 Langages Spécifiques à un Domaine

Un langage spécifique à un domaine, aussi abrégé DSL, permet de définir une grammaire capable d'adresser les préoccupations liées à un domaine particulier. Ce langage doit être facilement manipulable par les personnes concernées par le domaine, en proposant des notations adaptées [105, 45]. Ainsi, un DSML, défini en section 2.2.1, est un cas particulier de DSL. En outre, un DSL est accompagné d'un ensemble d'outils pour le mettre en œuvre, à l'instar d'interpréteurs et de compilateurs [31]. Il existe deux manières de concevoir un DSL [19] :

- **DSL interne** : certains langages de programmation permettent de définir un DSL en leur sein. Celui-ci est alors mis en œuvre en utilisant adroitement la syntaxe du langage. Certains langages présentent une syntaxe suffisamment flexible pour permettre l'expression d'un DSL comme *Groovy* [28], tandis que

d'autres mettent en place des mécanismes puissants pour les définir aisément, à l'instar des macros de *Rust* [29] ;

— **DSL externe** : au moyen d'un analyseur lexical tel *ANTLR* [4] ou *Lark* [106], il est possible de définir une grammaire indépendante. Celle-ci sera analysée et chargée dans un langage de programmation particulier, restreint par l'analyseur lexical.

Un même DSL peut également présenter différentes implémentations. Ce mécanisme est appelé *surcharge de DSL*. Ainsi, cela permet à un DSL d'être traduit dans plusieurs langages différents tout en exhibant un comportement unique [31].

2.4 Développement Mobile Multiplateforme

De nombreuses différences distinguent les multiples plateformes mobiles existantes. Celles-ci résident tant au niveau logiciel, avec des langages de programmation spécifiques, que matériel, où certains constructeurs ont la mainmise sur le développement des composants présents [42]. Il existe des solutions dites *cross-platforms*, qui permettent de faire abstraction de ces particularités, à l'instar du langage de programmation *Kotlin* [24], ainsi que d'un paradigme de développement hybride ou de développement web [27]. Ces solutions sont possibles au prix de multiples concessions, telle une perte significative de performance [27]. Cependant, car le cas d'application *BetterSoft* [31], repris en section 3.1, s'intéresse aux systèmes d'exploitation *Android* et *iOS* et plus précisément aux langages de programmation *Java* [51] et *Swift* [94], les solutions *cross-platforms* ne sont pas explorées dans ce cadre-ci. Les tables 2.1 et 2.2 présentent les principales différences entre *Android* et *iOS* et indiquent quelles tactiques sont disponibles pour chacun des langages de programmation associés. Les IDE *Android Studio* et *XCode* permettent de développer complètement des applications mobiles simples vers *Android* et *iOS*, respectivement, et ce sans rendre obligatoire l'utilisation de frameworks tiers. Ainsi, dans le cadre de ce mémoire, ces IDE seront utilisés pour concevoir les différentes configurations. Les tactiques qu'il est possible d'utiliser de concert avec ceux-ci sont identiques à celles des langages qu'ils supportent, comme présenté par la table 2.2.

Caractéristique	***Android***	***iOS***
Kit de développement	Android SDK	iOS SDK
IDE	*Android Studio* (, *Eclipse*)	*XCode* (, *AppCode*)
Architecture logicielle	MVVM [3] (& MVC [57])	
Fichiers de contrôle	« .java » d'activité	« .swift » de *ViewController*
Fichiers de métadonnées	« AndroidManifest.xml »	« .plist »
Composants graphiques	« .xml »	« .xib » ou « .storyboard »
Composants statiques	*drawables*	*image sets*
Gestion des paramètres d'une application	« .xml » & *shared preferences* [87]	« .plist » et *user defaults* [102]

TABLE 2.1 – Tableau de comparaison de l'architecture logicielle entre *Android* et *iOS* dans le cadre du développement mobile

Tactique	***Java* (*Android*)**	***Swift* (*iOS*)**
Architecture générique	Toutes les tactiques mentionnées	
Métaprogrammation	Macros, réflexive (introspection structurelle), orientée aspects, générative (génériques et transformation de l'AST) [61], à l'aide de traits [78]	Non supportée
Programmation conditionnelle	Instruction conditionnelle native, compilation conditionnelle avec *FreeMaker* [53]	Instruction conditionnelle native, compilation conditionnelle simulable [109]

Altération de fichiers	Compatible	
Programmation orientée objets	Toutes les tactiques mentionnées	
DSL	Externe [4], surcharge de DSL	Interne [108], externe [4], surcharge de DSL

TABLE 2.2 – Tableau de comparaison des tactiques d'implémentation entre *Java* et *Swift*

La table 2.1 met en évidence des similarités flagrantes entre les outils disponibles pour les plateformes *Android* et *iOS*. Elles mettent toutes deux en place les modèles *View Controller* [57] et *View ViewModel* [3] qui divisent interface graphique et logique métier, possèdent toutes deux des fichiers de métadonnées exploitables et présentent toutes deux un ensemble exploitable presque identique de tactiques d'implémentation. Ainsi, il serait judicieux d'en tirer parti et de chercher à obtenir un ensemble restreint et homogène de tactiques, tout en gardant à l'esprit que celles-ci devront tout de même être adaptées en fonction de la plateforme. En effet, malgré ces similitudes, de nombreuses différences primordiales persistent et requièrent des compétences précises, à l'instar des formats de données ou des spécificités physiques de chacun des appareils [98]. Les différentes caractéristiques intrinsèques des plateformes *Android* et *iOS* et des langages de programmation associés sont explorées en détail dans la section 5.2 du chapitre 5.

2.5 Métamodèle de la Méthodologie *Bespoke*

La structure méthodologique *Bespoke* propose un répertoire réunissant les informations relatives à celle-ci sur lequel se base ses différentes étapes. À titre de rappel, ces étapes sont explicitées par la figure 1.1 dans le chapitre 1. Ainsi, le répertoire reprend l'information liée aux modèles, méthodes, tactiques, artefacts et règles méthodologiques [31]. La figure 2.7 présente un métamodèle qui détaille la structure de ce répertoire. Celui-ci est divisé en six panneaux [32] :

— Le **panneau 1** détaille le métamodèle du langage de modélisation de caractéristiques *FeatAll*. Celui-ci est utilisé pour représenter les caractéristiques reprises dans un métamodèle particulier sous forme d'un diagramme de caractéristiques. Ce panneau est lié à la troisième étape du plan *SPL Engineer* ;

— Le **panneau 2** reprend les différentes annotations qu'il est possible d'ajouter au diagramme de caractéristiques. Ce panneau est lié à la quatrième étape du plan *SPL Engineer* ;

— Le **panneau 3** définit les stratégies employées pour implémenter la ligne de produits logiciels guidée par le diagramme de caractéristiques. Celles-ci sont composées de tactiques qui agissent sur un point de variation, un attribut ou une variante. Plusieurs stratégies peuvent être définies pour un même diagramme de caractéristiques. Ce panneau est lié à la cinquième étape du plan *SPL Engineer* ;

— Le **panneau 4** catalogue les artefacts pertinents pour la conception de l'usine à logiciels, employés par les différentes tactiques définies. Ce panneau est lié aux quatrième et cinquième étapes du plan *SPL Engineer* ;

— Le **panneau 5** formalise un ensemble de types de tactique. Ceux-ci représentent des combinaisons de procédures appelées par les tactiques ainsi définies à invoquer sur les artefacts correspondants. Ce panneau est lié à la cinquième étape du plan *SPL Engineer* ;

— Le **panneau 6** présente la manière dont les tactiques sont coordonnées entre elles, dans l'optique d'établir un plan d'exécution de celles-ci. Ce panneau est lié à la sixième et dernière étape du plan *SPL Engineer*.

2.6 Conclusion

Ce chapitre a permis de mettre en lumière les principes qui régissent les lignes de produits logiciels. En particulier, les points de variation sont définis comme des concepts permettant l'expression des différences entre les exigences émises par les clients. L'usine à logiciels est formalisée comme la mise en œuvre concrète d'une ligne de produits logiciels. En outre, dans l'optique d'exprimer formellement les exigences, différentes modélisations sont présentées. Un ensemble de tactiques, classées en catégories distinctes, est également proposé afin d'aider à l'expression de la variabilité au sein de l'usine à logiciels. Des tableaux comparatifs des spécificités des langages de programmation *Swift* et *Java* et des plateformes *Android* et *iOS* sont également dressés dans l'optique de circonscrire les possibilités offertes par chacun d'eux. Les concepts ainsi définis seront mis en pratique tout au long de ce document. Enfin, le formalisme des différents outils employés pour mettre en œuvre la structure méthodologique *Bespoke* est présenté sous forme de métamodèle décrivant un répertoire des connaissances à exploiter par les différentes étapes de la méthodologie.

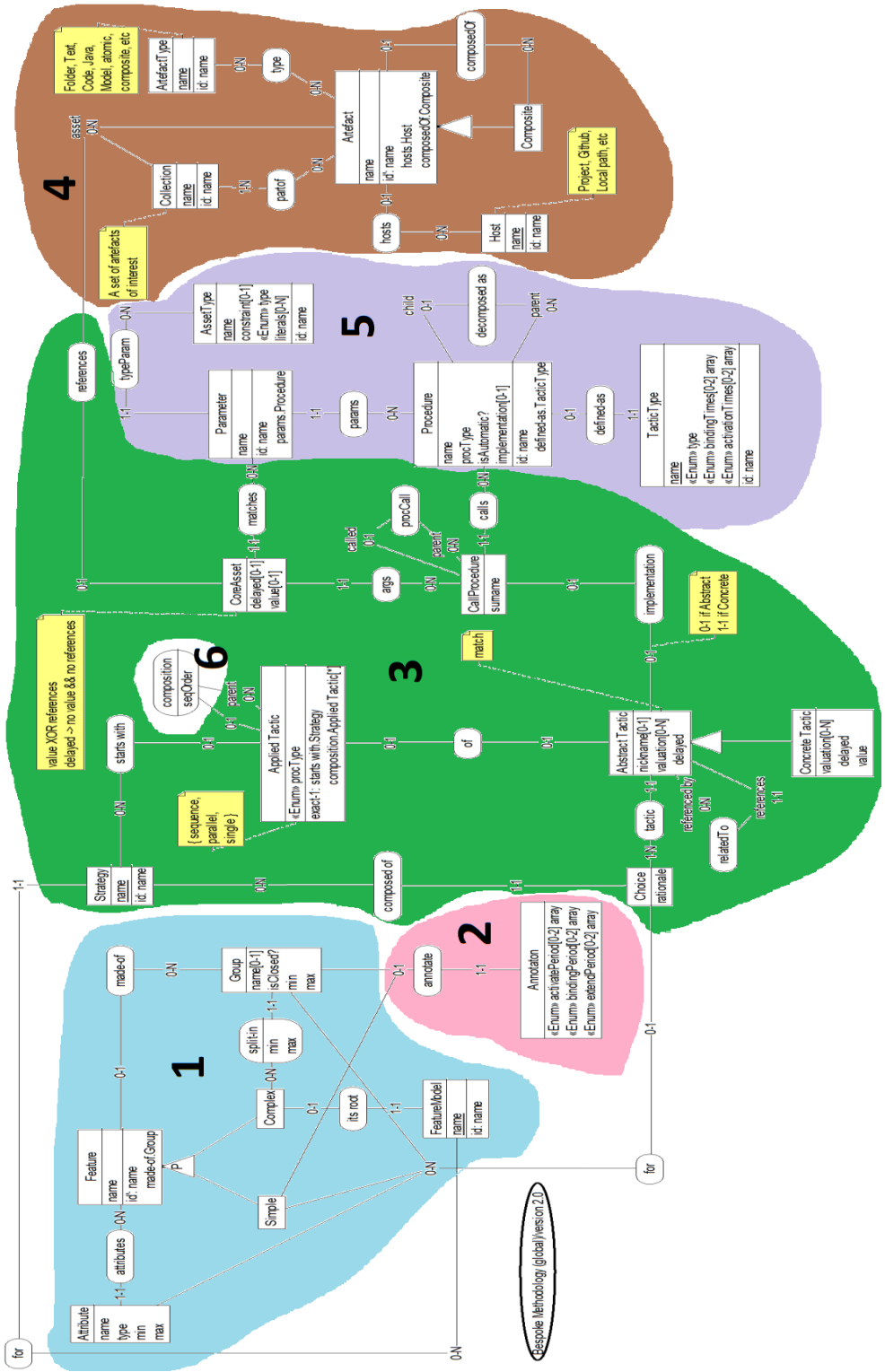


FIGURE 2.7 – Métamodèle de la méthodologie *Bespoke*

Chapitre 3

Définition du Domaine

Ce chapitre vise à introduire le cas d'utilisation qui accompagnera la validation de la méthodologie *Bespoke* tout au long de ce mémoire. À cette fin, une étude succincte de son domaine d'application est effectuée selon la structure établie par l'ouvrage accompagnant la méthodologie *Bespoke* [31], conformément à l'hypothèse (H1). Une visualisation complémentaire de l'importance des caractéristiques est également présentée. Il est important de noter que la définition du domaine ne prétend pas être exhaustive ni approfondie, car il s'agirait alors d'un exercice d'analyse d'entreprise à réaliser d'un point de vue économique. À ce titre, la définition du marché ne sera pas abordée ici non plus.

3.1 Cas d'Application *BetterSoft*

L'entreprise fictive *BetterSoft* [31] est une entreprise spécialisée dans la confection de logiciels sur mesure. En particulier, celle-ci se spécialise dans les applications touristiques. Les logiciels produits par l'entreprise sont traités individuellement. Ainsi, *BetterSoft* a créé et maintient entre autres une application mobile de visite de musées et une application mobile de gestion de parcs de loisirs, chacune pour des clients différents. Des plateformes de gestion liées aux applications, utilisées par les gestionnaires des sites touristiques, sont également produites et maintenues. Les clients sont pleinement satisfaits du service proposé et cela en attire indéniablement de nouveaux. L'entreprise est alors approchée par un client souhaitant commander une application capable de gérer à la fois des visites de musées et ses parcs de loisirs. Suite à cela, l'entreprise réalise qu'il est possible de tirer parti des similitudes entre ses différentes applications. Elle décide donc d'adopter une approche plus générique et d'implémenter une ligne de produits logiciels, en utilisant la structure méthodologique *Bespoke*.

3.2 Vocabulaire

Certains termes employés dans la suite peuvent prêter à confusion. À ce titre, les définitions non ambiguës de ceux-ci sont données ci-après :

Site Un espace géographique, non nécessairement d'un seul tenant, offrant des activités de loisir et/ou culturelles à des visiteurs. Un site peut couvrir une région arbitrairement grande que l'on peut traverser intégralement en une journée avec un moyen de transport courant - vélo, voiture, bus, train, ... Il ne dépend que d'une seule législation nationale [31]. À ne pas confondre avec un site Internet ;

Site touristique Un site touristique est un lieu de passage, mais non de séjour, car il est sans fonction d'hébergement, ou à capacité d'hébergement sans commune mesure avec sa fréquentation. Il s'agit d'un type de lieu touristique créé par invention, c'est-à-dire par le regard et l'usage des touristes [79] ;

Client La personne ou entité qui fait l'acquisition d'un ou de plusieurs systèmes conformes à une configuration [31] ;

Utilisateur Une personne qui utilise l'un des systèmes sur un site. Synonyme : visiteur ;

Déficient visuel Un utilisateur du système qui présente au moins un des handicaps visuels suivants : presbytie [76], hypermétropie [48] ou daltonisme [26] sous ces différentes formes [59]. Les autres handicaps visuels, à l'instar de la myopie ou de l'astigmatisme, ne sont pas pris en compte ;

Service Une activité de services se caractérise essentiellement par la mise à disposition d'une prestation technique ou intellectuelle. À la différence d'une activité industrielle, elle ne peut pas être décrite par les seules caractéristiques d'un bien tangible acquis par le client. Son produit final est immatériel, il n'est ni stockable, ni transportable [50]. Exemple : acheter des places de théâtre.

3.3 Étude Succincte

En utilisant une approche orientée ligne de produits logiciels, l'entreprise *BetterSoft* poursuit plusieurs objectifs. Dans un premier temps, il s'agit pour l'entreprise de pouvoir aisément traiter de multiples commandes de produits en parallèle. L'approche guidée par les lignes de produits permettrait de factoriser les préoccupations au niveau de celle-ci. Ainsi, l'entreprise pourrait se concentrer sur les exigences des clients et libérer des ressources humaines et techniques, monopolisées à l'heure actuelle par les différentes applications en production. Ce serait également une pratique intéressante pour diminuer les coûts de production. Dans un second temps, cela permettrait à l'entreprise d'accueillir de nouveaux clients, car elle disposerait alors d'une solution capable de répondre à leurs besoins de façon rapide [70]. Enfin, il s'agit d'initier les informaticiens de l'entreprise à une approche plus qualitative, dirigée et structurée du point de vue de la production logicielle.

L'adoption de ce paradigme de production logicielle s'inscrit donc dans un souci d'évolution entrepreneuriale. Ce faisant, l'entreprise espère atteindre de nouveaux clients engagés dans la gestion de sites d'intérêt touristique, à l'instar de monuments historiques ou d'événements culturels. L'entreprise se limite au cadre européen pour l'instant mais espère notamment s'implanter en Asie. Les applications mobiles développées seront distribués sous les plateformes *Android* et *iOS* et les logiciels de gestion associés seront soit hébergés sur les serveurs de l'entreprise, soit sur l'infrastructure du client concerné. En outre, l'entreprise dispose de différents atouts pour confectionner l'usine à logiciels. Celle-ci dispose en effet d'informaticiens compétents et d'applications en production, écrites en *Java* et *Swift*, dont il est possible de tirer parti. Enfin, l'équipe informatique adhère au nouveau principe de conception et est prête à s'y investir au maximum.

3.4 Définition des Caractéristiques

Dans le but d'établir le métamodèle du domaine d'application détaillé dans le chapitre 4, il est nécessaire d'expliciter les caractéristiques attendues de la ligne de produits logiciels. L'ouvrage syllabus [31] définit les caractéristiques suivantes :

OS le support des smartphones ou tablettes *Android* et *iOS* ;

CTRL une plateforme de gestion WEB ;

SPEC La prise en compte des personnes à handicaps visuels ;

PAY Le paiement en ligne de certains services ;

BOOK La possibilité de réserver certains services ;

NAV L'aide à la navigation ;

INT Une aide à l'interprétation qui est contextuelle et moderne ;

PUB La possibilité de faire financer l'application par de la publicité ;

SNET L'intégration avec les réseaux sociaux.

D'autres caractéristiques sont intéressantes dans le cadre du domaine d'application. Ainsi, les caractéristiques suivantes complètent l'ensemble défini ci-avant :

MEET La possibilité de trouver des gens avec qui passer la journée. Les personnes le souhaitant peuvent se rendre à certains endroits du site afin de former un groupe avec d'autre visiteurs. Cette caractéristique nécessite la présence de la caractéristique **NAV** pour le rassemblement et la navigation ;

FUN L'organisation d'événements sur mesure. Il s'agit de permettre au client de choisir entre diverses activités, à l'instar d'un quiz, et de déployer l'événement nouvellement créé vers les appareils disposant du logiciel. Cette caractéristique nécessite la présence de la caractéristique **CTRL** pour créer les événements.

D'autres caractéristiques pourraient être ajoutées aux listes précédentes, mais n'offriraient vraisemblablement pas de véritable avantage concurrentiel. Les caractéristiques ainsi mentionnées composent donc le domaine d'application et seront prises en charge et combinées de manière optimale par l'usine à logiciels. Les caractéristiques **SPEC**, **BOOK**, **INT**, **SNET**, **MEET** et **FUN** sont les caractéristiques stratégiques capables de donner un avantage concurrentiel à l'entreprise *BetterSoft*.

3.5 Visualisation des Caractéristiques

Dans l'optique d'exprimer l'importance des différentes caractéristiques mentionnées ci-avant au sein du domaine d'application, l'ouvrage syllabus [31] propose l'utilisation de divers tableaux. Le tableau 3.1 en est une illustration. Celui-ci s'intéresse à la valeur ajoutée et au montant que les clients sont prêts à débourser afin d'acquérir ces caractéristiques. Cependant, bien qu'informatif, il ne permet pas de visualiser aisément les différences entre celles-ci. Il ne permet pas non plus de faire ressortir leur importance. Or, présenter une visualisation adéquate des données est tout aussi important que les données elles-mêmes, car « une image vaut mieux que mille mots » [101].

Caractéristique	Clients	Avantage concurrentiel	Investissement pécuniaire
OS	Tous	4	1
CTRL	Tous	4	5
SPEC	Majorité	3	4
PAY	Majorité	5	2
BOOK	Minorité	3	2
NAV	Minorité	5	4
INT	Tous	5	4
PUB	Beaucoup	4	4
SNET	Certains	3	3
MEET	Certains	2	2
FUN	Majorité	5	5

TABLE 3.1 – Tableau de visualisation de l'importance des caractéristiques définies

Ainsi, une modélisation complémentaire à l'aide d'un graphique à bulles est proposée. L'utilisation combinée d'une table et d'un graphe devrait permettre de mieux visualiser les différences entre les caractéristiques [104]. Dès lors, le graphique à bulles de la figure 3.1 classe également les différentes caractéristiques en fonction de l'avantage concurrentiel qu'elles apportent, du montant que les clients sont prêts à investir pour les acquérir et de la proportion de clients potentiellement intéressés par celles-ci. Ces informations se rapportent respectivement à l'axe des ordonnées, à celui des abscisses et enfin au diamètre des bulles. Il est toutefois important de noter que les différentes valeurs des axes ne sont pas liées entre elles par une quelconque relation. Il s'agit simplement d'utiliser ce formalisme afin de présenter une visualisation complémentaire. Par conséquent, les valeurs des différents axes sont similaires aux valeurs retrouvées dans la table 3.1 correspondante.

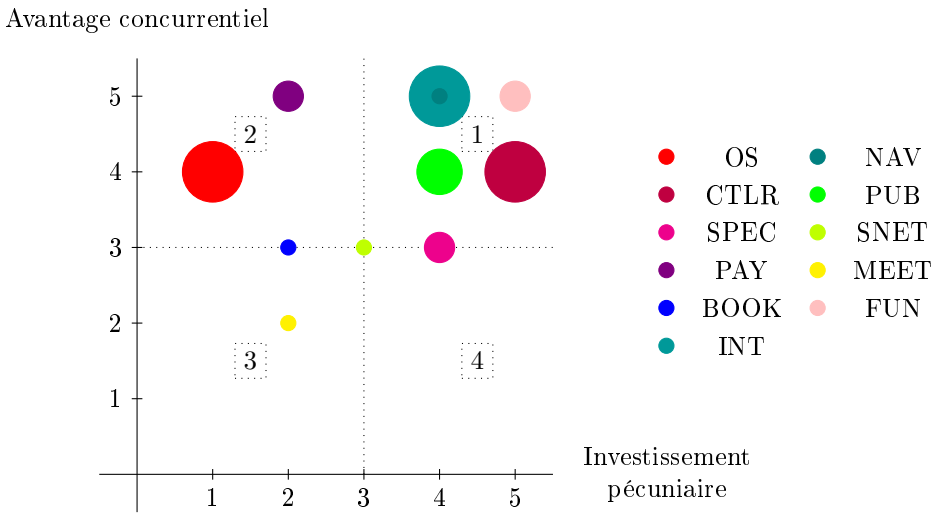


FIGURE 3.1 – Graphe de visualisation de l'importance des caractéristiques

Ce graphique à bulles est divisé en quatre quadrants distincts. Ceux-ci sont représentés par leur numéro, inscrit dans un rectangle, sur le graphe.

— Le **quadrant *1*** contient les caractéristiques les plus importantes. Celles-ci présentent un avantage concurrentiel certain et les clients sont potentiellement prêts à dépenser un montant élevé afin de les acquérir ;

— Le **quadrant *2*** contient quant à lui les caractéristiques que les clients s'attendent à trouver dans toute application rudimentaire ;

— Le **quadrant *3*** regroupe les caractéristiques plus en retrait, plus secondaires. Elles n'apportent en elles-mêmes que peu de valeur ajoutée, mais restent néanmoins bienvenues ;

— Le **quadrant *4*** est conceptuellement difficile à interpréter. Il contiendrait des caractéristiques sans réel avantage concurrentiel du point de vue des clients, mais ceux-ci seraient tout de même enclins à dépenser un certain montant pour les acquérir. Il pourrait, par exemple, s'agir de caractéristiques indispensables pour eux afin de rester concurrentiel. Celles-ci viseraient alors à garder fidèle leur clientèle.

Somme toute, en observant l'emplacement d'une caractéristique particulière ainsi que le diamètre de sa bulle, il est possible de rapidement mettre en évidence son importance aux yeux des différents clients. Cette visualisation peut également servir dans une moindre mesure à la planification de l'ordre d'implémentation des différentes caractéristiques, facultatives au niveau des configurations, dans l'usine à logiciels. Si d'aventure l'objectif de *BetterSoft* est de maximiser leurs bénéfices, l'entreprise pourrait classer les quadrants selon l'ordre d'importance suivant : $1 \rightarrow 4 \rightarrow 2 \rightarrow 3$.

Remarque(Rem·I) ► Priorisation des Caractéristiques

La visualisation des caractéristiques permet de mettre en avant celles qui présentent les plus grands avantages stratégiques. Ceci est l'objectif poursuivi par la visualisation complémentaire offerte par le graphique à bulles de la figure 3.1. Elle permet ainsi de prioriser l'implémentation de ces caractéristiques selon un ordonnancement particulier définit par l'entreprise. Si d'aventure la ligne de produits logiciels s'avère trop ambitieuse pour être implémentée d'une seule traite et que la décision est prise de mettre sa version distribuée à jour au fur et à mesure, cette visualisation pourrait offrir une certaine aide quant à la priorisation des caractéristiques.

3.6 Conclusion

Ce chapitre a permis d'introduire le cas d'application *BetterSoft* dont les spécificités guideront la conception de l'usine à logiciels dans la suite du document. Une étude succincte de l'entreprise est réalisée afin de cerner leurs besoins. Les caractéristiques du domaine d'application sont élicitées et une visualisation complémentaire de celles-ci est également proposée. Ce chapitre permet ainsi de formaliser le domaine d'application dont dépendent les chapitres suivants.

Chapitre 4

Analyse du Domaine

Le chapitre 3 a permis de circonscrire le domaine d'application et par conséquent de remplir les conditions nécessaires à l'utilisation de la structure méthodologique *Bespoke*. Ainsi, ce chapitre met en œuvre les quatre premières étapes du plan *SPL Engineer* de la méthodologie, soit de la définition du métamodèle à l'obtention du diagramme de caractéristiques annoté. Somme toute, ce chapitre vise à représenter les caractéristiques énoncées de manière expressive et formelle, dans l'optique de poser les bases de la ligne de produits logiciels. À cette fin, le domaine d'application est représenté sous la forme d'un métamodèle permettant d'exprimer les préoccupations communes et les différences des multiples logiciels que l'entreprise souhaiterait produire. Ce métamodèle est ensuite converti selon une transformation formelle en diagramme de caractéristiques permettant de visualiser les dépendances et particularités de chacune des caractéristiques. Cette transformation fait par ailleurs l'objet de quelques précisions. Afin de rendre la démarche plus concrète, les différentes représentations élicitées ci-avant sont illustrées au moyen d'un scénario de commande d'un logiciel particulier auprès de l'entreprise *BetterSoft*. En outre, il serait également judicieux et fort recommandé de réaliser une ingénierie des exigences afin définir formellement les différentes caractéristiques du domaine d'application. Cependant, cet exercice prenant trop de temps, il ne sera pas réalisé. Ainsi, des libertés créatives seront prises au niveau de l'implémentation de ces caractéristiques, abordée dans les chapitres 5 et 6.

4.1 Modélisation du Domaine

Les deux premières étapes consistent en la définition d'un métamodèle et d'une version annotée de celui-ci, respectivement. Ceux-ci vont permettre d'exprimer les exigences qu'un client émet à propos d'un logiciel particulier à produire. Le métamodèle se doit donc d'être intuitif car il sera implicitement manipulé par un client. À cette fin, une syntaxe concrète graphique est définie pour le manipuler.

4.1.1 Métamodèle

La figure 4.1 représente le métamodèle qui décrit le domaine d'application. La figure A.1 disponible en annexe en présente une version agrandie. On retrouve sur celui-ci l'entièreté des caractéristiques définies dans la section 3.4 du chapitre 3 :

Plateforme Contrôle Le serveur qui héberge la plateforme de contrôle ;

Application L'application demandée par le client ;

Appareil Les appareils physiques, de type GSM ou tablette, accueillant l'application demandée. La relation d'héritage est sémantiquement similaire à une spécialisation *is-a* de type P du formalisme *Entité-Association* ;

Caractéristiques Matérielles Les particularités matérielles des appareils mentionnés ci-avant. Ce concept regroupe les caractéristiques dites *hardware* et le système d'exploitation de l'appareil ;

Persona Un utilisateur de l'application ;

Handicap Visuel Un possible handicap visuel présenté par un utilisateur particulier. Une personne atteinte d'hypermétropie ou de presbytie peut agrandir la taille de la police présentée par l'application. Une personne daltonienne peut modifier le contraste des couleurs en fonction du type de daltonisme dont elle est atteinte ;

Fonctionnalité Les fonctionnalités parmi lesquelles il est possible de choisir, dont la relation d'héritage est sémantiquement similaire à une spécialisation *is-a* de type P du formalisme *Entité-Association*. Il

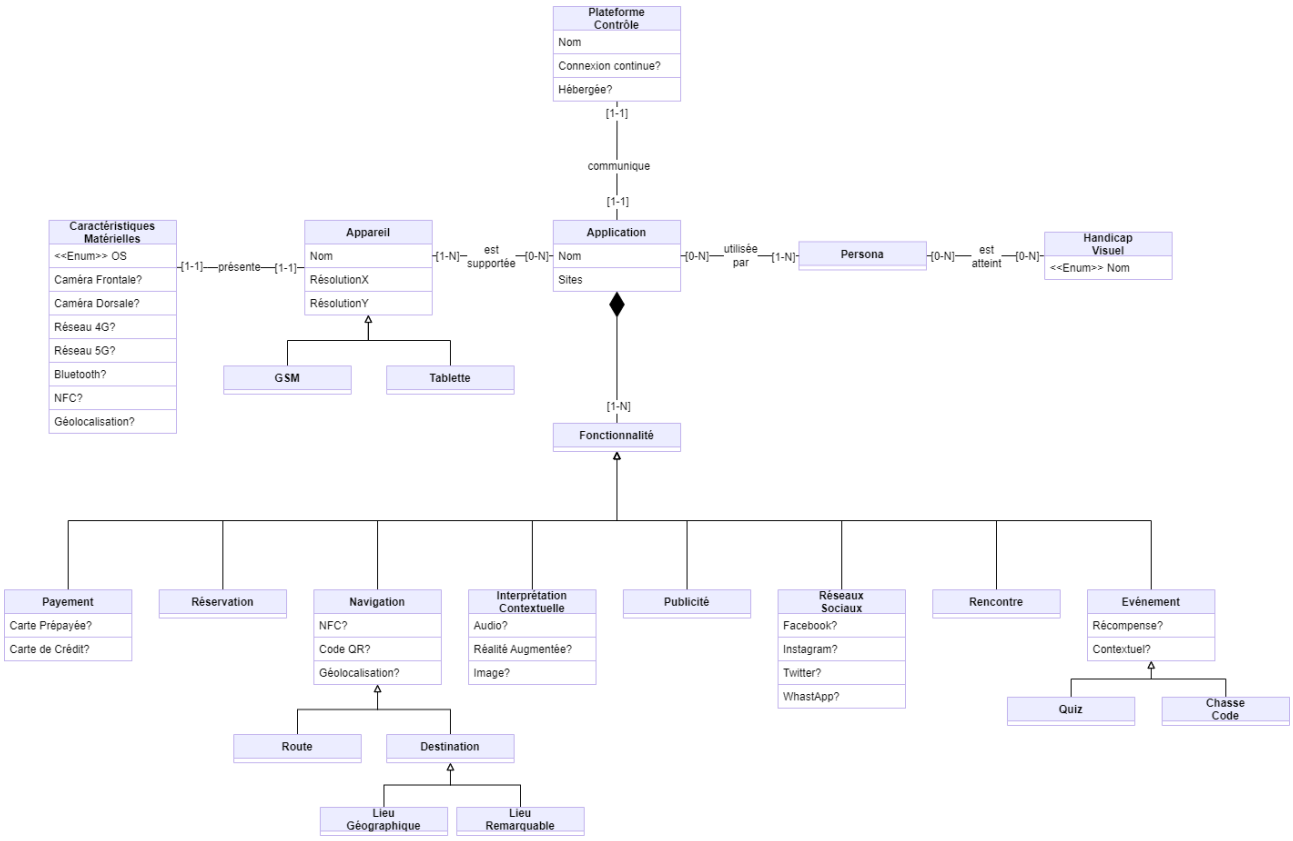


FIGURE 4.1 – Métamodèle du domaine d'application - notation *UML* simplifiée

s'agit des fonctionnalités de payement, de réservation, de navigation, d'interprétation contextuelle, de publicité, d'intégration avec les réseaux sociaux, de rencontre et enfin d'événements. L'entièreté des relations d'héritage retrouvées dans les sous-graphes à partir des variantes du concept de *Fonctionnalité* sont sémantiquement similaires aux spécialisations *is-a* de type T du formalisme *Entité-Association*.

Certains concepts du métamodèle présentent également des attributs. Ceux-ci représentent des précisions quant aux différentes préoccupations propres aux concepts du domaine d'application. Par exemple, quelles sont les caractéristiques physiques d'un appareil importantes à prendre en compte ? La réponse à cette question influencera les configurations possibles. Ainsi, si un GSM ne possède pas de système de géolocalisation, cela influencera entre autres la manière de concevoir la fonctionnalité de navigation et les possibilités liées à une configuration. Ce métamodèle permet donc de définir le formalisme syntaxique que devront respecter les modèles suggérés par les différents clients.

4.1.2 Jeu de Contraintes

Afin d'assurer la validité des modèles qui seront générés par les clients, les contraintes suivantes sont ajoutées au métamodèle. Celles-ci sont exprimées en langage formel et doivent être satisfaites.

(C1) Une même *Fonctionnalité* ne peut être sélectionnée plus d'une fois ;

(C2) Chaque instance de *Caractéristiques Matérielles* doit présenter au moins un des attributs suivants : *Réseau 4G*, *Réseau 5G* ;

(C3) Si la fonctionnalité *Navigation* est sélectionnée, alors tous les *Appareils* doivent présenter la caractéristique matérielle correspondant à l'option choisie pour la navigation. Ainsi :

- (C3a) Si l'attribut *NFC* est sélectionné, alors toutes les instances de *Caractéristiques Matérielles* présentent l'attribut *NFC* ;
- (C3b) Si l'attribut *Code QR* est sélectionné, alors toutes les instances de *Caractéristiques Matérielles* présentent soit l'attribut *Caméra Frontale*, soit *Caméra Dorsale* ou les deux ;
- (C3c) Si l'attribut *Géolocalisation* est sélectionné, alors toutes les instances de *Caractéristiques Matérielles* présentent l'attribut *Géolocalisation*.

(C4) Si la fonctionnalité *Interprétation Contextuelle* est sélectionnée, alors toutes les instances de *Caractéristiques Matérielles* présentent soit l'attribut *Caméra Frontale*, soit *Caméra Dorsale* ou les deux ;

(C5) Si la fonctionnalité *Rencontre* est sélectionnée, alors la fonctionnalité *Navigation* l'est aussi ;

(C6) Si la fonctionnalité *Evénement* est sélectionnée, alors toutes les instances de *Caractéristiques Matérielles* présentent au moins un des attributs suivants : *Réseau 4G*, *Réseau 5G*, *Bluetooth*.

(C6a) En particulier, si la sous-fonctionnalité *Chasse Code* est sélectionnée, alors toutes les instances de *Caractéristiques Matérielles* présentent soit l'attribut *Caméra Frontale*, soit *Caméra Dorsale* ou les deux.

4.1.3 Métamodèle Annoté

Dans l'optique de transformer automatiquement le métamodèle présenté ci-avant en un diagramme de caractéristiques, la méthodologie *Bespoke* propose de l'annoter. Ces notations permettent de formaliser et diriger la conversion. Celles-ci ainsi que les règles de conversion sont présentées dans la section 5.2, *Modélisation des Features*, de l'ouvrage syllabus [31]. Ces annotations permettent d'offrir un sens de parcours afin de transformer le métamodèle en une structure arborescente en précisant la manière d'interpréter et de transformer au besoin les concepts rencontrés lors de la visite. La figure 4.2 présente une version annotée du métamodèle du domaine d'application. La figure A.2 disponible en annexe en présente une version agrandie. Le jeu de contraintes défini dans la section 4.1.2 reste pertinent.

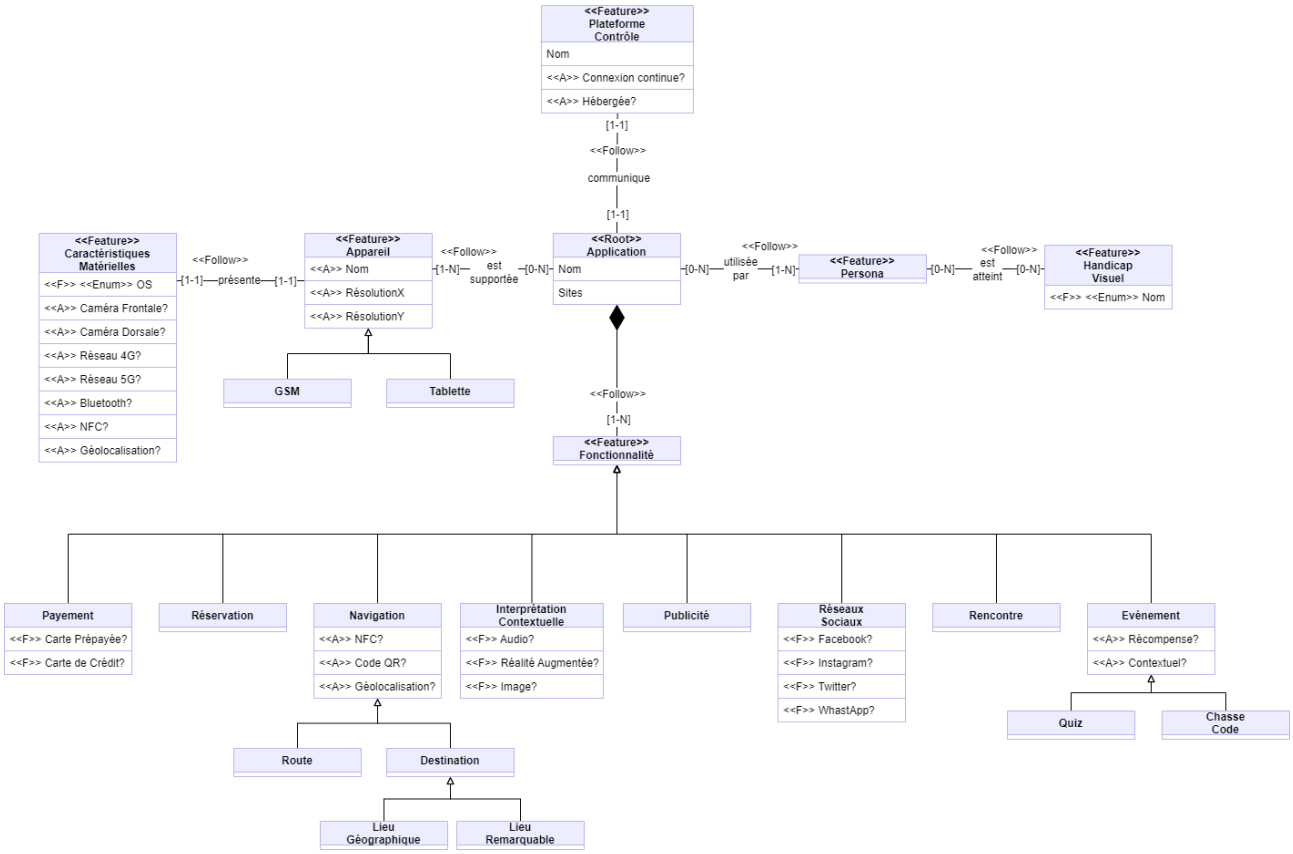


FIGURE 4.2 – Métamodèle annoté du domaine d'application - notation *UML* simplifiée

Le concept *Application* est caractérisé par l'annotation *«Root»*. Ainsi, celui-ci représente le point d'entrée de la méthode de conversion. L'entièreté des concepts restants sont quant à eux caractérisés par l'annotation *«Feature»*. Ces concepts représentent donc les différentes caractéristiques des configurations qu'il sera possible de générer. Leurs attributs annotés représentent des variations dans les caractéristiques correspondantes. Ceux dénotés par *«A»* représentent des préoccupations internes aux caractéristiques, tandis que ceux dénotés par *«F»* seront transformées en caractéristiques à part entière par la suite, car ils présentent une variabilité plus grande. Par exemple, le concept *Caractéristiques Matérielles* présente les attributs *Caméra Frontale* et *OS*. Le premier, annoté par *«A»*, désigne simplement une caractéristique du matériel : l'appareil possède-t-il une caméra frontale ? Le second, annoté par *«F»*, s'intéresse au système d'exploitation de l'appareil. S'agit-il d'*Android*, d'*iOS* ou

encore d'un autre système d'exploitation ? La nuance est subtile. Par ailleurs, certains attributs ne sont pas annotés. Bien que ceux-ci soient intéressants au niveau de la modélisation du domaine d'application, ils ne représentent pas une information utile pour la conception de l'usine à logiciels. Ces attributs seront alors laissés tombés lors de la transformation du métamodèle annoté en diagramme de caractéristiques.

4.1.4 Syntaxe Concrète

Afin de manipuler le métamodèle présenté par la figure 4.1, une syntaxe concrète lui est associée. Celle-ci peut être textuelle ou graphique, mais se doit de faciliter l'expression des exigences d'un client. Étant donné que ceux-ci ne sont pas initiés aux techniques et formalismes informatiques, il serait plus judicieux d'adopter une approche graphique interactive. Celle-ci se présenterait en même temps comme un outil de configuration et comme un prototype du logiciel à développer. Elle devrait permettre de produire un visuel plus accessible pour les clients. La figure 4.3 propose ainsi une syntaxe concrète qui répond à ces critères. Celle-ci marie boutons utilisables et icônes intuitifs [49]. La capacité à définir une telle syntaxe concrète est un avantage stratégique indéniable de l'approche des lignes de produits grâce à un DSML mais nécessite toutefois l'investissement de ressources supplémentaires dans l'élaboration de ladite syntaxe.

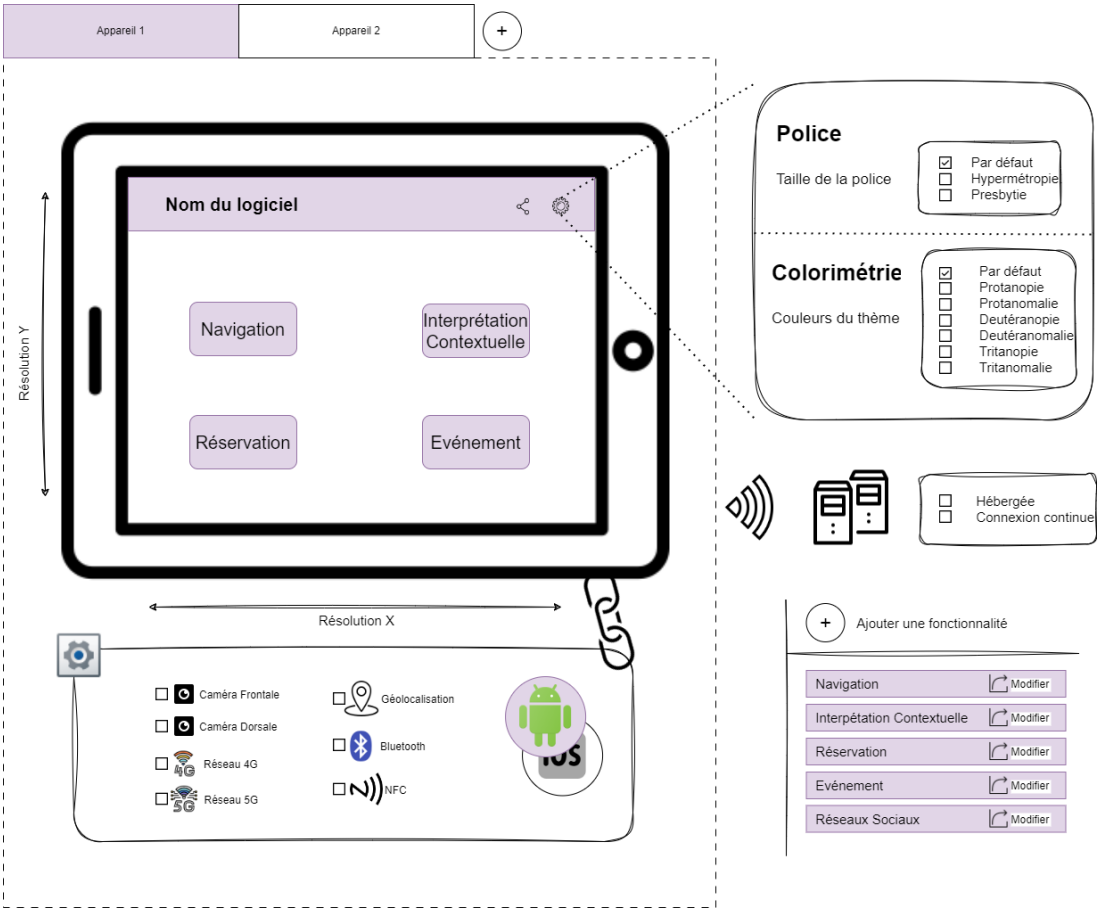


FIGURE 4.3 – Extrait de la syntaxe concrète du DSML

L'outil de modélisation qui met en avant cette syntaxe concrète implémente les contraintes mentionnées dans la section 4.1.2, permettant par conséquent d'assurer l'élaboration d'un modèle licite. Il met en avant un panneau de création d'appareils dont il est possible de modifier les caractéristiques matérielles. Les différents appareils définis sont liés à un ensemble de caractéristiques communes : ils partagent tous la même page de paramètres, prenant en charge les handicaps visuels, la même plateforme de contrôle et les mêmes fonctionnalités. Un visuel de la page d'accueil du logiciel est présenté au centre de l'appareil sélectionné et s'adapte en fonction de celui-ci. Enfin, il est possible de rajouter des fonctionnalités et de les modifier dans une fenêtre séparée. La figure 4.4 met en avant la fonctionnalité d'interprétation contextuelle qu'il est possible de personnaliser en cochant les carrés liés à celle-ci.

Remarque(Rem·II) ► Approche par DSML

L'approche des lignes de produits logiciels au moyen d'un DSML permet la définition d'une syntaxe concrète destinée à le manipuler. Celle-ci peut être définie sur mesure par l'ingénieur en charge de la conception de l'usine à logiciels et n'est pas restreinte par un ensemble de limitations liées à une méthodologie ou un outil. Par conséquent, la syntaxe concrète du DSML peut prendre la forme optimale élaborée par l'ingénieur pour communiquer au client les préoccupations et subtilités liées au domaine d'application. Dans le cas de l'entreprise *BetterSoft*, la syntaxe concrète présentée par la figure 4.3 est la combinaison d'un configurateur interactif et d'un prototype d'un logiciel que le client souhaiterait obtenir. La conception d'une telle syntaxe peut être guidée par les principes d'interactions homme-machine.

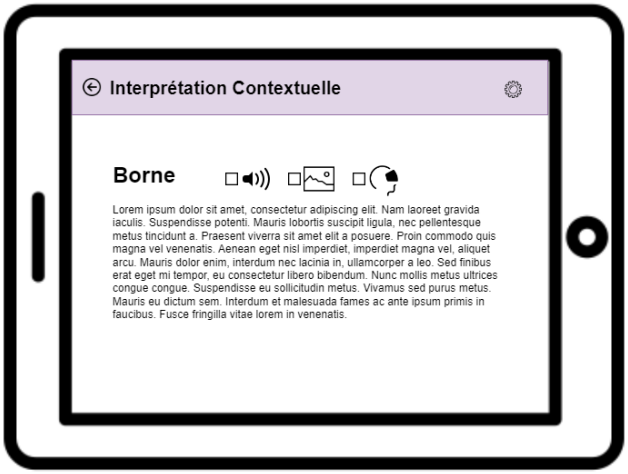


FIGURE 4.4 – Exemple de fonctionnalité modifiable

4.1.5 Scénario de Commande du Logiciel *JarreDeux*

Afin de rendre la démarche plus concrète, un scénario de commande d'un logiciel particulier auprès de l'entreprise *BetterSoft* est introduit ci-après. Ce scénario va permettre, entre autres, de donner corps à la méthodologie et de se concentrer sur un ensemble réduit de fonctionnalités. En effet, implémenter l'usine à logiciels telle que décrite par le métamodèle de la figure 4.1 nécessiterait des ressources matérielles et temporelles trop importantes dans le cadre de ce mémoire, ainsi qu'une priorisation des caractéristiques conforme à ce qui est décrit dans la section 3.5 et la figure 3.1 du chapitre 3. Le scénario est le suivant :

L'entreprise *JarreDin* est une petite entreprise qui gère un ensemble de jardins remarquables ouverts au public. Les visiteurs peuvent découvrir les beautés cachées de la nature sur un site d'un seul tenant regroupant une multitude de jardins de cultures et styles différents. Ces jardins sont disposés de part et d'autre du site et sont reliés entre eux par des sentiers pittoresques qui serpentent entre les arbres. Devant chacun de ces jardins est disposée une borne. Ces bornes présentent des informations détaillées sur le jardin qu'elles référencent.

Dans l'espoir d'attirer une nouvelle clientèle, l'entreprise a pris la décision de numériser ses activités. Ainsi :

— Les différentes bornes sont chacune équipées d'une code QR. L'entreprise espère relier ces codes à une application capable de lire à haute voix le contenu de la borne, ainsi que de présenter des images adaptées des différents jardins, par exemple une image par saison ;

— Ces bornes disposent également d'un marqueur géographique signalant en tout temps leur position à un serveur particulier que l'entreprise possède, afin de les localiser. L'entreprise *JarreDin* pense pouvoir les mettre à profit afin de proposer aux visiteurs la possibilité de se rendre à un jardin particulier grâce au nom dudit jardin ;

— L'entreprise a déjà fait les démarches auprès de fournisseurs afin d'acheter des GSM sous *iOS* et des tablettes sous *Android*. Ces appareils pourront être empruntés par les visiteurs à l'entrée du site afin de, par exemple, se rendre à un endroit précis et d'obtenir des informations contextuelles sur ce lieu ;

— Les appareils, qui seront donc mis à disposition des visiteurs pour la durée de la visite, sont autonomes et indépendants. Ils contiendront l'entièreté des informations liées aux jardins. Cependant, ils seront parfois mis à jour, par exemple pour modifier la description d'une borne, lorsqu'ils ne sont pas utilisés. Pour ce faire, l'entreprise souhaite disposer d'une plateforme centralisant ces données avec laquelle les appareils pourront communiquer. Celle-ci serait hébergée sur leur propre serveur minimaliste et manipulée par un informaticien sur place ;

— Soucieuse d'offrir la meilleure expérience possible et de partager leur passion des jardins, l'entreprise aimerait que les visiteurs atteints de daltonisme *rouge-vert* puisse disposer d'une application qui modifierait les images contextuelles fournies grâce aux bornes selon leur déficience visuelle ;

— L'entreprise aimerait également pouvoir organiser des chasses au code QR afin de faire découvrir aux visiteurs des jardins de saison, ou d'un certain thème.

L'entreprise *JarreDin* réalise donc une commande de logiciel à l'entreprise *BetterSoft* pour une application nommée *JarreDeux*. Les exigences de l'application correspondent à celles décrites ci-avant.

4.1.6 Modèle du Logiciel *JarreDeux*

La figure 4.5 présente une partie du modèle correspondant aux exigences émises par l'entreprise *JarreDin*. Celui-ci est conforme au métamodèle établi et détaillé par la figure 4.2 et respecte le jeu de contraintes imposé en section 4.1.2. En particulier, les contraintes (C1), (C2), (C3c), (C4) et (C6a), potentiellement mises à mal, ont été satisfaites. L'entièreté du modèle, fragmenté en plusieurs figures, est disponible dans l'annexe A.3.

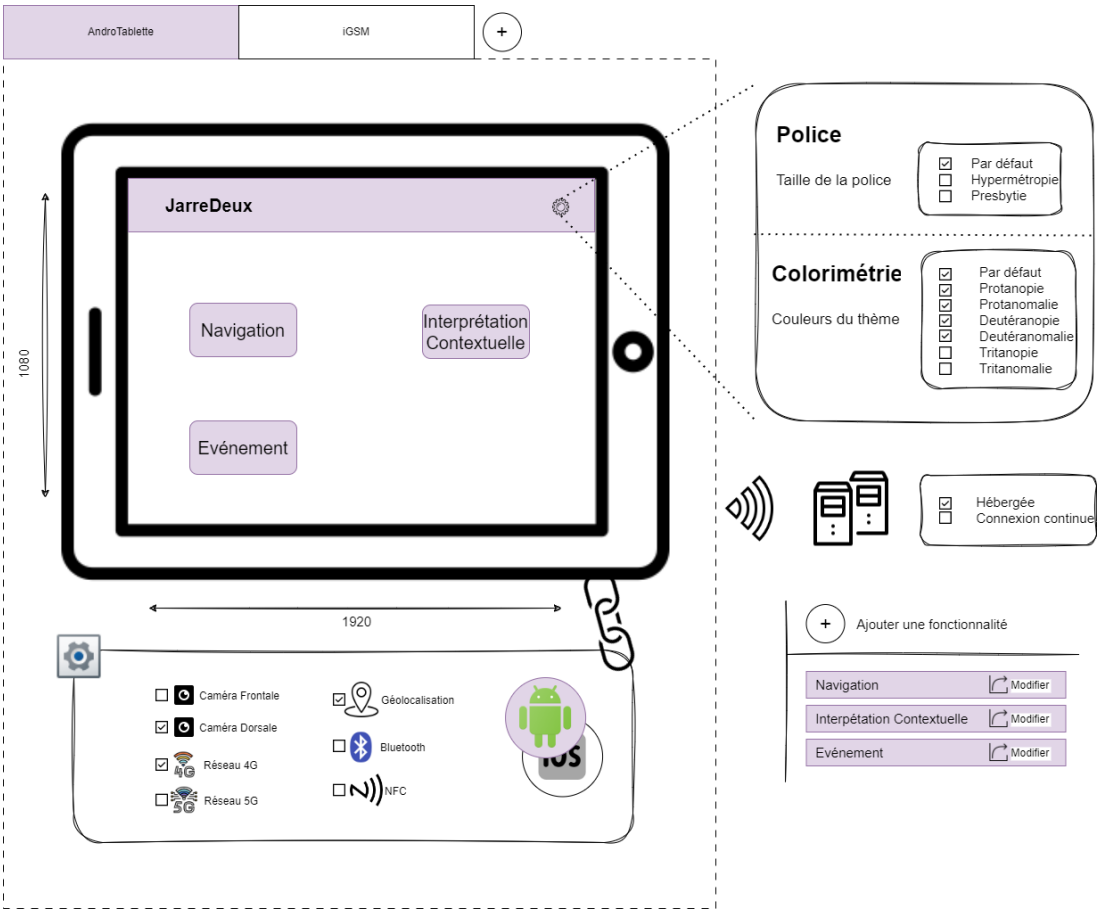


FIGURE 4.5 – Extrait du modèle du logiciel *JarreDeux*, conforme au métamodèle 4.2

4.2 Modélisation des Caractéristiques

Alors que le métamodèle permet de mettre en lumière les exigences d'un client, il n'est pas idéal pour concevoir l'usine à logiciels. Ainsi, un diagramme de caractéristiques, dérivé à partir de celui-ci, lui est préféré. La conversion du métamodèle et l'obtention du diagramme de caractéristiques constituent la troisième étape

du plan *SPL Engineer* de la méthodologie *Bespoke* Ce diagramme est conforme à la spécification donnée par le panneau 1 de la figure 2.7. Celui-ci est ensuite annoté d'informations cruciales au guidage de la génération de code, tel que prescrit par la quatrième étape dudit plan. Ces annotations permettront de choisir les tactiques d'implémentation à employer pour réaliser un point de variation, ses attributs et ses variantes. Ces tactiques sont abordées dans le chapitre 5. Les annotations ainsi rajoutées correspondent au panneau 2 de la figure 2.7.

4.2.1 Diagramme de Caractéristiques

La transformation s'effectue au moyen des primitives τ_e(E : Type d'entité) et τ_a(F : Feature, E : Type d'entité, A : Attribut) définies dans la section 5.2, *Modélisation des Features*, de l'ouvrage syllabus [31]. Celle-ci est effectuée de manière manuelle car son automatisation est hors de portée de ce document, comme indiqué par l'hypothèse (H3). L'annexe A.1 apporte quelques précisions au sujet de cette transformation.

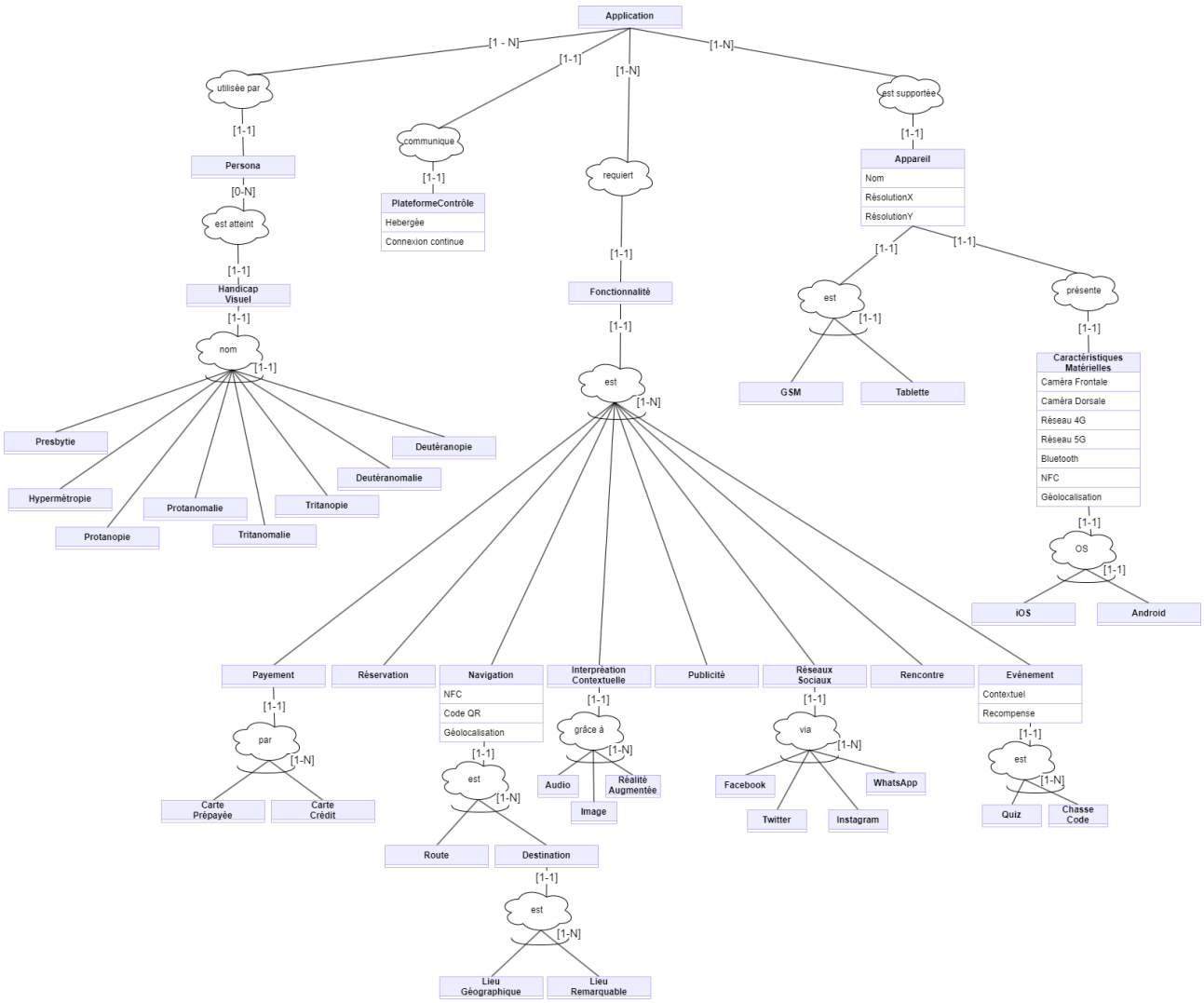


FIGURE 4.6 – Diagramme de caractéristiques dérivé du métamodèle 4.2

La figure 4.6 présente le diagramme de caractéristiques, converti à l'aide des primitives τ_e et τ_a mentionnées ci-avant. La figure A.3 disponible en annexe en présente une version agrandie. Dans la mesure où le métamodèle n'était pas complexe, la conversion s'est déroulée sans gêne. En effet, celui-ci ne contenait aucun cycle, ni association *n-aire* - avec n $>$ 2. Somme toute, le diagramme de caractéristiques ne doit donc pas faire l'objet de modifications ultérieures. Les cardinalités indiquées sur le diagramme représentent la sémantique logique de choix d'une caractéristique, ou d'un groupe de caractéristiques. Il en existe quatre : *OR* ou *1-N*, *XOR* ou *1-1*, *AND* ou *N-N* ou encore aucune des trois, ou *0-N*. Par exemple, un groupe caractérisé par la construction *XOR*, ou *1-1*, signifie que seule une des caractéristiques présentes dans ce groupe peut être sélectionnée.

Il est intéressant de noter que de nouveaux concepts, apparus sur le diagramme, n'étaient pas présents sur le métamodèle. Ceci est dû à la primitive de conversion τ_a, qui convertit les attributs annotés par *«F»* en caractéristiques à part entière. Ainsi, le nom d'un handicap visuel est devenu un groupe de caractéristiques portant la cardinalité *1-1*, qui représente une construction *XOR*. Il en va de même pour le système d'exploitation des appareils. Les fonctionnalités de payement, d'interprétation contextuelle et d'intégration avec les réseaux sociaux représentent quant à eux une construction *OR*, à l'instar des relations de spécialisation *is-a* de type T.

Remarque(Rem·III) ► Conversion du Métamodèle

La conversion du métamodèle en un diagramme de caractéristiques semble automatisable en l'état selon les primitives τ_e et τ_a uniquement dans la mesure où le métamodèle ne présente pas de constructions complexes. En effet, si d'aventure celui-ci présente des cycles ou des associations n-aires avec n strictement supérieur à 2, le processus de conversion est mis à mal et contraint de s'arrêter. Il existe deux possibles solutions à ce problème. La première est d'interdire toute construction complexe. Cependant, cela restreint l'expressivité du métamodèle et pourrait le rendre inapte à exprimer les besoins du domaine d'application. La seconde consiste à étendre le métamodèle et à élaborer des règles de conversion plus complètes, par exemple à l'aide d'heuristiques ou d'annotations ad hoc supplémentaires. Conformément à l'hypothèse (H3), cette problématique ne sera pas approfondie dans ce document.

4.2.2 Jeu de Contraintes Étendu

La section 4.1.2 a défini un ensemble de contraintes sur le métamodèle. Celles-ci s'appliquent également au diagramme de caractéristiques. Par conséquent, ces contraintes doivent également être satisfaites par les configurations obtenues à partir de modèles particuliers, car celles-ci se doivent d'être conformes au diagramme de caractéristiques. Ces contraintes sont complétées par celle définie ci-après :

(C7) Si, pour une configuration donnée, de multiples instances du concept *Appareils* existent, alors le concept *OS* instancié est toujours identique. Par exemple, si un appareil présente le système d'exploitation *Android*, alors tous les autres appareils doivent le posséder aussi. Si tel n'est pas le cas, la configuration doit être répliquée autant de fois qu'il y a de systèmes d'exploitation différents, en répliquant les autres concepts à l'identique.

L'ajout de la contrainte (C7) est nécessaire car, dans l'exemple des logiciels produits par *BetterSoft*, les langages de programmation sont différents en fonction des systèmes d'exploitation. Par conséquent, il est fort probable que la génération du produit lié à une configuration s'effectue différemment d'une plateforme à l'autre, étant donné que ces langages ont des spécificités propres qui ne se retrouvent pas intégralement dans d'autres langages. Ces différences ont été précédemment illustrées par les tables 2.1 et 2.2 dans le chapitre 2.

Remarque(Rem·IV) ► Scission d'une Configuration

La structure méthodologique *Bespoke* autorise dans sa définition la scission d'une configuration en de multiples fragments distincts. Cependant, aucun mécanisme n'est mis en place pour prendre en charge cette scission et les futurs problèmes qu'elle pourrait introduire. Dans le cas du logiciel *JarreDeux*, qui scinde sa configuration selon les systèmes d'exploitation, les problèmes sont les suivants :

- La fonctionnalité *Plateforme Contrôle* est dédoublée. Cependant, il s'agit de la même instance de celle-ci dans les deux configurations créées. En effet, la plateforme de contrôle est indépendante des systèmes d'exploitation *Android* et *iOS* car elle ne réside pas sur une plateforme mobile. Il n'existe aucun moyen de communiquer cette problématique d'unicité à l'usine à logiciels car les configurations sont traitées séparément. Une solution possible à cette problématique serait de permettre le traitement parallèle des configurations ainsi scindées et d'annoter ces caractéristiques particulières. Dans le cas de la caractéristique *Plateforme Contrôle*, l'étiquète ajoutée pourrait être « distinct ». L'usine à logiciels ne produirait alors qu'une seule instance de ladite caractéristique. Une solution alternative consisterait quant à elle en l'isolement de cette caractéristique dans un diagramme dédié, à nouveau grâce à une étiquète nouvellement ajoutée ;
- Le choix des tactiques définies dans le chapitre 5 pour mettre en œuvre les différents concepts dépend du système d'exploitation présenté par une configuration distincte. En effet, l'architecture logicielle et les artefacts liés à celle-ci présentent des subtilités propres à chacune des plateformes. Il est donc impératif d'élaborer un mécanisme permettant de résoudre cette problématique si d'aventure l'objectif est d'automatiser le processus d'application des tactiques. Cette problématique est abordée plus en détail par la remarque (Rem·VII).

4.2.3 Configurations du Logiciel *JarreDeux*

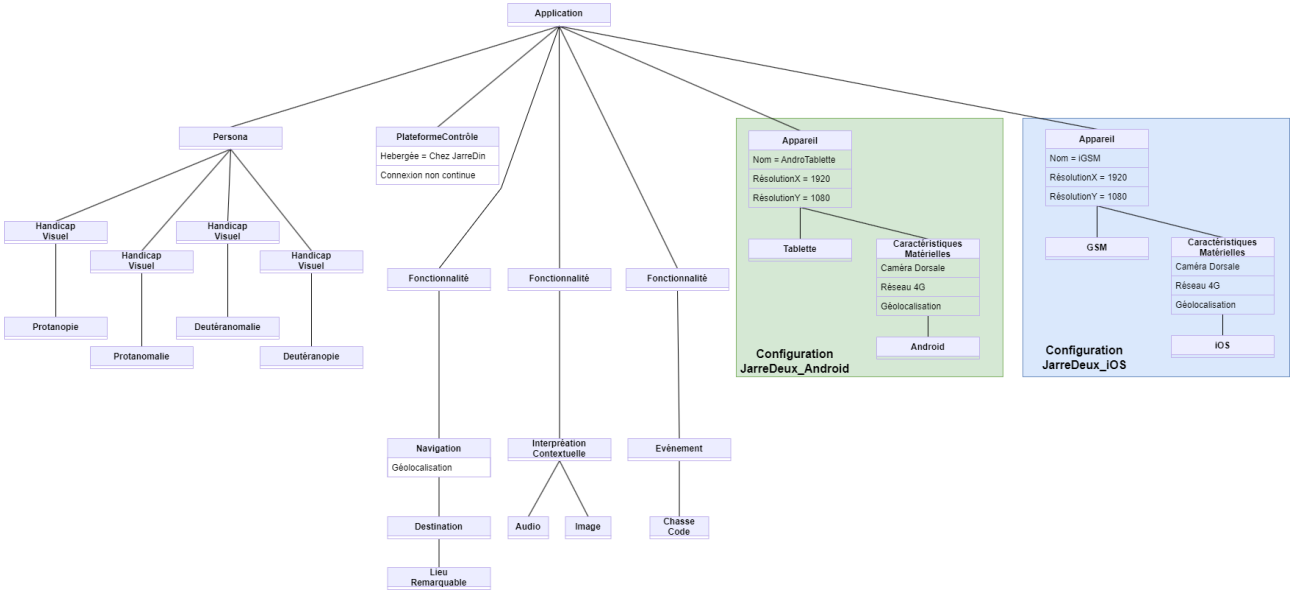


FIGURE 4.7 – Configurations du logiciel *JarreDeux*, conformes au diagramme de caractéristiques 4.6

La figure 4.7 détaille les deux configurations obtenues grâce à la conversion du modèle du logiciel *JarreDeux*. La figure A.4 disponible en annexe en présente des versions agrandies. Celles-ci sont au nombre de deux suite à la satisfaction de la contrainte (C7). Ces configurations, également transformées à l'aide des primitives τ_e et τ_a, sont par conséquent conformes au diagramme de caractéristiques de la figure 4.6. Par souci de concision, les deux configurations sont représentées en une seule et même figure. Le panneau vert intitulé *Configuration JarreDeux_ Android* reprend les appareils sous le système d'exploitation *Android*, tandis que celui intitulé *Configuration JarreDeux_ iOS* reprend ceux sous le système d'exploitation *iOS*. Les autres concepts sont identiques pour les deux configurations. Aussi, bien que la caractéristique *Plateforme Contrôle* soit par conséquent présente deux fois, une seule plateforme de contrôle sera utilisée pour les deux configurations. Comme précisé par la remarque (Rem·IV), cette caractéristique pourrait être isolée dans un diagramme de caractéristiques dédié.

4.2.4 Diagramme de Caractéristiques Annoté

L'annotation du diagramme de caractéristiques consiste en la précision de certaines particularités associées à celles-ci. Ces précisions vont permettre de choisir dans le chapitre 5 les tactiques associées à chaque point de variation et variante. Pour chacun d'eux où il est pertinent de réaliser cela, les informations mentionnées ci-après sont ajoutées. La table 4.1 met en évidence ces informations.

— **Ouverture** : un attribut booléen qui indique si le point de variation peut être étendu par de nouvelles variantes durant une phase d'extension. Cette phase est explicitée en section 2.1.2. La table 4.1 représente cet attribut au moyen des valeurs *Ouverte* et *Fermée* ;

— **Phase de sélection** : noté *[Sélection]* dans la table 4.1, cet attribut précise la phase de sélection du point de variation ou de ses variantes. Ces phases sont détaillées par la figure 2.1 et sont abrégées en prenant les deux premières lettres de chaque phase. En ce qui concerne les phases *runtime.launch*, *runtime.init* et *runtime.run*, ceux-ci sont abrégés *rl*, *ri* et *rr*, respectivement ;

— **Phase d'activation** : noté *[Activation]* dans la table 4.1, cet attribut précise la phase d'activation du point de variation ou de ses variantes. À l'instar de la phase de sélection, les phases correspondantes sont détaillées par la figure 2.1. Les abréviations sont également similaires ;

— **Impacts** : cet attribut propose un aperçu des artefacts, déjà existants ou à développer, qui seront impactés par le point de variation ou les variantes concernés. Les artefacts précis sont élaborés lors de la définition des tactiques les impactant dans la section 5.2 du chapitre 5.

Dans l'optique de manipuler le diagramme de caractéristiques annoté ainsi que les configurations générées, un langage de modélisation textuel est plus approprié pour les représenter. Celui-ci permet à des outils informatiques de manipuler les modèles produits et d'effectuer des processus automatiques. Ainsi, le diagramme de caractéristiques et les configurations du logiciel *JarreDeux* sont exprimés par les listings A.1 et A.2, respectivement, selon le langage FeatAll [31].

Remarque(Rem·V) ► Annotation du Diagramme de Caractéristiques

L'annotation du diagramme de caractéristiques ainsi proposée par la table 4.1 diffère de celle préconisée par la structure méthodologique *Bespoke*. En effet, cette dernière demande de préciser pour chaque point de variation et variante une tactique suggérée ainsi que des artefacts précis. Cependant, dans la mesure où ces artefacts dépendent des tactiques définies sur l'architecture de base d'une configuration, il semble plus pertinent de repousser l'élaboration de ces détails à l'étape suivante, soit à la définition des stratégies. Les tactiques ainsi définies et les artefacts impactés sont par conséquent abordés dans le chapitre 5. En outre, ces détails, n'apparaissent pas dans le panneau 2 du métamodèle présenté par la figure 2.7. Il semble par conséquent y avoir des inconsistances entre le métamodèle et la méthodologie prescrite.

4.3 Conclusion

Ce chapitre a permis de formaliser les exigences des clients par rapport au domaine d'application défini dans le chapitre 3. Le métamodèle présenté par la figure 4.2 permet d'exprimer ces exigences de manière aisée grâce à une syntaxe concrète. Celle-ci est illustrée par le modèle du logiciel *JarreDeux* repris en annexe A.3. Le métamodèle annoté est alors converti en un diagramme de caractéristiques, exprimé par la figure 4.6, selon les primitives τ_e et τ_a. Cette transformation permet ainsi de convertir un modèle produit par un client en une configuration particulière. Ce mécanisme est illustré par les configurations 4.7 du logiciel *JarreDeux*. Le diagramme de caractéristiques annoté, détaillé par la table 4.1 et le listing A.1, servira par la suite d'artefact primordial au guidage du processus de génération de code.

Caractéristique ou association	Abstraction	[Sélection]	[Activation]	Impacts
Application	/	[de-co]		Logo
Plateforme Contrôle	/	[de-co]	[pr-co]	Serveur et module de connexion de l'application
Appareil	Ouverte	[an-co]	[rr]	Interface graphique, orientation
Appareil.est	/			
Caractéristiques Matérielles	/	[de-co]		Fonctionnalités
Caractéristiques Matérielles.OS	Fermée			Tout le projet (langage de programmation)
Handicap Visuel	Ouverte	[an-co]	[rr]	Interface graphique, images
Handicap Visuel.nom	/			
Fonctionnalité	Ouverte	[an-rr]		Logique métier
Fonctionnalité.est	/			
Payement	Ouverte	[pr-co]		Fonctionnalité **PAY**
Payement.par	/	[pr-co]	[rr]	
Réservation	/	[pr-co]		Fonctionnalité **BOOK**
Navigation	Fermée	[de-co]	[de-rr]	Fonctionnalité **NAV**, carte
Navigation.est	/			
Destination	Fermée			
Destination.est	/			
Interprétation Contextuelle	Ouverte	[an-co]	[pr-co]	Fonctionnalité **INT**
Audio	/	[de-co]		
Image				Fonctionnalité **INT**, images
Réalité Augmentée		[an-co]		Fonctionnalité **INT**
Publicité	/	[de-co]		Fonctionnalité **PUB**
Réseaux Sociaux	Ouverte	[pr-co]		Fonctionnalité **SNET**
Réseaux Sociaux.via	/	[pr-co]	[pr-rr]	
Rencontre	/	[pr-co]		Fonctionnalités **MEET**, **NAV**
Evénement	Ouverte	[an-co]	[rr]	Fonctionnalités **FUN**, **CTRL**
Evénement.est	/			

TABLE 4.1 – Annotation du diagramme de caractéristiques. Le contenu de deux cellules est fusionné si celui-ci est identique et qu'elles font toutes deux parties d'une même construction sémantique, en étant par exemple liées à une même fonctionnalité. Par ailleurs, les intervalles, qui décideront des tactiques, reprennent fréquemment des phases distinctes car, à cette étape de la ligne de produits logiciels, les tactiques à employer sont encore inconnues. La table ne présente pas la caractéristique *Persona* car, étant trop abstraite, celle-ci ne fait pas l'objet d'une variation notable. En outre, les variantes des points de variation ne sont pas représentées directement. En effet, celles-ci sont reprises par l'association liée au point de variation qu'elles étendent.

Chapitre 5

Conception du Domaine

Ce chapitre détaille les tactiques d'implémentation mises en place pour réaliser les points de variation, leurs attributs pertinents et leurs variantes. Ces tactiques permettent de modifier l'architecture commune partagée entre les différentes configurations possibles dans l'optique de produire le logiciel demandé par le client. Les artefacts nécessaires à la mise en œuvre des tactiques et à l'adaptation de l'architecture commune sont par ailleurs définis sur base de celles-ci. Les caractéristiques sélectionnées par l'entreprise *JarreDin*, présentées par la figure 4.7 dans le chapitre 4, sont ensuite mises en place grâce à la définition de stratégies d'utilisation des tactiques associées à ces caractéristiques. Chaque stratégie vise un système d'exploitation particulier en conséquence de l'application de la contrainte (C7). La conception des tactiques et des stratégies sont réalisées et implémentées de façon concrète grâce aux langages *TacticType* et *Strategy* respectivement, définis dans l'ouvrage syllabus de la méthodologie [31]. La définition des tactiques et l'implémentation de ces caractéristiques grâce à différentes stratégies correspondent à la cinquième étape de la structure méthodologique *Bespoke*.

5.1 Architecture Commune *BetterSoft*

Remarque(Rem·VI) ► Formalisation d'une Architecture Commune

Les tactiques définies dans la suite de ce chapitre visent à modifier l'architecture commune des configurations afin produire le logiciel demandé par le client. Cependant, la définition de celle-ci est un aspect manquant de la conception de l'usine à logiciels telle que présentée par la structure méthodologique *Bespoke*. Étant donné que ces tactiques agissent sur cette architecture, il semble essentiel de développer celle-ci. Cette nouvelle étape se place entre la quatrième et la cinquième étape, soit entre l'annotation du diagramme de caractéristiques et la définition des stratégies d'implémentation. Cette étape est d'autant plus cruciale dans le cas d'une architecture « maison » car il n'existerait alors aucune documentation préalable. La formalisation de celle-ci est complétée par son adaptation, intervenant à la suite de la définition des tactiques et artefacts. Une telle architecture pourrait ainsi être définie au sein d'une tactique particulière, dont la tâche serait d'importer celle-ci afin que d'autres tactiques la modifient. Cette adaptation est par ailleurs discutée dans la remarque (Rem·X).

Les plateformes *Android* et *iOS* mettent en avant les mêmes architectures logicielles : le modèle *View Controller* [57] et son alternative récente *View ViewModel* [3]. Cette dernière est mise en place par défaut par les IDE *Android Studio* et *XCode*. Par conséquent, celle-ci est l'architecture logicielle choisie pour concevoir l'architecture commune des logiciels à produire. Il existe ainsi une architecture commune à adapter sur chacune des plateformes prises en charge. Celle-ci est directement dérivée de la syntaxe concrète présentée dans la section 4.1.4 du chapitre 4. Les différents artefacts qui composent cette architecture sont quant à eux explicités dans la section 2.4 du chapitre 2. Les structures générées par les IDE *Android Studio* et *XCode* étant suffisantes pour rencontrer les besoins émis par l'usine à logiciels *BetterSoft*, celles-ci seront employées telles quelles. Par conséquent, aucune architecture ad hoc n'est développée ici. Les figures B.1 et B.2 de l'annexe B.1 présentent respectivement l'architecture commune pour les configurations *iOS* et *Android*.

5.2 Ensemble des Tactiques *BetterSoft*

Tactique étiquette ▶ Concept visé
Artefacts : Les artefacts impactés par la tactique
Tactique : *La tactique proprement dite*
Sélection : Phase de sélection
Activation : Phase d'activation
Description : Une brève description du choix de tactique
Motivation : Une brève motivation de la justesse du choix de tactique

FIGURE 5.1 – Définition d'une tactique

L'élaboration d'un ensemble de tactiques d'implémentation correspond à la première partie de la cinquième étape du plan *SPL Engineer* de la méthodologie *Bespoke*. Comme mentionné précédemment, chaque point de variation ainsi que leurs attributs pertinents et variantes font l'objet d'une définition de tactiques. Celles-ci sont introduites dans la section 2.3 du chapitre 2 qui leur est dédiée. Par ailleurs, plusieurs tactiques peuvent être combinées pour réaliser un point de variation, un attribut ou une variante. Ces derniers peuvent en effet nécessiter des modifications de différentes natures. Ces tactiques peuvent également différer d'un langage de programmation et d'une plateforme à l'autre, en fonction des caractéristiques présentées par ceux-ci. La figure 5.1 présente le formalisme utilisé pour définir les tactiques employées dans la suite, mettant en avant les informations requises par le panneau 3 de la figure 2.7. L'étiquette de couleur située dans le coin supérieur droit du cadre représente la plateforme ciblée par la tactique :

- **Tactique multiplateforme** : une tactique ne nécessitant aucune adaptation particulière, fonctionnant de manière identique sur les plateformes *Android* et *iOS* ;
- **Tactique adaptative** : une tactique fonctionnant de manière similaire sur les plateformes *Android* et *iOS*, moyennant des adaptations propres à chacune ;
- **Tactique *Android*** : une tactique exclusive à la plateforme *Android* en raison des caractéristiques intrinsèques du langage *Java* et de l'architecture logicielle d'*Android* ;
- **Tactique *iOS*** : une tactique exclusive à la plateforme *iOS* en raison des caractéristiques intrinsèques du langage *Swift* et de l'architecture logicielle d'*iOS*.

Si d'aventure plusieurs tactiques sont définies pour un même concept visé, il existe deux cas de figure possibles :

1. Toutes les tactiques sont multiplateformes et/ou adaptatives. Elles se combinent ainsi sans encombre. La caractéristiques *Plateforme Contrôle* illustre cette combinaison avec les tactiques (Multi·1a) et (Adapt·2a) ;
2. Certaines tactiques sont spécifiques à une plateforme particulière. Dès lors, celles-ci se combinent exclusivement avec les tactiques spécifiques à cette même plateforme ainsi que celles multiplateformes et adaptatives également mentionnées. Ainsi, si un même concept présente des tactiques *Android* et *iOS*, il existe donc deux ensembles de tactiques distincts dans lesquels les tactiques multiplateformes et adaptatives sont dédoublées. Les caractéristiques *Presbytie* et *Hypermétropie* illustrent cette distinction. L'ensemble des tactiques destinées à être mises en place sur la plateforme *Android* sont les tactiques (Adapt·5a) et (Android·3a), tandis que celles destinées à être mises en place sur la plateforme *iOS* sont les tactiques (Adapt·5a), (iOS·3a) et (iOS·3b).

Le nombre de tactiques différentes définies dans la suite est relativement restreint. Cela permet entre autres de réduire la charge de développement et de maintenance de celles-ci et de limiter les compétences requises pour les mettre en place. En particulier, la tactique de *patching* est employée à maintes reprises car elle présente pour l'usine à logiciels des avantages considérables par rapport aux autres. En effet, comme discuté dans la section 5.1, l'architecture commune de l'usine à logiciels est mise en place grâce à l'architecture logicielle *View ViewModel* sur les plateformes *Android* et *iOS*. Ainsi, dans la plupart des cas, des préoccupations similaires sont exécutées de manière identique et sont situées dans des artefacts équivalents. Cette homogénéité favorise l'élaboration d'un ensemble restreint de tactiques et en particulier celle de *patching*. Dans le présent cas, le *patching par soustraction* est préféré au *patching par addition* car toutes les caractéristiques, à l'exception d'*Android* et d'*iOS*, sont valides ensembles. Grâce à l'application de la contrainte (C7), cela permet ainsi d'assembler tous les artefacts au préalable et d'observer leurs interactions et de pallier les éventuels problèmes de façon préventive. Une attention particulière doit toutefois être apportée à l'implémentation de cette tactique lorsqu'il s'agit de

modifier les ressources « storyboard » de la plateforme *iOS*, car celles-ci ne sont pas conçues pour être modifiées sans passer par l'IDE *XCode*. La figure B.3 en annexe illustre cette manipulation. Bien que ces ressources soient des fichiers *XML* déguisés qu'il est ainsi possible de modifier, l'IDE attribue des identifiants automatiques aux balises lors de leur manipulation sous-jacente, la rendant par conséquent délicate. La totalité des tactiques définies ci-après sont reprises dans le tableau 5.1 afin d'offrir une vue plus claire des ensembles distincts de tactiques *Android* et *iOS*.

5.2.1 Communication

Application Les logos proposés par l'usine à logiciels doivent être remplacés par ceux fournis par le client.

Tactique (Adapt·1a) ▶ Application

Artefacts *Android* : Les ressources des *mipmaps*
Artefacts *iOS* : Les ressources des *image sets*
Tactique : *Substitution de composants*
Sélection : compile
Activation : compile
Description : Les logos fournis par défaut par l'usine à logiciels sont remplacés par ceux de la configuration à produire.
Motivation : Cette tactique est optimale car il suffit de substituer quelques fichiers par d'autres nommés de manière identique pour remplacer les logos.

Plateforme Contrôle Le serveur qui correspond à la plateforme de contrôle est indépendant du langage de programmation choisi pour concevoir l'application et ne réside pas sur un appareil *Android* ou *iOS*. Par conséquent, la technologie employée pour le concevoir peut être différente de celles employées pour développer les configurations.

Tactique (Multi·1a) ▶ Plateforme Contrôle

Artefacts : Serveur
Tactique : *Framework par héritage Django*
Sélection : program
Activation : program
Description : Le framework Django permet de déployer des serveurs rapidement, dont la mise à l'échelle est aisée. Les spécificités propres à chaque configuration sont à implémenter manuellement après génération de celle-ci.
Motivation : Un framework par héritage propose tous les outils nécessaires pour mettre en place un serveur de manière aisée sans écrire du code dit *boilerplate*.

Tactique (Adapt·2a) ▶ Plateforme Contrôle

Artefacts *Android* : Module de connexion d'une configuration
Artefacts *iOS* : Module de connexion d'une configuration
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Le module de connexion présente dans sa forme canonique toutes les méthodes pour assurer le fonctionnement de l'entièreté des fonctionnalités. Ainsi, le code source des fonctionnalités non sélectionnées est retiré.
Motivation : Cette tactique permet de s'assurer au préalable de la justesse et validité des interactions entre les composants et d'ensuite retirer aisément le code non utilisé.

5.2.2 Appareils Physiques

Appareil Cette caractéristique met en lumière la résolution de l'écran d'un appareil grâce aux attributs *RésolutionX* et *RésolutionY*. Ceux-ci sont concrétisés par l'utilisation des tactiques mentionnées ci-après. Étant donné que le métamodèle présenté par la figure 4.1 dans le chapitre 4 autorise la sélection de plusieurs appareils dont les résolutions peuvent différer, il est impératif de choisir des tactiques permettant de gérer habilement cette hétérogénéité.

Tactique (Adapt·3a) ▶ RésolutionX, RésolutionY

Artefacts *Android* : Les ressources des *drawables*
Artefacts *iOS* : Les ressources des *image sets*
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : compile
Activation : runtime.run
Description : Sur *Android*, en spécifiant plusieurs répertoires différenciés par des qualificatifs bien précis contenant des images de résolutions différentes, la configuration choisit automatiquement la plus adaptée [91] à l'exécution de l'application. Sur *iOS*, il suffit d'importer les différentes versions d'une image dans un même répertoire en les nommant selon leur résolution, au moyen de qualificatifs conventionnels - @2x & @3x. À l'instar d'*Android*, la configuration choisit automatiquement la version la plus adaptée [1] à l'exécution de l'application.
Motivation : Cette tactique permet d'exploiter la capacité des plateformes *Android* et *iOS* à sélectionner la meilleure résolution d'une image à l'exécution de la configuration.

GSM, Tablette Les proportions des écrans entre les GSM et tablettes diffèrent. Par conséquent, la manière de présenter l'information diffère également. Ces caractéristiques étant des spécialisations de la caractéristique *Appareil*, pouvant donc aussi être sélectionnées de multiples fois, il est impératif de choisir des tactiques permettant d'assurer une mise en page adaptée en fonction de l'appareil.

Tactique (Android·1a) ▶ GSM, Tablette

Artefacts *Android* : Les ressources « xml » des *layouts*
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : compile
Activation : runtime.run
Description : En spécifiant plusieurs répertoires différenciés par des qualificatifs bien précis, contenant des mises en page différentes, la configuration choisit automatiquement la plus adaptée [92] à l'exécution de l'application. Ainsi, si plusieurs appareils dont les proportions sont différentes sont présentés par une même configuration, l'adaptation s'effectue de manière automatique.
Motivation : Cette tactique permet d'exploiter la capacité de la plateforme *Android* à sélectionner la meilleure mise en page à l'exécution de la configuration.

Tactique (iOS·1a) ▶ GSM, Tablette

Artefacts *iOS* : Les ressources « storyboard » des *layouts*
Tactique : *Aucune (caractéristique intrinsèque)*
Sélection : program
Activation : runtime.run
Description : En mettant à profit l'approche à base de contraintes *Auto Layout*, il est possible de créer une mise en page qui s'adapte à l'espace disponible sur l'écran de l'appareil [7]. Cette approche est également disponible sur la plateforme *Android*. Cependant, celle-ci met en place un mécanisme de multiplicité des *layouts* plus intéressant en ce qui concerne les tactiques à employer. Par conséquent, la tactique (Android·1a) reste inchangée.
Motivation : La plateforme *iOS* supporte nativement une mise en page adaptative. Il n'est donc pas nécessaire d'utiliser une tactique sur mesure.

Caractéristiques Matérielles Cette caractéristique reprend les particularités physiques d'un appareil. Les tactiques associées doivent donc permettre à une configuration donnée de demander la permission d'accéder aux particularités sélectionnées.

Tactique (Adapt·4a) ▶ Caractéristiques Matérielles & Attributs

Artefacts *Android* : Fichier « AndroidManifest.xml »
Artefacts *iOS* : Fichier « Info.plist »
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Les permissions non nécessaires sont retirées du fichier de métadonnées. Celles-ci sont directement dérivées des attributs présentés par la caractéristique *Caractéristiques Matérielles*.
Motivation : Cette tactique permet de prévoir toutes les permissions nécessaires et d'ensuite retirer celles inutiles.

Android Cette caractéristique force l'utilisation du langage de programmation *Java* et de l'IDE *Android Studio* pour la configuration demandée.

Tactique (Android·2a) ▶ Android

Artefacts *Android* : Tous les artefacts liés à une configuration, excepté le serveur
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : design
Activation : design
Description : Cette tactique charge l'architecture de base d'une configuration *Android*, tel que discuté dans la remarque (Rem·VI).
Motivation : Cette tactique est l'unique tactique capable de dupliquer et charger du code écrit sur mesure. Elle s'impose d'elle-même.

iOS Cette caractéristique force l'utilisation du langage de programmation *Swift* et de l'IDE *XCode* pour la configuration demandée.

Tactique (iOS·2a) ▶ iOS

Artefacts *iOS* : Tous les artefacts liés à une configuration, excepté le serveur
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : design
Activation : design
Description : Cette tactique charge l'architecture de base d'une configuration *iOS*, tel que discuté dans la remarque (Rem·VI).
Motivation : Cette tactique est l'unique tactique capable de dupliquer et charger du code écrit sur mesure. Elle s'impose d'elle-même.

5.2.3 Handicaps Visuels

Persona Cette caractéristique n'a aucun impact sur les configurations générées. En effet, celles-ci sont déjà destinées à être manipulées par les visiteurs.

Handicap Visuel Par défaut, la configuration générée présente une page pour modifier ses paramètres. Si d'aventure la gestion d'un handicap visuel est sélectionnée, la configuration générée doit présenter un paramètre associé permettant de modifier les handicaps de l'utilisateur courant. Ce paramètre influence ainsi le reste de l'application, en fonction des handicaps sélectionnés. Le choix des tactiques employées à cette fin est délégué aux différents handicaps définis. La plateforme *Android* met à disposition un fichier partagé [87] qu'il est possible de mettre à profit pour modifier l'application en fonction des préférences sélectionnées par les utilisateurs. La plateforme *iOS* en fait par ailleurs de même [102]. Ceux-ci facilitent grandement la gestion des préférences au travers d'une configuration donnée.

Presbytie, Hypermétropie Ces handicaps visuels permettent de modifier la taille de la police d'écriture de l'application à l'exécution d'une configuration. Ainsi, les champs de texte sont affectés par la préférence active. Les tailles proposées pour répondre à la presbytie ou l'hypermétropie sont similaires.

Tactique (Adapt·5a) ▶ Presbytie, Hypermétropie

Artefacts *Android* : Les ressources « xml » des préférences et des listes
Artefacts *iOS* : La ressource « plist » des préférences
Tactique : *Patching par soustraction*
Sélection : compile
Activation : runtime.run
Description : L'usine à logiciels met à disposition des utilisateurs des tailles de police différentes. Si aucun des deux handicaps n'est sélectionné, alors les polices agrandies sont retirées des préférences.
Motivation : Cette tactique permet de prévoir toutes les préférences nécessaires et d'ensuite retirer celles inutiles.

Tactique (Android·3a) ▶ Presbytie, Hypermétropie

Artefacts *Android* : Tous les fichiers « java » d'activité
Tactique : *Trait*
Sélection : program
Activation : runtime.run
Description : Le trait sert d'interface concrète pour accéder aux préférences choisies par les utilisateurs. Celui-ci est par conséquent implémenté pour toutes les activités de l'application. Il met à disposition une méthode pour changer la taille de la police de l'entièreté des champs de texte de l'application. Celle-ci est ensuite redémarrée.
Motivation : Cette tactique permet d'ajouter aisément des fonctionnalités concrètes à une classe et de tirer parti de l'outil *singleton* des préférences partagées.

Tactique (iOS·3a) ▶ Presbytie, Hypermétropie

Artefacts *iOS* : Tous les fichiers « swift » de *ViewController*
Tactique : *Patron de conception Observeur*
Sélection : program
Activation : runtime.run
Description : Les différents *ViewControllers* s'inscrivent à une classe dont le rôle est de notifier ceux-ci en cas de changement dans les préférences enregistrées par l'utilisateur. Cette classe remplit le même rôle que le trait défini par la tactique (Android·3a), mais la plateforme *iOS* permet de facilement appliquer ces changements sans devoir redémarrer l'application. Cependant, à l'inverse de la tactique (Android·3a) qui propose également une méthode de changement de taille de police, le langage de programmation *Swift* propose une solution plus élégante et requiert ainsi la tactique complémentaire (iOS·3b).
Motivation : Ce patron de conception permet d'aisément ajuster la configuration aux préférences sélectionnées en tirant parti de l'outil *singleton* des préférences de l'utilisateur.

Tactique (iOS·3b) ▶ Presbytie, Hypermétropie

Artefacts *iOS* : Un fichier « swift » dédié
Tactique : *Aucune (caractéristique intrinsèque)*
Sélection : program
Activation : program
Description : Il est possible d'étendre un type grâce au mot-clef *extension*. Celui-ci permet entre autres de modifier ou rajouter des méthodes au type choisi. Cette technique est utilisée pour greffer une nouvelle méthode de changement de police au type *UIView* afin de pouvoir utiliser celle-ci dans les fichiers « swift » de *ViewController*.
Motivation : Le langage de programmation *Swift* permet nativement d'étendre un type, rendant cet ajout directement disponible à toutes les instances de ce type. Ceci permet d'éviter la définition du trait de la tactique (Android·3a).

Protanopie, Protanomalie, Deutéranopie, Deutéranomalie, Tritanopie, Tritanomalie Les types de daltonisme permettent de changer le contraste des couleurs d'une configuration à l'exécution de celle-ci. Ainsi, les thèmes et les images sont affectés par la préférence active.

Tactique (Adapt·6a) ▶ Protanopie → Tritanomalie

Artefacts *Android* : Les fichiers « java » des filtres
Artefacts *iOS* : Les fichiers « swift » des filtres
Tactique : *Patron de conception Stratégie*
Sélection : program
Activation : runtime.run
Description : Différents algorithmes sont implémentés dans l'optique d'appliquer un filtre sur les images présentes dans une configuration. En fonction du thème sélectionné, une stratégie spécifique est utilisée.
Motivation : Cette tactique permet d'appliquer aisément un algorithme sans se soucier de son implémentation.

Tactique (Android·4a) ▶ Protanopie → Tritanomalie

Artefacts *Android* : Tous les fichiers « java » d'activité
Tactique : *Trait*
Sélection : program
Activation : runtime.run
Description : Ce trait est le même que celui défini par la tactique (Android·3a). Il présente donc des méthodes supplémentaires afin de récupérer aisément le style à appliquer en utilisant les préférences enregistrées par l'utilisateur. Les fichiers d'activités qui l'implémentent peuvent ainsi appeler la méthode *setTheme* proposée par *Android* pour mettre à jour le thème courant.
Motivation : La motivation est identique à celle de la tactique (Android·3a).

Tactique (iOS·4a) ▶ Protanopie → Tritanomalie

Artefacts *iOS* : Tous les fichiers « swift » de *ViewController*
Tactique : *Patron de conception Observeur*
Sélection : program
Activation : runtime.run
Description : À l'inverse de la plateforme *Android* qui permet un changement de thème aisé grâce à l'application d'une unique méthode, dont tire parti la tactique (Android·4a), la plateforme *iOS* force la mise à jour explicite des éléments d'une vue. Étant donné les différences entre chaque vue, la responsabilité de modifier le thème courant est laissée à celles-ci. Cette mise à jour s'opère en s'inscrivant à une classe qui surveille les changements dans les préférences utilisateur enregistrées par l'utilisateur et notifie les objets abonnés. Cette classe est la même que celle définie par la tactique (iOS·3a).
Motivation : La motivation est identique à celle de la tactique (iOS·3a).

Tactique (Adapt·6b) ▶ Protanopie → Tritanomalie

Artefacts *Android* : Les ressources « xml » des préférences, des listes et des thèmes et « java » des tactiques (Adapt·6a) et (Android·4a)
Artefacts *iOS* : Les ressources « plist » des préférences et « swift » des thèmes et des tactiques (Adapt·6a) et (iOS·4a)
Tactique : *Patching par soustraction*
Sélection : compile
Activation : runtime.run
Description : Les types de daltonisme non sélectionnés par le client dans une configuration particulière sont retirés des ressources « xml » (*Android*) et « swift » et « plist » (*iOS*) correspondantes. Sur *iOS*, les différents thèmes sont implémentés grâce à un protocole étendu par chacun d'eux, justifiant ainsi l'utilisation d'une ressource « swift ». Les stratégies liées aux types de daltonisme non sélectionnés sont également retirées.
Motivation : Cette tactique permet de prévoir toutes les préférences et stratégies nécessaires et d'ensuite retirer celles inutiles.

5.2.4 Fonctionnalités

Fonctionnalité Les fonctionnalités partagent un ensemble commun de caractéristiques au niveau de leur création. Celles présentées par *BetterSoft* sont suffisamment simples pour permettre d'abstraire cet ensemble. Cette hypothèse peut se révéler insuffisante dans d'autres situations.

Tactique (Adapt·7a) ▶ Fonctionnalité

Artefacts *Android* : Fichiers « java » d'activité correspondant à la fonctionnalité sélectionnée et de fonctionnalité abstraite
Artefacts *iOS* : Fichiers « swift » de *ViewController* correspondant à la fonctionnalité sélectionnée et de fonctionnalité abstraite
Tactique : *Patron de conception Template*
Sélection : program
Activation : program
Description : Une fonctionnalité abstraite est étendue par les différentes fonctionnalités disponibles. Celle-ci définit un ensemble de méthodes concrètes et abstraites à compléter selon les besoins.
Motivation : Cette tactique permet de tirer parti des similarités partagées par les différentes fonctionnalités, tout en proposant un ensemble de méthodes pré-établies. Celle-ci est préférée au trait, capable de prouesses similaires, car elle est plus adaptée d'un point de vue sémantique.

Tactique (Adapt·7b) ▶ Fonctionnalité

Artefacts *Android* : Fichiers *layout* correspondant à la fonctionnalité
Artefacts *iOS* : Fichiers de ressources « storyboard » correspondant à la fonctionnalité
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : compile
Activation : compile
Description : Sur *Android*, le *layout* correspondant à la fonctionnalité doit être ajouté à la configuration. De même, sur *iOS*, il faut ajouter le fichier *storyboard* correspondant.
Motivation : Cette tactique peut être remplacée par un retrait de fichiers. Cependant, dans l'optique de rester cohérent avec les tactiques (Adapt·3a) et (Android·1a) qui ajoutent des ressources visibles par l'utilisateur, l'ajout de fichiers lui est préféré.

Tactique (Adapt·7c) ▶ Fonctionnalité

Artefacts *Android* : Fichiers « java » d'activité principal, *layout* associé, « AndroidManifest.xml » et « java » de fonctionnalité non sélectionnée
Artefacts *iOS* : Fichier de ressources « storyboard » principal et « swift » de fonctionnalité non sélectionnée
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Sur *Android*, le fichier d'activité principal, qui permet d'accéder à ces fonctionnalités, est adapté en retirant celles non sélectionnées. Il en va de même pour le fichier de *layout* associé ainsi que pour le fichier de métadonnées. Sur *iOS*, il suffit de retirer dans le « storyboard » principal les boutons associés et les références aux « storyboards » correspondant à ces fonctionnalités. Enfin, quelle que soit la plateforme, le fichier correspondant à la fonctionnalité non sélectionnée est retiré.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Payement Si d'aventure cette caractéristique est sélectionnée, sa mise en place effectuée par les tactiques (Adapt·7a) à (Adapt·7c). Ses variantes sont quant à elles mises en place par les tactiques définies ci-après.

Tactique (Adapt·8a) ▶ Carte Prépayée, Carte Crédit

Artefacts *Android* : Fichier « java » d'activité et *layout* correspondant
Artefacts *iOS* : Fichier « swift » de *ViewController* et « storyboard » correspondant
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Les méthodes de payement non sélectionnées sont retirées des fichiers correspondants.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Tactique (Adapt·8b) ▶ Carte Prépayée, Carte Crédit

Artefacts *Android* : Les fichiers « java » de payement
Artefacts *iOS* : Les fichiers « swift » de payement
Tactique : *Patron de conception Stratégie*
Sélection : program
Activation : runtime.run
Description : Différentes stratégies de payement sont implémentées et l'algorithme développé pour chacune d'elles est activé par l'utilisateur lors du choix du mode de payement. Celles non sélectionnées sont retirées grâce à la tactique (Adapt·8a).
Motivation : Cette tactique permet d'appliquer aisément un algorithme sans se soucier de son implémentation.

Réservation Si d'aventure cette caractéristique est sélectionnée, sa mise en place effectuée par les tactiques (Adapt·7a) à (Adapt·7c).

Navigation Si d'aventure cette caractéristique est sélectionnée, sa mise en place effectuée par les tactiques (Adapt·7a) à (Adapt·7c). Ses variantes sont quant à elles mises en place par les tactiques définies ci-après. *BetterSoft* tire parti du service *OpenStreetMap* proposant une carte exploitable hors-ligne. Celle-ci est téléchargée pour les sites touristiques correspondants et sa couleur est modifiée selon le thème sélectionné grâce à la tactique (Adapt·6a). S'il s'avère impossible d'exploiter la carte, il faudra alors soit changer de service, soit développer un service ad hoc et ainsi remanier les différentes tactiques mentionnées ci-après.

Tactique (Multi·2a) ▶ Carte *OpenStreetMap*

Artefacts : Fichier contenant la carte
Tactique : *Ajout de fichiers (Patching absolu)*
Sélection : design
Activation : design
Description : La carte souhaitée est téléchargée et ajoutée à la configuration.
Motivation : Cette tactique permet d'ajouter une carte sur mesure pour une configuration. Elle s'impose d'elle-même car l'architecture commune ne fournit pas de carte par défaut.

Tactique (Adapt·9a) ▶ Géolocalisation, Code QR, NFC

Artefacts *Android* : Fichier « java » d'activité
Artefacts *iOS* : Fichier « swift » de *ViewController*
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Les moyens d'obtention de la position non sélectionnés sont retirés des fichiers correspondants. Il s'agit du code déclenché associé. Les permissions liées aux moyens susmentionnés sont déjà gérées par la tactique (Adapt·4a).
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Tactique (Adapt·9b) ▶ Géolocalisation, Code QR, NFC

Artefacts *Android* : Les fichiers « java » de navigation
Artefacts *iOS* : Les fichiers « swift » de navigation
Tactique : *Patron de conception Stratégie*
Sélection : program
Activation : runtime.run
Description : Différentes stratégies d'obtention de la position sont implémentées et l'algorithme correspondant est activé par l'utilisateur en fonction de la stratégie désirée. Celles non sélectionnées sont retirées grâce à la tactique (Adapt·9a).
Motivation : Cette tactique permet d'appliquer aisément un algorithme sans se soucier de son implémentation.

Tactique (Adapt·10a) ▶ Route, Destination & Variantes

Artefacts *Android* : Fichier « java » d'activité
Artefacts *iOS* : Fichier « swift » de *ViewController*
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Les moyens de navigation non sélectionnés sont retirés des fichiers correspondants. Il s'agit des options ajoutées depuis un fichier source « java » sur *Android* et « swift » sur *iOS* ainsi que du code déclenché associé.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Tactique (Adapt·10b) ▶ Route, Destination & Variantes

Artefacts *Android* : Les fichiers « java » de localisation
Artefacts *iOS* : Les fichiers « swift » de localisation
Tactique : *Patron de conception Stratégie*
Sélection : program
Activation : runtime.run
Description : Différentes stratégies de navigation sont implémentées et l'algorithme correspondant est activé par l'utilisateur en fonction de la navigation désirée. Celles non sélectionnées sont retirées grâce à la tactique (Adapt·10a).
Motivation : Cette tactique permet d'appliquer aisément un algorithme sans se soucier de son implémentation.

Interprétation Contextuelle Si d'aventure cette caractéristique est sélectionnée, sa mise en place effectuée par les tactiques (Adapt·7a) à (Adapt·7c). Ses variantes sont quant à elles mises en place par les tactiques définies ci-après. Lorsqu'un code QR est analysé grâce à une caméra par la configuration, une page dédiée à l'information indiquée par celui-ci est mise en avant. L'utilisateur peut ensuite choisir une façon d'interpréter cette information en utilisant les boutons associés aux variantes de ce point de variation. La mise en place de celui-ci est par ailleurs effectuée par les tactiques (Adapt·7a) à (Adapt·7c).

Tactique (Adapt·11a) ▶ Audio, Image & Réalité Augmentée

Artefacts *Android* : Fichiers « java » d'activité et *layouts* correspondants
Artefacts *iOS* : Fichiers « swift » de *ViewController* et unique « storyboard » correspondant
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Sur *Android*, les boutons des variantes non sélectionnées sont retirés et les *layouts* non utilisés sont supprimés. Le code source correspondant est adapté en retirant les méthodes de gestion des boutons. Sur *iOS*, le « storyboard » correspondant est adapté pour ne laisser que les variantes désirées. Enfin, en ce qui concerne la variante *Réalité Augmentée*, le code source relatif à celle-ci, dédié à la superposition d'information sur la caméra, est également retiré. Les marqueurs permettant d'identifier la structure à « augmenter virtuellement » sont des spécificités propres à chaque logiciel commandé et sont implémentés au cas par cas.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Publicité Afin d'inclure des publicités sur les configurations demandées, il est possible de tirer parti du service multiplateforme *AdMob* offert par Google [44]. Il existe des SDK pour les plateformes *Android* et *iOS*.

Tactique (Adapt·12a) ▶ Publicité

Artefacts *Android* : Fichiers « AndroidManifest.xml », « build.gradle » au niveau du module et « java » d'activité principal
Artefacts *iOS* : Fichiers « Info.plist », « Podfile » et « swift » de *ViewController* principal
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Si la fonctionnalité n'est pas sélectionnée, les permissions correspondantes sont retirées et les fichiers d'activité principaux sont adaptés en conséquence.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Réseaux Sociaux Les plateformes *Android* et *iOS* permettent de partager facilement de l'information entre différentes applications à l'aide de *Sharesheets* [86, 99]. Ainsi, même si le client ne sélectionne pas certaines variantes, ce sont les applications installées sur les appareils physiques qui déterminent les possibilités de partage. Les configurations générées permettent de partager une *carte de visite* indiquant le passage de l'utilisateur par le site touristique. Ceci pourrait par exemple être étendu au partage d'informations contextuelles.

Tactique (Adapt·13a) ▶ Réseaux Sociaux & Variantes

Artefacts *Android* : Fichier « java » d'activité principal et *layout* correspondant
Artefacts *iOS* : Fichier « swift » de *ViewController* principal et « storyboard » correspondant
Tactique : *Patching par soustraction*
Sélection : compile
Activation : compile
Description : Si la fonctionnalité n'est pas sélectionnée, le code relatif à celle-ci est retiré des fichiers correspondants.
Motivation : Cette tactique permet de prévoir toutes les fonctionnalités nécessaires dans l'optique de vérifier leurs interactions et d'ensuite retirer celles inutiles.

Rencontre Si d'aventure cette caractéristique est sélectionnée, sa mise en place effectuée par les tactiques (Adapt·7a) à (Adapt·7c).

Evénement Une page Internet dédiée à la création d'événements sur mesure est mise à disposition du client sur le serveur. Une fois créés, ceux-ci sont communiqués aux appareils physiques associés. La mise en place de cette caractéristique sur les plateformes mobiles est effectuée par les tactiques (Adapt·7a) à (Adapt·7c).

Tactique (Multi·3a) ▶ Evénement, Attributs & Variantes

Artefacts : Application WebAssembly et page Internet associée.
Tactique : *DSL externe*
Sélection : design
Activation : runtime.run
Description : Une page Internet du serveur présente un créateur d'événement mis en place grâce à un DSL graphique, à l'instar de l'outil Figma [37]. Ce créateur permet au client de créer un événement de façon aisée, communiqué ensuite aux différents appareils selon un format défini - voir tactique (Adapt·2a).
Motivation : Un DSL permet de proposer un langage adapté pour résoudre la problématique de la création d'événements. Cette tactique s'impose d'elle-même dans la mesure où le client n'est sans doute pas un expert dans le domaine informatique.

Tactique (Adapt·14a) ▶ Evénement, Attributs & Variantes

Artefacts *Android* : Fichier « java » d'activité
Artefacts *iOS* : Fichier « swift » de *ViewController*
Tactique : *Surcharge de DSL*
Sélection : program
Activation : runtime.run
Description : En fonction des données envoyées par le serveur, les applications mobiles créent de manière dynamique les mises en page. L'interprétation des données s'effectue de manière similaire pour les plateformes *Android* et *iOS*, afin d'assurer un comportement unique à travers les différentes configurations.
Motivation : Le choix de cette tactique découle de la tactique (Multi·3a). Une surcharge permet d'assurer un comportement identique du DSL pour de multiples implémentations.

5.2.5 Récapitulatif des Tactiques

La table 5.1 propose un tableau récapitulatif des tactiques définies précédemment. Elle offre une vue concise du nombre de tactiques associées à un concept et met en avant la plateforme ciblée par chacune d'elles. On y retrouve ainsi deux ensembles distincts de tactiques à appliquer, visant par conséquent la plateforme *Android* ou *iOS*, où les tactiques multiplateformes et adaptatives sont dédoublées. On remarque également qu'il existe une multitude de tactiques adaptatives. Ceci est dû à l'architecture logicielle similaire proposée par les plateformes *Android* et *iOS* et l'utilisation de tactiques similaires. La figure 5.2 présente ainsi le diagramme de caractéristiques annoté de la ligne de produits logiciels *BetterSoft*.

Concept	**Tactiques**	
	Android	*iOS*
Application	(Adapt·1a)	
Plateforme Contrôle	(Multi·1a) (Adapt·2a)	
Appareil	(Adapt·3a)	
GSM, Tablette	(Android·1a)	(iOS·1a)
Caractéristiques Matérielles	(Adapt·4a)	
Android	(Android·2a)	
iOS		(iOS·2a)
Persona		
Handicap Visuel		
Presbytie, Hypermétropie	(Adapt·5a)	
	(Android·3a)	(iOS·3a) (iOS·3b)
Protanopie → Tritanomalie	(Adapt·6a) (Adapt·6b)	
	(Android·4a)	(iOS·4a)
Fonctionnalité	(Adapt·7a) (Adapt·7b) (Adapt·7c)	
Payement	(Adapt·8a) (Adapt·8b)	
Réservation		
Navigation	(Multi·2a) (Adapt·9a) (Adapt·9b) (Adapt·10a)	

	(Adapt·10b)
Interprétation Contextuelle	(Adapt·11a)
Publicité	(Adapt·12a)
Réseaux Sociaux	(Adapt·13a)
Rencontre	
Événement	(Multi·3a) (Adapt·14a)

TABLE 5.1 – Tableau récapitulatif des tactiques définies en section 5.2

Remarque(Rem·VII) ► Résolution des Tactiques par Plateforme

Le tableau 5.1 met en évidence les tactiques à mettre en œuvre en fonction de la plateforme. Il apparait ainsi que la majorité des tactiques définies sont déployables sur les plateformes *Android* et *iOS*, moyennant des adaptations propres à chacune, tandis que d'autres sont exclusives à une plateforme particulière. Par conséquent, il est nécessaire de disposer d'un mécanisme capable de différencier ces tactiques dans l'optique d'appliquer celles correspondant à la plateforme courante. Dans le cas des tactiques adaptatives, il s'agit de déterminer la bonne variante de celles-ci en sélectionnant les artefacts correspondant à la plateforme choisie. Dans le cas des tactiques exclusives, il s'agit uniquement d'opter pour celles définies pour la plateforme sélectionnée. Les solutions suivantes sont proposées pour résoudre cette problématique :

— Ajouter une annotation manuelle dépendante du contexte de l'usine à logiciels permettant un tri automatique des tactiques en différents ensembles. Dans le cas de l'entreprise *BetterSoft*, ces annotations pourraient être « Android » et « iOS ». En ce qui concerne les tactiques adaptatives, ces annotations se placeraient au niveau des différentes listes d'artefacts impactés. Ces annotations sont déjà effectives avec les tactiques présentées dans cette section. En effet, les étiquètes de couleur permettent de différencier les plateformes des tactiques exclusives. En outre, les différentes listes d'artefacts triés par plateforme permettent de créer les variantes des tactiques adaptatives ;

— Effectuer un tri automatique sur base des artefacts indiqués par chaque tactique. Si les artefacts sont clairement identifiés, il est possible d'inférer son appartenance à un ensemble dépendant du contexte de l'usine à logiciels. Dans le cas de l'entreprise *BetterSoft*, ces ensembles sont déterminés par les plateformes *Android* et *iOS*. À titre, d'exemple si le moteur d'inférence ou tout autre mécanisme capable de classification rencontre un fichier « storyboard », la tactique correspondante sera placée dans l'ensemble *iOS*. Ce mécanisme fonctionne tant pour les tactiques exclusives que pour les variantes des tactiques adaptatives.

Les solutions proposées sont en accord avec le panneau 3 du métamodèle présenté par la figure 2.7. En effet, plusieurs stratégies de composition des tactiques peuvent être mises en place et utilisées au besoin. Il suffirait ainsi de faire correspondre un ensemble à une stratégie particulière. La remarque (Rem·XI) aborde cet aspect de la méthodologie en détail.

Remarque(Rem·VIII) ► Multiplicité des Tactiques par Concept

Certains concepts regroupent plusieurs tactiques pour leur mise en œuvre. À l'intérieur d'un ensemble particulier, illustrés par la table 5.1, ces tactiques sont complémentaires et non pas exclusives. Ainsi, l'entièreté des tactiques définies pour un concept doivent être réalisées. Le besoin d'en définir plusieurs survient suite à la nécessité d'effectuer des changements de différentes natures pour un même concept, potentiellement durant des phases différentes. À titre d'exemple, les tactiques complémentaires (Adapt·5a), (iOS·3a) et (iOS·3b) sur la plateforme *iOS* agissent toutes trois sur les artefacts liés aux variantes *Presbytie* et *Hypermétropie*. Ces changements sont tous de différentes natures et il semble impossible de les unifier en une unique tactique.

La structure méthodologique *Bespoke* propose cependant un mécanisme capable de regrouper des changements de différentes natures. En effet, comme présenté par le panneau 5 du métamodèle détaillé par la figure 2.7, il est possible de développer des « procédures » réutilisables et d'en appeler plusieurs à la

suite. Chaque tactique est ultimement implémentée grâce à une telle procédure. Cependant, la principale raison d'être de celles-ci est la réutilisabilité. Ainsi, dans l'exemple du logiciel *JarreDeux*, une composition de procédures peut être employée pour factoriser les couples de tactiques qui apparaissent plus d'une fois. Dès lors, les multiples couples de tactiques « *Stratégie - Patching par soustraction* » associés à un même concept peuvent être refactorisés en utilisant ce mécanisme. Il s'agit des couples (Adapt·6a) et (Adapt·6b), (Adapt·8a) et (Adapt·8b), (Adapt·9a) et (Adapt·9b) et enfin (Adapt·10a) et (Adapt·10b). Chaque couple de tactiques devient alors une nouvelle tactique composite réutilisable à part entière. Les responsabilités de chaque tactique qui la compose en deviennent cependant moins claires.

À l'inverse, il semble impertinent de combiner les autres ensembles de tactiques proposés grâce à ces procédures. En effet, les changements apportés sont trop différents pour permettre une refactorisation, car ces tactiques composites ne seraient utilisés qu'une unique fois. L'exemple précédent des tactiques complémentaires (Adapt·5a), (iOS·3a) et (iOS·3b) sur la plateforme *iOS* illustre bien cette constatation. Il serait alors judicieux d'effectuer une division claire des responsabilités de chaque tactique et des changements qu'elles introduisent, division qui est brouillée dans le cas d'une tactique composite. Par conséquent, il serait bienvenu de transformer, dans le panneau 3 du métamodèle de la figure 2.7, la cardinalité *0-1* de l'association *for* du type d'entité *Choice* en *0-N*. Somme toute, ce changement est très subtil. Il s'agit de pouvoir, dans certains cas où la division des responsabilités est plus importante qu'un potentiel regroupement de modifications, présenter un ensemble de tactiques plutôt qu'un ensemble de procédures. Le produit final généré par l'usine à logiciels resterait identique et l'ingénieur aurait ainsi la possibilité de choisir entre un ensemble de tactiques, une tactique composite ou encore une combinaison des deux. La remarque (Rem·IX) propose une solution alternative à cette problématique.

Remarque(Rem·IX) ► Sémantique d'une Tactique

Si d'aventure la résolution proposée par la remarque (Rem·VIII) est malvenue et ne s'aligne pas avec la philosophie mise en avant par la structure méthodologique *Bespoke*, celle-ci peut être rejetée. Comme mentionné précédemment par ladite remarque, il suffirait alors d'appliquer une succession de procédures afin d'arriver au même résultat. Cependant, cela laisserait en suspens un problème d'ordre sémantique.

Une « tactique » est définie comme une technique d'implémentation de la variabilité. Le patching par soustraction et la compilation conditionnelle en sont des exemples. Chacune d'entre elles ne permet que des changements d'une nature bien précise. Le patching par soustraction enlève certains éléments d'un fichier particulier tandis que la compilation conditionnelle permet d'orienter la compilation du code source en fonction des arguments spécifiés. Le patching par soustraction est incapable de réaliser les changements mis en œuvre par la compilation conditionnelle et inversement. Par conséquent, une « tactique » telle que définie ne peut effectuer des changements que d'une unique nature spécifique. En limitant leur sélection à maximum une tactique, les changements possibles sont également limités à maximum un changement d'une seule nature. Il serait alors malvenu de réaliser une tactique non composite grâce à une succession de procédures opérant potentiellement des changements d'une nature différente de celle autorisée par la tactique qu'elles implémentent. Comme mentionné par la remarque (Rem·VIII), les tactiques complémentaires (Adapt·5a), (iOS·3a) et (iOS·3b) sur la plateforme *iOS* illustrent bien cette problématique. Ainsi, si la définition d'une tactique reste inchangée, il serait judicieux de renommer les éléments correspondants de façon sémantiquement appropriée. Par exemple :

— Une « tactique » deviendrait un « ensemble de tactiques ». Cet ensemble conserverait la cardinalité *0-1* d'une « tactique » ;

— Une « procédure » deviendrait une « tactique ». Cette « tactique » conserverait la cardinalité *0-N* d'une « procédure ».

Ce renommage permettrait ainsi d'éviter une certaine confusion sémantique tout en conservant les mêmes grandes étapes de la structure méthodologique. En effet, il serait toujours possible d'appliquer une succession de tactiques (anciennement procédures) pour arriver au même résultat que celui mis en avant par le changement de cardinalité de la remarque (Rem·VIII), tout en respectant la nature unique des changements liés à une tactique. La multiplicité des natures serait alors autorisée par l'union des tactiques au sein d'un ensemble. D'autres changements au métamodèle de la figure 2.7 devront peut-être être introduits pour accommoder ce renommage.

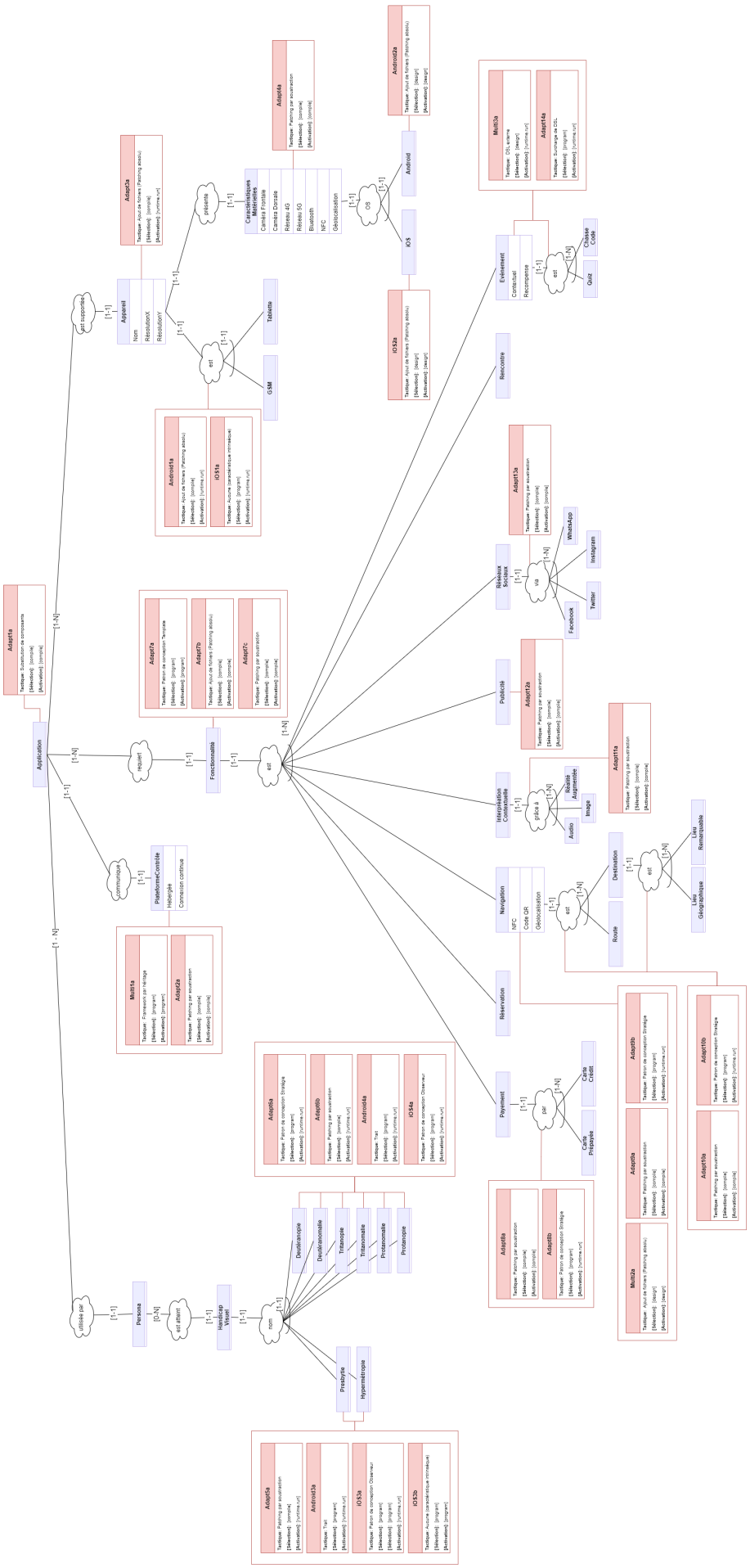


FIGURE 5.2 – Diagramme de caractéristiques annoté. Les concepts repris dans le tableau 4.1 du chapitre 4 s'appliquent également. Si le diagramme s'avère être illisible, la page référencée par le code QR de la figure B.4 disponible en annexe en présente une version WEB interactive.

5.3 Répertoire des Artefacts

Les artefacts sont les composants logiciels à développer pour mettre en œuvre les tactiques d'implémentation. Ceux-ci sont ainsi élaborés en fonction des tactiques définies et sont manipulés par celles-ci pour assembler une configuration précise. Ces artefacts correspondent à ceux spécifiés par le panneau 4 de la figure 2.7 et sont représentés de manière graphique selon une arborescence de répertoires.

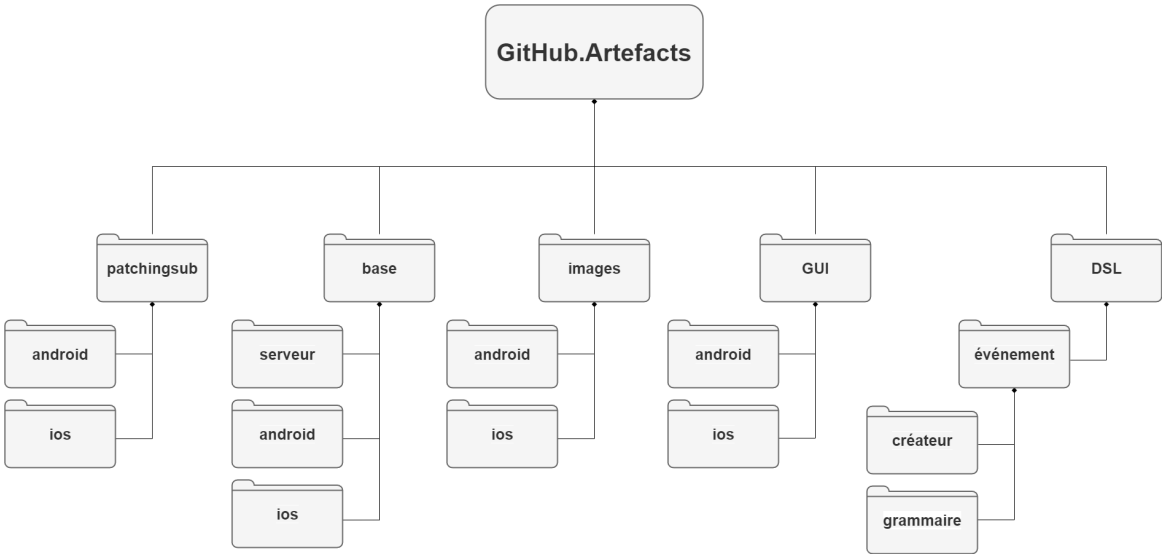


FIGURE 5.3 – Artefacts repris dans un répertoire *GitHub*

5.3.1 Définition des Artefacts

Grâce à la définition des tactiques d'implémentation effectuée précédemment, il est possible de formaliser les artefacts dont celles-ci ont besoin afin d'être mises en œuvre. La figure 5.3 présente un répertoire *GitHub* contenant les différents artefacts triés par nature. Le répertoire *patchingsub* contient les fichiers de patching nécessaires pour transformer les fichiers cibles. Le répertoire *base* définit les architectures communes du serveur et des configurations propres à chaque plateforme. Le répertoire *images* reprend un ensemble d'images déclinées en plusieurs variantes dont les résolutions diffèrent. Ces variantes sont nommées selon les qualificatifs prescrits par chaque plateforme. Il contient également les logos des configurations. Le répertoire *GUI* propose les différentes mises en page à ajouter en fonction des fonctionnalités sélectionnées et de la taille des appareils physiques. Enfin, le répertoire *DSL.événement* inclut l'outil de création d'événements à compiler vers *WebAssembly* et la grammaire liée à ceux-ci.

Répertoire	**Tactiques liées**
patchingsub.android	(Adapt·2a), (Adapt·4a), (Adapt·5a), (Adapt·6b), (Adapt·7c), (Adapt·8a), (Adapt·9a), (Adapt·10a), (Adapt·11a), (Adapt·12a), (Adapt·13a)
patchingsub.ios	(Adapt·2a), (Adapt·4a), (Adapt·5a), (Adapt·6b), (Adapt·7c), (Adapt·8a), (Adapt·9a), (Adapt·10a), (Adapt·11a), (Adapt·12a), (Adapt·13a)
base.serveur	(Multi·1a), (Multi·3a)
base.android	(Android·2a), (Android·3a), (Adapt·6a), (Android·4a), (Adapt·7a), (Adapt·8b), (Adapt·9b), (Adapt·10b), (Adapt·14a)
base.ios	(iOS·1a), (iOS·2a), (iOS·3a), (iOS·3b), (Adapt·6a), (iOS·4a), (Adapt·7a), (Adapt·8b), (Adapt·9b), (Adapt·10b), (Adapt·14a)
images.android	(Adapt·1a), (Adapt·3a)
images.ios	(Adapt·1a), (Adapt·3a)

GUI.android	(Android·1a), (Adapt·7b)
GUI.ios	(Adapt·7b)
DSL.événement.créateur	(Multi·3a)
DSL.événement.grammaire	(Multi·3a), (Adapt·14a)

TABLE 5.2 – Tactiques liées à chaque artefact

La table 5.2 reprend les tactiques liées à chacun des répertoires. La tactique (Multi·2a) n'apparait pas dans celle-ci car elle désigne un artefact trop spécifique pour être prévu dans l'usine à logiciels. Par conséquent, les cartes des différents sites sont ajoutées lors de la réalisation d'une configuration. À l'inverse, certaines tactiques apparaissent plusieurs fois car elles agissent sur de multiples artefacts. La table 5.2 permet ainsi de visualiser l'impact de chacune d'entre elles sur les artefacts à prévoir. Certains impacts ne sont cependant pas mis en évidence. En effet, l'entièreté des tactiques liées aux répertoires *patchingsub.android* et *patchingsub.ios* agissent sur d'autres artefacts de manière insidieuse. Cependant, les impacts précis de ces tactiques sont indiqués dans leurs définitions respectives.

5.3.2 Adaptation de l'Architecture Commune

Remarque(Rem·X) ► Adaptation de l'Architecture Commune

La table 5.2 met en évidence les tactiques qui agissent sur l'architecture commune définie dans la section 5.1 et argumentée dans la remarque (Rem·VI), contenue dans les répertoires *base.android* et *base.ios*. Par conséquent, ladite architecture doit être adaptée en lui appliquant les tactiques qui lui sont liées. Dans le cas contraire, les tactiques liées aux autres répertoires ne peuvent pas être appliquées car l'architecture commune ne respecte pas les préconditions nécessaires à l'application de celles-ci. L'adaptation de l'architecture commune n'est pas référencée par la structure méthodologique *Bespoke* mais semble nécessaire pour assurer une conception optimale de l'usine à logiciels.

Étant donné l'utilisation intensive de la tactique *patching par soustraction*, l'architecture commune se doit de présenter dans sa forme canonique la quasi-totalité des points de variation, de leurs variantes et de leurs attributs. Les concepts non sélectionnés lors de l'élaboration d'une nouvelle configuration seront par conséquent retirés de l'architecture commune conformément aux altérations prescrites par les fichiers de *patching* correspondants.

5.4 Stratégies de l'usine à logiciels *BetterSoft*

Remarque(Rem·XI) ► Multiplicité des Stratégies

La structure méthodologique *Bespoke* permet l'élaboration de plusieurs stratégies d'implémentation des tactiques. Cette capacité est illustrée par le panneau 3 du métamodèle présenté par la figure 2.7 et plus précisément par le type d'entité *Strategy*. Ce mécanisme peut être mis à contribution dans l'optique d'orienter l'implémentation d'une configuration en fonction des choix réalisés par le client. En effet, en fonction des caractéristiques sélectionnées au sein d'une configuration, une stratégie particulière est employée, celle-ci permettant par conséquent de répondre aux besoins émis et de réaliser une séparation claire des modifications à apporter. Ainsi, prévoir plusieurs stratégies différentes permet d'exprimer plus aisément la variabilité au sein d'une usine à logiciels. Dans le cas de l'usine à logiciels *BetterSoft*, deux stratégies peuvent ainsi être établies : *Android* et *iOS*. Celles-ci sont déployées en fonction du système d'exploitation éponyme demandé par une configuration.

Comme énoncé précédemment dans la remarque (Rem·XI), deux stratégies d'implémentation peuvent être définies pour l'usine à logiciels *BetterSoft*. Cependant, dans la suite, seule la stratégie *Android* sera abordée. En effet, au vu des similitudes exhibées par les tactiques définies pour les plateformes *Android* et *iOS*, la définition de la stratégie *iOS* ne semble apporter que peu de valeur ajoutée. En outre, seules certaines des caractéristiques sélectionnées par les configurations du logiciel *JarreDeux* ainsi que quelques autres choisies arbitrairement afin d'illustrer les tactiques associées sont développées dans la suite du document. Ainsi, l'ensemble des tactiques implémentées est présenté par la table 5.3.

Type de tactique	Artefacts	Tactiques effectives liées
Trait	base.android	(Android·3a), (Android·4a)
Patron de conception *Template*	base.android	(Adapt·7a)
Patron de conception *Stratégie*	base.android	(Adapt·6a), (Adapt·9b), (Adapt·10b)
Patching par soustraction	patchingsub.android	(Adapt·4a), (Adapt·5a), (Adapt·6b), (Adapt·7c), (Adapt·9a), (Adapt·10a), (Adapt·11a)
Ajout de fichiers (patching absolu)	images.android	(Adapt·3a)
	GUI.android	(Android·1a)
	base.android	(Android·2a)
Substitution de composants	images.android	(Adapt·1a)

TABLE 5.3 – Tactiques effectives de l'usine à logiciels *BetterSoft*

5.4.1 Conception d'une Tactique

La structure méthodologique *Bespoke* définit un langage théorique appelé *TacticType* permettant de formaliser la conception d'un type de tactique [31]. Aucune implémentation n'étant disponible à l'heure actuelle, quelques libertés sont prises quant à celle-ci au sein de l'usine à logiciels *BetterSoft*. Toute tactique définie dans la table 5.3 ne doit cependant pas faire l'objet d'une telle formalisation. En effet, les tactiques *Trait* et *Patrons de conception Template & Stratégie* agissent sur l'architecture de base d'une configuration et doivent ainsi être implémentées au niveau du code source dans le langage de programmation prescrit. À l'inverse, les tactiques *Patching absolu & par soustraction* et *Substitution de composants* doivent être implémentées selon le langage théorique *TacticType*. Ces implémentations sont réalisées au moyen du langage de programmation *Python* et sont détaillées dans la suite.

Ajout de Ressources

Le listing B.1 disponible en annexe détaille l'implémentation d'un type de tactique abstrait permettant d'ajouter des ressources à un répertoire indiqué. Une ressource est représentée par la classe de données « Res » contenant les informations nécessaires à l'ajout de fichiers, respectivement les chemins d'accès source et destination. Une telle ressource est manipulée par la méthode statique « AddRes.apply() » dans l'optique de compléter l'opération d'ajout de ressources telle que prescrite. Cette méthode définit trois paramètres différents permettant d'orienter la manipulation d'une ressource :

1. « *res : Res » représente une séquence non vide de ressources à ajouter. Celles-ci doivent être directement passées en argument et être séparées par des virgules ;
2. « order : Order » est un paramètre d'ordonnancement indiquant l'ordre d'application des ressources passées en argument. Cet ordre peut être séquentiel ou parallèle et celui-ci est déterminé par l'utilisateur lui-même. Un ordre d'application parallèle implique la création d'un ensemble de fils d'exécution en arrière-plan ;
3. « substitute : bool » est un paramètre optionnel supplémentaire offrant une plus grande versatilité à ce type de tactique abstrait en autorisant l'écrasement de fichiers présents au préalable. Ceci permet d'employer ce type de tactique afin d'implémenter l'ajout de fichiers ainsi que la substitution de composants.

Le listing 5.1 présente un exemple d'utilisation de ce type de tactique abstrait dans le cadre de la substitution des logos définie par la tactique (Adapt·1a). Une liste déballée de ressources indiquant l'emplacement des différentes versions d'un même logo sont utilisées comme arguments et doivent être appliquées de manière parallèle. Une telle parallélisation est possible car les ressources n'interfèrent pas entre elles.

```
1  AddRes.apply(
2      *[Res(src=os.path.join(self.artefacts_root_folder, f"images/android/mipmap-{dpi}/"),
3            dst=os.path.join(self.release_root_folder, f"app/src/main/res/mipmap-{dpi}/"))
4        for dpi in (DPI.MDPI, DPI.HDPI, DPI.XHDPI, DPI.XXHDPI, DPI.XXXHDPI, DPI.ANYDPI_V26)],
5      order=Order.PARALLEL,
6      substitute=True
7  )
```

Listing 5.1 – Substitution des logos pour la plateforme *Android*, telle que définie par la tactique (Adapt·1a)

Patching par Soustraction

Le listing B.2 également disponible en annexe détaille l'implémentation du type de tactique abstrait « Patching par soustraction ». Un fichier de patching par soustraction est représenté par la classe de données « Patch » composée des informations suivantes :

1. « yaml_src: str | Path » indique l'emplacement d'un fichier *YAML* contenant les informations nécessaires à l'application du patch. Le format *YAML* est intéressant à employer car il permet une représentation déclarative des opérations à effectuer ;
2. « base_target: str | Path » indique le chemin d'accès vers le dossier source de la nouvelle configuration ;
3. « skip_qualifier : Iterable[str] » est une structure de données itérable spécifiant les éléments d'un patch à ne pas appliquer, en fonction des caractéristiques choisies par le client dans une configuration particulière.

```
tactic:
  type: substraction
  start: <Debut de bloc de tactique>
  end: <Fin de bloc de tactique>
  sepsign: <Separateur>
  variants order: <Ordre>
  variants:

    - variant:
      qualifier: <Qualificatif unique>
      files order: <Ordre>
      files:
        - file:
          mode: <Mode>
          in: <Chemin vers le fichier source>
        - file:
          ...

    - variant:
      ...
```

Listing 5.2 – Structure d'un fichier *YAML* de patching par soustraction

Ainsi, un fichier de patching par soustraction peut être manipulé par la méthode statique « PatchingSub.apply() » afin d'appliquer les soustractions prescrites. La composition d'un tel fichier est régie par une structure *YAML* précise qu'il est impératif de respecter afin que les opérations de soustraction se complètent sans encombre. Celle-ci est détaillée par le listing 5.2, tandis que le listing B.3 disponible en annexe propose quant à lui un exemple exhaustif des possibilités offertes par une telle structure. Les différentes clefs de celle-ci sont explicitées ci-après :

— Plusieurs clefs « variant » peuvent être indiquées et plusieurs clefs « file » peuvent l'être également pour chaque variante. Celles-ci permettent d'énumérer l'entièreté des modifications apportées par les différentes variantes d'un point de variation au sein d'un même fichier de patching. À chacun de ces concepts est associé un paramètre d'ordonnancement à appliquer, dénoté par les clefs « variants order » et « files order », respectivement. Celui-ci peut être séquentiel ou parallèle, respectivement dénoté par la valeur « sequential » ou « parallel ». Un ordre d'application parallèle implique la création d'un ensemble de fils d'exécution en arrière-plan ;

— Les clefs « start », « end », « sepsign » et « qualifier » permettent de créer des balises à rechercher dans un fichier source particulier, indiquant par conséquent le bloc de code à soustraire. Chaque variante « variant » doit définir un qualificatif « qualifier » unique. À titre d'exemple, le listing 5.4 présente les balises reconstruites à rechercher sur base des valeurs arbitraires données par le listing 5.3. Le code ainsi délimité par les balises sera ensuite soustrait si d'aventure la valeur de la clef « qualifier » ne figure pas dans la séquence de qualificatifs à ne pas appliquer « skip_qualifier : Iterable[str] » d'un « Patch », défini auparavant ;

```
start = "@TACTIC('Adapt5a')"
end = "@END_TACTIC"
sepsign = " = "
qualifier = HYPERMETROPIE
```

Listing 5.3 – Exemple de valeurs

```
@TACTIC('Adapt5a') = HYPERMETROPIE
...
...
@END_TACTIC
```

Listing 5.4 – Balises à rechercher

— La clef « mode » indique la manière de modifier un fichier particulier. Celle-ci peut soit être l'altération, dénotée par la valeur « alter », soit la suppression, alors dénotée par la valeur « delete ». Dans le second cas, le fichier complet indiqué est simplement supprimé.

Le listing 5.5 présente un exemple d'utilisation de ce type de tactique abstrait dans le cadre de la soustraction définie par la tactique (Adapt·6b) des types de daltonisme. Ceux sélectionnés sont indiqués par la figure 4.7 du chapitre 4 présentant les configurations du logiciel *JarreDeux*. Le fichier de patching par soustraction est quant à lui mis en avant par le listing B.3 disponible en annexe.

```
1     PatchingSub.apply(
2         Patch(yaml_src=os.path.join(self.artefacts_root_folder, f"patchingsub/android/adapt6b.yaml"),
3               base_target=self.release_root_folder,
4               skip_qualifier=('PROTANOPIE', 'PROTANOMALIE', 'DEUTERANOPIE', 'DEUTERANOMALIE'))
5     )
```

Listing 5.5 – Soustraction des types de daltonisme non sélectionnés pour la plateforme *Android*, telle que définie par la tactique (Adapt·6b)

Remarque(Rem·XII) ▶ Implémentation Déclarative d'un Type de Tactique

L'implémentation proposée pour le type de tactique abstrait « Patching par soustraction » met l'accent sur la déclaration des opérations à effectuer pour une soustraction particulière. À ce titre, le format de représentation des données *YAML* est un candidat intéressant, car il propose une syntaxe lisible, simple et est suffisamment flexible pour permettre d'éviter l'implémentation ad hoc d'un DSL dédié, tel qu'initialement proposé par le langage TacticType. À ce titre, mais également par manque de temps, aucun DSL n'est implémenté dans ce document. En séparant ainsi ces déclarations de l'implémentation de leur moteur d'interprétation, ici développé en *Python*, la philosophie agnostique au niveau technologique de la structure méthodologique *Bespoke* est préservée. En effet, le moteur d'interprétation pourrait faire l'objet d'une réécriture dans un autre langage de programmation sans toutefois entraver le développement et l'exécution de l'usine à logiciels. Celle-ci pourrait en outre s'effectuer à n'importe quel moment du cycle de vie de l'usine, tout en conservant ainsi les opérations à appliquer au sein de fichiers déclaratifs séparés.

Par ailleurs, l'implémentation du type de tactique abstrait « Ajout de ressources » pourrait également être remaniée afin d'inclure l'aspect déclaratif susmentionné. Il suffirait pour cela d'extraire les informations contenues dans la classe de données « Res » vers un fichier *YAML*. Celui-ci présenterait dès lors les mêmes informations, ainsi que l'ordonnancement des ressources ainsi indiquées. Un répertoire dédié devrait également être créé dans le répertoire source des artefacts à employer. Somme toute, la structure d'un tel fichier d'ajout de ressources serait fortement similaire à celle d'un fichier de patching par soustraction. Cependant, étant donné la nature simpliste des modifications indiquées par la classe de données « Res », cette transformation ne semble apporter que peu de valeur ajoutée à la conception de l'usine à logiciels *BetterSoft* et n'est à ce titre pas appliquée.

5.4.2 Conception d'une Stratégie

Les types de tactique abstraits présentés dans la section 5.4.1 précédente permettent ainsi d'implémenter de façon abstraite les tactiques définies dans la section 5.2. Ces implémentations abstraites s'organisent au sein d'une stratégie particulière. Comme énoncé précédemment, seule la stratégie *Android* est concrétisée dans ce document. Afin d'implémenter une tactique spécifique, il est nécessaire de prendre connaissance des artefacts impactés par celle-ci. Il suffit dès lors de parcourir à nouveau la section 5.2 qui met les impacts de chaque tactique en évidence et de faire correspondre ceux-ci avec la table 5.2 des artefacts, qui indique leur emplacement au sein d'une arborescence de fichiers.

Remarque(Rem·XIII) ▶ Caractéristique Atomique d'une Tactique

Les modifications apportées par les différentes tactiques sont d'une nature unique. Ceci est le résultat d'une préoccupation émise par la remarque (Rem·VIII). Celle-ci argumente le choix de la définition d'un ensemble de tactiques pour une unique caractéristique d'une configuration. Chaque tactique ainsi définie n'effectue dès lors que des changements d'une nature unique. En outre, en accord avec l'hypothèse (H4), les liens d'ingérence entre tactiques ne sont pas explorés et ont ainsi été évités lors de la définition des caractéristiques. Enfin, comme exemplifié par le fichier de patching par soustraction présenté par le listing B.3 disponible en annexe, certains fichiers sont impactés par les différentes variantes d'un point de variation. Certains d'entre eux le sont également par d'autres points de variation, à l'instar du fichier de permissions « AndroidManifest.xml » de la plateforme *Android*, référencés par les tactiques (Adapt·4a), (Adapt·7c) et (Adapt·12a). Par conséquent, grâce à leur nature unique des changements apportés, l'ab-

sence de liens d'ingérence entre les caractéristiques et la probabilité non nulle de partager un artefact à modifier, les tactiques conçues à l'aide d'un type de tactique abstrait dans la section 5.4.1 sont atomiques. En d'autres termes, chacune d'entre elles peut ainsi s'appliquer sans interruption afin d'assurer la complétion des modifications à apporter. Cette caractéristique permet de traiter l'entièreté des modifications apportées par une tactique d'un seul bloc et d'opérer une division claire des responsabilités de chacune au sein du code source. Certaines tactiques interférant entre elles, non représentées dans ce document, ne présentent pas cette caractéristique. Leurs modifications respectives ne peuvent alors pas être effectuées d'un seul bloc et doivent ainsi être décomposées. Cette décomposition est abordée dans la remarque (Rem·XIV).

Cette caractéristique atomique est parfaitement illustrée par les tactiques (Adapt·4a), (Adapt·7c) et (Adapt·12a) qui agissent toutes sur le fichier de permissions « AndroidManifest.xml » de la plateforme *Android*. Bien que toutes modifient le fichier susmentionné, chacune est confinée à certains blocs du code source avec lesquels les autres n'interfèrent pas. Celles-ci sont par conséquent atomiques et peuvent ainsi s'exécuter sans interruption les unes à la suite des autres. L'ordonnancement précis des tactiques est discuté dans le chapitre 6.

Il est possible d'inférer un résultat pratique de la remarque (Rem·XIII). En effet, étant donné que chaque tactique de la section 5.4.1 est atomique, il suffit simplement de définir une unique fonction pour chacune d'entre elles et d'appliquer ces fonctions selon un ordonnancement précis. Celui-ci est par ailleurs établi dans le chapitre 6. Le listing 5.6 détaille l'implémentation abstraite de la tactique (Adapt·3a) au sein d'une méthode éponyme dédiée. La variable « drawable_suffixes » regroupe les résolutions d'images à ajouter à l'application sur base des informations données par la configuration du logiciel *JarreDeux* présentée par la figure 4.7. Cet exemple est représentatif des autres tactiques implémentées dans la stratégie *Android*, chacune des méthodes éponymes appartenant ainsi à la même classe, celle-ci représentant la stratégie associée. Dès lors qu'un ordonnancement des tactiques est établi, il suffit d'invoquer la méthode « strategy.apply() » sur une instance de la stratégie *Android* et l'application sera ensuite modifiée de manière automatique. Cette automatisation reste en accord avec l'hypothèse (H3), qui précise cependant que l'objectif de ce document n'est pas une automatisation exhaustive de la structure méthodologique *Bespoke*.

```
1     def _adapt3a(self) -> None:
2         """Implements the eponymous tactic
3         """
4
5         # Retrieving the resolutions from the configuration
6         drawable_suffixes = [
7             ...
8         ]
9
10        # Creating a Res for every appropriate drawable found
11        AddRes.apply(
12            *[Res(src=os.path.join(self.artefacts_root_folder, f"images/android/drawable{dpi}/"),
13                  dst=os.path.join(self.release_root_folder, f"app/src/main/res/drawable{dpi}/"))
14              for dpi in drawable_suffixes],
15            order=Order.PARALLEL if len(drawable_suffixes) > 1 else Order.SEQUENTIAL,
16        )
```

Listing 5.6 – Implémentation abstraite de la tactique (Adapt·3a) pour la plateforme *Android* au sein de la stratégie associée

5.5 Conclusion

Ce chapitre s'est intéressé en profondeur à la conception de tactiques d'implémentation afin de réaliser les points de variation, leurs attributs pertinents et leurs variantes. Il détaille les différents choix réalisés dans cette optique et argumente l'utilisation d'un ensemble de tactiques pour réaliser une caractéristique donnée. La sémantique d'une tactique est ainsi remise en question. Une explicitation de l'architecture commune est également proposée comme étape préalable à la définition des tactiques afin de formaliser les différents artefacts préexistants et de faciliter la compréhension de celles-ci. Cette explicitation est complétée par une adaptation de l'architecture commune et la création d'un répertoire d'artefacts à manipuler par les tactiques, survenant suite aux décisions prises lors de la définition de celles-ci. Les concepts ainsi formalisés sont concrétisés grâce à une conception et une implémentation abstraite de tactiques atomiques au sein d'une stratégie *Android* dans l'optique de créer le logiciel demandé par l'entreprise *JarreDin*.

Chapitre 6

Implémentation du Domaine

Le chapitre 5 a permis de formaliser la conception de l'usine à logiciels en définissant plusieurs ensembles de tactiques dont l'implémentation de certaines a été concrétisée dans la section 5.4. Cependant, l'ordre d'application de ces tactiques n'y a pas été défini. Ainsi, ce chapitre aborde l'élaboration d'un plan d'exécution visant à ordonner entre elles et de manière optimale les tactiques reprises dans la table 5.1. Cette étape correspond à la sixième et ultime étape du plan *SPL Engineer* de la structure méthodologique *Bespoke*. Ce chapitre détaille également l'implémentation concrète de l'usine à logiciels capable de produire de manière semi-automatique la configuration *Android* du logiciel *JarreDeux*, en présentant les divers éléments qui la constituent. Ce chapitre se veut ainsi être le point d'orgue et la clôture de la réalisation de l'usine à logiciels *BetterSoft*.

6.1 Ordonnancement des Tactiques

Le tableau récapitulatif 5.1 du chapitre 5 a présenté deux ensembles de tactiques à mettre en œuvre pour modifier à l'aide d'artefacts l'architecture commune des logiciels à produire. À chacun de ces ensembles est associée une stratégie d'implémentation des tactiques, abordées dans la section 5.4 du chapitre 5. Le panneau 6 du métamodèle de la figure 2.7 fait ainsi correspondre à chaque stratégie un ordre d'application des tactiques qui la composent. La section 5.4 du chapitre 5 n'ayant abordé que la stratégie relative à la plateforme *Android*, seul le plan d'exécution relatif à celle-ci est établi dans ce document. Ainsi, les tactiques reprises dans la stratégie *Android* doivent être ordonnées d'une manière telle que l'usine à logiciels soit capable de produire sans encombre le résultat escompté par le client. En observant les différentes phases de sélection définies pour chacune d'elles, il apparaît que seulement trois phases uniques du cycle de vie d'un logiciel sont évoquées : les phases de design, de programmation et de compilation. Cet ensemble restreint de phases est la conséquence de l'utilisation de tactiques identiques pour plusieurs points de variation, attributs et variantes différents. Dès lors, bien qu'il soit évident qu'il faille mettre en œuvre les tactiques de design avant celles de programmation et ces dernières avant celles de compilation, il est plus compliqué de déterminer l'ordre d'application des tactiques au sein d'une même phase. Cependant, il est possible de le déterminer en observant les artefacts impactés par ces tactiques et la nature des changements qu'elles apportent.

— Si deux tactiques appartenant à une même phase de sélection agissent sur des artefacts différents, alors leur ordre n'a pas d'importance. Les tactiques (Android·1a) et (Adapt·3a) illustrent parfaitement ce cas de figure ;

— Dans le cas contraire, il est impératif de déterminer la nature des changements effectués par chacune des tactiques :

1. Si, au sein d'un même artefact, deux tactiques agissent sur la même fonctionnalité et que leurs changements respectifs doivent survenir dans un ordre précis indiqué par les tactiques, alors leur ordre de mise en œuvre est évident. À titre d'exemple, la tactique (Adapt·11a) agit sur des fichiers de mise en page créés par la tactique (Android·1a) et doit par conséquent être appliquée après cette dernière ;
2. Si, au sein d'un même artefact, deux tactiques agissent sur des fonctionnalités différentes, il est impératif de prêter une attention particulière aux endroits modifiés. Cependant, si ces fonctionnalités sont correctement découplées, l'ordre d'application des tactiques ne devrait pas avoir d'impact. Ce cas de figure est illustré par les tactiques (Adapt·4a) et (Adapt·7c) qui agissent toutes deux sur le fichier de permissions « AndroidManifest.xml » pour des raisons différentes, mais dont les changements n'interfèrent pas entre eux ;

3. Si, au sein d'un même artefact, deux tactiques agissent sur la même fonctionnalité alors que leurs changements respectifs sont de natures différentes, il existe un lien d'ingérence entre ces tactiques. Celui-ci peut être résolu en décomposant les tactiques selon leurs opérations *Create-Read-Update-Delete* et en appliquant celles-ci de manière enchevêtrée [12]. La remarque (Rem·XIV) aborde plus en détail les problèmes liés à cette décomposition. Cependant, comme énoncé par la remarque (Rem·XIII), les changements apportés par les tactiques définies dans la section 5.2 sont d'une nature unique et celles-ci sont en outre atomiques. Par conséquent, ce cas de figure ne se présente pas dans le cadre de l'usine à logiciels *BetterSoft*. De plus, les liens d'ingérence ne sont pas traités dans ce document, comme indiqué par l'hypothèse (H4).

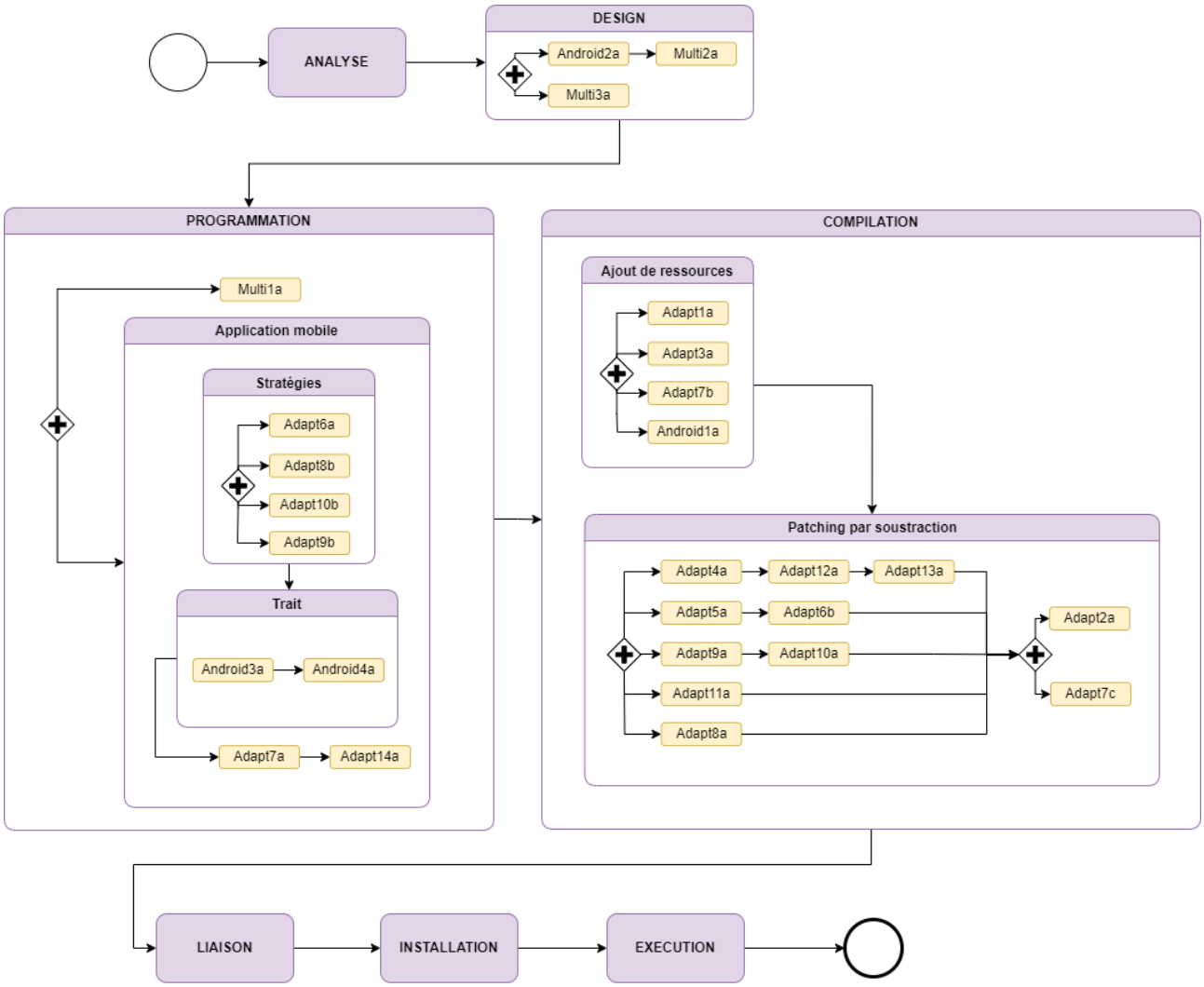


FIGURE 6.1 – Plan d'exécution de l'usine à logiciels *BetterSoft*, contenant uniquement l'ensemble des tactiques appartenant à la stratégie *Android* - notation BPMN simplifiée

La figure 6.1 présente le plan d'exécution établi dans l'optique d'ordonner les tactiques appartenant à la stratégie *Android* uniquement. Celui-ci est réalisé grâce à la notation BPMN et présente les tactiques triées et ordonnées selon les différentes phases du cycle de vie d'un produit ainsi que les artefacts impactés par les tactiques. Certaines d'entre elles, à l'instar de la tactique (Adapt·13a) peuvent être placées à différents endroits sans que cela ne pose un quelconque problème. Plusieurs conteneurs sont par ailleurs ajoutés afin de faciliter la compréhension du plan d'exécution en groupant certaines tactiques dont la sémantique est proche. Les constructions « + » inscrites dans un losange représentent une activation parallèle de tous les chemins liés. Si d'aventure les tactiques relatives à la stratégie *iOS* avaient également fait l'objet d'une définition du plan d'exécution, la figure 6.1 présenterait alors à certains endroits une construction « x » inscrite dans un losange, symbolisant l'activation exclusive d'un seul des chemins liés. Par ailleurs, le plan d'exécution ne présente pas de constructions « o » inscrites dans un losange, représentant quant à elles la possibilité d'activer un nombre arbitraire de chemins liés. Cela est dû à la conception du type de tactique « Patching par soustraction » telle que réalisée dans la section 5.4.1 du chapitre 5. En effet, les variantes d'un point de variation non sélectionnées

dans une configuration seront incluses dans les qualificatifs à ne pas retirer, tandis que celles non sélectionnées seront quant à elle bel et bien incluses. Somme toute, le plan d'exécution présenté par la figure 6.1 permet de définir un ordonnancement exploitable des tactiques dans une implémentation concrète de l'usine à logiciels *BetterSoft*, concrétisant dès lors la dernière partie manquante de celle-ci.

Remarque(Rem·XIV) ▶ Décomposition d'une Tactique

Les tactiques définies dans la section 5.2 sont atomiques. Ainsi, le plan d'exécution qui en résulte ne contient que des constructions simples et détaille chaque tactique une seule et unique fois. Si d'aventure certaines tactiques avaient dû être décomposées selon leurs opérations *Create-Read-Update-Delete* [12], la définition du plan d'exécution, bien que plus complexe, ne serait toutefois pas mise à mal. En effet, il suffirait simplement d'inclure ces décompositions aux endroits adéquats. Cependant, cette modification n'est pas permise par les panneaux 3 et 6 du métamodèle de la figure 2.7, qui n'autorise d'ordonner que les tactiques elles-mêmes. Plusieurs solutions sont possibles pour pallier ce problème et certaines sont discutées ci-dessous :

— Déplacer l'ordonnancement au niveau des procédures. En attribuant un ordre à chaque procédure définie dans l'usine à logiciels, dont sont composées les différentes tactiques, les multiples décompositions d'une tactique pourraient être assemblées avec les décompositions d'autres tactiques. Cela complexifierait cependant la maintenance de l'usine à logiciels car il faudrait maintenir à jour cet ordonnancement global des procédures au fur et à mesure des modifications apportées à l'usine ;

— Ajouter un type d'entité « Tactic Fragment » dont est composé une tactique particulière et déplacer l'ordonnancement vers celui-ci. Cette solution fait écho à l'argumentation bâtie par la remarque (Rem·VIII) qui détaille le choix d'utilisation de plusieurs tactiques par concept. Décomposer une tactique selon ses opérations *Create-Read-Update-Delete* nécessiterait cependant une modification complexe et en profondeur du métamodèle de la figure 2.7.

6.2 Implémentation de l'Usine à Logiciels *BetterSoft*

Cette section détaille l'implémentation concrète de l'usine à logiciels *BetterSoft*. Celle-ci est réalisée en *Python* et est uniquement disponible en utilisant une interface en ligne de commande. Les arguments qui la composent sont explicités dans les différentes sous-sections suivantes. Cette implémentation fait usage des types de tactique et stratégies conçus dans la section 5.4 du chapitre 5. Dans l'optique de proposer une automatisation de la création d'un logiciel, les éléments techniques qui composent l'usine à logiciels et permettent une telle automatisation sont présentés de manière brève. L'implémentation réalisée est ensuite illustrée grâce à la génération du logiciel *JarreDeux* sur la plateforme *Android*. La figure 6.2 détaille le processus dans son ensemble de manière graphique.

6.2.1 Composants Techniques

Afin d'offrir une certaine automatisation localisée du processus de génération d'un logiciel, l'hypothèse selon laquelle une configuration *XML* existe au préalable est formulée. Celle-ci serait le produit de la conversion d'une configuration telle que proposée par le listing A.2 écrit en *Groovy* en un format de représentation des données exploitable par ledit générateur de logiciels. Afin de rendre celui-ci plus souple et modulaire, un fichier de mapping *TOML* [97] est adjoint à la configuration *XML*, permettant de faire correspondre les balises *XML* à des artefacts précis du code source.

Configuration *XML*

Tel qu'énoncé précédemment, l'hypothèse selon laquelle une configuration valide sous le format de représentation des données *XML* existe est formulée. Le listing C.2 disponible en annexe en présente un exemple concret. Les données contenues dans ce fichier sont directement exploitables par la plupart des langages de programmation. Ainsi, les informations mises en avant par cette configuration sont utilisées pour compléter certains paramètres nécessaires à l'application des tactiques. Pour ce faire, un module contenant des fonctions de résolution de balises *XML* est implémenté afin de récupérer les informations correspondantes. Les données ainsi récupérées permettent d'orienter le processus de génération d'un logiciel particulier. Il est important de noter que l'implémentation dudit module est dépendante d'une structure *XML* précise qu'il est impératif de respecter. Celle-ci est illustrée par le listing C.2. L'argument permettant d'indiquer l'emplacement de la configuration *XML* à exploiter est le suivant, où « <chemin> » doit être remplacé par le chemin d'accès correspondant :

```
--xml-configuration <chemin>
```

Mapping *TOML*

Un fichier au format de représentation des données *TOML* doit également être indiqué au générateur de logiciels afin de faire le pont entre les balises *XML* et leurs attributs et les artefacts du code source à manipuler. Ce fichier de correspondance, aussi appelé *mapping*, permet de développer de manière plus découplée les différentes parties qui composent l'usine à logiciels. Celles-ci sont ensuite liées entre elles grâce à ce fichier de mapping. Le format de représentation des données *TOML* se prête parfaitement à cette fin car il permet d'exprimer aisément des ensembles « clef-valeur » et ce de manière minimale. Le listing C.1 disponible en annexe présente un exemple d'un tel fichier. Seul le contenu des variables peut être modifié car le générateur de logiciels dépend d'une structure *TOML* bien précise. À titre d'exemple, la variable « mapping » appartenant à la table « [persona.handicap_visuel.nom.couleurs] » contient les correspondances pour le daltonisme entre les balises *XML* et leurs attributs, ainsi que les qualificatifs du fichier de patching par soustraction présenté par le listing B.3 disponible en annexe. L'argument permettant d'indiquer l'emplacement du mapping *TOML* à exploiter est le suivant, où « <chemin> » doit être remplacé par le chemin d'accès correspondant :

```
--toml-mapping <chemin>
```

Implémentation Concrète d'une Tactique

Le listing 6.1 illustre l'implémentation concrète de la tactique (Adapt·3a) pour la plateforme *Android*. En particulier, il s'agit du code source complété présenté par le listing 5.6 dans le chapitre 5 lors de la conception d'une stratégie. Les valeurs assignées à la variable « drawable_suffixes » proviennent de la configuration *XML* explicitée par le listing C.2, telles qu'indiquées par le mapping *TOML* détaillé par le listing C.1. L'entièreté des tactiques relatives à la stratégie *Android* font ainsi l'objet d'une définition similaire et sont par conséquent toutes implémentées de façon concrète. Les composants techniques ainsi mentionnés permettent dès lors de générer de manière automatique un logiciel particulier.

```python
    def _adapt3a(self) -> None:
        """Implements the eponymous tactic
        """

        # Retrieving the resolutions from the xml and
        # Converting them into drawable qualifiers to load the appropriate drawables
        drawable_suffixes = [
            DPI.drawable_dpi_from_resolution(*pair)
            for pair in xml_resolver.find_qualifiers_from_resolution(
                self.xml_content,
                self.toml_mapping['appareil']['xml_name'],
                self.toml_mapping['appareil']['resolution_x'],
                self.toml_mapping['appareil']['resolution_y'])
        ]

        # Creating a Res for every appropriate drawable found
        AddRes.apply(
            *[Res(src=os.path.join(self.artefacts_root_folder, f"images/android/drawable{dpi}/"),
                  dst=os.path.join(self.release_root_folder, f"app/src/main/res/drawable{dpi}/"))
              for dpi in drawable_suffixes],
            order=Order.PARALLEL if len(drawable_suffixes) > 1 else Order.SEQUENTIAL,
        )
```

Listing 6.1 – Implémentation concrète de la tactique (Adapt·3a) pour la plateforme *Android* au sein de la stratégie associée

6.2.2 Création d'un Logiciel

Chaque configuration requiert un environnement dédié pour être développée de manière isolée et optimale. Ainsi, pour chaque produit demandé par un client, une branche dédiée à celui-ci est créée sur la plateforme *GitHub* dans laquelle les tactiques seront mises en œuvre. Conformément au plan d'exécution illustré par la figure 6.1, la tactique (Android·2a) ou (iOS·2a) est appliquée en premier lieu. Celle-ci copie l'architecture mobile de base correspondante dans la branche nouvellement créée. La carte du site touristique indiquée par la tactique (Multi·2a) et le serveur pris en charge par la tactique (Multi·1a) sont également ajoutés à ladite branche. Les tactiques de la phase de programmation sont ainsi automatiquement mises en place grâce à la copie de l'architecture commune. Les tactiques de la phase de compilation s'articulent alors entre elles conformément au plan d'exécution établi, en altérant les fichiers nouvellement copiés et en ajoutant les artefacts correspondants. Lorsque les tactiques ont achevé les modifications qu'elles devaient apporter à l'architecture commune, les

développeurs de l'entreprise *BetterSoft* s'emploient à implémenter les spécificités propres à chaque configuration. En effet, celles-ci étant trop particulières pour être capturées et abstraites par une tactique, elles font l'objet d'une implémentation sur mesure. Cette étape nécessite cependant une ingénierie des exigences qui met ces spécificités en lumière par rapport aux exigences capturées par les tactiques. Cette ingénierie peut être effectuée lors de l'analyse du domaine, comme explicité dans le chapitre 4. Enfin, la configuration ainsi parachevée est livrée au client selon les méthodes proposées par l'entreprise. La branche *GitHub* relative à la configuration est conservée afin de maintenir une empreinte des modifications apportées à celle-ci et de faciliter sa maintenance si d'aventure elle était amenée à évoluer et ainsi être générée à nouveau. Il existe trois derniers arguments relatifs à la création d'un nouveau logiciel :

1. « --branch-name » spécifie le nom de la branche *GitHub* sur laquelle créer le logiciel, où « <branche> » doit être remplacé par la valeur correspondante ;
2. « --artefacts-root-folder » indique le chemin d'accès menant aux artefacts à manipuler, où « <chemin> » doit être remplacé par la valeur correspondante ;
3. « --release-root-folder » précise le chemin d'accès où créer le répertoire dudit nouveau logiciel. Celui-ci ne doit pas exister au préalable car il sera créé par le générateur de logiciels, où « <chemin> » doit être remplacé par la valeur correspondante.

```
1 --branch-name <branche> --artefacts-root-folder <chemin> --release-root-folder <chemin>
```

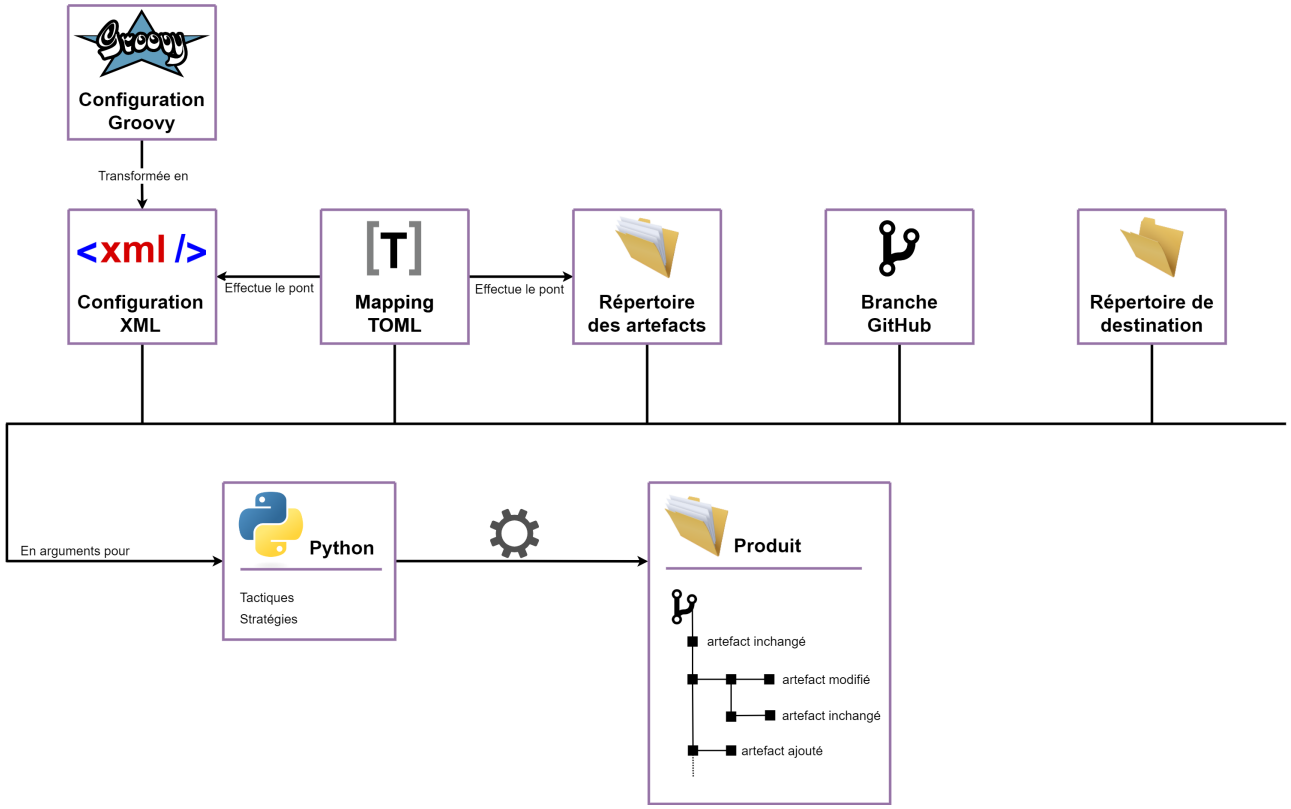


FIGURE 6.2 – Représentation graphique du processus de production d'un logiciel. L'implémentation de l'usine à logiciels à l'aide du langage de programmation *Python* requiert l'élicitation de cinq arguments distincts, détaillés dans les sous-sections 6.2.1 et 6.2.2. Ainsi, une configuration au format *XML*, dérivée de la configuration *Groovy* correspondante, un chemin d'accès vers le répertoire des artefacts, un fichier de correspondance au format *TOML* ainsi que le nom d'une branche *GitHub* et d'un répertoire de destination doivent être explicités. Grâce aux tactiques organisées au sein de stratégies, telles que définies dans la section 5.4 du chapitre 5, le produit demandé par le client est généré de manière automatique. Ce dernier est ensuite isolé dans le répertoire de destination précédemment spécifié au sein d'une branche dédiée du répertoire *GitHub* de l'usine à logiciels *BetterSoft*.

6.2.3 Illustration du Logiciel *JarreDeux*

Cette section présente la création du logiciel *JarreDeux* grâce à l'usine à logiciels créée à cette fin. Elle détaille les différents arguments indiqués à celle-ci, guidant ainsi que les modifications à appliquer par les tactiques afin de livrer le produit demandé. Enfin, quelques figures illustrant les altérations apportées par les tactiques présentent le résultat final obtenu.

Arguments

Les arguments relatifs à la création du produit *JarreDeux* par l'usine à logiciels *BetterSoft* sont détaillés par la table 6.1. Ainsi, la branche qui accueillera celui-ci se nomme « JarreDeux » et ledit produit est mis en œuvre par les éléments définis dans les chapitres 5 et 6 de ce document. En particulier, les modifications apportées par les tactiques, guidées par la configuration *XML* mentionnée en argument, sont listées de manière exhaustive dans la section 5.2 du chapitre 5. À ce titre, celles-ci ne sont pas détaillées dans la suite.

Argument	Valeur
« −−branch−name »	« JarreDeux »
« −−artefacts−root−folder »	Chemin d'accès vers le répertoire source de la table 5.2 du chapitre 5
« −−release−root−folder »	« ./release »
« −−xml−configuration »	Chemin d'accès du listing C.2 disponible en annexe
« −−toml−mapping »	Chemin d'accès du listing C.1 disponible en annexe

TABLE 6.1 – Arguments relatifs à la création du logiciel *JarreDeux*

Résultats

Dès le parachèvement de l'exécution du générateur à logiciels, une branche nouvellement créée intitulée « JarreDeux » est alors disponible sur la plateforme *GitHub* de l'entreprise *BetterSoft*. Le contenu de celle-ci, hérité de la branche principale de l'usine à logiciels, contient en outre un répertoire « release » au sein duquel réside le produit *Android* nouvellement assemblé. Les captures d'écran de l'interface graphique du logiciel *JarreDeux*, similaire à l'interface définie et illustrée par le modèle correspondant à la syntaxe concrète du DSML détaillée dans la section 4.1.4 du chapitre 4, sont disponibles dans l'annexe C.2. Les figures 6.3, 6.4, 6.5, 6.6 et 6.7 présentées dans la suite de cette section détaillent un échantillon représentatif des modifications apportées par les tactiques en effectuant quelques comparaisons *avant-après*. En effet, étant donné la kyrielle de modifications apportées aux différents fichiers qui constituent le produit demandé par le client, il semble complexe de toutes les détailler dans ce document, qui plus est de manière avenante.

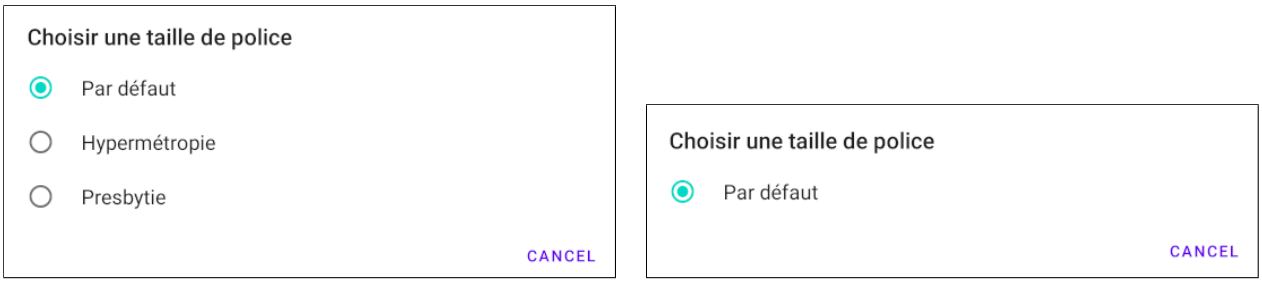
```
<!-- @TACTIC('Adapt4a') = GEOLOCALISATION -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<!-- @END_TACTIC -->
<!-- @TACTIC('Adapt4a') = BLUETOOTH -->
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />

<uses-feature
    android:name="android.hardware.bluetooth"
    android:required="false" />
<uses-feature
    android:name="android.hardware.bluetooth_le"
    android:required="false" />
<!-- @END_TACTIC -->
```

(a) Fichier « AndroidManifest.xml » *avant*

```
<!-- @TACTIC('Adapt4a') = GEOLOCALISATION -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<!-- @END_TACTIC -->
<!-- @TACTIC('Adapt4a') = BLUETOOTH -->
<!-- @END_TACTIC -->
```

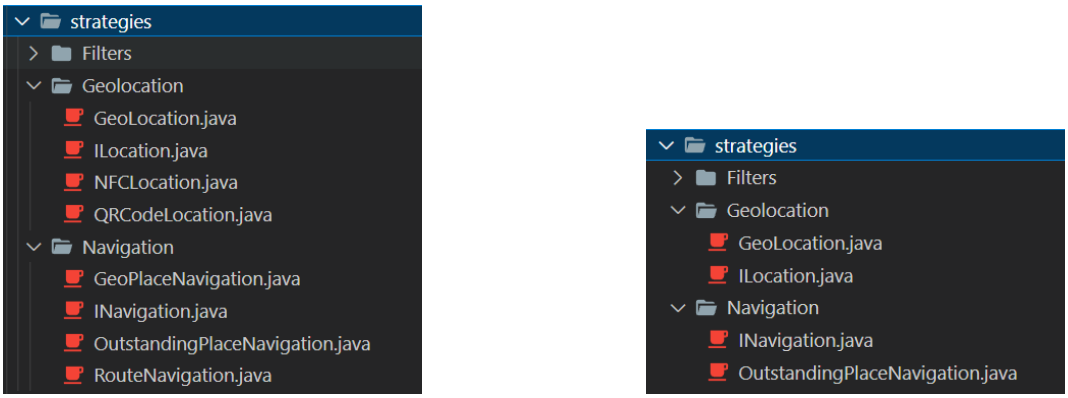
(b) Fichier « AndroidManifest.xml » *après*

FIGURE 6.3 – Illustration des modifications apportées par la tactique (Adapt·4a) au fichier « AndroidManifest.xml ». La configuration *XML* indiquée en argument précise ne pas avoir besoin des permissions liées au *Bluetooth*. Ainsi, comme structuré par le fichier de patching par soustraction relatif à cette tactique, détaillé dans la section 5.4 du chapitre 5, les permissions inutiles sont retirées.



(a) Tailles de police disponibles *avant* (b) Tailles de police disponibles *après*

FIGURE 6.4 – Illustration de la suppression, mise en place par la tactique (Adapt·5a), des tailles de police disponibles dans les paramètres du produit généré. Cette suppression est similaire à celle mise en place par la tactique (Adapt·6b) pour les filtres de couleurs liés aux différents types de daltonisme. À l'instar du patching par soustraction présenté par la figure 6.3, les tailles de police non mentionnées dans la configuration *XML* sont retirées conformément aux altérations prescrites par le fichier de patching par soustraction relatif à cette tactique.



(a) Stratégies de navigation et géolocalisation *avant* (b) Stratégies de navigation et géolocalisation *après*

FIGURE 6.5 – Illustration de la suppression des stratégies non sélectionnées. Les stratégies de géolocalisation et de navigation définies par les tactiques (Adapt·9b) et (Adapt·10b), respectivement, sont supprimées par les tactiques de patching par soustraction (Adapt·9a) et (Adapt·10a), respectivement. Ces modifications sont réalisées conformément aux fichiers de patching par soustraction éponymes, détaillés dans la section 5.4 du chapitre 5.

(a) Répertoire des ressources *avant* (b) Répertoire des ressources *après*.

FIGURE 6.6 – Illustration de l'ajout de fichiers au répertoire des ressources. Les répertoires « layout-w600dp » et « drawable-xhpi sont ajoutés par les tactiques (Android·1a) et (Adapt·3a), respectivement. Ces répertoires regroupent respectivement les mises en page et résolutions d'images adaptées aux appareils mentionnés dans la configuration *XML* indiquée en argument. Ces modifications sont réalisées conformément à la définition du type de tactique « Ajout de ressources » détaillé dans la section 5.4 du chapitre 5.

(a) Logo du produit *avant*

(b) Logo du produit *après*

FIGURE 6.7 – Illustration de la substitution du logo mise en place par la tactique (Adapt·1a). Le logo du logiciel *JarreDeux* est substitué à celui par défaut. Cette substitution, à l'instar des modifications présentées par la figure 6.6, est réalisée conformément à la définition du type de tactique « Ajout de ressources » détaillé dans la section 5.4 du chapitre 5.

6.3 Conclusion

Ce chapitre a présenté le plan d'exécution de l'usine à logiciels *BetterSoft*. En particulier, il permet ainsi d'ordonner les tactiques implémentées au sein de la stratégie *Android* afin de générer le produit demandé par le client. La décomposition d'une tactique est également abordée afin de présenter une vue exhaustive des modifications à apporter à la structure méthodologique *Bespoke*, en opposition avec les tactiques atomiques définies dans le chapitre 5. Enfin, l'implémentation de ladite usine à logiciels est présentée et les multiples composants techniques qui la constituent sont détaillés. Les résultats ainsi obtenus sont illustrés par la génération du logiciel *JarreDeux*, commandé par l'entreprise fictive *JarreDin*, et les modifications apportées par les différentes tactiques implémentées au sein de l'usine à logiciels.

Chapitre 7

Conclusion

La structure méthodologique *Bespoke* est explorée en détail au travers des multiples chapitres proposés dans ce document. L'élaboration d'une usine à logiciels est guidée par le cas d'application *BetterSoft* [31], qui dépeint une entreprise fictive éponyme désireuse d'exploiter les similitudes des logiciels touristiques mobiles qu'elle développe. Ce cas d'application permet de mettre en exergue les mécanismes proposés par la méthodologie *Bespoke*. En particulier, ce document s'intéresse à la conception d'une usine capable de prendre en charge des produits destinés à des systèmes d'exploitation multiples. La structure méthodologique *Bespoke* est ainsi abordée en symbiose d'un point de vue *ingénieur*, explicitant les choix effectués lors des différentes étapes de la conception de l'usine à logiciels et d'un point de vue *critique*, mis en avant au moyen de remarques liées aux choix d'ingénierie opérés.

7.1 Rétrospective

Dans un premier temps, les concepts et pratiques relatifs à l'ingénierie du domaine sont abordés en détail dans le chapitre 2, permettant ainsi la formalisation des notions d'expression de la variabilité employées tout au long du mémoire. Des mécanismes d'implémentation de la variabilité, aussi appelés *tactiques d'implémentation*, sont énumérés afin d'élaborer un outillage utilisable lors de la conception de l'usine à logiciels. Enfin, car là réside le cœur du document, les caractéristiques des plateformes mobiles *Android* et *iOS* sont exposées et mises en relation avec les tactiques d'implémentation.

La définition du domaine effectuée dans le chapitre 3 a permis de circonscrire le spectre d'action de l'usine à logiciels à concevoir, en définissant et en organisant les différentes fonctionnalités relatives à celle-ci. Les fondations ainsi posées sont formalisées dans le chapitre 4 sous forme d'un métamodèle, grâce à l'approche par DSML de la conception d'une ligne de produits logiciels adoptée par la méthodologie *Bespoke*. Cette approche est saluée par la remarque (Rem·II) car elle permet l'élaboration d'une syntaxe concrète destinée à le manipuler par des individus non-initiés. Cette capacité est illustrée par la commande du logiciel *JarreDeux*. Le métamodèle est alors manuellement converti en un diagramme de caractéristiques, dont la forme est plus adaptée à une manipulation automatique par des programmes tierces. Chaque caractéristique fait l'objet d'une annotation particulière destinée à orienter le choix des tactiques d'implémentation. Le contenu de ces annotations est par ailleurs restreint car, comme détaillé par la remarque (Rem·V), certaines préoccupations semblent prématurées à cette étape de la conception de l'usine. Le modèle du logiciel *JarreDeux* est également converti en parallèle en une configuration, en mettant en évidence la scission de celle-ci par la remarque (Rem·IV). Cette scission est inévitable en conséquence des caractéristiques intrinsèques observées et des hypothèses formulées sur les différences technologiques entre les plateformes *Android* et *iOS* par les tables 2.1 et 2.2 du chapitre 2. Les étapes suivantes de la méthodologie *Bespoke* s'en retrouvent alors grandement impactées.

Le chapitre 5 aborde la conception de l'usine à logiciels grâce à l'articulation de tactiques d'implémentation au sein de stratégies définies sur mesure. La remarque (Rem·VI) introduit en premier lieu la nécessité de définir explicitement l'architecture commune des configurations à générer, car c'est sur cette dernière que les tactiques à définir agissent. Cette formalisation est naturellement complétée par son adaptation et l'élaboration des artefacts impactés en fonction des tactiques ainsi définies, comme détaillé par la remarque (Rem·X). Le chapitre 5 s'épanche ensuite sur ladite définition des tactiques d'implémentation. Plusieurs catégories distinctes de tactiques sont élaborées afin d'offrir une classification directe et intuitive, en fonction des modifications à apporter à chaque caractéristique et du système d'exploitation présenté par la configuration courante. Il existe ainsi des tactiques *Android*, *iOS*, *multiplateformes* et *adaptatives* et plusieurs d'entre elles sont alors définies

pour réaliser un même concept. La remarque (Rem·VII) propose des méthodes manuelles et automatiques de résolution de la classification des tactiques. Par ailleurs, la remarque (Rem·VIII) argumente une utilisation plurielle des tactiques par concept. En effet, les tactiques d'implémentation ne permettent que des changements d'une seule nature distincte. Or, certains concepts nécessitent des changements de plusieurs natures, justifiant ainsi l'utilisation de multiples tactiques par concept. En parallèle, la remarque (Rem·IX) propose une résolution alternative de cette nature plurielle en redéfinissant certaines notions du métamodèle de la méthodologie *Bespoke*, conformément à la sémantique d'une tactique. Ces tactiques d'implémentation sont ensuite organisées et mises en œuvre au sein de stratégies. La structure méthodologique *Bespoke* permet l'élaboration de plusieurs stratégies et cette capacité est mise à profit dans la remarque (Rem·XI) afin de classer les tactiques en fonction de leur plateforme respective. Cette multiplicité permet de répondre aux besoins émis par un client en appliquant une stratégie particulière, en fonction du système d'exploitation présenté par une configuration.

Les tactiques ainsi définies font l'objet d'une implémentation abstraite, où certains arguments sont laissés en suspens. Chaque tactique de la stratégie *Android* est encapsulée dans une méthode spécifique, afin d'effectuer les modifications prescrites sans interruption, à l'instar d'une transaction. Cette caractéristique atomique est mise en lumière par la remarque (Rem·XIII). Les tactiques de patching par soustraction sont par ailleurs implémentées de manière déclarative, grâce à un fichier prescriptif au format *YAML*. La remarque (Rem·XII) justifie une telle implémentation en mettant en avant sa capacité intrinsèque à se concentrer sur la description des opérations à effectuer. Le chapitre 6 articule enfin ces tactiques en les ordonnant entre elles au sein d'un plan d'exécution. L'implémentation concrète de l'usine à logiciels *BetterSoft* est alors concrétisée en unissant l'entièreté des concepts définis tout au long du document et en définissant quelques composants techniques ad hoc. Celle-ci est finalement illustrée en générant la configuration *Android* du logiciel *JarreDeux*.

Somme toute, le cas d'application *BetterSoft* a permis de traverser intégralement la structure méthodologique *Bespoke*. Les multiples avantages et limitations de celle-ci ont été mis en lumière au travers du cas d'application holistique choisi. L'objectif de la méthodologie, à savoir l'élaboration agnostique d'un point de vue technologique d'une ligne de produits logiciels, a été rencontré au terme du chapitre 6, qui présente l'implémentation de l'usine à logiciels *BetterSoft*. Cependant, certains mécanismes proposés par la structure méthodologique se sont révélés insuffisants ou peu utiles, et des pistes d'améliorations ont ainsi été proposées et explorées tout au long du document. La page ix propose une table regroupant l'entièreté des remarques formulées. La majorité de ces critiques sont émises à propos des mécanismes liés à l'élaboration des tactiques et leurs impacts sur une architecture commune. En effet, le point d'orgue de ces remarques est le besoin fort de pouvoir définir plusieurs tactiques pour un même concept, car chacune des tactiques présentées dans le chapitre 2 n'effectue que des opérations et changements d'une seule nature. En outre, ces multiples tactiques doivent ensuite pouvoir être classées en différents ensembles selon les contraintes posées, à l'instar de la contrainte (C7), qui introduit la scission d'une configuration selon le système d'exploitation. À chacun de ces ensembles correspondrait alors une stratégie d'implémentation des tactiques. Néanmoins, la majorité des mécanismes proposés par la structure méthodologique *Bespoke* se sont révélés adéquats et ont permis la confection de l'usine à logiciels multiplateforme *BetterSoft* de manière guidée et aisée.

7.2 Limitations

Le travail réalisé dans ce document souffre de quelques limitations. Les fonctionnalités définies dans le chapitre 3, ayant guidés l'élaboration de la ligne de produits logiciels, sont des *mocks* et ne sont ainsi pas implémentées de façon concrète, conformément à l'hypothèse (H6). Des fonctionnalités complètes pourraient mettre en lumière d'autres avantages ou limitations de la structure méthodologique *Bespoke* et modifier les tactiques définies dans le chapitre 5. En outre, par manque de temps mais également de ressources, les langages *FeatAll*, *TacticType* et *Strategy* n'ont pas été implémentés. Cela ne permet pas une critique adéquate de ceux-ci. C'est pourquoi la remarque (Rem·XII) propose une implémentation alternative. Enfin, seule la stratégie *Android* a fait l'objet d'une implémentation concrète dans ce document. Bien que similaire, il pourrait être intéressant de réaliser à son tour à la stratégie *iOS*.

7.3 Travail Futur

Le travail réalisé dans ce document ouvre la voie à de nombreuses questions intéressantes. Celles-ci peuvent faire l'objet d'un travail de recherche futur. Les hypothèses restrictives (H3), (H4) et (H5) formulées dans le chapitre 1 méritent d'être explorées plus en détail. Ainsi, il serait intéressant de chercher à automatiser et par conséquent lier les différentes transformations de modèle, à l'instar de la conversion du métamodèle annoté en

un diagramme de caractéristiques, réalisée dans le chapitre 4 et explorée par le remarque (Rem·III). Cela permettrait de s'approcher d'une conception semi-automatique d'une ligne de produits logiciels. Par conséquent, il serait judicieux de s'intéresser à l'implémentation des langages *FeatAll*, *TacticType* et *Strategy*. En outre, il serait également intéressant de définir des fonctionnalités qui interfèrent entre elles, afin de provoquer des liens d'ingérence et de discuter de leur résolution, à l'instar de la remarque (Rem·XIV). Enfin, reprendre l'usine à logiciels implémentée dans le chapitre 6 et la soumettre à une phase d'évolution permettrait d'observer les changements à apporter ainsi que leurs conséquences. De manière analogue, il serait intéressant de s'interroger sur l'inclusion de code source existant au sein d'une usine à logiciels nouvellement créée, permettant ainsi l'adoption d'une telle approche de conception de logiciels à tout moment au sein d'une entreprise. Par exemple, une telle intégration nécessiterait éventuellement une certaine priorisation de l'implémentation des caractéristiques de la ligne de produits logiciels, comme proposée par la remarque (Rem·I). Somme toute, répondre à ces multiples interrogations permettrait ainsi de faire un nouveau pas vers une adoption plus large du paradigme de création de logiciels grâce à une ligne de produits.

Bibliographie

[1] *Adding images to your Xcode project | Apple Developer Documentation*. 1987. URL : https://developer.apple.com/documentation/xcode/adding-images-to-your-xcode-project (visité le 16/03/2023).

[2] Mauricio Alférez et al. « Modeling variability in the video domain : language and experience report ». In : *Software Quality Journal* 27 (2019), p. 307-347.

[3] Chris Anderson. « The model-view-viewmodel (MVVM) design pattern ». In : *Pro Business Applications with Silverlight 5*. Springer, 2012, p. 461-499.

[4] *ANTLR*. 1997. URL : https://www.antlr.org (visité le 24/02/2023).

[5] Sven Apel et al. *Feature-oriented software product lines*. Springer, 2016.

[6] *ArchiMate - Wikipedia*. 2001. URL : https://en.wikipedia.org/wiki/ArchiMate (visité le 22/05/2023).

[7] *Auto Layout Guide : Understanding Auto Layout*. 1987. URL : https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/ (visité le 16/03/2023).

[8] Kacper Bąk et al. « Clafer : unifying class and feature modeling ». In : *Software & Systems Modeling* 15 (2016), p. 811-845.

[9] Mahdi Bashari, Ebrahim Bagheri et Weichang Du. « Dynamic software product line engineering : a reference framework ». In : *International Journal of Software Engineering and Knowledge Engineering* 27.02 (2017), p. 191-234.

[10] Rabih Bashroush et al. « CASE Tool Support for Variability Management in Software Product Lines ». In : *ACM Comput. Surv.* 50.1 (mars 2017). ISSN : 0360-0300. DOI : 10.1145/3034827. URL : https://doi.org/10.1145/3034827.

[11] Martin Becker et al. « Comprehensive variability modelling to facilitate efficient variability treatment ». In : *Software Product-Family Engineering : 4th International Workshop, PFE 2001 Bilbao, Spain, October 3–5, 2001 Revised Papers 4*. Springer. 2002, p. 294-303.

[12] Maouaheb Belarbi. *Ingénierie d'usines à logiciels*. Rapp. tech. Université de Namur, jan. 2023.

[13] Thorsten Berger et al. « What is a feature ? a qualitative study of features in industrial software product lines ». In : *Proceedings of the 19th international conference on software product line*. 2015, p. 16-25.

[14] D. Beuche. « Modeling and Building Software Product Lines with Pure : :Variants ». In : *Software Product Line Conference, International*. Los Alamitos, CA, USA : IEEE Computer Society, sept. 2008, p. 358. DOI : 10.1109/SPLC.2008.53. URL : https://doi.ieeecomputersociety.org/10.1109/SPLC.2008.53.

[15] Sebastien Blacks. « A Software Factory Engine ». Mém. de mast. Université de Namur, 2022.

[16] Marco Brambilla et Piero Fraternali. *Interaction flow modeling language : Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[17] Lisa Brownsword et Paul Clements. *A Case Study in Successful Product Line Development*. Rapp. tech. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1996.

[18] Davide Brugali et Nico Hochgeschwender. « Software product line engineering for robotic perception systems ». In : *International Journal of Semantic Computing* 12.01 (2018), p. 89-107.

[19] Walter Cazzola et Edoardo Vacchi. « Language components for modular DSLs using traits ». In : *Computer Languages, Systems & Structures* 45 (2016), p. 16-34.

[20] *CelsiusTech Systems AB – SPLC*. 2004. URL : https://splc.net/fame/celsiustech-systems-ab/ (visité le 20/02/2023).

[21] Lianping Chen, Muhammad Ali Babar et Nour Ali. « Variability management in software product lines : a systematic review ». In : *Proceedings of the 13th International Software Product Line Conference*. Citeseer. 2009, p. 81-90.

[22] Paul Clements et Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[23] *Conditional (computer programming) - Wikipedia*. 2001. URL : https://en.wikipedia.org/wiki/Conditional_(computer_programming) (visité le 23/02/2023).

[24] *Create your first cross-platform app | Kotlin Documentation*. 2013. URL : https://kotlinlang.org/docs/multiplatform-mobile-create-first-app.html (visité le 28/02/2023).

[25] Krzysztof Czarnecki, Simon Helsen et Ulrich Eisenecker. « Staged configuration using feature models ». In : *Software Product Lines : Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings 3*. Springer. 2004, p. 266-283.

[26] *daltonisme - Définitions, synonymes, conjugaison, exemples | Dico en ligne Le Robert*. 1997. URL : https://dictionnaire.lerobert.com/definition/daltonisme (visité le 14/02/2023).

[27] Paulo RM De Andrade et al. « Cross platform app : a comparative study ». In : *arXiv preprint arXiv :1503.03511* (2015).

[28] *Domain-Specific Languages*. 2011. URL : http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html (visité le 24/02/2023).

[29] *DSL (Domain Specific Languages) - Rust By Example*. 2010. URL : https://doc.rust-lang.org/rust-by-example/macros/dsl.html (visité le 24/02/2023).

[30] Stéphane Ducasse et al. « Traits : A mechanism for fine-grained reuse ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2 (2006), p. 331-388.

[31] Vincent Englebert. *Ingénierie d'usines à logiciels*. Rapp. tech. Université de Namur, sept. 2022.

[32] Vincent Englebert et Maouaheb Belarbi. « A Methodological Framework for SPL Engineering from DSML ». In : *Model-Driven Engineering and Software Development - 10th International Conference, MODELSWARD, February 6-8, 2022*. Sous la dir. de Luís Ferreira Pires, Slimane Hammoudi et Edwin Seidewitz. Communications in Computer and Information Science. Accepté, en attente de publication. Springer, 2022.

[33] Mohamed E Fayad, Douglas C Schmidt et Ralph E Johnson. *Building application frameworks : object-oriented foundations of framework design*. John Wiley & Sons, Inc., 1999.

[34] Peter H Feiler, David P Gluch et John J Hudak. *The architecture analysis & design language (AADL) : An introduction*. Rapp. tech. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[35] Adel Ferdjoukh et al. « A CSP approach for metamodel instantiation ». In : *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE. 2013, p. 1044-1051.

[36] Dániel Fey, Róbert Fajta et András Boros. « Feature modeling : A meta-model to enhance usability and usefulness ». In : *Software Product Lines : Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings*. Springer. 2002, p. 198-216.

[37] *Figma is powered by WebAssembly*. 1999. URL : https://www.figma.com/fr/blog/webassembly-cut-figmas-load-time-by-3x/ (visité le 14/03/2023).

[38] Xavier Franch et al. « The i* framework for goal-oriented modeling ». In : *Domain-Specific Conceptual Modeling : Concepts, Methods and Tools* (2016), p. 485-506.

[39] Critina Gacek et Michalis Anastasopoules. « Implementing product line variabilities ». In : *Proceedings of the 2001 symposium on Software reusability : putting software reuse in context*. 2001, p. 109-117.

[40] Erich Gamma et al. « Design patterns : Abstraction and reuse of object-oriented design ». In : *ECOOP'93–Object-Oriented Programming : 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7*. Springer. 1993, p. 406-431.

[41] Erich Gamma et al. *Design patterns : elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[42] Mark H Goadrich et Michael P Rogers. « Smart smartphone development : iOS versus Android ». In : *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, p. 607-612.

[43] Hassan Gomaa. *Real-time software design for embedded systems*. Cambridge University Press, 2016.

[44] *Google AdMob – Monétisation des applications mobiles*. 1997. URL : https://admob.google.com/home/ (visité le 14/03/2023).

[45] Jack Greenfield et Keith Short. « Software factories : assembling applications with patterns, models, frameworks and tools ». In : *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2003, p. 16-27.

[46] Ruben Heradio et al. « A bibliometric analysis of 20 years of research on software product lines ». In : *Information and Software Technology* 72 (2016), p. 1-15.

[47] Jose-Miguel Horcas, Mónica Pinto et Lidia Fuentes. « Software Product Line Engineering : A Practical Experience ». In : *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. New York, NY, USA : Association for Computing Machinery, 2019, 164–176. isbn : 9781450371384. doi : 10.1145/3336294.3336304. url : https://doi.org/10.1145/3336294.3336304.

[48] *hypermétropie - Définitions, synonymes, conjugaison, exemples | Dico en ligne Le Robert*. 1997. url : https://dictionnaire.lerobert.com/definition/hypermetropie (visité le 14/02/2023).

[49] *Icônes, illustrations clipart, photos & musique gratuites*. 2017. url : https://icones8.fr (visité le 17/02/2023).

[50] *Insee - Institut national de la statistiques et des études économiques*. 1996. url : https://www.insee.fr/fr/metadonnees/definition/c1161 (visité le 07/02/2023).

[51] *Java Documentation - Get Started*. 1988. url : https://docs.oracle.com/en/java/ (visité le 28/02/2023).

[52] Frédéric Jouault et Jean Bézivin. « KM3 : a DSL for Metamodel Specification ». In : *Formal Methods for Open Object-Based Distributed Systems : 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006. Proceedings 8*. Springer. 2006, p. 171-185.

[53] *jpp*. 2018. url : https://github.com/mkowsiak/jpp (visité le 28/02/2023).

[54] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Rapp. tech. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[55] Christian Kastner, Sven Apel et Don Batory. « A Case Study Implementing Features Using AspectJ ». In : *11th International Software Product Line Conference (SPLC 2007)*. 2007, p. 223-232. doi : 10.1109/SPLINE.2007.12.

[56] Gregor Kiczales. « Aspect-oriented programming ». In : *ACM Computing Surveys (CSUR)* 28.4es (1996), 154-es.

[57] Glenn E Krasner, Stephen T Pope et al. « A description of the model-view-controller user interface paradigm in the smalltalk-80 system ». In : *Journal of object oriented programming* 1.3 (1988), p. 26-49.

[58] Christian Kästner et Svan Apel. « Integrating Compositional and Annotative Approaches for Product Line Engineering ». In : *Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE)* (oct. 2008).

[59] *Les types de daltonisme*. 2019. url : https://lesyeuxdudaltonisme.fr/les-types-de-daltonisme/ (visité le 14/02/2023).

[60] Daniela Lettner et al. « Feature modeling of two large-scale industrial software systems : Experiences and lessons learned ». In : *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2015, p. 386-395.

[61] Yannis Lilis et Anthony Savidis. « A Survey of Metaprogramming Languages ». In : *ACM Comput. Surv.* 52.6 (oct. 2019). issn : 0360-0300. doi : 10.1145/3354584. url : https://doi.org/10.1145/3354584.

[62] Frank J Van der Linden, Klaus Schmid et Eelco Rommes. *Software product lines in action : the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[63] Barbara Liskov. « Keynote address-data abstraction and hierarchy ». In : *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*. 1987, p. 17-34.

[64] Maíra Marques et al. « Software product line evolution : A systematic literature review ». In : *Information and Software Technology* 105 (2019), p. 190-208. issn : 0950-5849. doi : https://doi.org/10.1016/j.infsof.2018.08.014. url : https://www.sciencedirect.com/science/article/pii/S0950584918301848.

[65] Flávio Medeiros et al. « Colligens : A Tool to Support the Development of Preprocessor-Based Software Product Lines in C ». In : *Proc. Brazilian Conf. Software : Theory and Practice (CBSoft)*. 2013.

[66] Jens Meinicke et al. *Mastering software variability with FeatureIDE*. Springer, 2017.

[67] Andreas Metzger et Klaus Pohl. « Software product line engineering and variability management : achievements and challenges ». In : *Future of software engineering proceedings* (2014), p. 70-84.

[68] *Middleware - Wikipedia*. 2001. url : https://en.wikipedia.org/wiki/Middleware (visité le 24/02/2023).

[69] Elisabeth Niehaus, Klaus Pohl et Günter Böckle. *Software Product Line Engineering : Foundations, Principles and Techniques, Kapitel Product Management*. 2005.

[70] Linda Northrop et al. « A framework for software product line practice, version 5.0 ». In : *SEI.–2007–http ://www. sei. cmu. edu/productlines/index. html* (2007).

[71] Miika Oksanen et al. « A comparison of two SPLE tools : Pure : : Variants and Clafer tools ». In : (2018).

[72] Gilles Perrouin et al. « Pairwise testing for software product lines : comparison of two approaches ». In : *Software Quality Journal* 20 (2012), p. 605-643.

[73] *Petri net - Wikipedia*. 2001. URL : https://en.wikipedia.org/wiki/Petri_net (visité le 21/02/2023).

[74] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[75] Dhananjay Prasanna. *Dependency injection : design patterns using spring and guice*. Simon et Schuster, 2009.

[76] *presbytie - Définitions, synonymes, conjugaison, exemples | Dico en ligne Le Robert*. 1997. URL : https://dictionnaire.lerobert.com/definition/presbytie (visité le 14/02/2023).

[77] Tim Rentsch. « Object oriented programming ». In : *ACM Sigplan Notices* 17.9 (1982), p. 51-57.

[78] John Reppy et Aaron Turon. « Metaprogramming with traits ». In : *ECOOP 2007–Object-Oriented Programming : 21st European Conference, Berlin, Germany, July 30-August 3, 2007. Proceedings 21*. Springer. 2007, p. 373-398.

[79] *Ressources de géographie pour les enseignants — Géoconfluences*. 2002. URL : http://geoconfluences.ens-lyon.fr/glossaire/site-touristique (visité le 07/02/2023).

[80] Youssef Ridene et al. « A DSML for Mobile Phone Applications Testing ». In : *Proceedings of the 10th Workshop on Domain-Specific Modeling*. DSM '10. New York, NY, USA : Association for Computing Machinery, 2010. ISBN : 9781450305495. DOI : 10.1145/2060329.2060340. URL : https://doi.org/10.1145/2060329.2060340.

[81] James Robertson et Suzanne Robertson. « Volere ». In : *Requirements Specification Templates* (2000).

[82] Marko Rosenmüller et al. « Flexible feature binding in software product lines ». In : *Automated Software Engineering* 18 (2011), p. 163-197.

[83] *SAP Software Solutions | Business Applications and Technology*. 1995. URL : https://www.sap.com/index.html (visité le 23/02/2023).

[84] Nathanael Schärli et al. « Traits : Composable units of behaviour ». In : *ECOOP 2003–Object-Oriented Programming : 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings 17*. Springer. 2003, p. 248-274.

[85] Douglas C Schmidt et al. *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013.

[86] *Sending simple data to other apps | Android Developers*. 1997. URL : https://developer.android.com/training/sharing/send (visité le 14/03/2023).

[87] *Settings | Android Developers*. 1997. URL : https://developer.android.com/develop/ui/views/components/settings (visité le 10/03/2023).

[88] Marco Sinnema et Sybren Deelstra. « Classifying variability modeling techniques ». In : *Information and software technology* 49.7 (2007), p. 717-739.

[89] Jean-Sébastien Sottet, Alfonso García Frey et Jean Vanderdonckt. *Human centered software product lines*. Springer, 2017.

[90] *Storyboard*. 1987. URL : https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html (visité le 27/03/2023).

[91] *Support different pixel densities | Android Developers*. 1997. URL : https://developer.android.com/training/multiscreen/screendensities (visité le 10/03/2023).

[92] *Support different screen sizes | Android Developers*. 1997. URL : https://developer.android.com/guide/topics/large-screens/support-different-screen-sizes#alternative-layouts (visité le 10/03/2023).

[93] Mikael Svahnberg, Jilles Van Gurp et Jan Bosch. « A taxonomy of variability realization techniques ». In : *Software : Practice and experience* 35.8 (2005), p. 705-754.

[94] *Swift.org - Documentation*. 1996. URL : https://www.swift.org/documentation/ (visité le 28/02/2023).

[95] *Systems modeling language - Wikipedia*. 2001. URL : https://en.wikipedia.org/wiki/Systems_modeling_language (visité le 22/05/2023).

[96] Xhevahire TËRNAVA et Philippe COLLET. « On the diversity of capturing variability at the implementation level ». In : *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. 2017, p. 81-88.

[97] *TOML : Tom's Obvious Minimal Language*. 2015. URL : https://toml.io/en/ (visité le 11/05/2023).

[98] Kim W TRACY. « Mobile application development experiences on Apple's iOS and Android OS ». In : *IEEE Potentials* 31.4 (2012), p. 30-34.

[99] *UIActivityViewController | Apple Developer Documentation*. 1987. URL : https://developer.apple.com/documentation/uikit/uiactivityviewcontroller (visité le 14/03/2023).

[100] *Unified Modeling Language - Wikipedia*. 2001. URL : https://en.wikipedia.org/wiki/Unified_Modeling_Language (visité le 22/05/2023).

[101] Antony UNWIN. « Why is data visualization important ? what is important in data visualization ? » In : *Harvard Data Science Review* 2.1 (2020), p. 1.

[102] *UserDefaults | Apple Developer Documentation*. 1987. URL : https://developer.apple.com/documentation/foundation/userdefaults (visité le 15/03/2023).

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

[104] Iris VESSEY. « Cognitive fit : A theory-based analysis of the graphs versus tables literature ». In : *Decision sciences* 22.2 (1991), p. 219-240.

[105] Markus VOLTER. « DSLs for Product Lines : Approaches, Tools, Experiences ». In : *2011 15th International Software Product Line Conference*. IEEE. 2011, p. 353-353.

[106] *Welcome to Lark's documentation ! — Lark documentation*. 2014. URL : https://lark-parser.readthedocs.io/en/latest/ (visité le 24/02/2023).

[107] Danny WEYNS et al. « Claims and supporting evidence for self-adaptive systems : A literature study ». In : *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2012, p. 89-98.

[108] *What's New in Swift - WWDC19 - Videos - Apple Developer*. 1987. URL : https://developer.apple.com/videos/play/wwdc2019/402/ (visité le 28/02/2023).

[109] *xcode - #ifdef replacement in the Swift language - Stack Overflow*. 2014. URL : https://stackoverflow.com/questions/24003291/ifdef-replacement-in-the-swift-language (visité le 28/02/2023).

[110] Tewfik ZIADI, Loïc HÉLOUËT et Jean-Marc JÉZÉQUEL. « Towards a UML profile for software product lines ». In : *PFE* 3 (2003), p. 129-139.

Annexe A

Analyse du Domaine

A.1 Conversion du Métamodèle Annoté

La conversion du métamodèle annoté en diagramme de caractéristiques s'effectue donc à l'aide des primitives τ_e(E : Type d'entité) et τ_a(F : Feature, E : Type d'entité, A : Attribut) définies dans la section 5.2, *Modélisation des Features*, de l'ouvrage syllabus [31]. Cependant, celles-ci sont définies pour des modèles *Entité-Association*, alors que le métamodèle annoté présenté par la figure 4.2 dans le chapitre 4 est réalisé selon le formalisme *UML*. Dès lors, les précisions suivantes s'imposent :

— En *UML*, l'emplacement des cardinalités liées aux rôles sont inversées par rapport à leur emplacement en *Entité-Association*. Ainsi, lorsque la primitive τ_e définit la règle suivante :

$$\frac{E\text{–}\alpha(m_\alpha, n_\alpha)\text{–}R\text{–}\beta(m_\beta, n_\beta)\text{–}F}{F_E\text{–}[m_\alpha, n_\alpha]\text{–}G\text{–}(1,1)\text{–}F_F}$$

avec α et β des rôles et où $F_F = \tau_e(F)$, entre autres, la règle devient alors :

$$\frac{E\text{–}\alpha(m_\alpha, n_\alpha)\text{–}R\text{–}\beta(m_\beta, n_\beta)\text{–}F}{F_E\text{–}[m_\beta, n_\beta]\text{–}G\text{–}(1,1)\text{–}F_F}$$

En d'autres termes, les cardinalités définies sur le diagramme de caractéristiques sont les cardinalités du rôle β en *UML*, et celles du rôle α en *Entité-Association* ;

— La sémantique des relations de spécialisation *is-a* en *Entité-Association* est considérée comme équivalente à la sémantique d'héritage en *UML* exprimée au moyen de note. Dans le cas du métamodèle annoté de la figure 4.2, ces notes se trouvent dans le paragraphe explicatif associé ;

— La règle de conversion pour les relations de spécialisation *is-a* de type T est modifiée. En effet, la sémantique recherchée est celle d'une construction *OR*. Par conséquent, la cardinalité associée à celle-ci devrait être *1-N* et non *0-N*, car cette dernière permet de ne choisir aucune caractéristique associée.

Ces précisions permettent de garantir, dans le présent cas, que deux modèles sémantiquement équivalents, le premier en *UML* et le second en *Entité-Association*, produisent le même diagramme de caractéristiques. Cependant, des métamodèles plus complexes nécessiteront sans nul doute des précisions plus nombreuses, mais cette considération est hors de portée de ce document et ne sera pas approfondie.

A.2 Diagrammes

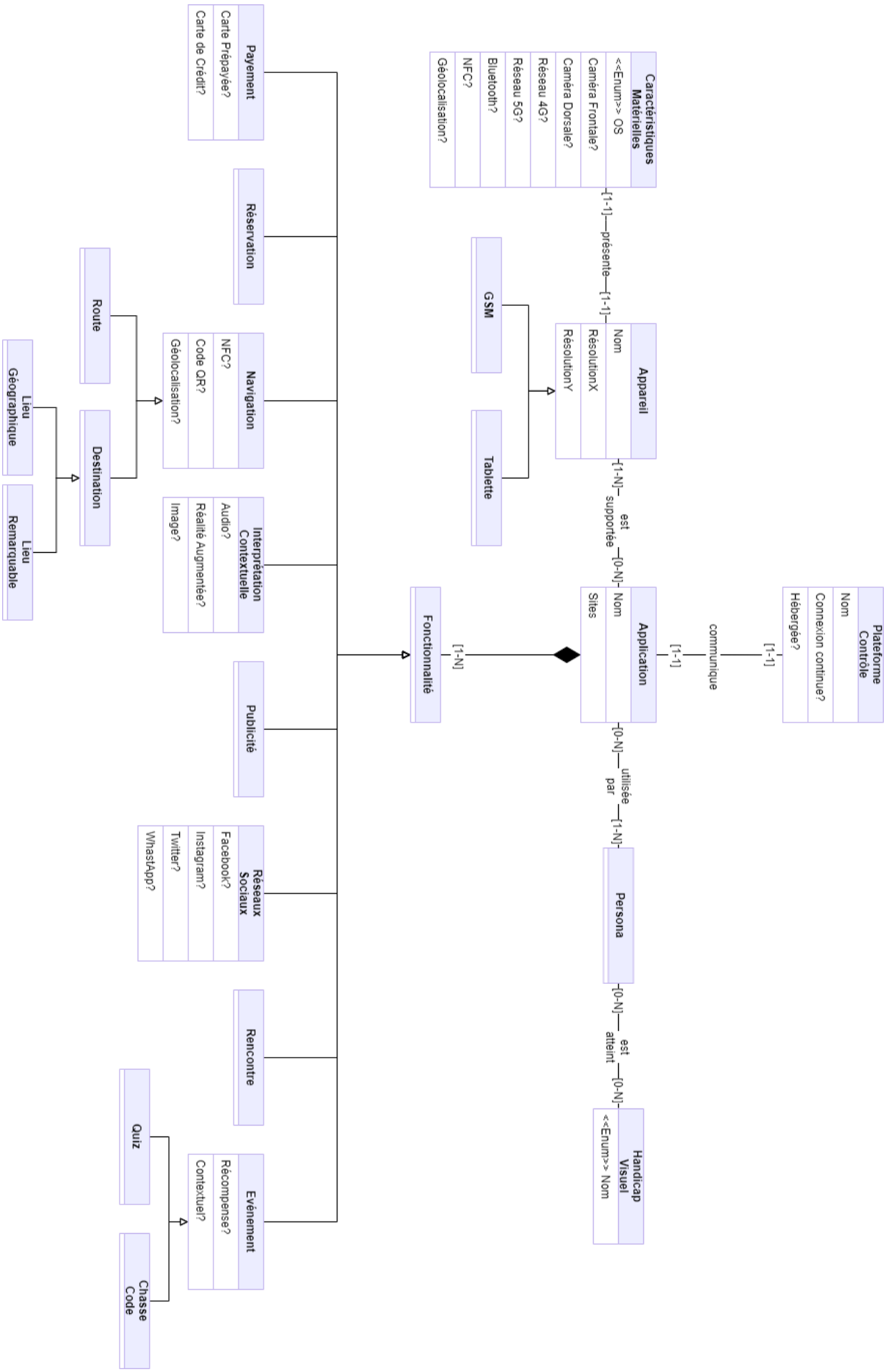


FIGURE A.1 – Métamodèle du domaine d'application - notation *UML* simplifiée

FIGURE A.2 – Métamodèle annoté du domaine d'application - notation *UML* simplifiée

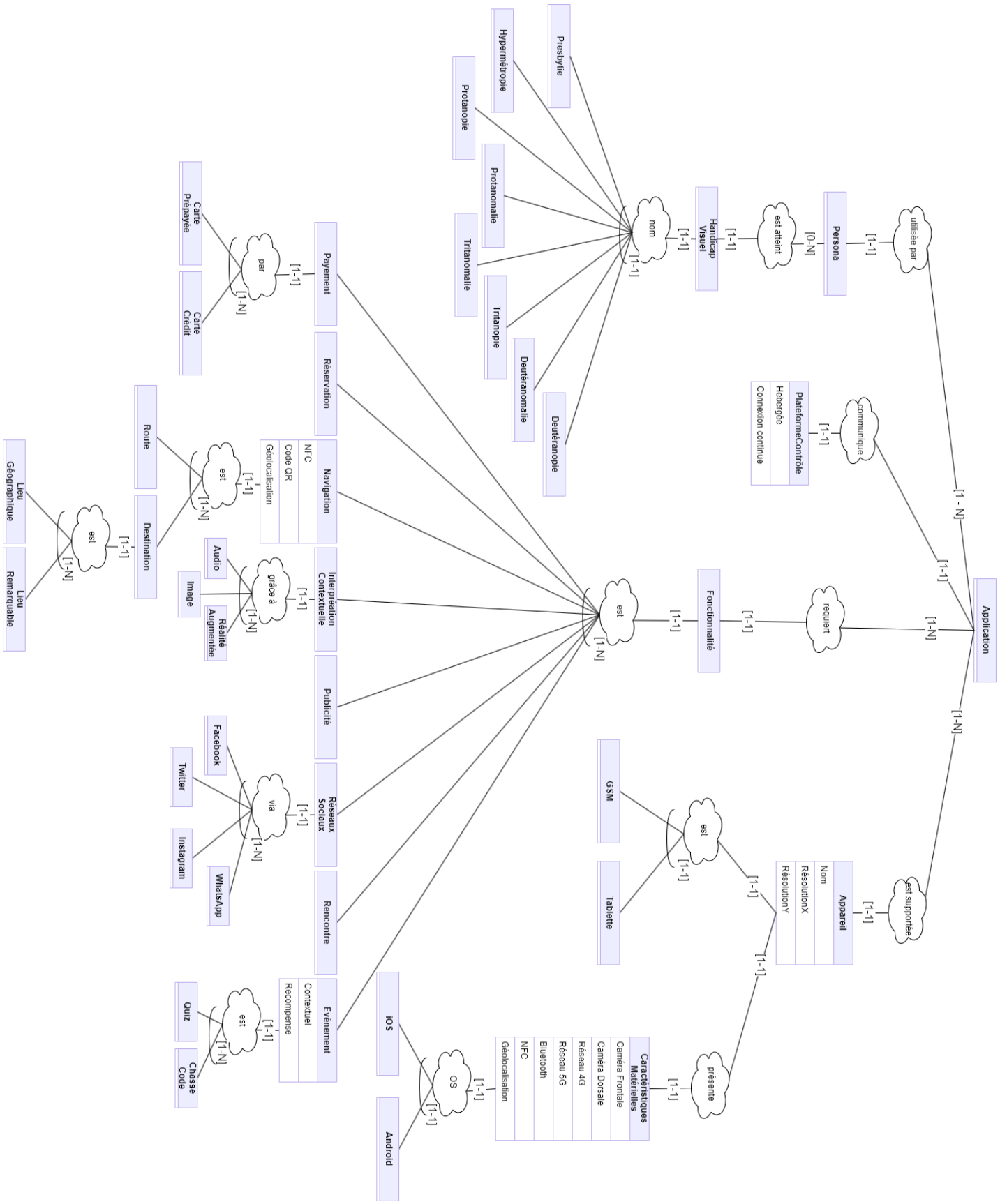


FIGURE A.3 – Diagramme de caractéristiques dérivé du métamodèle 4.2

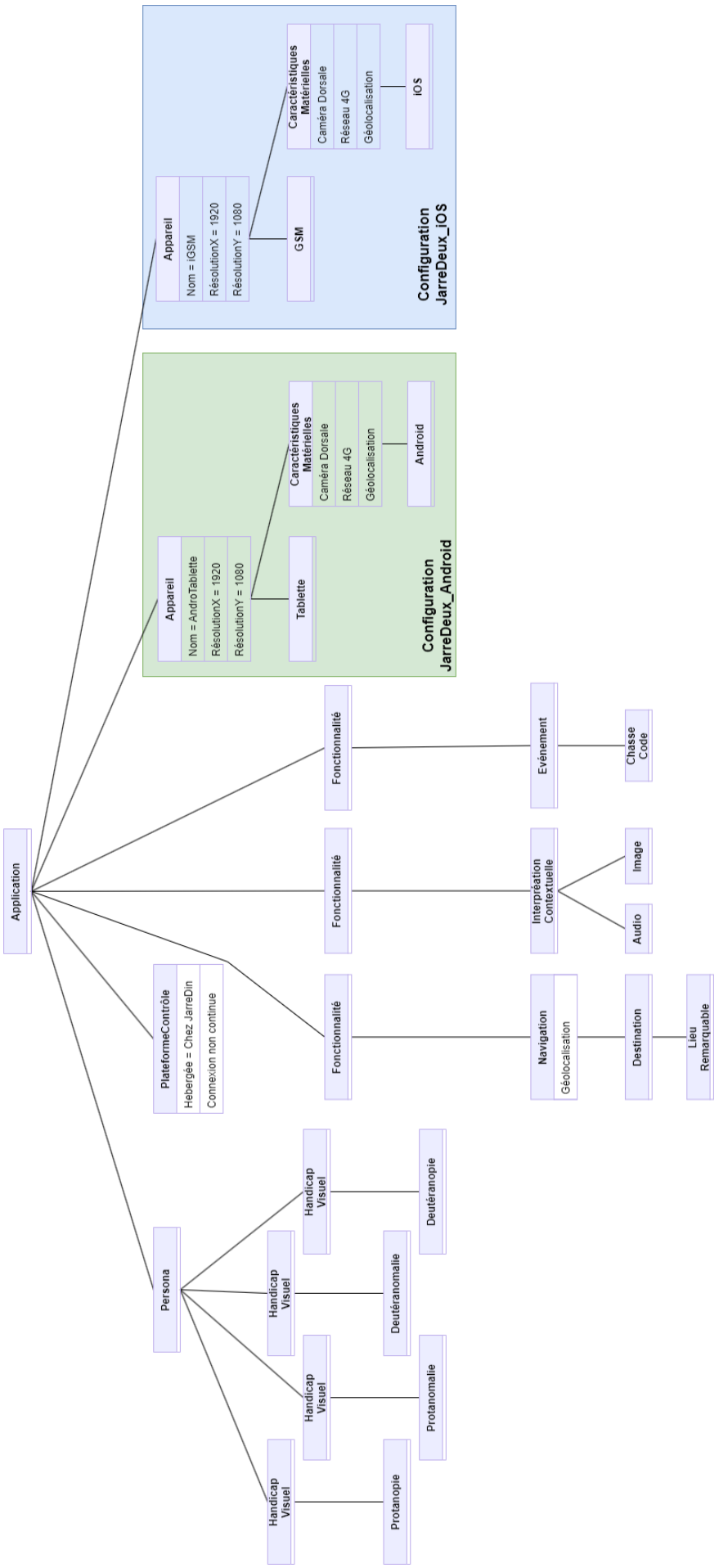


FIGURE A.4 – Configurations du logiciel *JarreDeux*, conformes au diagramme de caractéristiques 4.6

A.3 Modèle du logiciel *JarreDeux*

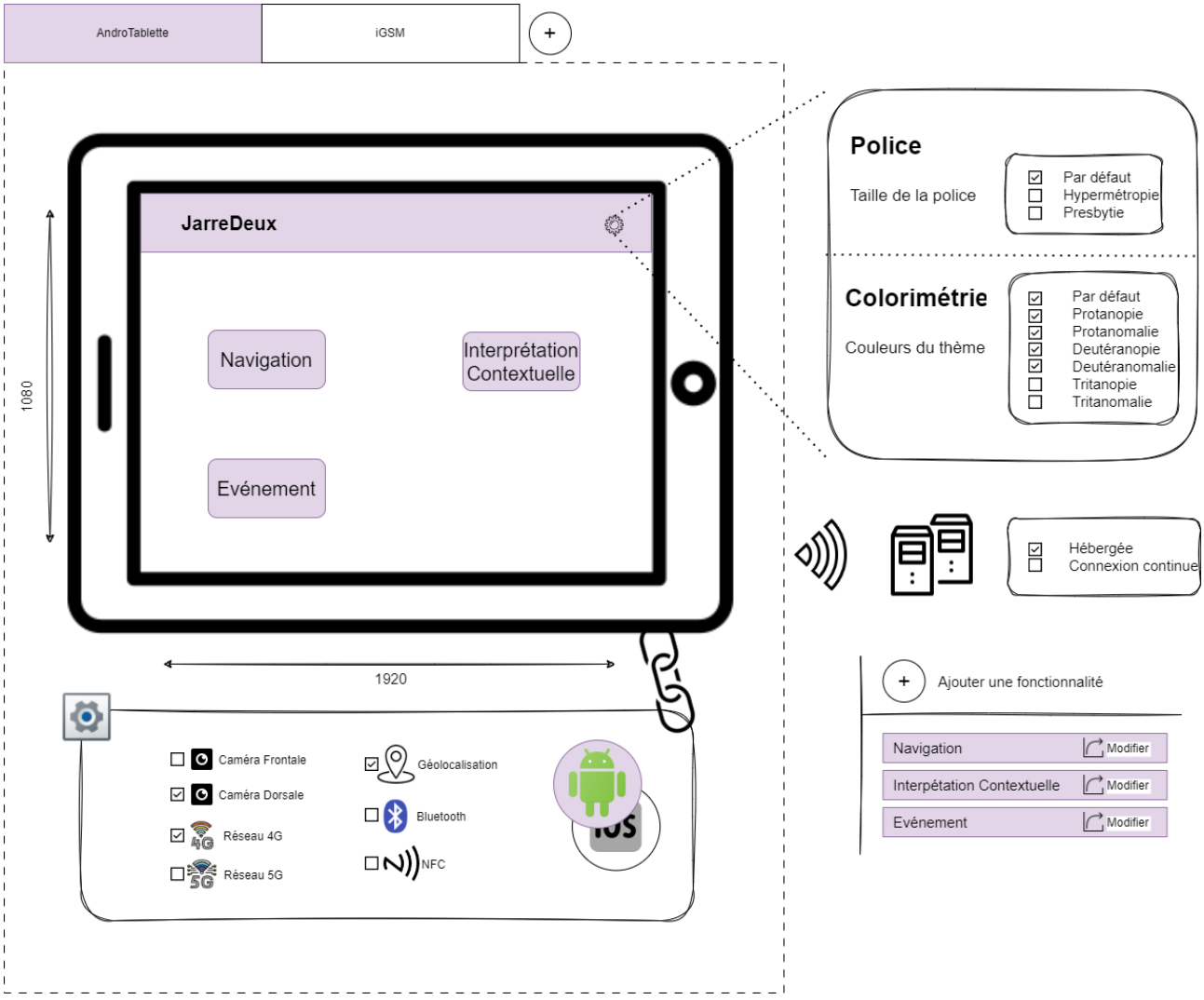


FIGURE A.5 – Appareil *AndroTablette*, ses caractéristiques matérielles et les caractéristiques communes

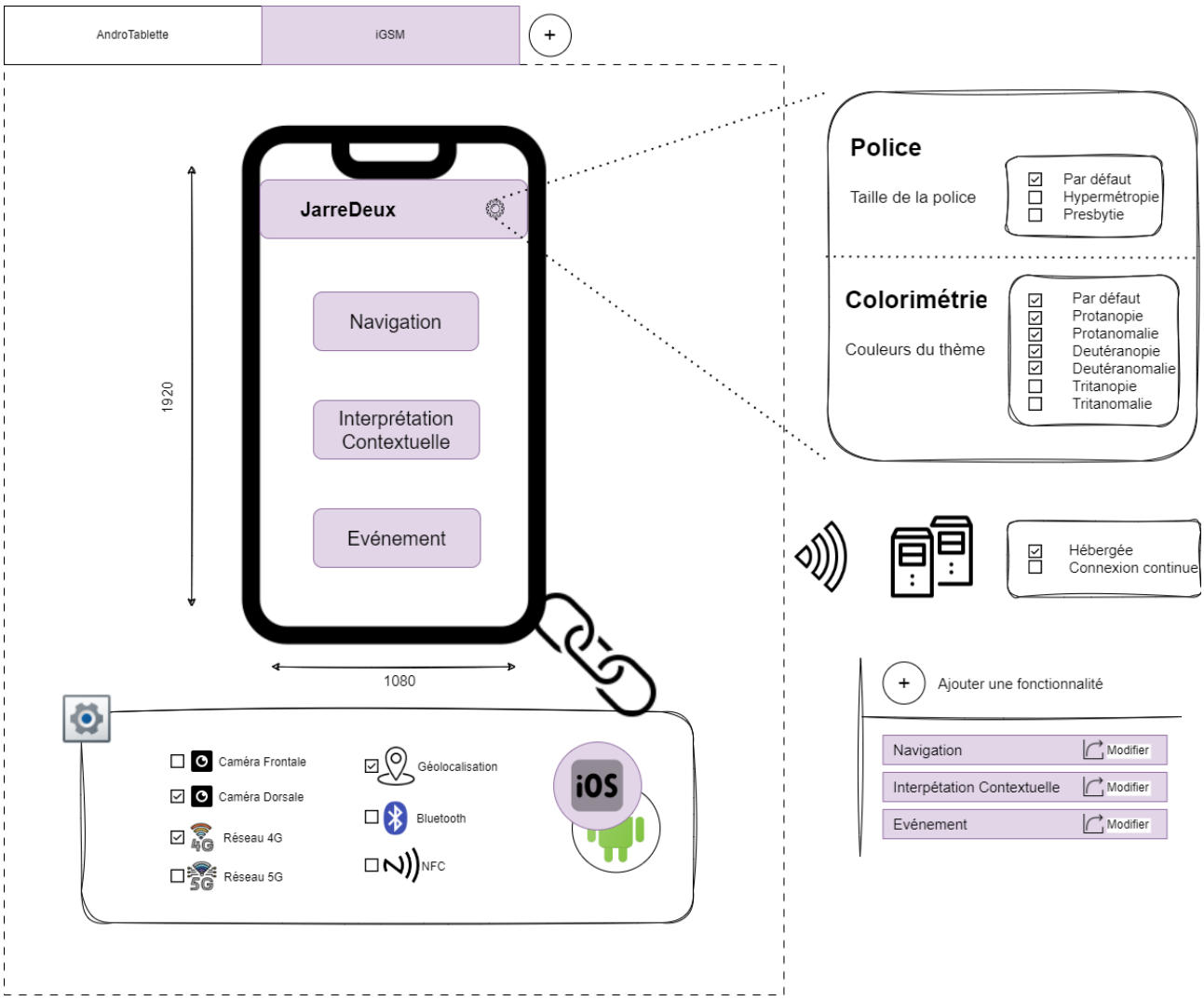


FIGURE A.6 – Appareil *iGSM*, ses caractéristiques matérielles et les caractéristiques communes

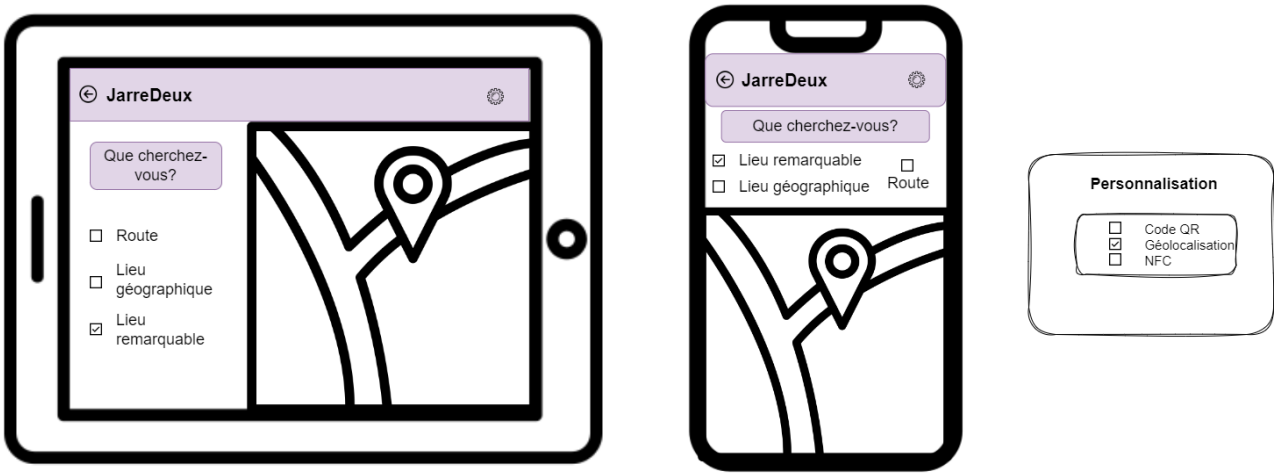


FIGURE A.7 – Personnalisation de la fonctionnalité *Navigation*. La boîte de dialogue *Personnalisation* permet de sélectionner les méthodes de localisation de l'appareil et n'appartiennent donc pas aux interfaces graphiques du logiciel.

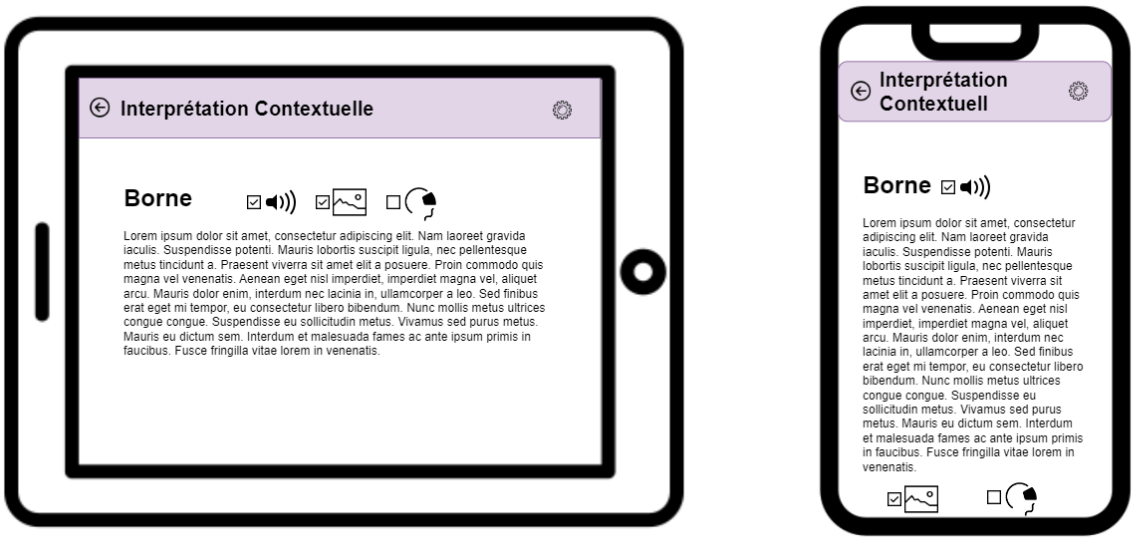


FIGURE A.8 – Personnalisation de la fonctionnalité *Interprétation contextuelle*

FIGURE A.9 – Personnalisation de la fonctionnalité *Evénement*. La boîte de dialogue *Personnalisation* permet de sélectionner les paramètres à prendre en charge et n'appartiennent donc pas aux interfaces graphiques du logiciel. Les événements sélectionnés font apparaître des boutons supplémentaires sur les interfaces.

A.4 Langage FeatAll

```
def BetterSoft() {
    featureModel("BetterSoft") {
        root("Application") {
            feature("PlateformeControle", Multiplicity.one) {
                bindingTime = Instants.design..Instants..compile
                activateTime = Instants.program..Instants..compile
                attribute(Types.boolean, "Hebergee")
                attribute(Types.boolean, "ConnexionContinue")
            }
            feature("Appareil", Multiplicity.atLeastOne) {
                bindingTime = Instants.analysis..Instants.compile
                activateTime = Instants.runtimeRun
                attribute(Types.string, "Nom")
                attribute(Types.integer, "ResolutionX")
                attribute(Types.integer, "ResolutionY")
                group(Multiplicity.one, "est", Multiplicity.one) {
                    isOpened()
                    bindingTime = Instants.analysis..Instants.compile
                    activateTime = Instants.runtimeRun
                    feature("GSM")
                    feature("Tablette")
                }
                feature("CaracteristiquesMaterielles", Multiplicity.one) {
                    bindingTime = Instants.design..Instants.compile
                    activateTime = Instants.design..Instants.compile
                    attribute(Types.boolean, "CameraFrontale")
                    attribute(Types.boolean, "CameraDorsale")
                    attribute(Types.boolean, "Reseau4G")
                    attribute(Types.boolean, "Reseau5G")
                    attribute(Types.boolean, "Bluetooth")
                    attribute(Types.boolean, "NFC")
                    attribute(Types.boolean, "Geolocalisation")
                    group(Multiplicity.one, "OS", Multiplicity.one) {
                        bindingTime = Instants.design..Instants.compile
                        activateTime = Instants.design..Instants.compile
                        feature("Android")
                        feature("iOS")
                    }
                }
            }
            feature("Persona", Multiplicity.atLeastOne) {
                bindingTime = Instants.analysis..Instants.compile
                activateTime = Instants.runtimeRun
                feature("HandicapVisuel", Multiplicity.any) {
                    bindingTime = Instants.analysis..Instants.compile
                    activateTime = Instants.runtimeRun
                    group(Multiplicity.one, "nom", Multiplicity.one) {
                        isOpened()
                        bindingTime = Instants.analysis..Instants.compile
                        activateTime = Instants.runtimeRun
                        feature("Presbytie")
                        feature("Hypermetropie")
                        feature("Protanopie")
                        feature("Protanomalie")
                        feature("Deuteranopie")
                        feature("Deuteranomalie")
                        feature("Tritanopie")
                        feature("Tritanomalie")
                    }
                }
            }
            feature("Fonctionnalite", Multiplicity.atLeastOne) {
                bindingTime = Instants.analysis..Instants.runtimeRun
                activateTime = Instants.analysis..Instants.runtimeRun
                group(Multiplicity.one, "est", Multiplicity.one) {
                    isOpened()
                    bindingTime = Instants.analysis..Instants.runtimeRun
                    activateTime = Instants.analysis..Instants.runtimeRun
                    feature("Payement") {
                        bindingTime = Instants.program..Instants.compile
                        activateTime = Instants.program..Instants.compile
                        group(Multiplicity.one, "par", Multiplicity.atLeastOne) {
                            isOpened()
                            bindingTime = Instants.program..Instants.compile
                            activateTime = Instants.runtimeRun
                            feature("CartePrepayee")
                            feature("CarteCredit")
```

Listing A.1 – Extrait de la représentation du diagramme de caractéristiques sous forme textuelle en *Groovy*, grâce au langage FeatAll. Un paramètre de cardinalité non pris en charge par la syntaxe classique est ajouté à la fonction *feature()*. Ceci permet d'exprimer des fonctionnalités qui peuvent être sélectionnées un nombre arbitraire de fois. Ce paramètre devient implicite lorsque la *feature()* est directement comprise dans un *group()*, car la cardinalité est déjà donnée par ledit *group()*.

```
def JarreDeux() {

    configuration("JarreDeux_Android", "BetterSoft") {
        root {
            feature("PlateformeControle") {
                Hebergee = false
                ConnexionContinue = false
            }
            feature("Appareil") {
                Nom = "AndroTablette" // Config "JarreDeux_iOS" :: Nom = "iGSM"
                ResolutionX = 1920
                ResolutionY = 1080
                group("est") {
                    feature("Tablette") // Config "JarreDeux_iOS" :: feature("GSM")
                }
                feature("CaracteristiquesMaterielles") {
                    CameraFrontale = false
                    CameraDorsale = true
                    Reseau4G = true
                    Reseau5G = false
                    Bluetooth = false
                    NFC = false
                    Geolocalisation = true
                    group("OS") {
                        feature("Android") // Config "JarreDeux_iOS" :: feature("iOS")
                    }
                }
            }
            feature("Persona") {
                feature("HandicapVisuel") {
                    group("nom") {
                        feature("Protanopie")
                        feature("Protanomalie")
                        feature("Deuteranopie")
                        feature("Deuteranomalie")
                    }
                }
            }
            feature("Fonctionnalite") {
                group("est") {
                    feature("Navigation") {
                        NFC = false
                        CodeQR = false
                        Geolocalisation = true
                        group("est") {
                            feature("Destination") {
                                group("est") {
                                    feature("LieuRemarquable")
                                }
                            }
                        }
                    }
                    feature("InterpretationContextuelle") {
                        group("gracea") {
                            feature("Audio")
                            feature("Image")
                        }
                    }
                    feature("Evenement") {
                        Contextuel = false
                        Recompense = false
                        group("est") {
                            feature("ChasseCode")
                        }
                    }
                }
            }
        }
    }
}
```

Listing A.2 – Représentation des configurations du logiciel *JarreDeux* sous forme textuelle en *Groovy*, grâce au langage FeatAll. La configuration *JarreDeux_Android* est exprimée en entière. La configuration *JarreDeux_iOS* étant pratiquement identique à la précédente, seules les variations y sont indiquées sous forme de commentaires - //. Certaines constructions ont être refactorisées, mais les configurations restent conformes au diagramme de caractéristiques annoté exprimé par le listing A.1. Enfin, les phases de sélection et d'activation ne sont pas mentionnées car elles sont identiques au diagramme de caractéristiques annoté.

Annexe B

Conception du Domaine

B.1 Arborescence de l'Architecture Commune

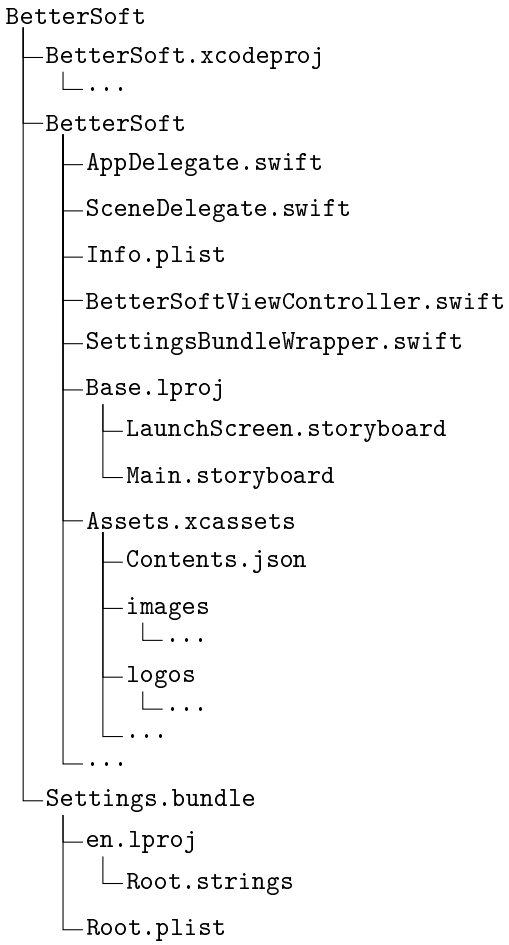


FIGURE B.1 – Architecture commune pour les configurations *iOS*

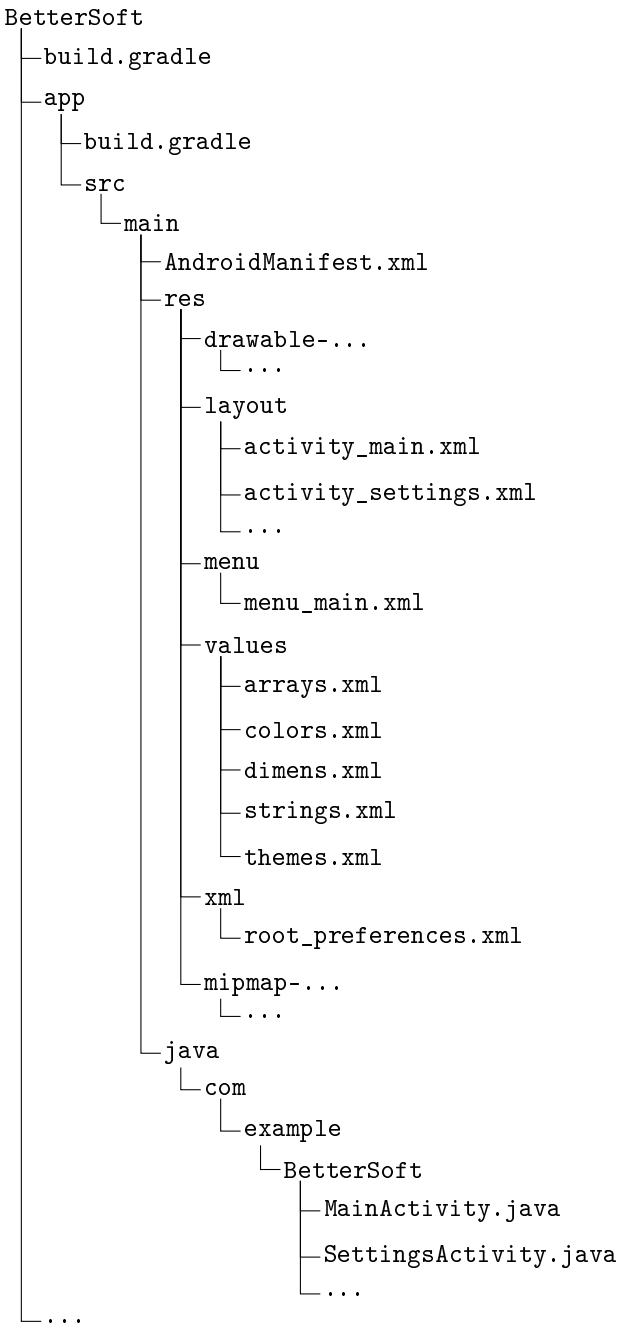


FIGURE B.2 – Architecture commune pour les configurations *Android*

B.2 Storyboard

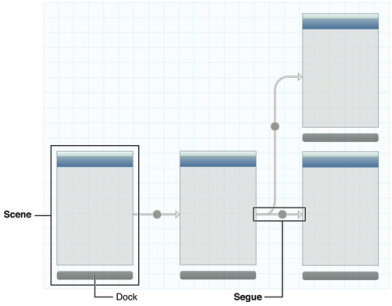


FIGURE B.3 – Fichier « storyboard » tel que montré par l'IDE *XCode* [90]

B.3 Langage TacticType

```python
from concurrent.futures import ThreadPoolExecutor
import logging

from dataclasses import dataclass
from os.path import exists
from pathlib import Path
from shutil import copytree

from src.tactics.values import BindingTimes, Order


logger = logging.getLogger('CLI')


@dataclass
class Res:
    """Represents a resource to add to the base project structure
    """
    src: str | Path
    dst: str | Path


class AddRes:
    """Provides a method to add a `Res` to the base project structure
    """

    binding_time = BindingTimes.COMPILE

    @staticmethod
    def apply(*res: Res, order: Order, substitute: bool = False) -> None:
        """Appends the provided `*res` to the base project structure

        :param res: resource to append. Multiple `res` can be provided
        :type res: Res
        :param order: order in which to apply the `*res`
        :type order: Order
        :param substitute: overwriting mode, defaults to False
        :type substitute: bool, optional
        """

        # User has to be careful when providing Order.PARALLEL
        # So that no resource is overlapping
        if order == Order.PARALLEL:
            with ThreadPoolExecutor(max_workers=4) as executor:
                # No result
                _ = [executor.submit(AddRes._worker, r, substitute)
                     for r in res]

        elif order == Order.SEQUENTIAL:
            for r in res:
                AddRes._worker(r, substitute)

        # No else statement
        # Because all values of Order have to be explicitly considered
        # For clarity reasons

    @staticmethod
    def _worker(res: Res, substitute: bool) -> None:
        """Appends one `res` to the base project structure

        :param res:  resource to append
        :type res: Res
        :param substitute: overwriting mode
        :type substitute: bool
        """
        if not exists(res.src):
            logger.info(f'Skipping res {res.src} as it does not exist')
            return

        if exists(res.dst) and not substitute:
            logger.info(f'Skipping res {res.dst} as it already exists')
            return

        copytree(res.src, res.dst, dirs_exist_ok=substitute)
```

Listing B.1 – Implémentation du type de tactique abstrait « Ajout de ressources » en appelant la méthode « AddRes.apply() ». Celui-ci peut être employé pour implémenter les tactiques « Ajout de fichiers » et « Substitution de composants » grâce au paramètre « substitute » permettant d'écraser ou non une ressource. L'ordonnancement des ressources à ajouter est quant à lui déterminé par le paramètre « order ».

```python
from concurrent.futures import ThreadPoolExecutor
import logging
import os

from dataclasses import dataclass
from pathlib import Path
from typing import Iterable

from yaml import safe_load

from src.tactics.values import BindingTimes, Order, FileAlterMode


logger = logging.getLogger('CLI')


@dataclass
class Patch:
    """Represents a patch file that indicates how to alter specific files
    of the base project stucture
    """

    yaml_src: str | Path
    base_target: str | Path
    skip_qualifier: Iterable[str]


class PatchingSub:
    """Provides a method to apply a `Patch` to the base project structure
    """

    binding_time = BindingTimes.COMPILE

    @staticmethod
    def apply(patch: Patch) -> None:
        """Applies the modifications provided by the `Patch`

        :param patch: patch to apply
        :type patch: Patch
        """

        # We assume the patch is always well formed
        with open(patch.yaml_src, 'r') as p:
            content = safe_load(p)['tactic']

        # Fetching metadata
        start: str = content['start']
        end: str = content['end']
        sepsign: str = content['sepsign']
        variants: list[dict] = content['variants']
        variants_order = Order(content['variants order'])

        # For each variant defined in the YAML file,
        # Substract content as indicated

        # User has to be careful when providing Order.PARALLEL
        # So that no resource is overlapping
        if variants_order == Order.PARALLEL:
            with ThreadPoolExecutor(max_workers=4) as executor:
                _ = [executor.submit(
                    PatchingSub._worker_variants(
                        patch=patch,
                        current_variant=variant,
                        start_guard=start + sepsign + variant['qualifier'],
                        end_guard=end
                    ))
                    for variant in variants if variant not in patch.skip_qualifier]

        elif variants_order == Order.SEQUENTIAL:

            for variant in variants:

                if variant['qualifier'] in patch.skip_qualifier:
                    continue

                PatchingSub._worker_variants(
                    patch=patch,
                    current_variant=variant,
                    start_guard=start + sepsign + variant['qualifier'],
                    end_guard=end
                )

        # No else statement
        # Because all values of Order have to be explicitly considered
        # For clarity reasons

    @staticmethod
    def _worker_variants(*, patch: Patch, current_variant: dict[str], start_guard: str, end_guard:
```

```
        str) -> None:
89          """Applies all the specified modifications to a specific variant
90
91          :param patch: patch to apply
92          :type patch: Patch
93          :param current_variant: current varitant to modify
94          :type current_variant: dict[str]
95          :param start_guard: guard that indicated the start of a potential area to modify in a
                specific file
96          :type start_guard: str
97          :param end_guard: guard that indicated the end of a potential area to modify in a specific
                file
98          :type end_guard: str
99          """
100
101         files_order = Order(current_variant['files order'])
102
103         # User has to be careful when providing Order.PARALLEL
104         # So that no resource is overlapping
105         if files_order == Order.PARALLEL:
106             with ThreadPoolExecutor(max_workers=2) as executor:
107                 _ = [executor.submit(
108                     PatchingSub._worker_files(
109                         patch=patch,
110                         res_file=res_file,
111                         start_guard=start_guard,
112                         end_guard=end_guard
113                     ))
114                     for res_file in current_variant['files']]
115
116         elif files_order == Order.SEQUENTIAL:
117             for res_file in current_variant['files']:
118
119                 PatchingSub._worker_files(
120                     patch=patch,
121                     res_file=res_file,
122                     start_guard=start_guard,
123                     end_guard=end_guard
124                 )
125
126         # No else statement
127         # Because all values of Order have to be explicitly considered
128         # For clarity reasons
129
130     @staticmethod
131     def _worker_files(*, patch: Patch, res_file: dict[str], start_guard: str, end_guard: str) -> None
            :
132         """Applies all the provided modifications to a specific file
133
134         :param patch: patch to apply
135         :type patch: Patch
136         :param res_file: current file resource
137         :type res_file: dict[str]
138         :param start_guard: guard that indicated the start of a potential area to modify in a
                specific file
139         :type start_guard: str
140         :param end_guard: guard that indicated the end of a potential area to modify in a specific
                file
141         :type end_guard: str
142         """
143         full_path = os.path.join(patch.base_target, res_file['in'])
144         mode = FileAlterMode(res_file['mode'])
145
146         # Switch on indicated mode
147         if mode == FileAlterMode.ALTER:
148             PatchingSub._alter(
149                 full_path=full_path,
150                 start_guard=start_guard,
151                 end_guard=end_guard
152             )
153         elif mode == FileAlterMode.DELETE:
154             if not os.path.exists(full_path):
155                 logger.warning(
156                     f'Cannot delete {full_path} as it does not exist. May have happened because it
                        was not selected in the configuration.')
157                 return
158             os.remove(full_path)
159
160         # No else statement
161         # Because all values of Order have to be explicitly considered
162         # For clarity reasons
```

Listing B.2 – Extrait de l'implémentation du type de tactique abstrait « Patching par soustraction ». Les opérations à appliquer sont indiquées dans le fichier *YAML* référencé dans la classe de données « Patch »

```yaml
version: 1

tactic:
  type: substraction
  start: "@TACTIC('Adapt6b')"
  end: "@END_TACTIC"
  sepsign: " = "
  variants order: sequential
  variants:

    - variant:
      qualifier: PROTANOPIE
      files order: parallel
      files:
        - file:
          mode: alter
          in: app/src/main/java/com/example/bettersoft/TVisualHandicap.java
        - file:
          mode: delete
          in: app/src/main/java/com/example/bettersoft/strategies/Filters/
              ProtanopieFilter.java
        - file:
          mode: alter
          in: app/src/main/res/values/arrays.xml
        - file:
          mode: alter
          in: app/src/main/res/values/themes.xml

    - variant:
      qualifier: PROTANOMALIE
      files order: parallel
      files:
        - file:
          mode: alter
          in: app/src/main/java/com/example/bettersoft/TVisualHandicap.java
        - file:
          mode: delete
          in: app/src/main/java/com/example/bettersoft/strategies/Filters/
              ProtanomalieFilter.java
        - file:
          mode: alter
          in: app/src/main/res/values/arrays.xml
        - file:
          mode: alter
          in: app/src/main/res/values/themes.xml

    - variant:
      qualifier: DEUTERANOPIE
      files order: parallel
      files:
        - file:
          mode: alter
          in: app/src/main/java/com/example/bettersoft/TVisualHandicap.java
        - file:
          mode: delete
          in: app/src/main/java/com/example/bettersoft/strategies/Filters/
              DeuteranopieFilter.java
        - file:
          mode: alter
          in: app/src/main/res/values/arrays.xml
        - file:
          mode: alter
          in: app/src/main/res/values/themes.xml

    - variant:
      qualifier: DEUTERANOMALIE
      files order: parallel
      files:
        - file:
          mode: alter
          in: app/src/main/java/com/example/bettersoft/TVisualHandicap.java
        - file:
          mode: delete
          in: app/src/main/java/com/example/bettersoft/strategies/Filters/
              DeuteranomalieFilter.java
        - file:
          mode: alter
          in: app/src/main/res/values/arrays.xml
        - file:
          mode: alter
          in: app/src/main/res/values/themes.xml
```

Listing B.3 – Extrait d'un exemple de fichier de patching par soustraction au format *YAML* pour la plateforme *Android*. Cet exemple est lié à la tactique (Adapt·6b).

B.4 Diagramme de Caractéristiques Annoté

FIGURE B.4 – Code QR renvoyant à une version WEB interactive du diagramme de caractéristiques annoté. Si le Code QR ne peut être scanné, le lien vers la page WEB correspondante est le suivant : « https://pierrejacobs.github.io ».

Annexe C

Implémentation du Domaine

C.1 Composants Techniques

```
[persona.handicap_visuel.nom.police]
mapping = { Hypermetropie = 'HYPERMETROPIE', Presbytie = 'PRESBYTIE' }

[persona.handicap_visuel.nom.couleurs]
mapping = { Protanopie = 'PROTANOPIE', Protanomalie = 'PROTANOMALIE', Deuteranopie = 'DEUTERANOPIE',
      Deuteranomalie = 'DEUTERANOMALIE',  Tritanopie = 'TRITANOPIE', Tritanomalie = 'TRITANOMALIE' }

[appareil]
xml_name = 'Appareil'
resolution_x = 'ResolutionX'
resolution_y = 'ResolutionY'

[appareil.caracteristiques_materielles]
xml_name = 'CaracteristiquesMaterielles'
mapping = { CameraFrontale = 'CAMERA', CameraDorsale = 'CAMERA', Reseau4G = 'INTERNET', Reseau5G = '
    INTERNET', Geolocalisation = 'GEOLOCALISATION', Bluetooth = 'BLUETOOTH', NFC = 'NFC', CodeQR = '
    CODE QR' }

[appareil.est.android]
# Layout qualifiers
mapping = { Tablette = '-w600dp', GSM = '' }

[fonctionnalite.est]
mapping = { InterpretationContextuelle = 'INTERPRETATION CONTEXTUELLE', Navigation = 'NAVIGATION',
    Evenement = 'EVENEMENT', Payement = 'PAYEMENT', Reservation = 'RESERVATION', Publicite = '
    PUBLICITE', ReseauxSociaux = 'RESEAUX SOCIAUX', Rencontre = 'RENCONTRE' }

[fonctionnalite.est.navigation]
xml_name = 'Navigation'
mapping = { Geolocalisation = 'GEOLOCALISATION', NFC = 'NFC', CodeQR = 'CODE QR' }

[fonctionnalite.est.navigation.est]
mapping = { LieuRemarquable = 'LIEU REMARQUABLE', LieuGeographique = 'LIEU GEOGRAPHIQUE', Route = '
    LIEU PARCOURS' }

[fonctionnalite.est.interpretation_contextuelle.est]
mapping = { Audio = 'AUDIO', Image = 'IMAGE', RealiteAugmentee = 'REALITE AUGMENTEE' }
```

Listing C.1 – Mapping TOML requis afin de lier les balises XML et leurs attributs aux artefacts du code source à manipuler

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<configuration feature_model="BetterSoft" name="JarreDeux_Android">

    <root name="Application">

        <feature name="PlateformeControle">
            <attribute name="Hebergee" type="boolean">false</attribute>
            <attribute name="ConnexionContinue" type="boolean">false</attribute>
        </feature>

        <feature name="Appareil">
            <attribute name="Nom" type="string">AndroTablette</attribute>
            <attribute name="ResolutionX" type="integer">1920</attribute>
            <attribute name="ResolutionY" type="integer">1080</attribute>
            <group association_name="est">
                <feature name="Tablette"/>
            </group>
            <feature name="CaracteristiquesMaterielles">
                <attribute name="CameraFrontale" type="boolean">false</attribute>
                <attribute name="CameraDorsale" type="boolean">true</attribute>
                <attribute name="Reseau4G" type="boolean">true</attribute>
                <attribute name="Reseau5G" type="boolean">false</attribute>
                <attribute name="Bluetooth" type="boolean">false</attribute>
                <attribute name="NFC" type="boolean">false</attribute>
                <attribute name="Geolocalisation" type="boolean">true</attribute>
                <group association_name="OS">
                    <feature name="Android"/>
                </group>
            </feature>
        </feature>

        <feature name="Persona">
            <feature name="HandicapVisuel">
                <group association_name="nom">
                    <feature name="Protanopie"/>
                    <feature name="Protanomalie"/>
                    <feature name="Deuteranopie"/>
                    <feature name="Deuteranomalie"/>
                </group>
            </feature>
        </feature>

        <feature name="Fonctionnalite">
            <group association_name="est">
                <feature name="Navigation">
                    <attribute name="NFC" type="boolean">false</attribute>
                    <attribute name="CodeQR" type="boolean">false</attribute>
                    <attribute name="Geolocalisation" type="boolean">true</attribute>
                    <group association_name="est">
                        <feature name="Destination">
                            <group association_name="est">
                                <feature name="LieuRemarquable"/>
                            </group>
                        </feature>
                    </group>
                </feature>
                <feature name="InterpretationContextuelle">
                    <group association_name="gracea">
                        <feature name="Audio"/>
                        <feature name="Image"/>
                    </group>
                </feature>
                <feature name="Evenement">
                    <attribute name="Contextuel" type="boolean">false</attribute>
                    <attribute name="Recompense" type="boolean">false</attribute>
                    <group association_name="est">
                        <feature name="ChasseCode"/>
                    </group>
                </feature>
            </group>
        </feature>

    </root>

</configuration>
```

Listing C.2 – Représentation XML d'une configuration, exploitable par un générateur de logiciels. Cette configuration est directement dérivée de celle présentée par le listing A.2

C.2 Interface Graphique du logiciel *JarreDeux*

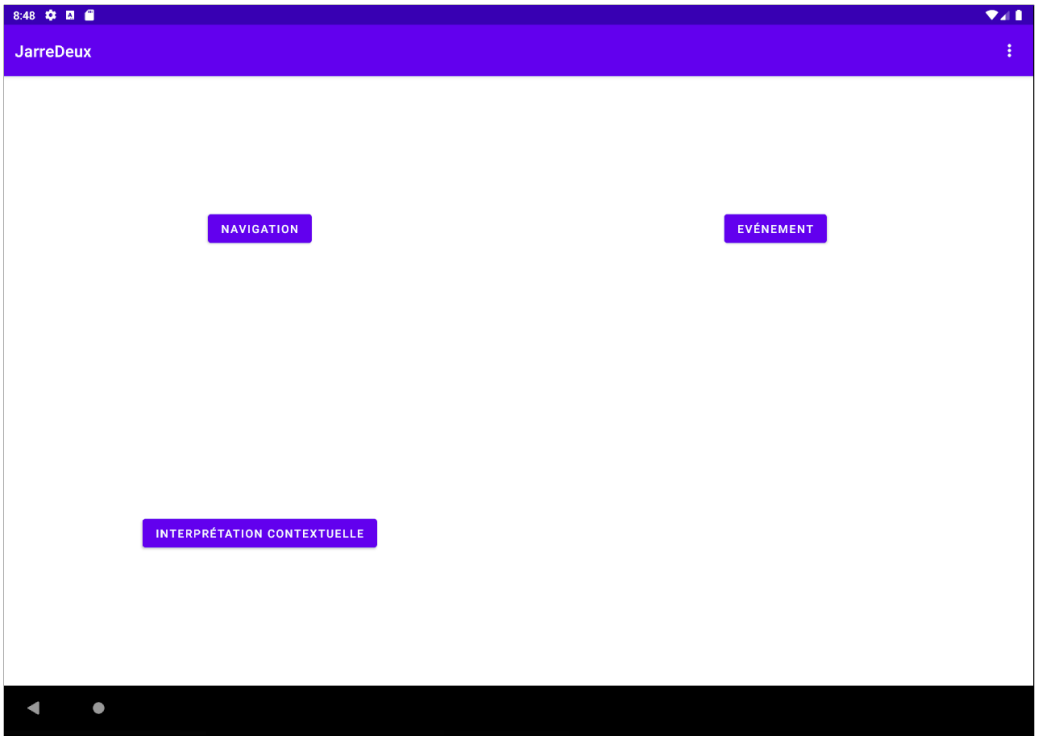


FIGURE C.1 – Capture d'écran de la page d'accueil du logiciel *JarreDeux* sur tablette. Cette mise en page est conforme à celle définie par le modèle du logiciel dans la figure A.5

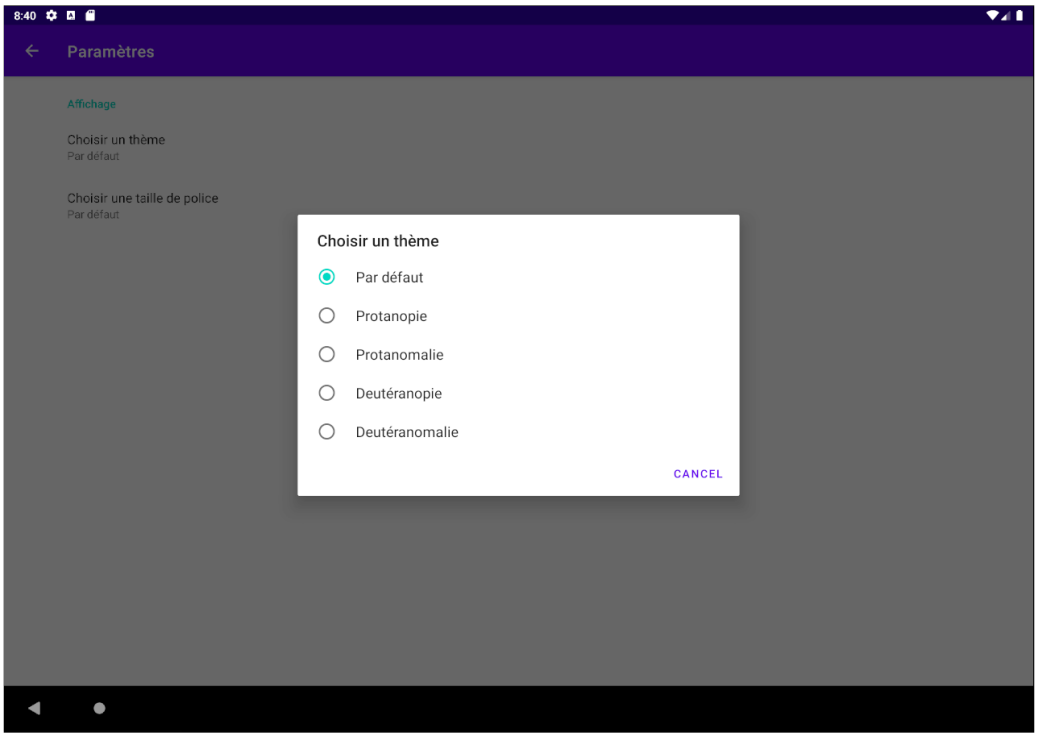


FIGURE C.2 – Capture d'écran des paramètres d'affichage du logiciel *JarreDeux* sur tablette. Cette mise en page est conforme à celle définie par le modèle du logiciel dans la figure A.5

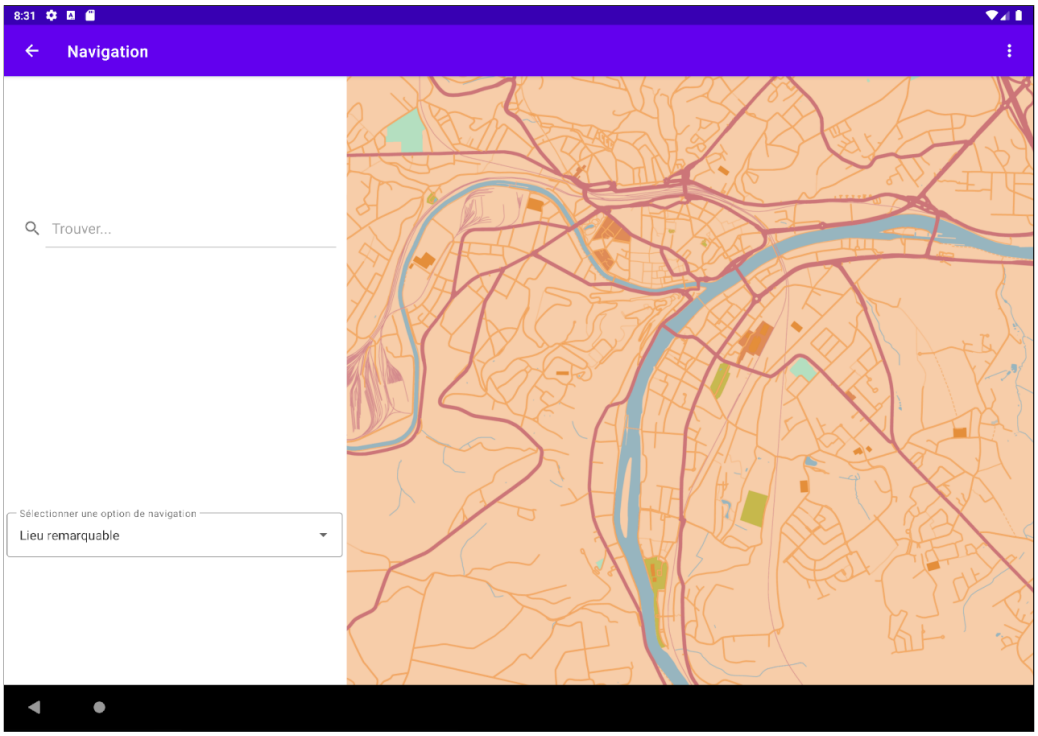


FIGURE C.3 – Capture d'écran de la page de navigation du logiciel *JarreDeux* sur tablette. Cette mise en page est conforme à celle définie par le modèle du logiciel dans la figure A.7

FIGURE C.4 – Capture d'écran de la page d'interprétation contextuelle du logiciel *JarreDeux* sur tablette. Cette mise en page est conforme à celle définie par le modèle du logiciel dans la figure A.8

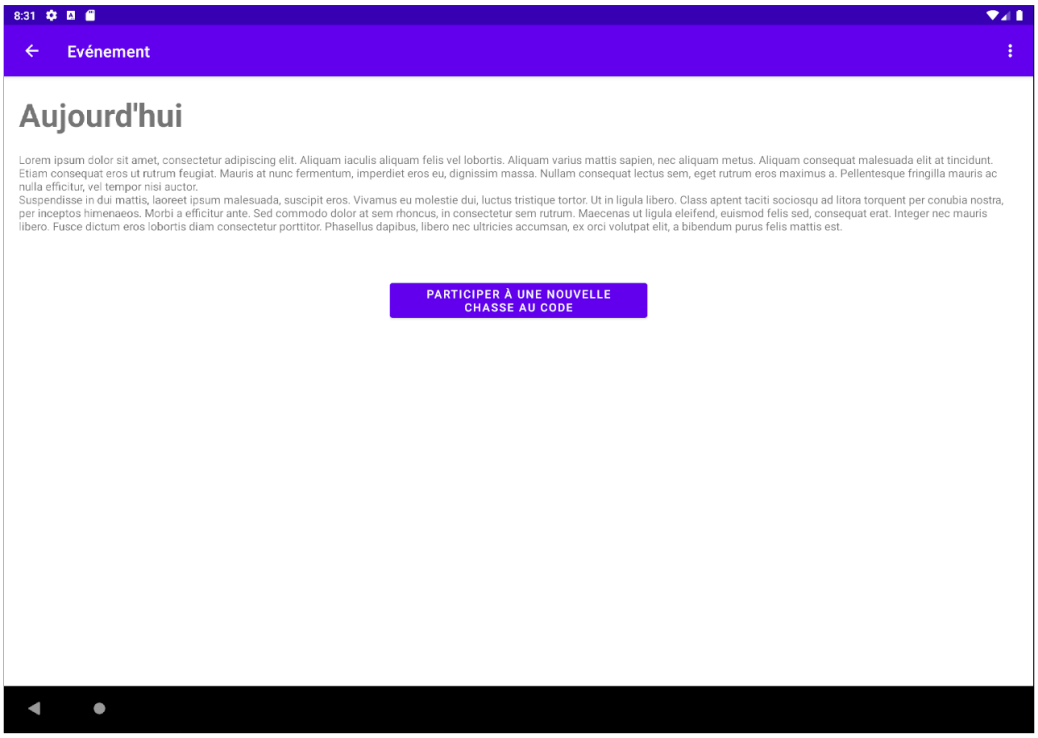


FIGURE C.5 – Capture d'écran de la page des événements du logiciel *JarreDeux* sur tablette. Cette mise en page est conforme à celle définie par le modèle du logiciel dans la figure A.9