MASTER EN SCIENCES INFORMATIQUES

A Software Factory Engine

Blacks, Sébastien

Award date:
2023

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2021–2022

A Software Factory Engine

Sébastien Blacks

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

Co-promotrice : Maouaheb Belarbi

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

Résumé

Les lignes de produits proposent une approche permettant la dérivation automatique d'un produit logiciel dans un ensemble de produits appartenant à une famille. Autrement dit, dans un domaine d'application, un ensemble de systèmes logiciels peuvent être définis pour gérer les aspects relatifs à ce domaine. Par conséquent, ces logiciels partagent un ensemble de caractéristiques communes et diffèrent entre eux selon des aspects variables. Dans ce contexte là, obtenir une ligne de produits pertinente dépend de la qualité de gestion et d'implémentation de la variabilité du domaine d'étude. Pour ce faire, les mécanismes de variabilité tels que la programmation orientée aspect, les plugins, les directives préprocesseur, sont fortement mis à contribution pour construire des blocs de code qui peuvent être assemblés à des points de variation. Dans ce mémoire, nous proposons un moteur d'une usine logicielle qui permettra de combiner plusieurs mécanismes de variabilité pour générer des produits logiciels. Pour débuter, nous allons introduire les lignes de produits en passant de la modélisation aux mécanismes d'implémentation de la variabilité et aux techniques de vérification de la configuration d'un logiciel. Ensuite, les techniques d'implémentation de variabilité proposées dans la littérature sont présentées dans l'état de l'art tout en respectant la taxonomie proposée dans le syllabus [20]. A ce niveau là, le moteur de l'usine à logicielle proposé peut générer différentes applications en assemblant les variants inclus dans la configuration en fonction de leurs mécanismes de variabilité correspondants.

Remerciements

Mes sincères remerciements s'adressent à Monsieur Vincent Englebert et Madame Maouaheb Belarbi pour leur supervision et leur guidance tout au long de ce mémoire.

Merci également à ma famille pour son suivi et son soutien dans la rédaction de ce mémoire et durant la totalité de mes années de cursus de sciences informatiques. Ils ont consacré du temps et de l'énergie pour des relectures et corrections qui ont été d'une aide bien précieuse.

Enfin, je tiens à remercier particulièrement mes amis et mes proches qui m'ont motivé et poussé à la réussite de ces années d'études universitaires, ce qui en a fait une aventure enrichissante à tout point de vue. Leur soutien, affection et encouragements ne seront jamais oubliés.

Table des matières

Chapitre 1

Introduction

Les systèmes informatiques gagnent en envergure au fil du temps mais doivent également répondre à des demandes grandissantes tant en proportion de projet qu'en qualité de code. Cependant, afin de rester compétitif d'un point de vue économique, il est indispensable que les coûts restent au plus bas et que le temps nécessaire au développement soit le plus court possible [31].

1.1 Les lignes de produits

Il existe de multiples méthodes, technologies, connaissances qui permettent d'améliorer la rentabilité d'une entreprise qui travaille sur un logiciel. Ce manuscrit va se concentrer sur l'approche des lignes de produits qui permettent de créer un ensemble de logiciels partageant des caractéristiques et des exigences communes tout en possédant pour chacun d'eux des variations propres [79]. Cette approche divise un code source en sections, de natures différentes en fonction des mécanismes de variabilité utilisés. Cette modularisation a pour objectif de faciliter la réutilisabilité et diminuer les coûts de développement [81]. Ci-dessous un schéma illustrant la progression des différentes étapes de développement d'une ligne de produits ainsi que les sections qui abordent chacune de ces étapes :

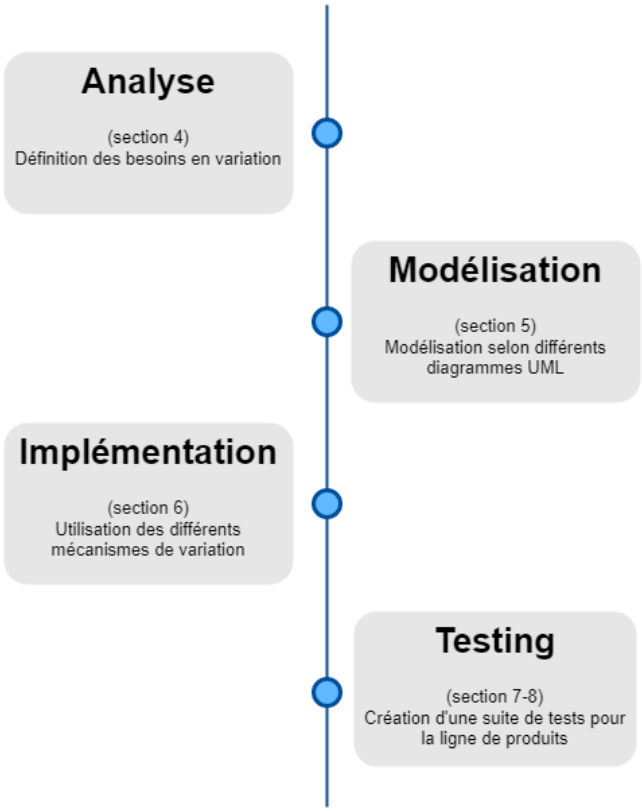

FIGURE 1.1 – Les différentes étapes de développement d'une ligne de produits

1.2 Contributions

Dans un premier temps, l'objectif des recherches effectuées pour ce mémoire, est d'avoir une vue globale des différentes techniques existantes qui permettent à un programme, et par extension à une ligne de produits, de posséder des points de variation. Dans un second temps, il s'agira de mettre en lumière, les différents outils disponibles qui permettent de faciliter le développement d'une ligne de produits. Il s'agira également de pouvoir mieux déterminer quels sont les mécanismes de variabilité les plus avantageux et les plus intéressants pour l'implémentation d'une ligne de produits.

Dans la continuité de cette première partie théorique, un travail pratique propose une plateforme qui permet de manière simple et élégante de construire et générer toute configuration possible d'une ligne de produits. Celle-ci ayant pour objectif de faciliter l'utilisation de lignes de produits et par conséquent augmenter son adoption. L'analyse effectuée en amont du développement de cette plateforme a permis de prendre connaissance des différentes techniques de variabilité ainsi que

des différentes visions de la conception d'une ligne de produits. Afin de pouvoir indiquer au programme les paramètres de la configuration exacte désirée, il est nécessaire également de développer une syntaxe ayant pour but de communiquer les informations utiles au moteur de ligne de produits. Une section est dès lors dédiée aux recherches qui ont permis de concevoir le langage qui après réflexion, est basé sur le format XML. Enfin, une ligne de produits a été conçue pour permettre de pouvoir tester un certain nombre de techniques de variation ce qui apportera une validation à la bonne conception de la plateforme résultant de ce mémoire.

1.3 Plan du mémoire

Dans un premier temps, ce mémoire présentera le résultat des recherches concernant les lignes de produits. Il sera question de comprendre leur nature ainsi que leur fonctionnement. Les sections correspondent aux différentes étapes du développement d'une ligne de produits. Après avoir introduit le sujet de recherche, la section 2 présente la méthodologie adoptée pour la réalisation du moteur de l'usine à logiciels. Ensuite, les sections 3 et 4 s'emploient à définir les lignes de produits qui représentent le domaine d'application de ce travail ainsi que la variabilité des programmes informatiques le contexte concret du projet de ce mémoire. La section 5 développe les différentes vues UML qui permettent de faire ressortir la variation d'un programme. La section 6 détaille une série de mécanismes de variation tandis que la section 7 développe une sélection de méthodes qui permettent de vérifier si une configuration est valide. La section 8 évoque les difficultés à développer des tests pour une ligne de produits, elle est suivie par la section 9 qui termine en présentant quelques programmes connus du grand public qui utilisent la variabilité à leur avantage.

Dans un second temps, il sera question de s'intéresser à l'objectif principal de ce mémoire : la conception d'un outil permettant la gestion d'une ligne de produits et de ses configurations possibles. Cependant, avant le développement, une analyse d'outils d'ingénierie de ligne de produits à été réalisée afin de connaître les diverses pratiques actuelles dans le domaine. Par après, suit une étude de la technique qui sera utilisée pour communiquer au configurateur la structure d'une ligne de produits ainsi que la configuration souhaitée. Sur cette base sera conçu un moteur de ligne de produits permettant d'enregistrer la structure d'une ligne de produits donnée ainsi que la possibilité de choisir une configuration de cette dernière et de disposer du programme résultant. Une situation pensée pour valider la bonne conception du moteur sera alors présentée. Cette situation permet de démontrer les capacités d'un moteur de ligne de produits. Enfin, pour conclure, il sera question d'évaluer la viabilité d'un moteur de ligne de produits et de tirer des conclusions sur cette implémentation ainsi que sur les recherches qui y ont mené.

1.4 Méthodologie

L'énoncé original de la recherche est « Mécanismes de variabilité et techniques de programmation dans les lignes de produits ». Après quelques investigations sur les lignes de produits, certaines techniques de variabilité sont ressorties comme étant massivement adoptées dans l'industrie du développement informatique.

Nos recherches furent menées par l'intermédiaire de la plateforme Google Schoolar qui nous a permis d'être redirigé vers une grande majorité des articles qui ont étayé la rédaction de ce document. Ce travail a été orienté par les mots clés suivants : « software product line, variability, point of variation, separation of concerns ». Afin d'avoir une meilleure compréhension de certaines technologies et d'avoir différents points de vue, la méthode de recherche snowballing a été mise à contribution.

Pour faciliter l'écriture de ce document, nous avons élaboré un lexique des articles collectés avec les informations suivantes :

- Le nom de l'article.
- Le lien URL permettant de retrouver l'article sur internet.
- Une liste des technologies évoquées.
- Le problème que l'article cherche à résoudre.
- Les possibles implémentations des technologies évoquées qui sont au stade de production.
- Les résultats obtenus avec la solution proposée.
- Un résumé d'une demi-page.

Chapitre 2

La variabilité et les lignes de produits

Introduction

Dans ce chapitre, nous définissons les notions basiques pré-requises pour la réalisation de ce mémoire. En effet, nous introduisons les lignes de produits comme domaine de définition du projet ainsi que les étapes par lesquelles un produit logiciel passe depuis la modélisation jusqu'à la dérivation et la validation. De plus, la notion de variabilité est la colonne vertébrale des lignes de produits. Ainsi, nous présentons également dans ce chapitre les mécanismes et les différentes caractéristiques de variabilité.

2.1 Les lignes de produits

Une ligne de produits définit une approche à la programmation avec une orientation sur les exigences clients. Elle a pour objectif de minimiser les coûts de développement en utilisant différents mécanismes de variabilité. Cette approche permet de créer une famille de produits dont plusieurs configurations sont possibles et facilement déployables [81]. La conception d'une ligne de produits vise à permettre de pouvoir fixer certains choix à un moment ultérieur dans la vie du logiciel. Cela nécessite d'introduire dans le code source des points de variation où il est possible, en fonction de la technique de variabilité, de choisir le variant à implémenter dans une configuration donnée [79].

La variabilité est définie [79] par ces termes : « software variability is the ability of a software system or artefact to be efficiently extended, changed, customi-

zed or configured for use in a particular context. » [1] L'action de construire une configuration d'une ligne de produits donnée est appelée « dériver un produit » [62].

La majorité des techniques évoquées dans ce document vise à permettre l'implémentation d'une ligne de produits et repose sur la séparation des fonctionnalités. Cependant certaines d'entre elles, telles que les directives préprocesseur, se passent de modularisation et offrent une variabilité au niveau des lignes de code source [7]. Un des bénéfices de cette séparation des responsabilités est également de faciliter l'évolution de la famille de produits [26]. En effet, la division en modules permet de faciliter la structure ainsi que la variation dans la ligne de produits. Des fragments de codes appelés variants peuvent être assemblés, rattachés aux fonctionnalités basiques du systèmes et personnalisés selon plusieurs contraintes du domaine. Malheureusement, chacune de ces contraintes peut également influencer plusieurs fonctionnalités [81, 79]. Un variant ne représentant pas une unique fonctionnalité, cela complique la concordance entre ces contraintes et les différents variants. Des règles de dépendance peuvent exister entre les différents variants d'une ligne de produits. Telle qu'une classe en utilisant une autre ou encore une classe nécessitant une structure de base de données spécifique. De ce fait, toutes les combinaisons de variants ne sont pas des configurations valides qui peuvent résulter en une compilation ou une exécution réussie [18, 83, 62]. Ces règles de dépendance inter-variants compliquent la séparation des responsabilités (SoC « separation of Concerns » en anglais) ayant pour objectif un découplage maximal entre les variants [5, 10]. C'est pourquoi modéliser clairement les dépendances inter-variants est une pratique importante dans le développement d'une ligne de produits [4].

Une étude économique doit être menée avant d'implémenter une ligne de produits. Si la famille de produits contient trop peu de configurations possibles avec des différences importantes, concevoir une ligne de produits peut se révéler plus coûteux qu'un développement traditionnel [27].

2.2 Caractéristiques de la variation

Une ligne de produits est composée de ce qui est communément appelé des variants. Un variant représente une option existant dans la ligne de produits et qui peut être choisie et liée au reste du programme à un moment donné défini soit par les stakeholders, soit par la technique de variabilité utilisée [26, 64, 33]. Les différents variants qui existent dans une ligne de produits peuvent être liés par un ou plusieurs points appelés « points de variation ». Ce sont ces points qui permettent à une famille de produits d'avoir des différences [81]. Un point de variation peut être soit optionnel, pour définir le choix de sélectionner zéro

1. Traduction de l'auteur :« La variabilité logicielle est la capacité d'un système logiciel ou d'un artefact à être efficacement étendu, modifié, personnalisé ou configuré pour être utilisé dans un contexte particulier »

ou plusieurs variants, soit obligatoire, si les variants doivent être nécessairement pris dans la configuration du logiciel. [68]. Il est important de connaître le type de variabilité pour un variant donné, car cela peut fortement contraindre le choix du mécanisme de variabilité[4]. De manière générale, diminuer au maximum le nombre de points de variation nécessaires pour la liaison d'un variant est une bonne pratique [79].

Il existe plusieurs techniques qui permettent de mettre en oeuvre des points de variation. Certains langages de programmation permettent d'utiliser plusieurs de ces techniques et d'autres non. De même, certaines de ces techniques peuvent être utilisées uniquement dans une sélection de langages de programmation [81]. Ces techniques présentent des avantages et des inconvénients et certaines sont plus adaptées que d'autres pour les caractéristiques d'un variant donné. Les besoins de variabilité au niveau client ne sont pas toujours facilement transcriptibles dans le langage de programmation [26]. Une sélection des techniques, parmi les plus connues et les plus utilisées, est détaillée dans cette section.

2.2.1 Cardinalité

Un variant peut être contraint par différents types de liaisons logiques.

— **Optionelle :** Un certain variant peut être inclu ou non dans le programme final.

— **Obligatoire :** Un certain variant doit être inclu dans le programme final car ce dernier est probablement une dépendance d'un autre variant. Le variant et sa dépendance peuvent être vus comme liés par l'opérateur *AND*.

— **Alternative :** Un certain point de variation peut offrir plusieurs choix de variants. Si un seul de ces variants peut être sélectionné, alors la relation respecte l'opérateur *XOR*. Si plusieurs variants peuvent être sélectionnés, alors la relation est similaire à un opérateur *OR*.

[62][26][79]

2.2.2 Liaison avec le variant

Déterminer le moment où un certain point de variation doit être figé est un choix cornélien. D'un coté, fixer la variabilité au début du développement nécessite des coûts afin de décider le variant voulu. D'un autre coté, avoir des variants en suspens jusqu'à l'exécution alourdit le programme final. De plus, il est clair dans ce cas que la variation aura dû traverser tous les stades de développement et cela augmente également les coûts de développement. Les choix du moment et de la technique utilisée pour fixer un point de variation ne sont donc pas à sous-estimer. L'instant et la technique utilisée vont également déterminer la facilité et les coûts de maintenance de la famille de produits ainsi que les possibilités de créer de nouveaux variants pour l'équipe de développement ou un développeur tiers [79].

Quelle que soit la technique utilisée pour un point de variation, la variabilité que le code source peut offrir, par les techniques développées dans ce mémoire, doit être fixée à un moment donné dans le cycle de vie du programme. Cela peut généralement se passer lors du design de l'application, de la compilation, du démarrage de l'application ou encore lors de son exécution. Figer un point de variation le plus tard possible, le rendra de manière générale plus flexible. Cependant, comme expliqué plus haut, le choix est plus complexe [62].

Les différents instants possibles de liaison d'un point de variation peuvent être résumés en deux catégories. Une liaison statique fait référence à une fixation de la variabilité avant le chargement de l'application sur l'ordinateur client. Si le variant n'est pas encore déterminé lors du chargement de l'application sur l'ordinateur client, il s'agit d'une liaison dynamique [67]. Lorsqu'il a été décidé d'utiliser un certain type de liaison pour un certain point de variation, il est très compliqué de changer de type de liaison étant donné que dans ce cas il faut généralement changer de technique de variation. Chaque type de liaison présente ses forces et ses faiblesses. Une liaison statique peut induire un surplus fonctionnel car, ne pouvant plus être modifié après la compilation, certaines fonctionnalités sont ajoutées mais non utilisées. Tandis qu'une liaison dynamique peut induire un surplus compositionnel dû à l'infrastructure variable présente lors de l'exécution. Les deux types de liaisons sont à considérer lors de l'implémentation d'une ligne de produits en regard à leurs avantages respectifs [67].

Statique

La liaison statique est préférable pour l'utilisation d'une granularité plus fine. Ce type de liaison étant fixé au plus tard lors de la compilation du programme, cela réduit en général l'impact de la variabilité sur les performances lors de l'exécution de ce dernier.[67] Ce type de liaison est utile pour certains choix de variants qui doivent être définis en amont du chargement du programme sur l'ordinateur client, tels que l'import des librairies ou le choix de système d'exploitation. Les techniques de variabilité utilisant ce type de liaisons ne permettent pas d'ouverture à un variant développé par des développeurs tiers, dû au fait que les variants sont fixés avant le déploiement de l'application. La liaison statique peut apporter un surplus de fonctionnalités qui ne seront pas utilisées car celles-ci sont incorporées au programme avant d'être fournies au client [67].

Dynamique

Plus tard un point de variation sera fixé, plus ce dernier offrira un niveau de flexibilité élevé. Cependant fixer un point de variation durant l'exécution apporte certains inconvénients tels que généralement, une consommation accrue de mémoire et de moins bonnes performances causées par le code nécessaire à la variabilité lors de l'exécution [67, 22]. Un point de variation fixé au moment de l'exécution peut permettre à des développeurs tiers de créer de nou-

veaux variants de type plug-in. Ceux-ci pouvant être liés au programme grâce à des fichiers de configuration ou des paramètres donnés au programme lors du démarrage, cela évite de devoir recompiler tout le projet pour y ajouter une fonctionnalité[67, 79]. Cependant certains appareils tels que des systèmes embarqués ne peuvent charger des plug-in lors de l'exécution, ce qui élimine la possibilité pour ces appareils d'utiliser la variabilité dynamique [67].

2.2.3 Granularité

Une contrainte fonctionnelle peut se traduire au niveau du code par un variant qui possède une granularité spécifique. Une feature peut nécessiter quelques lignes de code, des fonctions spécifiques, ou encore plusieurs classes. Ces différents niveaux de granularité peuvent être utilisés afin de catégoriser la taille d'un variant [79, 19, 3].

La taille du variant est un facteur déterminant sur le nombre de points de variation que ce variant nécessite afin de fonctionner correctement au sein du reste du programme. La structure d'un variant pouvant être complexe et fortement liée à plusieurs fonctionnalités, celui-ci peut avoir plusieurs points de variation qui utilisent différents mécanismes de programmation. Le variant peut donc être complètement lié à travers différentes étapes du développement [79]. De manière générale, il est préférable de minimiser au maximum le nombre de points de variation d'un variant en utilisant de préférence une granularité supérieure. Ceci améliore la qualité du code ainsi que sa maintenabilité mais également réduit le risque de bug [79, 9].

2.2.4 Méthodes

Les différentes méthodes existantes qui permettent d'introduire de la variabilité ont des empreintes différentes dans le code source. Selon cet impact sur le code source et la structure qu'elles impliquent, celles-ci peuvent soit être qualifiées de méthodes avec annotations ou au contraire de méthodes compositionnelles.

Méthodes avec annotations

Les méthodes avec annotations fonctionnent généralement au niveau des lignes de code et offrent une granularité fine. Comme l'indique le nom de cette catégorie, des annotations sont ajoutées au code source afin d'introduire un certain niveau de variation. Ces approches à la variabilité sont faciles d'accès, cependant elles peuvent aisément engendrer des erreurs syntaxiques [37]. Certaines de ces méthodes, telles que la compilation conditionnelle ou la technologie frame, deux techniques évoquées dans ce mémoire, sont facilement identifiables dans un programme car elles sont formées de blocs de code bien délimités [89, 3].

Méthodes compositionnelles

En opposition aux méthodes avec annotations, certains mécanismes de variabilité sont plus difficiles à identifier car ils ont une empreinte plus discrète sur le code source. Ces mécanismes de type compositionnel modifient le code afin de lier un plugin ou d'utiliser un certain composant. La granularité est relativement élevée au niveau des composants [37]. Grâce à ce niveau de granularité, ces méthodes possèdent généralement une meilleure lisibilité du code, elles facilitent la correction d'erreurs syntaxiques mais elles augmentent également la séparation des responsabilités. Cependant, certaines méthodes font exception telles que les méthodes de clonage ou de remplacement de module par exemple, qui sont difficiles à identifier pour une personne n'ayant pas une bonne connaissance du point de variation [89, 3].

2.2.5 Ouverture du point de variation

En tenant compte du moment de leur liaison, les points de variation d'une ligne de produits peuvent être de type « ouvert ». Un point de variation ouvert laisse la possibilité à un développeur externe de développer un nouveau variant selon ses besoins et de le lier à l'application, même si ce variant à été développé hors du développement originel de la ligne de produits. Si un variant externe ne peut être ajouté à un point de variation, ce dernier est qualifié de « fermé » [86].

2.2.6 Variation par défaut

Certaines techniques de variabilité permettent de définir un choix de variant qui sera utilisé par défaut [82]. Cela permet de diminuer la complexité de variation [89].

2.3 Modélisation

Une ligne de produits peut être composée d'un ensemble complexe de caractéristiques communes et variantes, c'est pourquoi, il est crucial de pouvoir modéliser clairement chaque variant de même que ses possibles relations et contraintes. Les différentes méthodes possibles de modélisation ont pour but de faciliter la compréhension ainsi que la gestion de la variabilité bien présente dans les lignes de produits, elles permettent également à des outils d'automatiser diverses taches. Certaines méthodes de modélisation peuvent être composées, en plus d'un schéma, d'une partie comprenant un ensemble de contraintes textuelles visant à compléter celui-ci. Ces contraintes permettent par exemple de vérifier automatiquement si une configuration de la ligne de produits respecte toutes celles-ci. Chaque vue de la ligne de produits permet de visualiser un aspect spécifique et l'utilité de celle-ci. Les sous sections suivantes développent une partie des vues possibles d'une ligne de produits [75].

2.3.1 Diagramme de classes

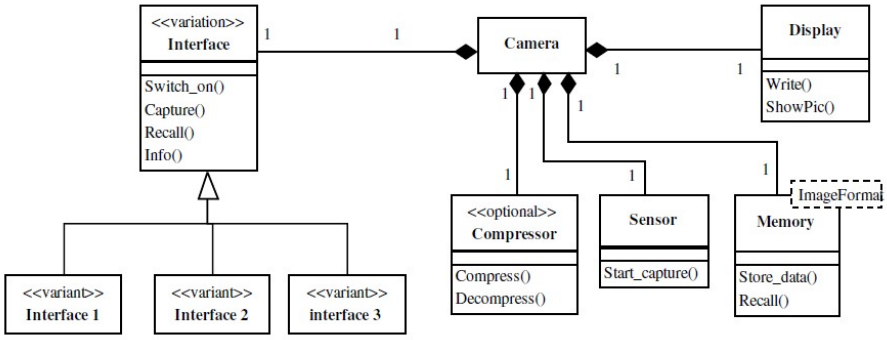

FIGURE 2.1 – Diagramme de classes

Un diagramme de classes permet de modéliser un programme informatique en mettant en avant la structure des classes et leurs relations. Des contraintes peuvent être définies à l'aide d'un langage OCL afin de limiter le choix de variants ou de définir des dépendances/incompatibilités entre les variants. Ce type de diagramme UML permet de modéliser la variabilité qui existe dans les lignes de produits de deux manières [90].

- **Optionnelle :** Une classe décorée du stéréotype << *optional* >> est une classe optionnelle et peut donc ne pas exister dans une certaine configuration de la ligne de produits. Une classe peut générer des incohérences si une classe non-optionnelle dépend d'une classe optionnelle [90].
- **Variation :** Un ensemble de variants peut être modélisé comme un héritage décoré de stéréotype. Une classe abstraite parent décorée du stéréotype << *variation* >> possède alors plusieurs classes enfant avec le stéréotype << *variant* >>[90].

2.3.2 Diagramme de features

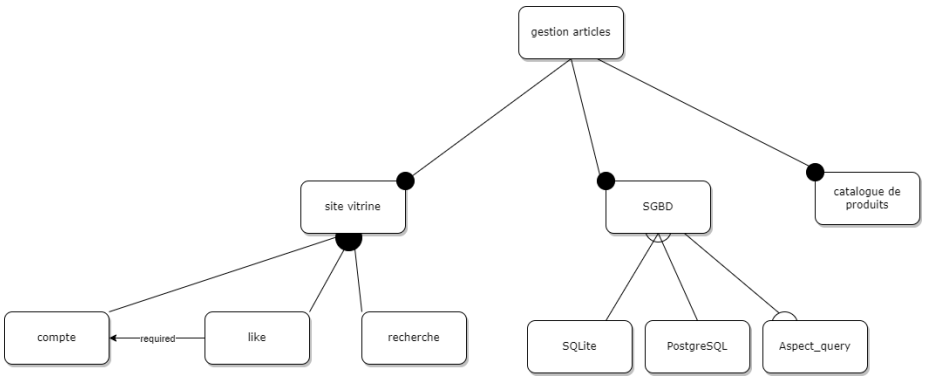

FIGURE 2.2 – Diagramme de features

Le diagramme de features permet de distinguer facilement les différents points de variation ainsi que les contraintes logiques qui les régissent. C'est pourquoi le diagramme de features est la notation la plus répandue pour représenter une ligne de produits. Différentes notations existent pour construire un diagramme de features telles que FODA, RSEB et FeatureRSEB [79].

FIGURE 2.3 – Notations FODA

Présentée en 1990 par le gouvernement américain, FODA est la première méthode de représentation graphique de la variabilité d'un point de vue fonctionnel [36]. Comme illustré dans le graphe ci-dessus, les fonctionnalités sont représentées par les noeuds d'un arbre. Les liens entres les différents noeuds peuvent être optionnels, représentés par un rond vide, tandis que les liens obligatoires sont représentés avec un rond plein. Des fonctionnalités alternatives mais dont au moins une d'entre elles doit être choisie, sont représentées avec un rond vide entre elles, alors que le rond plein représente une relation *or* entre les enfants d'une fonctionnalité[84].

2.3.3 Diagramme de séquence

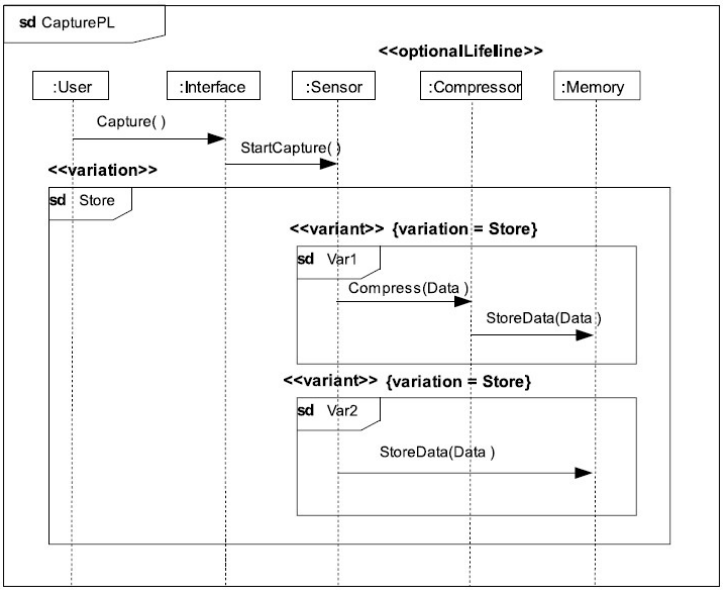

FIGURE 2.4 – Diagramme de séquence

Un diagramme de séquence permet d'avoir une vue sur l'aspect dynamique d'un programme. Les interactions entre différentes entités sont modélisées afin d'avoir une vue claire sur les interactions et également les dépendances dans un système [68]. Un diagramme de séquence permet également de préciser comment va être implémenté un *use case* en détaillant les objets impliqués [64]. En respectant le méta-model UML2.0, une ligne de produits peut être modélisée dans un diagramme de séquence à l'aide de trois constructions : Optionnel, Variation et Virtuel [90].

- **Optionnel :** Un objet du diagramme peut être modélisé comme optionnel avec le stéréotype $<< optionalLifeline >>$ ou $<< optionalInteraction >>$ pour une interaction entre deux objets qui pourrait être optionnelle.
- **Variation :** Un point de variation d'une ligne de produits peut posséder plusieurs variants. Le stéréotype $<< variation >>$ définit un bloc composé de sous-blocs avec le stéréotype $<< variant >>$ associé avec l'écriture : {variation = VariationName} afin d'indiquer le point de variation du variant.
- **Virtuel :** Une partie de diagramme de séquence possédant le stéréotype $<< virtual >>$ peut être redéfinie par un autre diagramme de séquence.

2.3.4 ConIPF

La méthodologie ConIPF (Configuration dans une ligne de produits industriels) a pour but de faciliter la dérivation d'une configuration d'une ligne de produits définie [31]. ConIPF distingue de multiples types d'entités ainsi que plusieurs types de relations entre ces entités. Ceci permet de former une structure hiérarchique dans les fonctionnalités disponibles dans la ligne de produits. Les différents types d'entités et de relations permettent de connaître les dépendances des différentes fonctionnalités et certaines valeurs paramétrables lors du choix de la configuration [31, 75]. Afin de faciliter la compréhension de la relation entre les demandes clients et les différentes entités de la ligne de produits tout en ayant une vue globale du projet, ConIPF propose une représentation graphique de celui-ci. Cependant la représentation graphique proposée dans la méthodologie ConIPF ne permet pas de modéliser l'ensemble des contraintes et paramètres. C'est pourquoi, une représentation textuelle qui utilise un langage appelé AMPL, permet de compléter la modélisation de la ligne de produits [75].

Suivant une sélection des fonctionnalités désirées de la part du client dans une ligne de produits, ConIPF offre la possibilité de faire le lien automatiquement avec les composants hardware nécessaires au bon fonctionnement de la configuration sélectionnée [75, 31].

2.4 Mécanismes de variabilité

Les sections suivantes mettent en avant une sélection de mécanismes permettant d'introduire la notion de variabilité dans un programme informatique, parmi les plus répandus et les plus prometteurs pour la conception d'une ligne de produits. Chaque technique permet d'introduire de la variation d'une manière qui lui est propre. Tout point de variation qui utilise l'une de ces méthodes nécessite d'être lié à un moment donné, au plus tard lors de l'exécution du programme. Il est important de noter également que certaines méthodes ne peuvent fonctionner dans certains langages de programmation.

Ci dessous, un tableau synthétisant les mécanismes de variabilité détaillés par après, selon les critères vus précédemment. Ce tableau permet de comparer sous plusieurs aspects les différentes techniques et par conséquent choisir celle qui est la plus adaptée à une situation donnée.

	Compilation conditionnelle	Héritage	Plugin	Frame technology	XVCL	POA	POD	POS
Liaison	statique	statique	dynamique	statique	statique	statique	statique	dynamique
Granularité	fine	moyenne	haute	moyenne	moyenne	moyenne	haute	haute
Annotations/Compositionnelles	annotations	composition	composition	annotations	annotations	composition	composition	composition
Ouverture du point de variation	fermé	fermé	ouvert	fermé	fermé	fermé	fermé	ouvert
Variant défaut	oui	non	non	oui	oui	non	oui	non

FIGURE 2.5 – Synthèse des différents mécanismes de variation

2.4.1 Compilation conditionnelle

```
#ifdef CONFIG_FOO
//Block 1
#else
//Block 2
#endif
```

FIGURE 2.6 – Directives préprocesseur

La compilation conditionnelle est mise en place à l'aide de conditions telles « #IFDEF » appelées directives préprocesseur [83]. Ces directives étant écrites dans le code source, cette méthode est donc dite avec annotations. Ces directives permettent au compilateur d'inclure certaines parties du code dans le binaire et d'en exclure d'autres [4, 20]. Cette technique peut nécessiter plusieurs points de variation et ces derniers sont difficilement identifiables. Utiliser la compilation conditionnelle fixe les différents variants lors de la compilation comme une méthode statique. Cela allège le binaire qui en résulte, cependant il est dès lors impossible de les modifier lors de l'exécution de l'application comme expliqué dans le chapitre sur la variation statique [79, 83].

2.4.2 Héritage

L'héritage est un mécanisme utilisé dans le paradigme de programmation orientée objet. Dans ce paradigme, le code est divisé en structures appelées classes. Ces dernières peuvent hériter d'une autre classe qui sert alors de modèle. Il existe différents types d'héritages correspondant à diverses situations. Cependant les principales formes d'héritages sont l'héritage simple ainsi que l'héritage multiple [4].
L'héritage est principalement utilisé par les design pattern strategy, celui-ci permet de choisir dans un ensemble défini, l'algorithme à exécuter lors de l'exécution du programme, mais également le design pattern factory, utilisé pour instancier un object selon un type abstrait dont il dérive.[82]

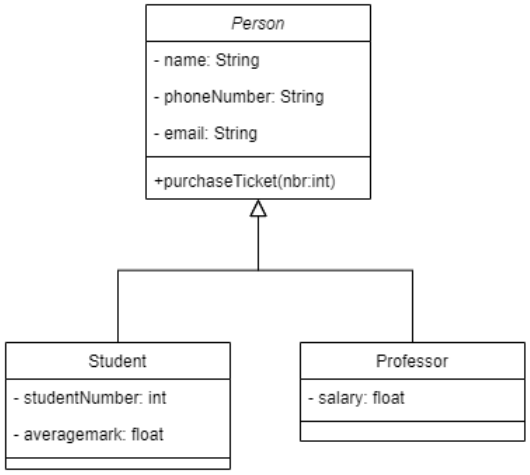

FIGURE 2.7 – Diagramme montrant l'héritage de classes

De multiples variants peuvent être créés dans différentes classes enfant d'une même super classe. Lors de la fixation du point de variation, il suffit de sélectionner une des classes enfant implémentées. L'héritage fait partie des méthodes compositionnelles et introduit une variation de type statique. L'inconvénient de cette méthode est principalement le nombre proportionnel de classes enfant par rapport au nombre de variants existants. Ceci peut rendre les diagrammes de représentation complexes. Avec cette méthode, il est également difficile, voire impossible, de combiner plusieurs options de variants [4].

2.4.3 Plugin

Certaines applications, qui veulent permettre à des développeurs tiers d'ajouter des fonctionnalités, alors qu'elles ont déjà été déployées, font appel à des plugins [56]. Les IDE tels que Eclipse ou Intellij par exemple, permettent grace à des plugins d'ajouter des fonctionnalités même lors de l'exécution du programme [6]. Ces applications, qui se basent sur des plugins afin d'ouvrir leurs portes à une communauté de développeurs, deviennent de plus en plus populaires [56]. Un plugin peut être vu comme un composant software dépendant d'un programme coeur, qui utilise le polymorphisme et le remplacement de modules afin de permettre d'ajouter des fonctionnalités à un programme existant, jusqu'au moment de l'exécution [47]. L'interface d'un plugin peut permettre également de modifier l'interface graphique du programme. Cependant agir de la sorte peut être complexe car il faut veiller à la représentation visuelle de tous les plugins [6]. Une application utilisant des plugins doit être conçue comme une application centrale sur laquelle les plugins vont pouvoir ajouter des fonctionnalités [66]. Les plugins étant généralement développés par des développeurs tiers, il n'est pas rare d'avoir des conflits entre différents plugins d'un même programme, par

exemple si l'un des plugins viole une assomption d'un autre plugin [56]. Il est nécessaire pour un programme qui veut utiliser des plugins d'avoir une procédure permettant de charger les fichiers des différents plugins pour les lier à ce dernier [6].

2.4.4 Programmation orientée aspect

```
aspect SimpleTracing {
    pointcut tracePoints():
        receptions(void FigureEditor.slide(int, int));

    static before(): tracePoints() {
        printMessage("Entering", thisJoinPoint);
    }
    static after(): tracePoints() {
        printMessage("Exiting", thisJoinPoint);
    }
}
```

FIGURE 2.8 – code source d'un aspect avec les directives after et before

La programmation orientée aspect est un paradigme de programmation utilisé afin d'apporter un niveau de variabilité à un programme. La programmation orientée objet permet d'implémenter facilement des structures hiérarchiques et utilise une variabilité compositionnelle. Tandis que la programmation orientée aspect a pour but de faciliter la séparation des responsabilités en rassemblant des dépendances transversales de plusieurs modules d'un programme dans un composant appelé aspect [4, 11]. Un aspect peut comprendre dans sa déclaration des actions à effectuer avant, après ou autour d'un de ses points de jonction [5, 83, 21]. Le tisseur d'aspect est un métaprogramme implémenté dans les frameworks présentés ci-dessous qui sert à lier les aspects en appliquant correctement les directives de jointures aux parties du programme qui ne sont pas orientées aspect [5, 29, 22]. La liaison des aspects, aussi appelée weaving, peut se faire lors de la compilation, du démarrage de l'application ou lors de son exécution [83, 5].

En pratique, la programmation orientée aspect possède quelques défauts tels qu'un grand nombre de lignes nécessaires à paramétrer les aspects. De même, le couplage des aspects avec un certain point de variation est assez élevé, ce qui peut amener des problèmes de modularité, car la séparation des responsabilités est difficile à respecter. Ce paradigme permet une bonne intégration avec différents design pattern améliorant la réutilisabilité ainsi que la modularité[10].

AspectJ

AspectJ est une extension java qui permet de faire de la programmation orientée aspect [5, 83]. La compatibilité de cette extension est assurée avec le code java traditionnel en plusieurs points [5] :

— Tout code java est accepté par AspectJ et inversement.
— Il est possible de programmer avec des outils déjà existants.

Hyper/J

Hyper/J offre la possibilité de séparer simultanément des responsabilités de différents types qui potentiellement interfèrent entre elles. Des méta-déclarations fournies par le développeur permettent au moteur Hyper/J d'extraire des méthodes de classes du programme et de recomposer une nouvelle classe représentant une responsabilité dans le programme [5, 60].

Spring

Spring est un framework open-source d'infrastructure répandu pour des programmes java. Ce framework offre aux développeurs de nombreuses fonctionnalités principalement architecturales, réparties dans une vingtaine de modules [35, 21]. Cette vingtaine de modules sont tous centrés autour d'un coeur fonctionnant par le design pattern inversion de controle. Ce design pattern qui influence tout le framework offre plusieurs avantages tels qu'une architecture simplifiée mais aussi des tests plus simples.[21] Un des modules de spring peut servir de tisseur d'aspect pour de la programmation orientée aspect. Les design patterns proxy et intercepteurs sont utilisés pour lier les aspects. Cela a comme avantage de ne pas nécessiter de tissage explicite et donc de diminuer l'empreinte d'annotations sur le code java, ce qui facilite son adoption. Cependant l'utilisation du design pattern proxy rend les capacités de spring plus faibles que AspectJ [41].

2.4.5 Frame technology

La Technologie Frame est un système permettant la gestion de composants appelés frames. Une frame est un composant d'une ligne de produits ainsi qu'une macro permettant de lier le composant. Une frame peut être elle-même composée de frames adaptées aux besoins spécifiques, ce qui crée un arbre hiérarchique des frames existantes [4]. Cependant, au vu des recherches effectuées pour ce travail, les outils utilisés jusqu'à présent pour implémenter cette solution sont fortement couplés au langage COBOL. Ceci pourrait être une hypothèse qui explique que la technologie Frame ne soit pas largement utilisée [29], malgré le fait que les coûts de production seraient réduits de 84% et le temps de développement accéléré de 70% par rapport aux standards pour de gros projets [80, 34].

Le processeur frame permet de manipuler, à l'aide d'annotations dans le code, les différentes frames et d'effectuer certaines actions tout en leur transmettant certains paramètres. Ces différentes commandes permettent [29] :

- D'invoquer une frame
- De modifier dans une frame un bloc à l'aide d'un identifiant donné à celui-ci
- D'ajouter une fonctionnalité ou un bloc dans une frame
- De paramétrer une valeur dans la frame
- D'effectuer un choix entre plusieurs sous-frames optionnelles

— De construire une boucle

Toutefois, cette méthode pose des problèmes de compréhension lorsqu'elle est utilisée dans une grande architecture. Le processus de frame est fixé lors d'une pré-compilation, les différents points de variation sont donc fixés à ce moment-là.

2.4.6 XVCL

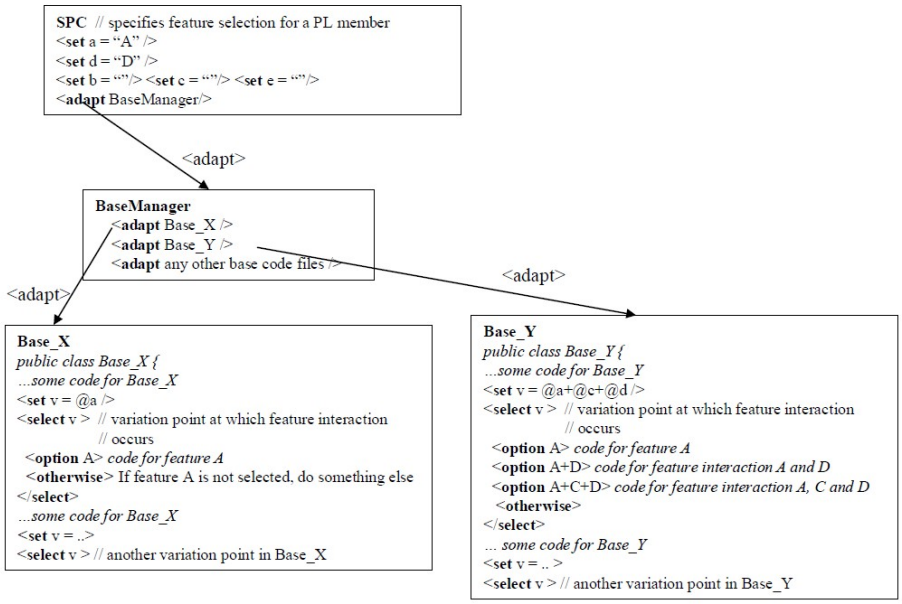

FIGURE 2.9 – illustration du fonctionnement de XVCL

XVCL (XML Variant Configuration Language) est fort similaire à la technologie frame. Cependant XVCL est totalement découplé de tout langage de programmation. Tout comme la technologie frame, il est possible avec XVCL de définir des modifications spécifiques aux différents points de variation d'une ligne de produits. Le grand atout de XVCL par rapport à la technologie frame est la possibilité d'être implémenté pour des projets orientés objet au lieu d'être restreint au COBOL. Contrairement à un DSL qui utilise de la sémantique propre à un domaine, cette approche est capable de s'adapter sans problème à tout domaine [34]. XVCL offre aussi de grandes possibilités d'évolution [28, 80].

2.4.7 Méta-programmation

Un programme informatique peut être écrit sous différents niveaux d'abstraction allant du code source haut niveau, écrit par un programmeur, jusqu'au bit-code utilisé par le processeur [16]. La méta-programmation a pour objectif d'introduire un niveau supérieur d'abstraction avec le programme en écrivant

des méta-programmes qui vont manipuler ce dernier [20]. La généralisation est donc au centre de ce paradigme de programmation afin de pouvoir générer une ligne de produits possédant des similitudes. Un méta-programme est un programme générique développé à l'aide d'un méta-langage qui à l'aide d'un certain nombre de paramètres permet de produire un programme d'une famille de programmes similaires [78, 16]. Les compilateurs sont considérés comme des outils de méta-programmation étant donné l'utilisation de méta-langages tels BNF afin de générer un programme [78]. Il existe différents frameworks ou technologies permettant de faire de la méta-programmation en offrant la capacité de paramétrer des niveaux d'abstraction inférieurs. Ces outils sont offerts dans certains langages de programmation tels que la programmation réflexive ou bien les open classes en ruby [24, 16]. Outre la difficulté de comprendre correctement l'approche de la méta-programmation, le principal obstacle à l'utilisation de la méta-programmation réside dans le fait qu'il est compliqué de décrire formellement les exigences fonctionnelles et non-fonctionnelles [13, 16].

Métadonnée

Les métadonnées sont les descriptions fournies afin de décrire et représenter différentes propriétés ou paramètres d'un niveau d'abstraction précis d'un méta-programme [78]. Cela permet également d'avoir davantage d'informations concernant le niveau d'abstraction. Une métadonnée peut être de différentes natures, d'une machine à états à une documentation écrite [16]. Les seules contraintes qu'un type de données doit respecter pour être utilisé comme métadonnée sont que cette dernière doit être liée au composant concerné mais surtout que ces données doivent pouvoir être traitées automatiquement [78].

2.4.8 Programmation orientée feature

La programmation orientée feature est un paradigme de programmation centré sur la modularité des fonctionnalités [20]. Celui-ci permet d'implémenter une ligne de produits en utilisant des modules, ce qui facilite grandement l'ajout et le retrait de fonctionnalités [70, 85, 7, 46, 67]. Ce paradigme se base essentiellement sur la méthodologie de développement itératif qui induit qu'un programme complexe peut être construit sur base d'un programme plus simple auquel des fonctionnalités ont été rajoutées itérativement [67, 7]. Contrairement à la programmation orientée delta, un problème présent dans la programmation orientée feature est que dans celui-ci il est complexe de partager du code entre deux modules de fonctionnalité. Cela nécessite qu'un certain module soit une multiplication de plus petits modules ou de braver la règle de séparation des responsabilités ou encore à une duplication du code entre les deux modules.

La programmation orientée aspect et la programmation orientée feature sont deux paradigmes de programmation relativement proches étant donné qu'ils ont tous les deux pour objectif de faciliter la gestion des dépendances transversales. Cependant la programmation orientée feature n'utilise qu'un sous-ensemble des

possibilités de la programmation orientée aspect, ce qui permet de faciliter la programmation itérative et le design des aspects. Ces deux paradigmes peuvent donc être compris comme héritant d'un paradigme commun plus général [7, 70].

AHEAD (Algebraic hierarchical Equations for Application Design) est une méthodologie de programmation orientée feature [85, 7, 20]. Dans cette méthodologie, il existe deux terminologies qui la distinguent de GenVoca, la méthodologie dont est dérivé AHEAD. GenVoca définit un programme comme une valeur, et une extension de ce programme est appelée une fonction. En contre partie, AHEAD définit la structure hiérarchique d'un ensemble d'artefacts comme étant une valeur et la fonction qui sélectionne certains artefacts de cette structure comme une extension [7]. AHEAD peut également être vu comme une généralisation de la méthodologie GenVoca à travers ces quatre points [7] :

- Il existe de multiples représentations d'un programme (code source, UML, grammaire, documentation, models,...). Une méthodologie orientée feature doit être capable de gérer ces différentes représentations.
- Ces différentes représentations d'un programme sont écrites dans différents langages. Une méthodologie orientée feature doit être ouverte à pouvoir utiliser une liste non finie de différents langages.
- Lorsqu'une nouvelle fonctionnalité est ajoutée, un ensemble de représentations du programme doit être mis à jour. Cela implique que la notion d'extension utilisée par AHEAD comprenne toutes les représentations du programme.
- Une méthodologie orientée fonctionnalité doit pouvoir utiliser la structure modulaire et hiérachique que ce paradigme de programmation requière.

En plus d'appliquer le paradigme orienté feature en découpant le code source d'un programme en terme de fonctionnalité, la méthodologie AHEAD utilise comme principe que les représentations d'un programme peuvent elles aussi être vues comme des fonctionnalités. A l'aide d'un traitement, il est possible, d'après une équation représentant les fonctionnalités désirées, d'optimiser l'équation du programme qui en résultera [7].

2.4.9 Programmation orientée delta

La programmation orientée delta est un paradigme inspiré de la programmation orientée feature avec cependant moins de restrictions. Ces deux paradigmes de programmation permettent le développement itératif, cependant leur grande différence est que la programmation orientée feature est utilisée pour l'implémentation de ligne de produits tandis que la programmation orientée delta a été pensée autour du développement des lignes de produits.

Dans ce paradigme pensé pour les lignes de produits qui n'est pas restreint à un langage de programmation, il existe un module central et un ensemble de modules delta [7]. Le terme de module central utilisé dans ce paradigme comprend le code nécessaire à l'implémentation d'une configuration complète et valide de la ligne de produits [71]. Utiliser un produit complet comme base de développe-

ment des autres fonctionnalités de la ligne de produits permet d'utiliser pour le développement du module central des techniques de programmation standard maîtrisées par tout bon développeur [70]. Sur base du module central, les différents modules delta contiennent des instructions de modification de ce dernier afin d'implémenter de nouvelles fonctionnalités [71]. Ces modules contiennent également des contraintes qui dictent si un certain module peut être appliqué sur une configuration donnée ou non. En respectant la logique propositionnelle, cet ensemble de conditions qui est défini dans la clause *when* d'un module delta, pose comme condition quel module doit ou non être déjà utilisé dans la configuration présente. Conditionner l'utilisation d'un module delta peut être complexe mais s'avère important afin d'éviter des problèmes de concurrence. Cela permet également de résoudre des problèmes de dépendance entre deux modules en permettant au paradigme d'autoriser la combinaison de deux fonctionnalités dans un delta module [70]. Ces contraintes présentes pour chaque module, permettent également de faciliter la correction de bugs en identifiant le delta module responsable du bug. L'ordre d'application des modules delta, déduit des clauses *when* et *after* de chaque module, permet de garantir que chaque implémentation d'une configuration donnée sera identique. Les modules delta, tels qu'on peut observer dans l'extrait de code ci-dessous, ont la permission d'ajouter, modifier ou supprimer une classe mais également de modifier la structure d'une classe en ajoutant, modifiant ou supprimant une méthode ou un attribut.

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

FIGURE 2.10 – Configuration d'un module delta

Le choix du produit valide utilisé comme module central est très libre, cela peut autant être une configuration minimale qu'une configuration avec de nombreuses fonctionnalités intégrées. Cependant, un module central avec beaucoup de fonctionnalités permet d'augmenter la part de développement avec des pratiques standards, ce qui offre une plus grande garantie de qualité avant d'introduire les modules delta qui compliquent le développement.

Les modules de la programmation orientée feature autorisent uniquement d'ajouter et modifier le code source tandis que les modules de la programmation orientée delta permettent également de supprimer du code. C'est cette différence qui permet de choisir n'importe quelle configuration valide comme module central pour la programmation orientée delta et offre plus de flexibilité à ce paradigme. Cependant les modules delta sont dès lors moins réutilisables dans une nouvelle ligne de produits. Un autre avantage de la programmation orientée delta réside dans la possibilité de pouvoir calculer une liste des modules delta applicables pour une certaine configuration. La programmation delta évolue mieux que la

programmation orientée feature dans de grandes lignes de produits possédant des problèmes de dépendances transversales [70].

2.4.10 Programmation orientée sujet

Le paradigme de programmation orientée sujet vise la séparation des responsabilités en découpant le code source en un ensemble de modules selon leur sujet [4, 11, 76]. Contrairement à la programmation orientée aspect, il n'y a pas de module principal. Malheureusement, cela implique que chaque module communique potentiellement avec tous les autres modules d'un programme [76]. Trois concepts sont définis dans ce paradigme :

- Un sujet définit un ensemble de classes indépendantes qui utilisent le paradigme de programmation orientée objet. Cet ensemble de classes concerne un sujet propre
- La partie label d'un sujet permet de connaître la structure de ce dernier
- Le dernier concept fait référence aux règles de composition. Celles-ci définissent des contraintes pour la composition des sujets pour le programme final

Chaque module peut être compilé indépendamment des autres ce qui permet de choisir un ensemble de sujets sans devoir recompiler le tout. Les informations contenues dans le label permettent de lier les différents modules. Le code de liaison n'est pas contenu à l'intérieur des différents modules mais dans un fichier séparé, contrairement à la programmation orientée aspect où la liaison des différents aspects est réalisée à l'intérieur de ceux-ci [76, 11]. Le code produit par le paradigme de programmation orientée sujet peut être plus complexe dû à la séparation par sujet. Néanmoins, ce paradigme permet de produire du code avec une meilleure réutilisabilité, sécurité accrue ainsi que de meilleures performances pour de grands projets [11]. La programmation orientée sujet offre une plus grande liberté que la programmation orientée aspect, cependant les règles de composition des modules sont plus complexes [76].

2.5 Configuration Valide

Il existe différentes méthodes qui permettent de vérifier si une certaine configuration pour une ligne de produits donnée, est acceptable ou non. Ces méthodes offrent un certain niveau d'automatisation ce qui est fort utile afin de générer des configurations pour effectuer des tests en autonomie.

2.5.1 SAT

Un solveur SAT ou solveur de satisfaisabilité en logique propositionnelle [27, 50] est un outil qui permet de résoudre un problème exprimé en une formule de logique propositionnelle si cette dernière possède une solution [7]. Déterminer si une formule écrite en logique propositionnelle est satisfaisable, est connu comme étant un problème NP-complet [27, 50]. Un solveur SAT peut aussi être

utilisé pour corriger des modèles possédant des erreurs [7]mais son efficacité sera dégradée si des contraintes groupées sont définies. Par conséquent, étant donné que FODA ne décrit pas des contraintes de groupes, cette méthode est bien utilisée par un solveur SAT [27]. Il est également faisable de compter le nombre de solutions possibles pour une formule donnée et ainsi connaître le nombre de configurations valides pour une ligne de produits [27].

DPLL est un algorithme [27] utilisé par plusieurs solveurs SAT. Cet algorithme de recherche permet de vérifier si une formule booléenne écrite en logique propositionnelle est satisfaisable. Pour cela, l'algorithme DPLL assigne itérativement une valeur booléenne aux différentes variables tant que celles-ci ne violent aucune clause. Dans le cas contraire, l'algorithme revient en arrière afin de corriger la variable problématique. L'algorithme finit quand toutes les variables de la formule ont été assignées de sorte que la formule soit satisfaite ou à l'inverse, que toutes les possibilités ont été essayées mais que la formule n'est pas satisfaisable [50]. Les solveurs SAT peuvent être classés en fonction du nombre de variables admis par clause. Un solveur admettant 2 littéraux par clause sera donc un solveur 2-SAt [50].

2.5.2 Arbre de décision binaire

Un arbre de décision binaire permet également de savoir si une certaine configuration de ligne de produits est valide. Cette technologie offre de meilleurs résultats pour de grandes lignes de produits, effectuant une vérification dans une durée en corrélation directe avec le nombre de noeuds de décision. Cependant elle impose un ordre fixe dans lequel les contraintes doivent être vérifiées. Trouver l'ordre optimal de vérifications pour une ligne de produits donnée est un problème NP-complet. Malgré tout, certains calculs permettent d'approximer l'ordre idéal avec moins de complexité. A la différence d'un solveur SAT, un arbre de décision binaire n'accepte que des contraintes textuelles et donc plus simplistes qu'un solveur SAT ce qui rend les contraintes moins fortes [27].

2.5.3 Automate

Des automates peuvent également servir à vérifier les différentes configurations valides pour une certaine ligne de produits. L'écriture d'une grammaire peut servir à définir les séquences de fonctionnalités acceptées dans la ligne de produits. L'ensemble des phrases validées par la grammaire définit l'ensemble des configurations autorisées. Les conditions pour les différents modules doivent être traduites en contraintes syntaxiques dans une grammaire définie. Utiliser une grammaire dans ce but revient à voir un diagramme de feature comme une grammaire. Cependant cette dernière a comme avantage de donner également comme information un certain ordre d'implémentation des features, ce qui est utile pour la programmation orientée delta et la programmation itérative [7].

2.6 Testing

La grande différence entre le testing d'un programme traditionnel et le testing d'une ligne de produits se situe dans la gestion de la variabilité. En effet, gérer correctement la variabilité d'un programme est compliqué car le nombre de variants dans une ligne de produits augmente de façon exponentielle le nombre de produits finaux. Ceci rend pratiquement impossible de tester toutes les possibilités de variations pour les grandes lignes de produits [33].
Étant donné que le moment de fixation des variants peut avoir lieu à de nombreux stades différents durant le cycle de vie du programme, ceci implique une grande influence sur les tests [44]. Effectuer les tests après la fixation des variants présente des désavantages :

— Les tests ne sont pas garantis de fonctionner avec d'autres variants et tester toutes les combinaisons de variants peut devenir impossible
— Le processus de test est rendu plus complexe

De plus, lorsque les tests sont effectués quand tous les variants voulus ont été fixés et qu'une erreur doit être corrigée, le coût de correction de cette erreur est élevé.
Procéder à des tests après la liaison de chaque variant est plus compliqué à implémenter, cependant, le coût de réparation d'un bug est moindre [33]. Dans la plupart des cas il est plus efficace et moins coûteux de tester les différents variants avant de les lier. Dans ce dernier cas, il est nécessaire de mettre en place des mocks lors des tests, afin de simuler la réponse d'un autre variant.

Deux types de tests sont reconnus pour les lignes de produits. Les tests de domaine sont utilisés pour tester de petites parties du programme afin de s'assurer de leur fonctionnement. Les tests d'application réutilisent principalement les tests de domaine, cependant les points de variation sont ajoutés aux tests afin de vérifier le bon fonctionnement de la variabilité de la ligne de produits [44].

2.6.1 CIT

Les tests d'interaction combinatoire (CIT) [62] sont une approche qui a pour but de réduire le nombre de combinaisons de variants sur lesquelles il est nécessaire d'effectuer les tests. Un des problèmes qui se présente lorsqu'on effectue des tests automatisés pour une ligne de produits consiste dans la sélection automatique d'une configuration valide. La satisfaction de contraintes est un exercice NP-complet ce qui rend l'automatisation des tests difficile pour une ligne de produits comportant beaucoup de configurations possibles. Le diagramme de feature est un outil très utilisé afin de connaître les différentes configurations valides. Augmenter le nombre de configurations de tests augmente la couverture de ces tests, cependant cela augmente également les ressources nécessaires à ces derniers [62].

2.6.2 Outils

Différents outils permettent de faciliter la phase de test d'une ligne de produits. Le framework GATE permet de générer automatiquement des cas de test d'après des paramètres de test unitaire [44]. Kesit est un outil qui utilise un résolveur SAT afin de générer une approche incrémentale pour tester une ligne de produits [85].

2.7 La variabilité en pratique

Il existe de nombreux exemples de logiciels utilisant des techniques de variabilité afin de proposer davantage de fonctionnalités ou permettre de pouvoir s'adapter à différentes installations.

2.7.1 Linux

Linux est l'un des systèmes d'exploitation les plus utilisés au monde et est l'un des plus gros programmes informatiques avec des points de variation permettant de configurer plus de 10.000 fonctionnalités [54]. Ces points de variation présents dans le noyaux Linux permettent de compiler ce dernier avec entre-autres les drivers d'une certaine plateforme sans alourdir l'installation de drivers non voulus. Le grand nombre de points de variation permet également de faciliter l'implémention d'une nouvelle fonctionnalité et de la rendre accessible par l'un de ces points [54, 18].

Cette variabilité est rendue possible par trois systèmes : le code source, les fichiers KCONFIG ainsi que les fichiers makefiles KBUILD [53].

KBUILD

Les fichiers makefiles KBUILD permettent de fournir à l'outil de compilation Make les fichiers nécessaires à la compilation de Linux, suivant certaines conditions.

KCONFIG

Le fichier KCONFIG, présent normalement dans chaque dossier, contient les paramètres des fonctionnalités du noyau Linux relatives au dossier ainsi que les interdépendances.

Code space

Le code source du noyau Linux est écrit en langage C et comprend des directives préprocesseur définies par des blocs « #ifdef » et « #ifndef ». Ces directives permettent de donner des conditions pour que certaines parties du code source ne soient pas compilées [53].

2.7.2 Mozilla

Fondée par la société américaine Netscape, Mozilla devient une fondation indépendante en 2003. A l'aide d'une architecture basée sur des composants, Mozilla a pour objectif de créer une plateforme web avec une architecture flexible et facilement modifiable. L'architecture de base créée par Mozilla permet à l'entreprise de développer un set de fonctionnalités sur cette base facilement variable. Cette architecture a également l'avantage de créer des composants indépendants de la plateforme. Ces derniers nécessitent uniquement les composants avec lesquels ils sont liés [79].

2.7.3 Wordpress

Wordpress est un framework PHP open-source pour plateforme web. Ce framework est utilisé par des millions de sites web et propose des milliers de plugins afin de répondre aux besoins de chacun. Du fait de son caractère open-source, n'importe qui peut ajouter ou modifier un plugin Wordpress et le proposer à la communauté. Cependant cela apporte un risque de compatibilité car tout le monde ne connaît pas les besoins de tous les plugins, ce qui peut engendrer des problèmes de compatibilité entre différents plugins. Dans Wordpress, les différents plugins choisis peuvent s'enregistrer dans une liste de plugins actifs et faire appel à des fonctions les avertissant d'évènements pouvant déclancher une action. De manière générale, les plugins Wordpress interagissent peu entre eux ce qui diminue les problèmes de dépendance et rend leur configuration plus simple [56, 46].

2.8 Outils pour lignes de produits

Il existe de nombreux outils ou frameworks qui permettent de faciliter les différentes étapes de création d'une ligne de produits. Cependant, rares sont ceux maintenus au-dela de la durée de vie du programme pour lequel ils ont été créés. La grande majorité de ces outils ne couvre qu'une partie du cycle de développement d'une ligne de produits car il est très complexe de maitriser toutes les fonctionnalités nécessaires pour prétendre couvrir l'entièreté du cycle de vie d'une ligne de produits. Ces outils se trouvent limités dans leurs performances lorsque les configurations possibles augmentent car cela représente un problème NP-complet [30].

2.8.1 Pure : :variants

Pure : :variants est un plugin commercial populaire pour Eclipse facilitant le développement des lignes de production [30]. Ce plugin définit un family model comme une structure hiérarchique qui décrit la structure des différents variants ainsi que leur liaison avec le programme [59]. Cet outil offre plusieurs facilités de programmation telles qu'un éditeur qui simplifie l'utilisation de structures en arbre, des méchanismes permettant de modulariser des features models ainsi

qu'un accès rapide à plusieurs métriques en rapport avec la ligne de produits comme le nombre de configurations valides. Pure :variants supporte de multiples techniques de variabilité dans plusieurs langages de programmation ainsi que des langages de modélisation tels que UMl [30].

Ce plugin offre une bonne flexibilité permettant l'intégration d'autres outils tels que Git. Il est l'un des outils pour développement de lignes de produits le plus abouti et mature, cependant son aspect commercial freine sa progression par rapport à d'autres outils [30].

2.8.2 FeatureIDE

Développé en 2004, FeatureIDE est un framework Eclipse open-source pour faire de la programmation orientée feature. FeatureIDE est un des rares outils qui couvrent tout le cycle de vie d'une ligne de produits car son objectif est de faciliter la création de ligne de produits en proposant un outil complet. Il est également possible de travailler par une interface graphique lors de la création du feature model. Cependant, d'après les recherches, FeatureIDE présente des difficultés avec les attributs qui doivent être enregistrés manuellement dans le fichier XMl [30]. FeatureIDE, en plus d'être open-source, propose une ouverture aux plugins, ce qui permet de personnaliser certains aspects de l'outil à des besoins précis ou également d'ajouter de nouveaux mécanismes de variabilité. Ce framework fonctionne relativement proche du code ce qui permet un bon support pour les techniques de variation qui sont proposées [30].

Différents outils font partie intégrante de FeatureIDE, cependant ils ne peuvent pas toujours être utilisés dans un même projet, de la même manière qu'il est complexe de combiner différents langages de programmation dans un même projet. Ces outils qui permettent entre autres, la programmation orientée feature, la programmation orientée aspect, la programmation orientée delta et les directives préprocesseur, sont [83, 30] :

- AHEAD : pour la programmation orientée Feature
- Aspectj : pour la programmation orientée Aspect
- FeatureHouse : pour la programmation orientée Feature
- FeatureC++ : pour la programmation orientée Feature
- Munge : pour la compilation conditionnelle
- Antenna : pour la compilation conditionnelle
- Colligens : pour les directives préprocesseur

Afin de garder la voie ouverte à d'autres outils, FeatureIDE enregistre les features models sous format standard XML. L'avantage certain d'utiliser des outils tels que pure : :variants ou FeatureIDE réside dans le fait qu'ils peuvent permettre des aides lors de la configuration d'un produit en calculant les possibilités de variations possibles, distinguer les variants qui ne peuvent être sélectionnés au risque de violer une contrainte ou encore des fonctionnalités de tracing des différentes features, etc [30, 83]. FeatureIDE propose également une approche nommée VELVET qui facilite la migration de variant d'une ligne de produits

vers une autre [83].

2.8.3 GEARS

GEARS est un framework développé par la société BigLever qui a pour objectif de faciliter le cycle de vie d'une ligne de produits. Ce framework répandu conçoit la division d'une ligne de produits en variants sur base de ses features [52]. BigLever propose également une API permettant de faire fonctionner GEARS avec d'autres outils [40].

2.8.4 SPLOT

Lancé en 2009 S.P.L.O.T., qui est toujours en phase expérimentale, offre une interface web permettant de configurer et traiter des feature models. Malgré plusieurs fonctionnalités qui ne sont pas encore disponibles, S.P.L.O.T. propose actuellement deux fonctionnalitées principales [49] :

— Résonnement : calcul de statistiques et mesures de propriétés d'un feature model et de ses différentes features.
— Configuration : configurateur capable de propager une configuration au différents endroits impactés par celle-ci, rendant la configuration plus aisée.

Conclusion

Ce chapitre était consacré à la présentation des fondements des lignes de produits et de variabilité adoptés pour la réalisation du projet mémoire présent. Certains problèmes de tests d'une ligne de produits sont également introduits et mis en parallèles à des solutions. Des outils de développement dédié à la gestion de la variabilité sont également présentés.

Chapitre 3

Conception d'un langage informatique

Introduction

Ce chapitre est dédié à la présentation des différents type de langages ainsi que certains outils existant dans la littérature pour écrire un nouveau langage. Ceci va nous permettre de faire un choix argumenté concernant un langage à utiliser pour notre moteur de ligne de produits.

3.1 DSL

Les langages de domaines spécifiques (DSL) offrent une possibilité de créer un langage de spécification pour une ligne de produits facilement compréhensible par l'utilisateur car il utiliserait des termes et concepts du domaine [34, 20]. Cependant, étant donné que les DSL sont spécifiques à un domaine donné, ils perdent dans leur facteur de généralité [14]. Idéalement, un DSL devrait être déclaratif afin de décrire les variants spécifiques voulus dans le programme final qui seront assemblés par l'interpréteur pour répondre aux demandes clients. Cela permettrait au client de spécifier uniquement ce qu'il a besoin et non comment faire ce dont il a besoin. Malheureusement, il est complexe de traiter un DSL uniquement déclaratif [34, 8]. Il est assez accessible pour tout programmeur de créer un DSL se basant par exemple sur la structure XML, car la syntaxe est relativement permissive et il est assez simple avec certains outils tels que JAXB de parser un fichier XML [65]. Selon la méthode d'implémentation utilisée, il est possible de distinguer deux types de DSL différents.

3.1.1 DSL interne

Dans un premier temps, il est possible de définir un DSL se basant sur un langage hôte. Ces DSL, dits internes, sont alors liés directement à la grammaire

du langage sur lequel ils se basent ce qui peut compliquer la conception du DSL [55, 20]. Un avantage de ces DSL est la possibilité d'accéder à des librairies prévues pour le langage hôte. De plus, la syntaxe étant héritée du langage hôte, les outils pour vérifier celle-ci peuvent servir pour le DSL interne. Il est plus aisé avec ce type de DSL d'adapter le modèle et les capacités du langage afin de l'adapter au mieux aux besoins des différents clients [48, 23].

3.1.2 DSL externe

Il est également possible de penser un DSL sans langage hôte en définissant une grammaire propre à ce nouveau langage. Cela a comme avantage d'apporter de la flexibilité dans la syntaxe ainsi que la possibilité pour un quelconque outil d'identifier le DSL sans le confondre avec un langage hôte [55, 20, 48]. Cependant, cela nécessite la tache complexe de concevoir un compilateur ou un interpréteur afin de traduire et rendre utilisable le DSL pensé soit directement par la machine ou en code source d'un autre langage [48, 23].

3.2 Outils pour DSL

3.2.1 Xtext

Xtext est un framework open source développé par eclipse. Ce framework vise à permettre de développer des langages de programmation ainsi que des DSLs [25]. L'outils Xtext offre la facilité de générer l'infrastructure complète nécessaire aux nouveaux langages créés ainsi qu'une intégration aux IDEs qui supportent le Language Server Protocol (LSP), avec par exemple l'auto-complétion ou la coloration syntaxique [25, 63]. Cependant, Eclipse bénéficie d'une meilleure intégration ainsi que davantage de fonctionnalités spécifiques à cet IDE [25, 12]. Il existe d'autres outils similaires tels que Textx. Cet outil inspiré par Xtext se différencie principalement car celui-ci est développé en Python et non en Java. Cette imitation ne bénéficie pas non plus d'une intégration aussi aboutie qu'avec les IDEs existants [25].

3.2.2 MPS

Jetbrains est un vendeur de logiciels informatiques qui veut aider les développeurs informatiques en proposant une suite d'outils. MPS (Meta Programming System) est l'un de ces outils qui est spécialisé dans le design de DSL [25]. Une spécificité de cet éditeur de DSL est l'utilisation d'une vue projectionnelle, ce qui épargne la complexité d'écrire une grammaire [15] qui facilite grandement les modifications dans le code du DSL. Cette vue projectionnelle permet également de pouvoir manipuler aisément, de manière textuelle, l'arbre syntaxique d'un langage [15, 69]. Jetbrains possédant plusieurs IDEs, MPS propose également un environnement de développement pour les langages créés [25, 69].

BaseLanguage est le méta métamodel proposé par MPS. Ce langage qui ressemble à Java, est utilisé comme langage de base qu'il est possible de modifier afin de définir un nouveau langage [15, 69]. Une fois le nouveau langage défini, il est possible de générer le code Java correspondant au modèle MPS [69].

3.2.3 MetaEdit+

Créé en 1995, MetaEdit+ est l'outil commercial de modélisation le plus utilisé [39, 17, 38]. MetaEdit+ existe sous forme de programme pour la plupart des plateformes et sous formes d'API qui respectent les standards de communication afin de faciliter son adoption et une possibilité de générer une grammaire WSDL pour la documenter [17]. La conception de DSL est la raison première de la création de MetaEdit+ [39]. Il est possible de travailler en équipe simultanément sur un même modèle à l'aide d'un serveur de sauvegardes partagées. Cet outil permet d'utiliser de multiples langages de modélisation et diverses méthodes de représentation simultanément. C'est pourquoi une synchronisation en temps réel est nécessaire au bon fonctionnement de cette solution [39, 38]. MetaEdit+ utilise un langage de modélisation appelé GOPPRR (Graph-Object-Property-Role-Relationship). Un métamodel est défini dans ce langage comme un set d'objets, dont MetaEdit+ attribue un symbole générique [17].

3.3 Grammaire

La syntaxe d'un langage peut être comparée à son orthographe. La forme et l'écriture des mots réservés d'un langage sont définies par sa syntaxe. Cette dernière peut être définie formellement dans une grammaire sous différentes syntaxes telles que EBNF, qui est utilisée entre autres par Antlr [72].

```
stat:   "if" expr "then" stat  "else" stat
    |   "while" expr "do" stat
    |   VAR ":=" expr ";"
    |   "begin" ( stat )+ "end"
    ;

expr:   atom ( "\+" atom )*
    ;

atom:   INT
    |   FLOAT
    ;
```

FIGURE 3.1 – Code d'une grammaire EBNF

La syntaxe de tout langage peut être exprimée à l'aide d'une grammaire. Cette grammaire se présente sous la forme d'un ensemble de règles syntaxiques pouvant définir la manière d'écrire une phrase dans le langage en question [32, 7]. Une grammaire peut être composée de symboles ou tokens terminaux et non-terminaux. Les tokens non-terminaux font appel à une autre règle de la grammaire tandis que les symboles terminaux peuvent être traduits textuellement. Il existe différentes classifications des types de grammaires en fonction par exemple de leur sensibilité au contexte, s'ils sont formels ou encore de leur

ambiguité[7, 42, 73].

Une grammaire peut être utilisée par un parser ou analyseur syntaxique qui est un programme qui a pour but d'analyser un code et vérifier que ce dernier respecte la structure définie dans cette grammaire [2, 77, 61]. Les parsers de type récursif descendant, nommés aussi LL, sont privilégiés par les programmeurs car ils ont une meilleure structure [73] qui offre une meilleure flexibilité. Ils sont plus faciles à développer [61] grace à une correlation attractive entre la grammaire et le parser [74]. Pour effectuer l'analyse lexicale, l'analyseur syntaxique doit pouvoir, en respectant les règles écrites dans la grammaire, regrouper des tokens et ainsi construire l'abre syntaxique abstrait [77]. Cette analyse permet également de bloquer les erreurs syntaxiques [61].

Il existe différentes syntaxes qui permettent d'écrire une grammaire pour un analyseur syntaxique. EBNF (Extended Backus-Naur Form) est une syntaxe connue pour écrire des grammaires, celle-ci est une évolution de la syntaxe BNF. Le grand avantage de EBNF est l'utilisation d'itérations au lieu de récursions, ce qui donne un gain de performance mais également la possibilité d'écrire des grammaires plus compactes et plus simples à écrire et maintenir. Dans une grammaire EBNF, la droite d'une formule est une expression régulière pouvant contenir des terminaux et non-terminaux [74].

Conclusion

Tout au long de ce chapitre nous avons introduit la notion de DSL ainsi que certains outils qui visent à faciliter leur création. Il est, à ce niveau, possible d'argumenter le choix du type de langage à utiliser dans l'implémentation d'un moteur de ligne de produits pour communiquer à celui-ci les informations nécessaires.

Chapitre 4

Implémentation du moteur de lignes de produits

Introduction

L'implémentation détaillée dans ce chapitre répond à l'objectif principal de ce mémoire sur les lignes de produits. Le travail sur cette plateforme se veut d'offrir une manière simple et élégante de construire et générer toute possible configuration d'une ligne de produits. Ce moteur entend faciliter l'utilisation de lignes de produits et permettre d'augmenter son adoption dans un premier temps avec une architecture ouverte, pour que des développeurs tiers puissent expérimenter les lignes de produits à leur niveau. Le moteur de ce projet est responsable de la gestion automatisée de la ligne de produits. Basé sur le langage Java, ce programme peut, à l'aide d'un fichier de configuration qui sera détaillé dans une des sections suivantes, compiler une configuration donnée d'une ligne de produits. Un lien GitHub est disponible contenant le code source à l'annexe 1.

4.1 Use cases

4.1.1 Conception d'une ligne de produits

Concevoir une ligne de produits pour qu'elle puisse être utilisée par ce moteur se réalise en plusieurs étapes :

1. Conception de la ligne de produits : la ligne de produits doit s'articuler autour d'une partie commune sur laquelle les différents variants viennent s'ajouter ou s'enlever.
2. Définition du FeatureModel : un fichier « *FeatureModel.xml* » doit être créé dans le dossier racine du module principal de la ligne de produits. Comme expliqué dans le chapitre « XML.FeatureModel », ce fichier dé-

finit les différents fichiers à ajouter ou actions à effectuer pour lier un variant.

3. Publication : la ligne de produits doit être publiée, soit sur un repository Github, soit dans un répertoire local en respectant les règles expliquées dans le chapitre « Import ».

4.1.2 Sélectionner une configuration d'une ligne de produits

Toute personne voulant utiliser ce moteur de ligne de produits devrait suivre les étapes suivantes :

1. Prendre connaissance des variants disponibles pour la ligne de produits donnée.
2. Définir la configuration voulue dans un fichier xml en respectant les règles détaillées dans le chapitre « XML.Configuration ».
3. Exécuter le moteur en fournissant le fichier de configuration.

Une personne qui voudrait obtenir sa propre configuration devrait exécuter la commande suivante :

```
java -jar engine.jar <chemin du fichier de configuration><exécution des tests>
```

Le premier paramètre est constitué simplement du chemin vers le fichier xml contenant la configuration de la ligne de produits voulue. Le second paramètre utilisé à l'exécution des tests à été ajouté afin de permettre de ne pas exécuter de tests lors de la création du package jar. Lors de la création du fichier jar pour le projet de test décrit dans le chapitre suivant, une erreur est survenue. Durant la génération d'une application qui utilise le framework Spring, Maven exécute une série de tests fournis par Spring avant de générer le fichier jar. Si par exemple, l'application à compiler utilise une base de données manipulée par Spring, le framework demandera à Maven la vérification de la connexion à cette base de données. Le moteur de ligne de produits ne pouvant pas connaitre tous les frameworks qui effecturaient ce genre de tests, il s'est montré plus pertinent d'ajouter un paramètre pour faire abstraction des tests lors de la génération du fichier jar par Maven. Ce paramètre est un boolean dont la valeur true fait abstraction des tests.

4.1.3 Vérification d'un featureModel

Afin d'aider à la bonne écriture du featureModel, duquel dépend la capacité du moteur à créer une configuration, celui-ci peut effectuer une vérification de la structure. Une méthode nommée « *getxsdDeclaration* » qui a la responsabilité de fournir du code XSD permet de valider la structure du featureModel pour les parties concernant une technique de variabilité précise. Pour ce faire, il suffit

de remplacer le paramètre fournissant le chemin vers un fichier de configuration par un chemin vers le featureModel à vérifier.

```
java -jar engine.jar <chemin du fichier featureModel>
```

4.2 Architecture

L'utilisation de plugin a semblé d'une grande importance dans ce projet étant donné le grand nombre de techniques de variation déjà existantes et les nombreuses futures techniques ayant des fins de variabilité. C'est pourquoi ce moteur de ligne de produits tourne autour de l'interface « *Interpreter* » qui définit le minimum de méthodes nécessaires à tout plugin voulant fonctionner avec ce projet.

Le moteur utilise majoritairement deux classes. La classe « *JavaFileManager* » est responsable de la gestion des fichiers. Cette classe permet de récupérer les informations d'un fichier XML, downloader les fichiers nécessaires depuis un répository Git, effectuer le traitement nécessaire pour vérifier le XML à l'aide de XSD et manipuler des fichiers. La classe « *Importer* » quant à elle, est chargée de vérifier tout ce qui tient à l'importation de la configuration sélectionnée. C'est à dire récupérer le FeatureModel, vérifier que la sélection est valide et vérifier s'il s'agit d'un import depuis un répository Git ou d'un dossier. L'interface « *Interpreter* » est très important car comme mentionné ci-dessus, elle permet l'ouverture pour l'utilisation de plugins qui sont chargés par la classe « *PluginLoader* ». Elle définit donc les méthodes obligatoires pour la conception de plugins afin de pouvoir traiter un point de variation suivant la méthode voulue, mais également de fournir le texte XSD pour vérifier la validité du FeatureModel et vérifier l'importation des fichiers nécessaires. Le diagramme de classes de ce projet se présente comme ci dessous :

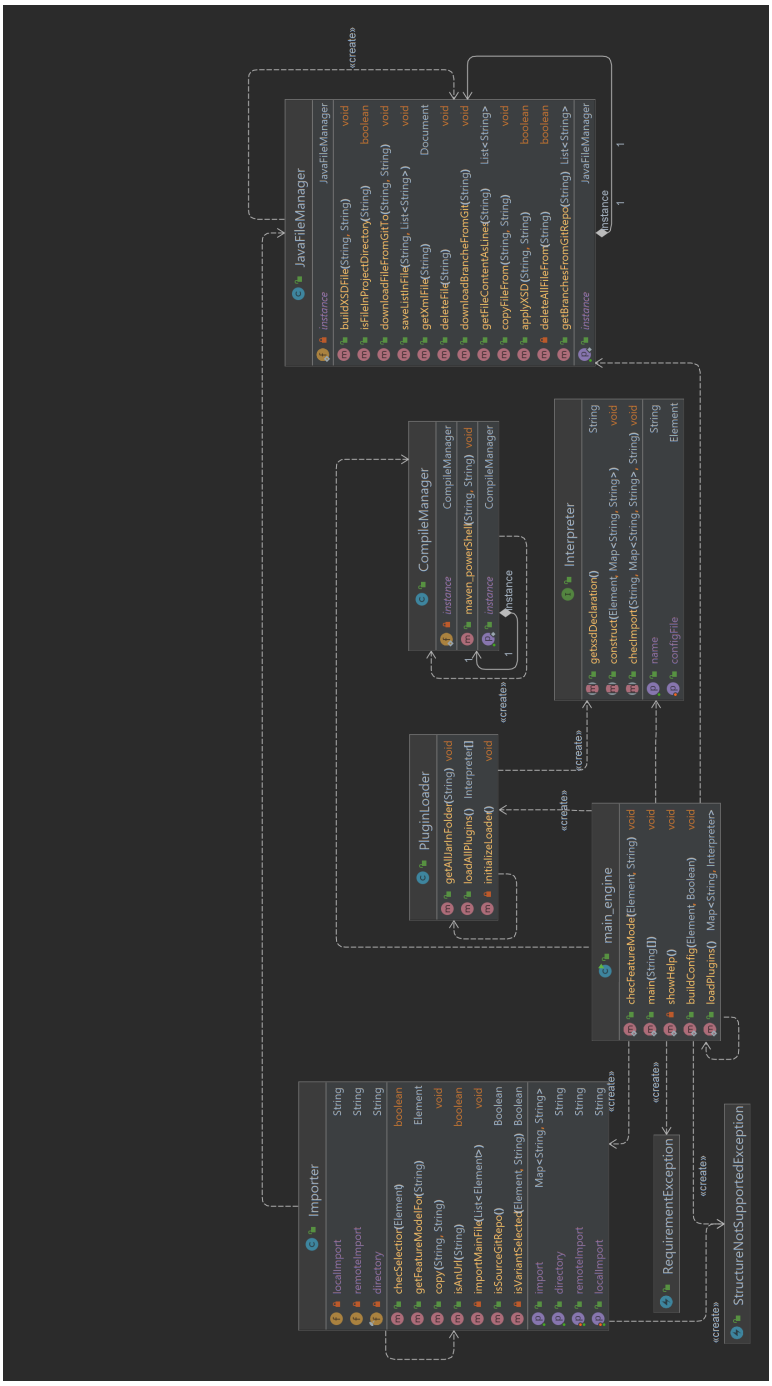

FIGURE 4.1 – Diagramme de classes du moteur de ligne de produits

4.3 XML

Après de longues réflexions, il a été décidé qu'écrire un DSL de type externe avec un outil tel que Antlr s'avèrerait relativement complexe et couteux en temps. L'approche XVCL a paru être une solution plus pertinente avec un meilleur potentiel d'évolution, ainsi qu'une meilleure ouverture pour des techniques de variabilité non encore implémentées. XML est donc utilisé comme base afin de définir un DSL interne dont on se sert pour communiquer les différentes caractéristiques d'une ligne de produits et pour sélectionner les différents variants souhaités pour une configuration.

La définition de la ligne de produits et le choix de la configuration se font dans deux fichiers distincts.

4.3.1 FeatureModel

Le featureModel est écrit dans un fichier *'FeatureModel.xml'* et contient la définition des différents variants qui existent pour la ligne de produits. Pour des soucis de gestion de fichier, celui-ci doit obligatoirement se trouver dans le dossier racine du module principal de la ligne de produits. Il est présenté comme ceci :

```
<?xml version="1.0" encoding="UTF-8"?>
<FeatureModel xmlns:th="http://www.w3.org/1999/xhtml">
    <SpringAspect name="logging" url="">
        <file path="\src\main\java\com\example\testspring\dbAccess\DBManagerAspect.java" />
        <class name="DBManagerAspect" />
    </SpringAspect>
    <SpringPreprocessor name="SGBD" url="">
        <file path="\src\main\resources\application.properties">
            <var datasource="mysql" />
        </file>
    </SpringPreprocessor>
    <Delta name="liker" require="compte">
        <modif...>
    </Delta>
    <Delta name="compte">
        <addFile...>
    </Delta>
    <Plugin name="recherche" url="">
        <file path="src/main/java/com/example/testspring/Plugin/Recherche.java" />
        <file path="src/main/java/com/example/testspring/Interface/PluginInterface.java" />
        <file path="src/main/resources/static/rechercheNavBar.html" />
    </Plugin>
</FeatureModel>
```

FIGURE 4.2 – Exemple de featureModel pour une ligne de produits

La structure du fichier XML, comprend l'élément FeatureModel comme racine ainsi que les différents variants présents à l'intérieur. Tout variant doit obligatoirement posséder l'attribut « *name* » afin de pouvoir être identifié dans

le fichier de configuration. Il est également possible de définir avec l'attribut « *require* » une dépendance avec un autre variant. L'attribut « *denied* » permet enfin de définir deux variants ne pouvant pas cohabiter dans une même configuration.

4.3.2 Configuration

Le moteur de ligne de produits accepte une configuration de ligne de produits sous forme d'un fichier xml dont le choix du nom est laissé libre. Ce fichier contient la liste des différents variants souhaités. Certaines techniques de variation, pourraient avoir besoin de paramètres. C'est le cas du plugin *SpringPreprocessor* qui nécessite la transmission d'une valeur par le fichier de configuration. Ce fichier est présenté comme ceci :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <import uri="C:\unamur\Master\memoire\SPL\main\featureModel.xml"/>
    <SpringAspect names="logging"/>
    <SpringPreprocessor name="SGBD">
        <var datasource="mysql" />
    </SpringPreprocessor>
    <Delta name="liker" />
    <Delta name="compte" />
    <Plugin name="recherche" />
</Configuration>
```

FIGURE 4.3 – Exemple d'une configuration pour une ligne de produits

4.4 Validation du XML

Le use case spécifiant la vérification du featureModel demande à faire appel à une méthode permettant de vérifier la structure du fichier XML. Principalement trois langages évoqués ci-dessous furent susceptibles de pouvoir effectuer cette vérification. Le format de fichier XML est utilisé de nos jours comme un moyen d'organiser de l'information à transmettre, par exemple par internet. Dans un premier temps, il est important de vérifier le bon respect de la syntaxe XML pour être qualifié de bien formé. Il faut également vérifier que ce fichier XML respecte la structure désirée afin d'être considéré comme valide [51]. En 1998, le langage DTD (Document Type Definition) a été inventé pour effectuer cette vérification de structure, cependant ce langage s'est avéré limité en vérification pour de multiples raisons [88]. C'est pourquoi d'autres langages et outils ont été proposés par la suite dont XSD, SOX, XSV, ...[45].

Comme expliqué dans la section précédente, ce moteur de ligne de produits utilise le format XML pour définir un featureModel et également pour sélec-

tionner une configuration voulue. Par conséquent, il a été nécessaire de pouvoir efficacement valider ces derniers. Pour ce faire, XSD a semblé être le meilleur choix étant donné sa popularité et ses fonctionnalités qui correspondent tout à fait aux besoins. Le moteur de ligne de produits ne pouvant pas connaître la structure XML définie par les plugins, ceux-ci doivent donc communiquer la structure XML qu'ils attendent respectivement. Ainsi, le moteur qui possède la structure de base pour tout featureModel peut assembler les morceaux des différents plugins et valider ou non tout featureModel qu'il rencontre.

4.4.1 DTD

DTD fut le premier langage dévoilé en 1998 ayant pour but de définir la structure d'un document XML. Un fichier DTD n'est pas obligatoire à l'existence d'un fichier XML. Depuis, d'autres langages ont été proposés en remplacement à cause des différentes limitations de DTD, mais aussi suite à l'évolution de l'utilisation du XML [45]. La grande difficulté de ce langage réside dans le fait que celui-ci possède une syntaxe différente du XML ce qui rend plus complexe son utilisation comparé par exemple à XSD qui utilise la même syntaxe que le XML. DTD ne fut pas qualifié de facile à utiliser [51].

```
<!ELEMENT BookCatalog (Book)*>
<!ELEMENT Book (Title, Authors, Publisher, ISBN, DatePublished,
Price)>
<!ATTLIST Book pubCountry CDATA #IMPLIED >
<!ELEMENT Title (#PCDATA)>
<!ATTLIST Title series CDATA #IMPLIED>
<!ELEMENT Authors (Author)+>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Publisher (#PCDATA) >
<!ELEMENT ISBN (#PCDATA) >
<!ELEMENT DatePublished (#PCDATA) >
<!ELEMENT Price (#PCDATA) >
```

FIGURE 4.4 – Fichier DTD permettant de valider le fichier XML

4.4.2 SOX

Développé par Commerce One, SOX est un autre langage permettant de définir la structure et sémantique d'un fichier XML. La particularité de ce langage par rapport à son ancêtre DTD est l'apport de la notion d'objet. Cette notion apporte, comme dans la programmation, des possibilités d'héritage ainsi que le choix de types de données ayant une certaine hiérarchie. Comme XSD, SOX offre l'avantage d'utiliser la syntaxe XML [43].

4.4.3 XSD

L'organisme de standardisation W3C en 2001 a introduit le langage XSD (XML Schema Definition). Basé sur le langage XML, XSD a pour but de décrire

formellement la structure d'un fichier XML. Ce langage a pour veut rendre utilisable le format XML de façon fiable pour échanger de l'information et éviter ainsi de nombreux tests pour s'assurer de la conformité du document [88]. Il est possible de définir exactement le type de données qui doit s'y trouver, qu'il soit simple, complexe ou compositionnel de type séquence, choix ou tous les enfants. Les attributs peuvent également être définis et il est possible de les rendre obligatoires, optionnels ou prohibés [45, 88]. Les deux grands avantages de XDS par rapport à DTD sont [43] :

— les capacités et fonctionnalités plus avancées pour définir le type de données attendu dans le document XML.
— un document XSD s'écrit en utilisant la syntaxe XML ce qui facilite grandement son apprentisage et utilisation, car éditable avec un simple éditeur XML.

Ci-dessous une partie du fichier XSD permettant au moteur de ligne de produits de valider le fichier XML qui contient un featureModel. Celui-ci est composé de différents plugins installés qui fournissent le XSD nécessaire à valider les parties de ce featureModel qui les concernent. C'est la méthode « *getxsdDeclaration* » présente dans l'interface « *Interpreter* » qui permet de récupérer auprès de chaque plugin le XDS qui s'y rapporte.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="FeatureModel">
    <xs:complexType>
    <xs:choice maxOccurs="unbounded">
        <xs:element name="SpringAspect">
        <xs:complexType>
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="file">
            <xs:complexType>
                <xs:attribute name="path" type="xs:string"/>
            </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="require" type="xs:string"/>
        </xs:complexType>
        </xs:element>
        <xs:element name="SpringPreprocessor">
        <xs:element name="Delta">
        <xs:element name="Plugin">
        <xs:complexType>
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="file">
            <xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="require" type="xs:string"/>
        </xs:complexType>
        </xs:element>
    </xs:choice>
    </xs:complexType>
    </xs:element>
</xs:schema>
```

FIGURE 4.5 – Fichier XSD permettant de valider le fichier XML d'un featureModel

4.5 Plugin

Nous avons choisi de laisser ouvert la possibilité d'introduire de nouvelles techniques de variabilité pour lier les variants. C'est pourquoi la méthode des plugins a été utilisée afin que de nouveaux types de connexions de variants puissent être utilisés. Ceux ci devront implémenter une interface comprenant ces méthodes :

- *getName* : permet d'identifier le plugin à appliquer sur un variant utilisé dans le fichier XML de configuration ou de FeatureModel
- *getxsdDeclaration* : afin de vérifier la bonne struture d'un variant dans le featureModel, cette méthode fournit le XSD permettant de valider chaque appel dans le featureModel à ce plugin
- *checImport* : vérifie que les différents fichiers à importer dans la configuration sont bien importés
- *construct* : méthode principale de chaque plugin, c'est cette méthode qui lance la liaison du variant
- *setConfigFile* : permet au variant d'accéder au fichier de configuration pour, par exemple, fournir une variable

Les mécanismes de variabilité implémentés au moment de la remise de ce mémoire sont au nombre de quatre. Ceci pour valider le moteur sur plusieurs techniques de variabilités connues des développeurs. La classe « *PluginLoader* » est responsable de charger le dossier « *plugins* » où devront être déposés les plugins à utiliser. Ceux-ci devront se présenter sous forme de fichiers *.jar* contenant ce qui est nécessaire au plugin. Chargés en mémoire, ils sont ensuite instanciés s'ils implémentent correctement l'interface « *interpreter* ».

4.5.1 SpringAspect

Ce plugin a comme responsabilité de gérer la programmation orientée aspect en utilisant le framework Spring qui offre un module qui prend en charge ce paradigme. La programmation orientée aspect avec Spring est relativement simple d'accès pour tout programmeur. Le plugin de ce moteur pour la programmation orientée aspect ne doit qu'importer les fichiers Java contenant les aspects désirés.

La structure de ce plugin dans le fichier *FeatureModel* doit traduire l'ajout du fichier contenant le code de l'aspect à ajouter. C'est pourquoi l'élement *SpringAspect* avec le paramètre *name* définissant le nom du variant en question, contient un élément *file* avec le paramètre *path* contenant le chemin depuis la racine du dossier de ce variant vers le fichier contenant l'aspect. Dans le fichier de configuration, il suffit de définir un élément XML nommé *SpringAspect* avec le paramètre *name* où le nom du variant doit être spécifié.

```
1 <xs:element name="SpringAspect">
2 <xs:complexType>
3 <xs:sequence minOccurs="1" maxOccurs="unbounded">
4     <xs:element name="file">
```

```
5     <xs:complexType>
6         <xs:attribute name="path" type="xs:string"/>
7     </xs:complexType>
8     </xs:element>
9 </xs:sequence>
10 <xs:attribute name="name" type="xs:string" use="required"/>
11 <xs:attribute name="require" type="xs:string"/>
12 </xs:complexType>
13 </xs:element>
```

Listing 4.1 – Code XSD pour valider le featureModel SpringAspect

Ci-dessus les lignes XSD du paradigme de programmation orientée aspect informent sur la structure du fichier featureModel à respecter. Les lignes 3 à 9 concernent les fichiers qui seront importés tandis que les lignes 10 et 11 informent sur les paramètres de l'élément principal de ce paradigme.

4.5.2 Delta

L'implémentation du plugin de programmation orientée delta fut le plugin le plus long à implémenter, compte tenu du fait des capacités nécessaires afin de prétendre pouvoir faire de la programmation delta.

La structure du fichier *FeatureModel*, pour la définition d'un variant utilisant la programmation delta, se divise en trois parties pour les trois opérations autorisées lors de la liaison du variant.

1. *addFile* permet d'ajouter une liste de fichiers définis par des éléments xml *file* avec un paramètre *path* contenant le chemin vers le fichier depuis la racine du dossier de ce variant
2. *deleteFile* permet de supprimer une liste de fichiers définis par des éléments xml *file* avec un paramètre *path* contenant le chemin vers le fichier depuis la racine du dossier de ce variant
3. *modif* permet de modifier une liste de fichiers définis par des éléments xml *file* avec un paramètre *path* contenant le chemin vers le fichier depuis la racine du dossier de ce variant ainsi qu'un paramètre *type* permettant de fournir au plugin le nom du langage à des fins de traitement avec la grammaire du langage concerné. Un sous-élément *add* contient enfin le contenant à ajouter au fichier

Le fichier de configuration doit simplement contenir un élément XML *Delta* avec un paramètre *name* contenant le nom du variant.

```
1 <xs:element name="Delta">
2 <xs:complexType>
3 <xs:choice maxOccurs="unbounded">
4     <xs:element name="modif">
5         <xs:complexType>
```

```
6         <xs:sequence minOccurs="1" maxOccurs="unbounded">
7             <xs:element name="file">
8                 <xs:complexType>
9                 <xs:sequence minOccurs="1" maxOccurs="unbounded">
10                    <xs:element name="add" />
11                </xs:sequence>
12                <xs:attribute name="path" type="xs:string"/>
13                <xs:attribute name="type" type="xs:string"/>
14                </xs:complexType>
15            </xs:element>
16        </xs:sequence>
17        </xs:complexType>
18    </xs:element>
19    <xs:element name="addFile">
20        <xs:complexType>
21        <xs:sequence minOccurs="1" maxOccurs="unbounded">
22            <xs:element name="file">
23            <xs:complexType>
24                <xs:attribute name="path" type="xs:string"/>
25            </xs:complexType>
26            </xs:element>
27        </xs:sequence>
28        <xs:attribute name="name" type="xs:string"/>
29        </xs:complexType>
30    </xs:element>
31    <xs:element name="deleteFile">
32        <xs:complexType>
33        <xs:sequence minOccurs="1" maxOccurs="unbounded">
34            <xs:element name="file">
35            <xs:complexType>
36                <xs:attribute name="path" type="xs:string"/>
37            </xs:complexType>
38            </xs:element>
39        </xs:sequence>
40        <xs:attribute name="name" type="xs:string"/>
41        </xs:complexType>
42    </xs:element>
43 </xs:choice>
44 <xs:attribute name="name" type="xs:string" use="required"/>
45 <xs:attribute name="require" type="xs:string"/>
46 </xs:complexType>
47 </xs:element>
```

Listing 4.2 – Code XSD pour valider le featureModel delta

Ci-dessus les lignes XSD du paradigme de programmation orientée delta informent sur la structure du fichier featureModel à respecter. Les lignes 4 à 18

concernent l'action de modifier un fichier avec les lignes qu'il faut ajouter au fichier voulu. Les lignes 19 et 30 informent sur l'action d'ajouter un fichier. Finalement, les lignes 31 et 24 concernent la suppression d'un fichier.

4.5.3 Plugin

Comme pour la programmation orientée aspect, le plugin permettant la gestion des plugins dans la ligne de produits doit uniquement pouvoir importer le fichier JAR de ceux-ci. Pour le projet servant à valider ce moteur de ligne de produits, les plugins sont uniquement chargés si ceux ci se trouvent dans un dossier « plugins » cependant pour d'autres lignes de produits, ces plugins peuvent devoir se trouver dans un autre dossier.

De même que pour le plugin *SpringAspect*, la structure de ce plugin dans le fichier *FeatureModel* communique les informations nécessaires à l'ajout du fichier *.jar* contenant le plugin à ajouter. C'est pourquoi l'élement *Plugin* avec le paramètre *name* qui définit le nom du variant en question, contient au minimum un élément *file* avec le paramètre *path* contenant le chemin depuis la racine du dossier de ce variant vers le fichier contenant le plugin. Dans le fichier de configuration, il suffit de définir un élément xml nommé *Plugin* avec le paramètre *name* où le nom du variant doit être spécifié.

```
1  <xs:element name="Plugin">
2  <xs:complexType>
3  <xs:sequence minOccurs="1" maxOccurs="unbounded">
4      <xs:element name="file">
5      <xs:complexType>
6          <xs:attribute name="path" type="xs:string"/>
7      </xs:complexType>
8      </xs:element>
9  </xs:sequence>
10 <xs:attribute name="name" type="xs:string" use="required"/>
11 <xs:attribute name="require" type="xs:string"/>
12 </xs:complexType>
13 </xs:element>
```

Listing 4.3 – Code XSD pour valider le featureModel plugin

Ci-dessus les lignes XSD du mécanismes plugin informent sur la structure du fichier featureModel à adopter. Les lignes 3 à 9 concernent les fichiers qui seront importés tandis que les lignes 10 et 11 informent sur les paramètres de l'élément principal de ce mécanisme.

4.5.4 SpringPreprocessor

Le langage Java n'accepte pas nativement de directives préprocesseur. Il existe certains outils tels que Manifold [1] qui permettent d'utiliser des directives préprocesseur, cependant, l'ouverture qu'offre l'architecture du moteur de ligne

de produits, permet de créer une syntaxe de préprocesseur personnalisée. Cela permet de faire fonctionner la liaison d'un variant donné avec une syntaxe choisie et de fonctionner comme des directives préprocesseur dans les langages tels que C. La syntaxe pensée pour ce plugin SpringPreprocessor accepte deux types de directives préprocesseur. Chacune de ces directives peut être influencée par une variable transmise depuis le fichier de configuration.

— Condition : la syntaxe *##ifdef* permet de définir une condition d'inclusion. La syntaxe *##else* permet de définir un bloc de code alternatif tandis que la syntaxe *##endif* permet d'indiquer la fin de la condition.
— Définition : Il est possible de définir une constante avec le préfixe *##define* et chaque itération de cette variable dans le fichier sera remplacée par sa valeur.

Le *FeatureModel* qui définit un variant utilisant *SpringPreprocessor* doit posséder, dans l'élément principal, un paramètre *name* spécifiant le nom du variant, dans l'élément principal. Chaque fichier possédant une directive préprocesseur à manipuler, doit être listé par un élément *file* avec le paramètres *path* contenant le chemin depuis la racine du dossier de ce variant. A l'intérieur de cet élément, se trouve un ou plusieurs éléments *var* avec, comme nom de paramètre, le nom de la variable et comme valeur de ce paramètre, la valeur choisie par défaut. Cette variable peut être modifiée dans le fichier de configuration afin de lui donner une valeur précise.

```
1  <xs:element name="SpringPreprocessor">
2      <xs:complexType>
3          <xs:sequence minOccurs="1" maxOccurs="unbounded">
4              <xs:element name="file">
5                  <xs:complexType>
6                      <xs:sequence minOccurs="0" maxOccurs="
       unbounded">
7                          <xs:element name="var">
8                              <xs:complexType>
9                                  <xs:simpleContent>
10                                     <xs:extension base="xs:string
       ">
11                                         <xs:anyAttribute
       processContents="lax"/>
12                                     </xs:extension>
13                                 </xs:simpleContent>
14                             </xs:complexType>
15                         </xs:element>
16                     </xs:sequence>
17                     <xs:attribute name="path" type="xs:string"/>
18                 </xs:complexType>
19             </xs:element>
20         </xs:sequence>
21         <xs:attribute name="name" type="xs:string" use="required"
```

```
    />
22          <xs:attribute name="require" type="xs:string"/>
23      </xs:complexType>
24  </xs:element>
```

Listing 4.4 – Code XSD pour valider le featureModel SpringPréprocessor

Ci-dessus les lignes XSD informent sur la structure du fichier featureModel à adopter afin de manipuler des directives préprocesseur. Les lignes 3 à 20 concernent les fichiers et variables nécessaires aux manipulations des directives préprocesseur tandis que les lignes 21 et 22 informent sur les paramètres de l'élément principal de ce mécanisme.

4.6 Import

Il a été décidé de permettre deux types d'imports où la structure et les fichiers de la ligne de produits sont stockés. Le premier système de fichiers autorisé est un répertoire local défini par un path avec un dossier spécifique pour chaque variant. Le deuxième système de fichiers accepté est Github utilisant le système de branches afin de séparer les fichiers des différents variants.

4.6.1 Github

Depuis le fichier de configuration, il est possible d'importer une ligne de produits qui est sauvegardée dans un repository Github. Afin que la configuration puisse fonctionner, chaque variant devra néanmoins être sauvegardé sur une branche nommée avec le même nom que celui spécifié dans le FeatureModel. Le fichier FeatureModel doit être à la racine de la branche *main*. La branche main est importée dans un premier temps et le moteur ajoute, lors de la configuration, les fichiers nécessaires sauvegardés dans les branches correspondantes aux variants désirés.

4.6.2 Directory

Afin de permettre l'utilisation un répertoire local, il a été nécessaire de définir une certaine structure à respecter.

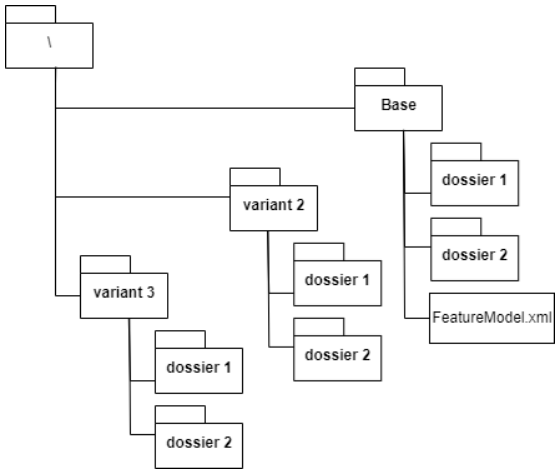

FIGURE 4.6 – Arborescence à respecter pour une ligne de produits

Comme dans le graphe ci-dessus, la structure des dossiers doit être comprise dans un dossier principal. Dans ce dernier, un premier dossier au nom libre, comprend la base choisie pour la ligne de produits ainsi que le fichier « *FeatureModel.xml* ». Les autres dossiers comprenant chacun les fichiers nécessaires aux différents variants, doivent obligatoirement être nommés avec le même nom qui est spécifié dans le FeatureModel.

4.7 Maven

Maven est un outil populaire open source de la fondation Apache, qui permet d'automatiser des opérations de développement du code Java telles que la gestion des dépendances ou la construction du package jar [21, 57]. C'est cet outil qui est utilisé pour assembler le fichier jar contenant le nécessaire pour faire fonctionner l'application qui correspond à la configuration de la ligne de produits choisie. Les commandes à exécuter par maven sont :*mvn build* et *mvn package*

La gestion des dépendances dans un projet utilisant Maven se fait dans un fichier appelé « *pom.xml* ». Il existe un repository sur internet permettant de downloader certaines dépendances. Une dépendance Maven peut être identifiée sans erreur à l'aide d'un *groupId*et un *artifactId* ainsi qu'une version et éventuellement un type (jar, war, EAR)[87].

4.8 Dépendances

Ce projet utilisant Maven comme gestionnaire de dépendances, celles-ci sont définies dans un fichier « *pom.xml* ». Plusieurs dépendances ont dû être appelées,

voici les principales :

- **jdom2** : Librairie permettant d'accéder et manipuler un fichier XML
- **jPowerShell** : Librairie pour exécuter des commandes PowerShell
- **org.eclipse.jgit** : Librairie offrant certaines manipulations d'un repository Github
- **jsoup** : Librairie servant de navigateur web dans une console
- **junit-jupiter** : Les tests unitaires nécessitent cette librairie

Conclusion

Dans ce chapitre nous avons présenté le moteur de l'usine à logicielles qui est la contribution essentielle de ce mémoire. Tout au long des sections nous avons présenté les différents aspects de ce moteur. Ce chapitre a aussi permis d'expliquer les décisions qui ont été prise que ce soit au niveau de l'architecture globale ou de l'utilisation d'une technologie en particulier.

Chapitre 5

Implémentation d'un proof of concept

Introduction

La situation utilisée pour valider le bon fonctionnement du moteur de ligne de produits est un site web d'e-commerce basé sur spring présentant une liste de produits sur la gauche de l'écran avec la possibilité d'afficher davantage de détails sur un produit sur la droite de l'écran. Il est également possible de mettre en place un système permettant de liker un produit en spécifiant un nom d'utilisateur. Dans les détails de ce produit il est alors possible de voir le nombre de personnes ayant liké ce produit. Dans la section « Ligne de produits », différents variants qui ont été pensés et implémentés concernant cette situation, sont détaillés.

5.1 Structure de l'application d'e-commerce

Un lien GitHub est disponible contenant le code source à l'annexe 1. Une représentation de l'interface utilisateur est disponible à l'annexe 2. On peut y voir sur la gauche de l'écran une liste de produits avec un code, un nom, un ean ainsi qu'une description. Sur la droite de l'écran se trouve une zone détail qui affiche les informations de l'article sélectionné. Dans cette zone peut se trouver, suivant la configuration, le nombre de like que le produit a reçu ainsi qu'un formulaire pour ajouter un like. Ces likes sont enregistrés dans une base de données MySQL ou PostgreSQL qui est gérée par le framework Spring. Sur le haut de l'écran se trouve une barre de navigation où se trouve le formulaire du plugin recherche.

« *ShopControler* » est la classe servant de lien avec la vue web. La gestion des requêtes pour la base de données sont centralisée dans la classe « *DBManager* » avant d'être traité par Hibernate. La possibilité d'ajouter des plugins à cette

WebApp a nécessité les classes « *PluginLoader* » pour charger les plugins, la classe « *PluginInterface* » dicte aux plugins ce qu'ils doivent implémenter et « *MainControler* » est utilisé comme controler backend à l'écart des requêtes du frontend. Le diagramme de classes de cette plateforme web se présente comme ci-dessous :

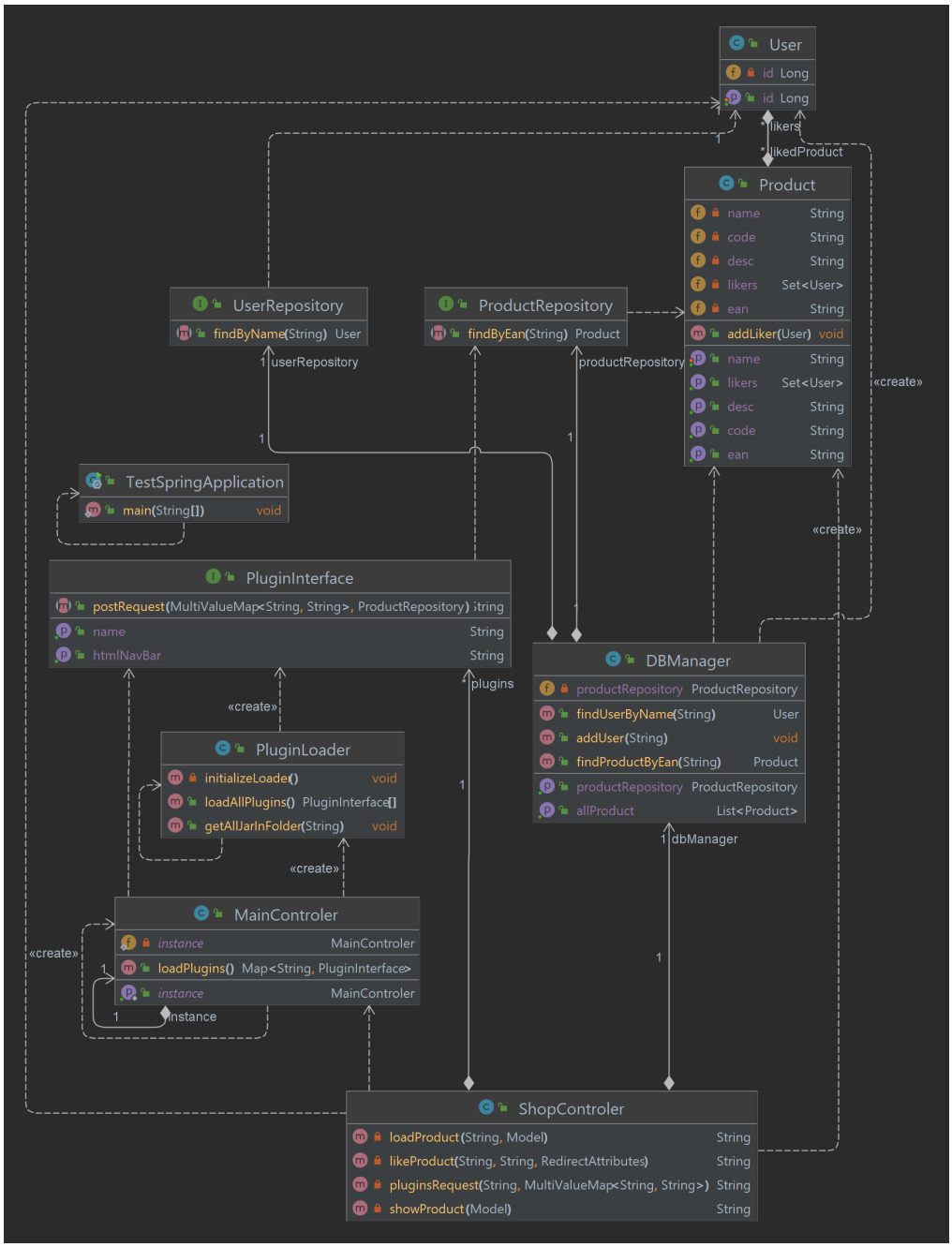

FIGURE 5.1 – Diagramme de classes de la plateforme web Shop

5.2 Spring

Cette application web utilise largement le framework Spring avec son module Spring web MVC prévu pour le développement d'application web et d'API. Mais ce framework a également servi pour la programmation orientée aspect telle qu'expliquée dans le chapitre concernant ce paradigme [57].

Spring permet également d'avoir accès à une base de données par l'intermédiaire de plusieurs framework ORM tels que Hibernate, JDO, JPA et IBATIS. Ces ORMs sont responsables de la persistance des données dans une base de données et de signaler à spring si une erreur survient [21]. Dans cette implémentation, l'ORM Hibernate est utilisé.

Hibernate est un famework de persistance de données très connu et utilisé depuis longtemps [21]. Cet ORM a pour but de faire le lien entre les objets du code source et les modèles relationnels utilisés en base de données. Grace à Hibernate, la gestion de la persistance des données se fait de manière beaucoup plus accessible. Un langage nomé HQL est utilisé et transformé lors de l'exécution de l'application afin de pouvoir s'adapter à la forme du SQL utilisé par la base de données [21]. Hibernate a été choisi pour ce projet compte tenu du fait de sa facilité d'accès ainsi que sa solide documentation et sa bonne intégration avec Spring. JPA est une API fournie par Java qui a pour but d'aider la liaison entre les classes ou objets Java et le modèle relationnel d'une base de données. Hibernate utilise cette api afin d'accéder aux objets Java [58, 57].

5.3 Ligne de produits d'e-commerce

Dans les chapitres suivants sont détaillés les variants qui ont été implémentés concernant la situation donnée à l'aide de différents mécanismes de variabilité. Pour résumer, voici la représentation visuelle grâce au diagramme de feature. Afin de montrer les capacités du moteur de ligne de produits, la situation a été réfléchie de sorte que tous les types de relations possibles entre variants soient compris dans la situation.

- un et un seul : Le type de SGBD requiert un choix entre SQLite et PostgeSQL. Cette sélection est effectuée en utilisant le plugin SpringPreprocessor.
- et : Si le variant like est choisi, il faut obligatoirement avoir le variant compte. Ces deux variants utilisent la programmation orientée delta.
- ou : Les variants plugin recherche et like sont des variants optionnels au fonctionnement du site web.

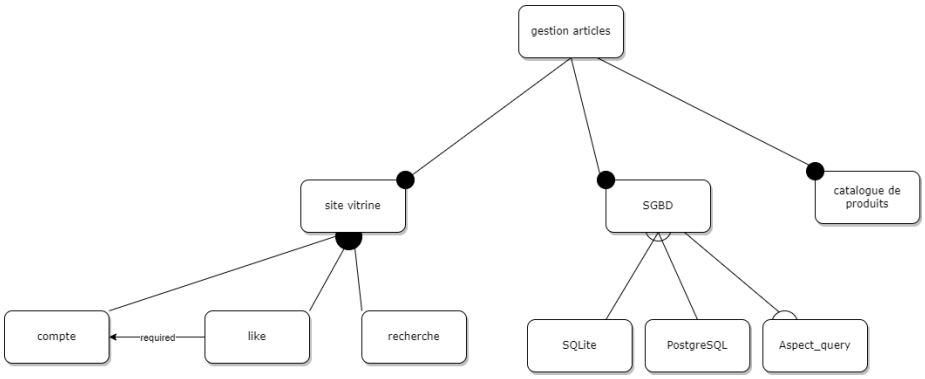

FIGURE 5.2 – Diagramme de features de la plateforme web Shop

5.3.1 Variation de la feature SGBD

Deux possibilités de SGBD ont été implémentées, soit MySQL ou PostgreSQL. Les paramètres Spring à modifier étant dans un fichier de configuration, la sélection se fait par un semblant de directives préprocesseur. Ceci confirme l'ouverture du moteur à une syntaxe personnelle ainsi que la manipulation des directives préprocesseur pour les langages supportant ces derniers, Java n'en faisant pas partie nativement.

Le point de variation de ce variant se joue dans le fichier « *application.properties* ». Ce fichier généré par Spring contient des paramètres par défaut utilisés par l'application. C'est dans ce fichier donc qu'il faut modifier les paramètres de connexion à la base de données afin de pouvoir se connecter à une base de données MySQL ou PostgreSQL. Comme expliqué dans le chapitre sur le moteur de ligne de produits, dans ce fichier, des lignes respectent une certaine syntaxe afin que certaines lignes soient ou non incluses dans le résultat final de ce fichier.

5.3.2 Variation du système de like

Un des variants implémentés a pour objectif de permettre d'enregistrer des likes pour un produit donné à l'aide d'un pseudo. Ce variant se lie au reste du programme en utilisant les principes de la programmation orientée delta. Ce variant a besoin du variant compte détaillé dans la section suivante afin de fonctionner correctement.

Outre les modifications nécessaires pour le variant compte, plusieurs fichiers doivent être modifiés, ajoutés et supprimés par le paradigme de programmation orientée delta afin d'inclure le variant like dans le produit final.

— *bibliotheque.html* doit être modifié afin d'ajouter le formulaire html qui permet à un utilisateur d'enregistrer son like.

— *DBManager.java* étant le fichier qui centralise les requêtes pour la base de données, celui-ci doit donc comprendre en plus deux méthodes. Une méthode qui permet de retrouver un utilisateur enregistré à l'aide de son nom et une autre qui est responsable d'enregistrer un nouvel utilisateur.
— *ShopControler.java* ce fichier est le controleur qui reçoit les différentes requêtes venant de l'utilisateur. Une méthode doit donc être ajoutée afin d'accepter une requête post venant du formulaire html qui envoie le like d'un utilisateur.
— *product.java* ce fichier contient le modèle représentant ce qu'est un produit dans le programme. Il faut ajouter un lien vers le model user afin de pouvoir lier les likes des produits vers le compte qui a déposé le like.

5.3.3 Variation de la feature compte

La programmation orientée delta a également été utilisée afin d'implémenter un variant pour la gestion de compte utilisateur. Grâce à Spring Data repositories et CrudRepository qui permettent de faciliter l'interaction avec la base de données, seul le modèle correspondant à un utilisateur doit être défini ainsi que le CrudRepository correspondant. La programmation orientée delta permet d'ajouter les fichiers : *User.java* et *UserRepository.java*. Dans ce variant, un utilisateur est enregistré dans la base de données comme un nom ainsi qu'un id afin de pouvoir l'identifier.

5.3.4 Variation de la feature log des requêtes

La programmation orientée aspect qui est autorisée par le framework Spring a permis de pouvoir effectuer un log lors des accès à la base de données. Spring permet de définir facilement un aspect simplement en ajoutant une classe et l'annotation « *Aspect* ». Pour ajouter ce variant, il suffit donc d'ajouter le fichier « *dbManagerAspect.java* » contenant l'aspect qui log toute action se passant dans le fichier *dbManager.java*.

5.3.5 Variation de la feature Recherche

La page web principale de ce projet a été dessinée avec une barre de navigation. Cette dernière est utilisée afin de pouvoir mettre en place un lien vers d'éventuels plugins. Pour prouver cette utilisation, un plugin de recherche a été implémenté. Celui-ci utilise un champ dans cette barre de navigation pour effectuer une recherche sur tous les champs composant un produit et affiche les détails de ce dernier.

Durant la conception de ce projet, il a été choisi de laisser libre au plugin de décider le code html qui serait affiché dans la barre de navigation afin de permettre à ce plugin d'afficher un formulaire au lieu d'un simple bouton qui redirigerait vers une nouvelle page. Ce plugin nécessite donc en plus du fichier « *Recherche.java* » qui définit les méthodes propres au plugin, le fichier « *rechercheNavBar.html* »

qui possède le code html qui sera affiché dans la barre de navigation soit un formulaire avec un champs texte et un bouton submit.

Conclusion

Ce chapitre a permis d'expliquer une situation réaliste, représentant un site web d'e-commerce où il est possible d'afficher des articles et de pouvoir les liker. Ce site web a été développé afin de pouvoir valider le moteur de ligne de produits détaillé dans ce mémoire.

Chapitre 6

Conclusion

Ce document s'emploie en premier lieu à faire un état des lieux de l'avancement des recherches dans le domaine des lignes de produits logiciels. Une attention particulière a été accordée aux recherches concernant les différentes méthodes qui permettent d'introduire des points de variation dans ces lignes de produits afin de connaître leurs différences et leurs méthodes d'implémentation. À travers les multiples articles scientifiques consultés, nous avons mis en exergue un certain nombre de techniques de programmation avec lesquelles la variabilité et l'adaptabilité peuvent être introduites dans un programme. Ce document offre également une base de comparaison de ces différentes méthodes de variation afin de pouvoir déterminer laquelle est la plus adaptée à une situation donnée.

La variation est un point clé dans le développement des lignes de produits. Cependant, celle-ci peut se présenter sous de multiples formes, selon différentes tailles de granularité et peut nécessiter une prise de décision à différents moments de la vie du logiciel, c'est à dire, de la phase de développement jusqu'à l'étape finale où le programme est déployé chez le client. Les principes de programmation évoqués dans ce document répondent chacun à des besoins de variabilité différents. C'est pourquoi, ces techniques ne sont pas vraiment concurrentes mais plutôt complémentaires. Une ligne de produits nécessite tout cet éventail de méthodes afin de posséder des points de variation qui répondent exactement aux critères en jeu. Ceci complique le travail des outils qui ont pour objectif de faciliter le développement de lignes de produits. Ceux-ci doivent pouvoir gérer le plus grand nombre de variations possibles afin de proposer un service le plus exhaustif possible pour ne pas être rapidement limité.

Certains outils sont évoqués car ils permettent un niveau d'automatisation de la mise en place et de la gestion d'une ligne de produits. Cependant la plupart des outils qui visent à aider le développement d'une ligne de produits, se concentrent sur un seul aspect du développement car la vie totale d'une ligne de produits est trop compliquée à gérer par un seul outil. De plus, il semble qu'aucun outil

ne puisse prétendre supporter tous ces mécanismes déjà existants tout en étant simples d'utilisation. Les lignes de produits sont un domaine de recherche relativement récent, il est nécessaire de poursuivre des recherches afin de pouvoir proposer un outil permettant de faciliter entièrement le développement ainsi que la gestion de celles-ci.

Il est apparu évident qu'une technique de variation ne peut supplanter toutes les autres méthodes. Lors de l'implémentation d'une ligne de produits, les différents variants ont chacun leurs caractéristiques et besoins de variabilité. Cela implique qu'une unique méthode ne peut être la seule solution pour lier tous les variants. Chaque méthode de variation évoquée dans ce document apporte un niveau de variation statique ou dynamique et permet ou non à des développeurs tiers de créer un nouveau variant. Malgré le fait que chaque méthode présente des inconvénients, chacune de celles-ci est envisageable lors de la conception d'une ligne de produits.

Sur base de ces recherches, l'implémentation personnelle d'un moteur d'assemblage de ligne de produits est proposée. Cela fut l'occasion de constater directement certaines difficultés qui sont apparues et qui peuvent expliquer certains blocages à l'intégration des lignes de produits dans le milieu professionnel. Il existe un nombre trop important de méthodes permettant d'introduire de la variabilité dans un programme. C'est pourquoi l'ouverture offerte par l'utilisation de plugins parait être le meilleur choix. Quatre méthodes de variabilité ayant pour but de valider la conception du moteur d'assemblage ont été étudiées. Cependant l'ouverture de cet aspect permet à n'importe quel utilisateur de créer son propre mécanisme de variabilité. Un langage basé sur le XML a été également développé afin de pouvoir communiquer au programme la structure d'une ligne de produits donnée ainsi qu'une configuration choisie.

Perspectives

L'implémentation de ce moteur d'assemblage de lignes de produits a pu mettre en lumière les difficultés engendrées par le niveau de variabilité que les lignes de produits impliquent. La solution proposée répond aux besoins exprimés tels que pouvoir aider à la construction d'une ligne de produits ainsi que permettre d'assembler une configuration pour une ligne de produits donnée. Toutefois, tels qu'exprimés dans le chapitre sur les possibles travaux futurs concernant le moteur d'assemblage, certains besoins fonctionnels accessoires peuvent être ajoutés en supplément. Afin de garder ce moteur de ligne de produits viable et attrayant, certains besoins non-fonctionnels doivent être conservés au premier plan.

Le besoin non-fonctionnel principal est évidemment que la création de plugins doit être une opération simple afin de ne pas effrayer d'éventuels développeurs tiers à créer de nouvelles techniques de variation. Cela semble être un des be-

soins clés pour la pertinence à long terme d'un tel outil. Il est également très important de garder une syntaxe XML la plus intuitive possible afin de faciliter au maximum le travail de description du featureModel ainsi que l'écriture de la configuration voulue.

Travail futur

Le programme développé et présenté dans ce mémoire permet ainsi de pouvoir concevoir une ligne de produits avec des variants utilisant la méthode de variation souhaitée, mais également de pouvoir construire une configuration et fournir le programme associé. Cela répond aux besoins énoncés pour ce travail dont la création d'une plateforme ayant les capacités de générer une configuration pour une ligne de produits donnée. Cependant, certaines améliorations restent sans conteste possibles.

6.0.1 GUI

Le moteur de ligne de produits fonctionne pour l'instant dans une fenêtre console. Il est tout à fait possible de l'utiliser en l'état. Pourtant, une utilisation par une personne lambda, ayant un bagage informatique limité, nécessite une interface graphique. De plus, certaines fonctionnalités peuvent alors être imaginées afin de rendre la sélection d'une configuration plus commode. Lister visuellement les variants disponibles d'après un featureModel et pouvoir griser dans cette liste les variants interdits suite à une règle de dépendance peut, par exemple, grandement faciliter la configuration.

6.0.2 Plugins

Actuellement, le nombre de techniques de variabilité acceptées par le moteur de ligne de produits est de quatre. Il est évident qu'il existe davantage de techniques de variabilité, cependant celles qui sont en place permettent de valider l'utilisation du moteur. Grace à l'utilisation de plugins, il est tout à fait possible d'ajouter d'autres méthodes de variabilité.

Bibliographie

[1] Java preprocessor. `https://github.com/manifold-systems/manifold/tree/master/manifold-deps-parent/manifold-preprocessor`. Accessed : 11-10-2022.

[2] Anthony A Aaby. Compiler construction using flex and bison. *Walla Walla College*, 2003.

[3] Fellipe A. Aleixo, Marília Freire, Daniel Alencar, Edmilson Campos, and Uir' Kulesza. A comparative study of compositional and annotative modelling approaches for software process lines. In *2012 26th Brazilian Symposium on Software Engineering*, pages 51–60, 2012.

[4] M. Anastasopoulos and Cristina Gacek. Implementing product line variabilities. In *Proceedings of the 2001 symposium on Software reusability : putting software reuse in context*, volume 26, pages 109–117, 05 2001.

[5] Frismadani Anggita. Object oriented programming. *IEEE Software*, July 2003.

[6] Boto Bako, Andreas Borchert, Norbert Heidenbluth, and Johannes Mayer. Linearly ordered plugins through self-organization. In *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, pages 8–8. IEEE, 2006.

[7] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 3–35, 2005.

[8] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 3–35, 2005.

[9] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *International Workshop on Software Product-Family Engineering*, pages 13–21. Springer, 2001.

[10] Nélio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thaís Batista, and Carlos Lucena. Composing design patterns : A scalability study of aspect-oriented programming. In *Proceedings of the 5th international conference on Aspect-oriented software development*, volume 2006, pages 109–121, 01 2006.

[11] Meriem Chibani, Brahim Belattar, and Abdelhabib Bourouis. Aspect and subject oriented programming paradigms : A comparative study. 12 2014.

[12] Justin Cooper, Alfonso De la Vega, Richard Paige, Dimitris Kolovos, Michael Bennett, Caroline Brown, Beatriz Sanchez Pina, and Horacio Hoyos Rodriguez. Model-based development of engine control systems : Experiences and lessons learnt. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 308–319. IEEE, 2021.

[13] James R Cordy and Medha Shukla. *Practical metaprogramming*. Queen's University of Kingston. Department of Computing and Information Science, 1992.

[14] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. Generative programming. In *European Conference on Object-Oriented Programming*, pages 15–29. Springer, 2002.

[15] Alberto Rodrigues Da Silva. Model-driven engineering : A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43 :139–155, 2015.

[16] Robertas Damaševičius and Vytautas Štuikys. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2), 2008.

[17] Philip De Smedt. Comparing three graphical dsl editors : Atom3, metaedit+ and poseidon for dsls. *Preprint, Submitted to Elsevier, University of Antwerp*, 2011.

[18] E. Dolstra, Gert Florijn, and Eelco Visser. Timeline variability : The variability of binding time of variation points. *Proceedings of the Workshop on Software Variability Management*, 01 2003.

[19] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Metrics for analyzing variability and its implementation in software product lines : A systematic literature review. *Information and Software Technology*, 106 :1–30, 2019.

[20] Vincent Englebert. Ingénierie d'usines à logiciels, Mars 2022.

[21] Paul Tepper Fisher and Brian D Murphy. *Spring persistence with Hibernate*. Springer, 2010.

[22] Michael Goedicke, Carsten Köllmann, and Uwe Zdun. Designing runtime variation points in product line architectures : Three cases. *Sci. Comput. Program.*, 53 :353–380, 12 2004.

[23] François-Xavier Guillemette. Vers l'utilisation de dsl et de langages dynamiques en entreprise : une étude de cas avec groovy/mémoire présenté comme exigence partielle de la maîtrise en informatique par françois-xavier guillemette ;[directeur de recherche, louis martin]. 2009.

[24] Sebastian Günther and Sagar Sunkle. Dynamically adaptable software product lines using ruby metaprogramming. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, pages 80–87, 2010.

[25] Patrik Harnoš and L'ubomír Dedera. Analysis of current trends in the development of dsls and the possibility of using them in the field of information security. *Science & Military Journal*, 16(2) :15–27, 2021.

[26] Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Betsbrugge. Adding variants on-the-fly : Modeling meta-variability in dynamic software product lines. *Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL @ SPLC 2009)*, 01 2009.

[27] Ruben Heradio, David Fernandez-amoros, Jose A. Cerrada, and Ismael Abad. A literature review on feature diagram product counting and its usage in software product line economic models. *International Journal of Software Engineering and Knowledge Engineering*, 23(08) :1177–1204, 2013.

[28] Chris Holmes and Andy Evans. A review of frame technology. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, 2003.

[29] Chris Holmes and Andy Evans. A review of frame technology. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, 01 2004.

[30] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. Empirical analysis of the tool support for software product lines. *Software and Systems Modeling*, pages 1–38, 2022.

[31] Lothar Hotz, Thorsten Krebs, and Katharina Wolter. Knowledge-based product derivation - research topics of the conipf project. *Künstliche Intell.*, 18(4) :58, 2004.

[32] Aye Thiri Htun and Swe Zin Hlaing. *Implementation of Syntax Analyzer by using JavaCC*. PhD thesis, MERAL Portal, 2009.

[33] Michel Jaring, Rene Krikhaar, and Jan Bosch. Modeling variability and testability interaction in software product line engineering. In *Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 120–129, 03 2008.

[34] Stan Jarzabek and Hongyu Zhang. Xml-based method and tool for handling variant requirements in domain models. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 166–173. IEEE, 2001.

[35] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, et al. The spring framework–reference documentation. *interface*, 21 :27, 2004.

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

[37] Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008.

[38] Steven Kelly. Comparison of eclipse emf/gef and metaedit+ for dsm. In *19th annual ACM conference on object-oriented programming, systems, languages, and applications, workshop on best practices for model driven software development*, 2004.

[39] Steven Kelly and Juha-Pekka Tolvanen. Collaborative modelling and metamodelling with metaedit+. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 27–34. IEEE, 2021.

[40] Charles Krueger and Paul Clements. Feature-based systems and software product line engineering with gears from biglever. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2*, pages 1–4, 2018.

[41] Raminvas Laddad. *Aspectj in action : enterprise AOP with spring applications*. Simon and Schuster, 2009.

[42] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. Dsl classification. In *OOPSLA 7th workshop on domain specific modeling*, 2007.

[43] Dongwon Lee and Wesley W Chu. Comparative analysis of six xml schema languages. *ACM Sigmod Record*, 29(3) :76–87, 2000.

[44] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. *ACM International Conference Proceeding Series*, 1, 09 2012.

[45] Jian Bing Li and James Miller. Testing the semantics of w3c xml schema. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 443–448. IEEE, 2005.

[46] Samer Al Masri, Nazim Uddin Bhuiyan, Sarah Nadi, and Matthew Gaudet. Software variability through c++ static polymorphism : A case study of challenges and open problems in eclipse omr. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, page 285–291, USA, 2017. IBM Corp.

[47] Johannes Mayer, Ingo Melzer, and Franz Schweiggert. Lightweight plug-in-based application development. In *Net. ObjectDays : International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 87–102. Springer, 2002.

[48] Mehdi MEHDARY, BenLahmar EL HABIB, and Abderrahim TRAGHA. Concevoir un dsl dédié au cloudcomputing.

[49] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot : software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, 2009.

[50] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240, 2009.

[51] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Dtd-miner : a tool for mining dtd from xml documents. In *Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000*, pages 144–151. IEEE, 2000.

[52] Qaiser Munir and Muhammad Shahid. Software product line : Survey of tools, 2010.

[53] Sarah Nadi and Ric Holt. Mining kbuild to detect variability anomalies in linux. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 107–116, 2012.

[54] Sarah Nadi and Ric Holt. The linux kernel : A case study of build system variability. *Journal of Software : Evolution and Process*, 26, 08 2014.

[55] Eman Negm, Soha Makady, and Akram Salah. Survey on domain specific languages implementation aspects. *International Journal of Advanced Computer Science and Applications*, 10(11), 2019.

[56] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 907–918, New York, NY, USA, 2014. Association for Computing Machinery.

[57] Tu Nguyen. Java spring framework in developing the knowledge article management application : A brief guide to use spring framework. 2018.

[58] Tu Nguyen. Java spring framework in developing the knowledge article management application : A brief guide to use spring framework. 2018.

[59] Miika Oksanen et al. A comparison of two sple tools : Pure : : Variants and clafer tools. 2018.

[60] Harold Ossher and Peri Tarr. Hyper/j : Multi-dimensional separation of concerns for java. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, page 734–737, New York, NY, USA, 2000. Association for Computing Machinery.

[61] Terence J. Parr and Russell W. Quong. Antlr : A predicated-ll (k) parser generator. *Software : Practice and Experience*, 25(7) :789–810, 1995.

[62] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise testing for software product lines : Comparison of two approaches. *Software Quality Journal*, 20, 09 2011.

[63] Yonni Chen Kuang Piao. *Nouvelle architecture pour les environnements de développement intégré et traçage de logiciel*. Ecole Polytechnique, Montreal (Canada), 2018.

[64] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering : foundations, principles, and techniques*, volume 1. Springer, 2005.

[65] Jaroslav Porubän, Michal Forgáč, and Miroslav Sabo. Annotation based parser generator. In *2009 International Multiconference on Computer Science and Information Technology*, pages 707–714, 2009.

[66] Elder Rodrigues, Avelino Zorzo, Edson OliveiraJr, Itana Gimenes, José Maldonado, and Anderson Domingues. Plugspl : an automated environment for supporting plugin-based software product lines. In *SEKE*, page 647, 07 2012.

[67] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Autom. Softw. Eng.*, 18 :163–197, 06 2011.

[68] Rajaa Saidi, Agnes Front, Dominique Rieu, Mounia Fredj, and Salma Mouline. Variability dimensions for business component reuse. In *The third edition of the International Conference on Web and Information Technologies (ICWIT'10)*, 11 2010.

[69] Dušan Savic, Alberto Rodrigues da Silva, Siniša Vlajic, Saša Lazarevic, Ilija Antovic, Vojislav Stanojevic, and Miloš Milic. Preliminary experience using jetbrains mps to implement a requirements specification language. In *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 134–137. IEEE, 2014.

[70] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines : Going Beyond*, pages 77–91, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[71] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. A model-based framework for automated product derivation. In *MAPLE@SPLC*, 2009.

[72] David A Schmidt. Programming language semantics. *ACM Computing Surveys (CSUR)*, 28(1) :265–267, 1996.

[73] Elizabeth Scott and Adrian Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7) :177–189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[74] Elizabeth Scott and Adrian Johnstone. Gll syntax analysers for ebnf grammars. *Science of Computer Programming*, 166 :120–145, 2018.

[75] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49(7) :717–739, 2007.

[76] Mark Skipper. The watson subject compiler & aspectj (a critique of practical objects. *Ossher et al.[891]*, 1999.

[77] Richard M. Stallman. Bison : The yacc-compatible parser generator. 2015.

[78] Vytautas Štuikys and Robertas Damaševičius. *Meta-programming and model-driven meta-program development : principles, processes and techniques*, volume 5. Springer Science & Business Media, 2012.

[79] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software : Practice and experience*, 35(8) :705–754, 2005.

[80] Soe Myat Swe, Hongyu Zhang, and Stan Jarzabek. Xvcl : a tutorial. In *Proceedings of the 14th international conference on software engineering and knowledge engineering*, pages 341–349, 2002.

[81] Xhevahire Tërnava and Philippe Collet. On the Diversity of Capturing Variability at the Implementation Level. In *the 21st International Systems and Software Product Line Conference - Volume B*, Sevilla, France, September 2017. ACM Press.

[82] Xhevahire Tërnava and 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*, pages 81–88, 2017.

[83] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide : An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79 :70–85, 2014.

[84] Jean-christophe Trigaux, Patrick Heymans, Pierre-yves Schobbens, and Andreas Classen. Comparative semantics of feature diagrams : Ffd vs. vdfd. In *Fourth International Workshop on Comparative Evaluation in Requirements Engineering (CERE'06 - RE'06 Workshop)*, pages 36–47, 2006.

[85] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 249–258, 2008.

[86] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE, 2001.

[87] Balaji Varanasi. *Introducing Maven : A Build Tool for Today's Java Developers*. Apress, 2019.

[88] R Allen Wyke and Andrew Watt. *XML Schema Essentials*, volume 5. John Wiley & Sons, 2002.

[89] Bo Zhang, Slawomir Duszynski, and Martin Becker. Variability mechanisms and lessons learned in practice. In *2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE)*, pages 14–20, 2016.

[90] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 129–139, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Chapitre 7

Annexes

7.1 Annexe 1

Lien vers le répository GitHub contenant moteur d'assemblage de ligne de produits :
https ://github.com/sebiba/memoire.git
Lien vers le répository GitHub contenant le proof of concept :
https ://github.com/sebiba/memoirePOC.git

7.2 Annexe 2

Navbar
Bibliothèque
Search
Search
code
Nom
ean
desc
GV9061C0
CALOR Centrales vapeur GV9061C0
1234567890123
Calor GV9061C0. Puissance du fer a repasser: 2400 W
BM6010
TEFAL Pese-personnes BM6010
2345678901234
Tefal Bodymaster BM6010. Type: Pese-personne electronique. Poids maximum (capacite): 160 kg. Exactitude/Precision: 100 g. ecran: LCD
MX-4154
TRISTAR SET MIXER PLONGEANT MX4154
3456789012345
Tristar MX-4154 Set Plongeur. Capacite du bol: 0.5 L. Type de commande: Boutons.
FV9770
CALOR FER ULTIMATE FV977
3948740194384
Calor Ultimate Anti-Calc. Type: Fer a repasser a sec ou a vapeur, Puissance: 3000 W
EAN: 3456789012345
Nom: TRISTAR SET MIXER PLONGEANT MX4154
Code article: MX-4154
Description: Tristar MX-4154 Set Plongeur. Capacite du bol: 0.5 L. Type de commande: Boutons.
Nombre de like: 1
Nom:
Submit