RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE.
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE.
UNIVERSITÉ MOULOUD MAMMERI DE TIZI OUZOU.
FACULTÉ DE GÉNIE ÉLECTRIQUE ET D'INFORMATIQUE.
DÉPARTEMENT D'INFORMATIQUE.

MÉMOIRE DE FIN D'ÉTUDES
EN VUE D'OBTENTION DU DIPLÔME
DE MASTER ACADÉMIQUE EN INFORMATIQUE

OPTION : CONDUITE DE PROJET INFORMATIQUE

THÈME

APPLICATION DES PRODUCT LINES AUX SYSTÈMES MULTI-AGENTS

RÉALISÉ PAR :

M. DIAF AGHILAS
M. CHERMAK YENNI

PROPOSÉ PAR :

MME. ZAHIA GUESSOUM

DIRIGÉ PAR :

M. RACHID AHMED OUAMER

Avant-propos

Ce thème nouveau, encore au stage de la recherche, nous a été proposé par Madame GUESSOUM Zahia, Professeur à l'université Paris VI. Nous lui adressons nos plus vifs remerciements d'autant plus que ce thème cadre bien avec le profil de notre formation.

Nous exprimons aussi toute notre gratitude à Monsieur Ahmed OUAMER, Professeur au département d'Informatique, FGEI, UMMTO, qui, du côté professionnel, nous a encadrés avec patience, rigueur et compétence. Nous le remercions d'avoir accepté de nous encadrer et diriger dans le cadre de ce mémoire de fin d'études.

Nos vifs remerciements vont également aux membres du jury, chacun par son nom, pour avoir bien voulu juger ce travail en y apportant leurs critiques et leurs contributions en vue de son amélioration.

À ceux qui nous ont aidés de près ou de loin, nous adressons nos vifs remerciements, sans oublier l'ensemble des enseignants du département Informatique qui ont déployé tous leurs efforts durant tout le long de notre formation.

Sommaire

Chapitre 3 : Application des Product Lines aux SMA

Liste des figures

Introduction générale

Au cours des dernières années, l'ingénierie logiciel orientée agent ou Agent-Oriented Sotware Engineering (AOSE), a émergé comme nouveau paradigme du génie logiciel pour permettre le développement d'applications distribuées et complexes qui sont caractérisées par un système composé de beaucoup de sous-systèmes en interaction. La plupart des méthodologies courantes de l'ingénierie logiciel orienté agent sont consacrées à développer un seul système multi-agent. Cependant, de nouvelles approches ont fait appel à des techniques de réutilisation en logiciel. Certaines d'entre elles n'ont pas été convaincantes. Néanmoins, Celles qui ont été approuvées ont exploré l'incorporation des technique de l'ingénierie des lignes de produits logiciels ou Software Product Lines Engineering (SPL) au contexte du développement en masse de systèmes multi-agent. Le but de ces approches est d'intégrer des techniques de l'ingénierie SPL et d'AOSE en incorporant leurs avantages respectifs et permettre l'exploitation industrielle de la technologie agent. Notre travail se basera sur l'étude de ces approches et, plus précisément, l'application des Product Lines aux Systèmes Multi-Agent (MAS-PL).

Ainsi, ce mémoire est organisé comme suit:

Dans le premier chapitre intitulé *« Généralité sur les Systèmes Multi-Agent »,* nous traitons les fondements des systèmes multi-agents en présentant un état de l'art sur les concepts d'agent, d'environnement, d'interaction et de communication.

Le deuxième chapitre intitulé « Software Product Lines» ou lignes de produits donne une présentation de cette nouvelle approche de réutilisation en logiciel. Ainsi, nous présentons successivement la réutilisation en logiciel dans l'industrie, les avantages des lignes de produits logiciels, la variabilité dans les SPL, l'analyse des besoins et l'ingénierie de domaine et d'application.

Dans le troisième et dernier chapitre intitulé « Application des Product Lines aux Système multi agents» est consacré, comme son titre l'indique, à l'application des lignes de produits aux systèmes multi-agents. 'Nous passons en revue, particulièrement, l'ingénierie logiciel orientée agent dans laquelle nous faisons apparaître les méthodologies et les plates-formes orientées agent, l'application des Product Lines aux systèmes multi-agents et les techniques d'implémentation de variabilité.

Ce mémoire est terminé par une conclusion générale et des perspectives ainsi que des références bibliographiques.

Premier chapitre

Généralités sur les systèmes multi-agents

1.1 Introduction

Après l'informatique centralisée considéré aujourd'hui comme ayant atteint ses limites à cause de la complexité de nouvelles applications et leur taille devenue croissante, la décentralisation a été développée en assurant la coopération entre modules et logiciels. Ensuite, la programmation orientée objet a permis déjà de décentraliser les fonctionnalités dans des composants autonomes et avec l'Informatique Distribuée (ID), on donne plus d'autonomie et d'initiative aux différents modules logiciels permettant ainsi à des applications hétérogènes de tourner sur des plates-formes hétérogènes et de communiquer facilement entre elles. C'est pour répondre à ces enjeux importants que le concept de Système Multi-Agent (SMA) a été proposé. Ainsi, un SMA est un système composé d'un ensemble d'agents situés dans un certain environnement et interagissant entre eux. Il constitue l'un des aspects de l'Intelligence Artificielle Distribuée (IAD) proposé vers la fin des années 1970 [1] pour modéliser des comportements de groupes dans différents domaines d'applications en utilisant un ensemble d'agents intelligents fonctionnant en commun et dotés de capacités de perception et

d'actions sur l'environnement. Les SMA évoluent généralement dans des environnements complexes ou larges, ouverts, dynamiques et imprévisibles dans lesquels il est très difficile et parfois impossible de définir correctement et complètement les systèmes lors de la phase de conception, avant leur utilisation sinon, on devrait connaître à l'avance toutes les conditions environnementales qui surviendraient dans le futur. Pour éviter ces difficultés, nous pouvons donner, à chaque agent, l'habileté d'améliorer ses propres performances et celles du groupe auquel il appartient. Ainsi, dans ce chapitre, il s'agit pour nous de présenter les fondements des SMA.

1.2 Notion d'agent

La compréhension du domaine des SMA commence par la compréhension de l'agent en tant que composant principal du système. Pour un agent, plusieurs définitions ont été proposées. Elles se ressemblent toutes, mais diffèrent selon le type d'application pour laquelle l'agent est conçu. Parmi les premières définitions, c'est celle de Ferber [1] qui est surtout retenue. Celui-ci définit un agent comme étant une entité physique ou virtuelle qui:

- est capable d'agir dans un environnement,
- peut communiquer directement avec d'autres agents,
- est mue par un ensemble de tendances sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser,
- possède ses propres ressources,
- perçoit, de manière limitée, son environnement,
- possède des compétences et offre des services,
- peut éventuellement se reproduire,
- a un comportement qui tend à satisfaire ses objectifs en tenant compte des ressources et des compétences dont elle dispose et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

De leur côté, Jennings, Sycara et Wooldridge [2] ont défini l'agent comme étant « *un système informatique,* ***situé*** *dans un environnement et qui agit de façon* ***autonome*** *et* ***flexible*** *pour atteindre les objectifs pour lesquels il a été conçu».*

Les notions "*situé*", "*autonome*" et "*flexible*" prennent différentes définitions. Ainsi,

- l'agent est *situé* s'il est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement.
- Il est *autonome*, s'il est capable d'agir sans être dirigé par un autre agent ou un operateur humain et il contrôle ses propres actions ainsi que son état interne.
- L'agent est dit *flexible* s'il est capable de *répondre à temps* et est *proactif* et *social*. Capable de répondre à temps signifie que l'agent doit être capable de percevoir son environnement et d'élaborer une réponse dans les temps requis. Etant *proactif,* l'agent doit exhiber un comportement opportuniste, tout en étant capable de prendre l'initiative au "bon" moment. Un agent est dit s*ocial* s'il doit être capable d'interagir avec les autres agents (logiciels et humains) quand la situation l'exige afin de compléter ses tâches ou aider ces agents à accomplir les leurs.

Ainsi, globalement, les agents n'ont qu'une représentation partielle de leur environnement, c'est-adire qu'ils n'ont pas de vision globale de tout ce qui se passe. C'est d'ailleurs ce qui se passe dans les réalisations humaines d'envergure comme la fabrication d'un véhicule dans lesquelles personne ne connait tous les détails de la réalisation, chaque spécialiste n'ayant qu'une vue partielle correspondant à son domaine de compétence. L'agent est ainsi une sorte «d'organisme vivant » dont le comportement vise à la satisfaction de ses besoins et de ses objectifs à partir de tous les autres éléments (perceptions, représentations, actions, communications et ressources) dont il dispose.

Concevoir un agent orienté vers un but précis et qui fonctionnera dans un environnement statique est relativement une tâche simple mais, lorsque plusieurs agents agissent simultanément dans le même environnement, ils

doivent être en mesure de réagir aux changements externes causée par les actions des autres agents.

D'autres part, un agent purement réactif, ne peut-être pas en mesure de répondre à ses objectifs individuels que s'il prend l'initiative de poursuivre ses buts. Ainsi, le défi est de concevoir des agents avec un bon équilibre entre l'ensemble des caractéristiques qui leurs permettent d'atteindre le but qui leurs est assigné.

1.2.1 Différents types d'agents

En fonction de la taille d'un agent, de sa complexité, de ses connaissances et de son raisonnement, nous pouvons classer les agents en trois grandes catégories : ***cognitif, réactif*** et ***hybride***.

- Les agents *cognitifs* disposent d'une capacité de raisonnement et d'exécution de tâches complexes sur une base de connaissances et d'une aptitude relative à la gestion des interactions avec les autres agents et avec l'environnement (Fig.1.1). Dans ce cas, le SMA sera composé d'un petit nombre d'agents "intelligents", chacun possédant une base de connaissance comprenant l'ensemble des informations et des savoir-faire nécessaires à la réalisation de sa tâche et à la gestion des interactions avec son environnement et les autres agents. Ils sont parfois dits « intentionnels », c'est-à-dire qu'ils possèdent des buts et des plans explicites leur permettant de les atteindre. On parlera alors de conduites téléonomiques, par opposition à conduites réflexives lesquelles caractérisent plus fréquemment les agents réactifs et dont les buts sont définis en réaction aux stimuli du milieu. Dans ce cadre cognitif, planifié au maximum, le SMA se conduira à la façon d'un petit groupe d'individus, régi par des règles sociales prédéfinies comme dans cas de situations conflictuelles où les agents seront amenés à négocier.

- Les agents *réactifs* sont des agents passifs qui réagissent seulement à un stimulus. Ce type d'agent ne dispose pas de module de raisonnement interne. Il ne possède pas, non plus, une représentation complète de son environnement. Il n'est pas capable de tenir compte de ses actions passées (Fig.1.2).

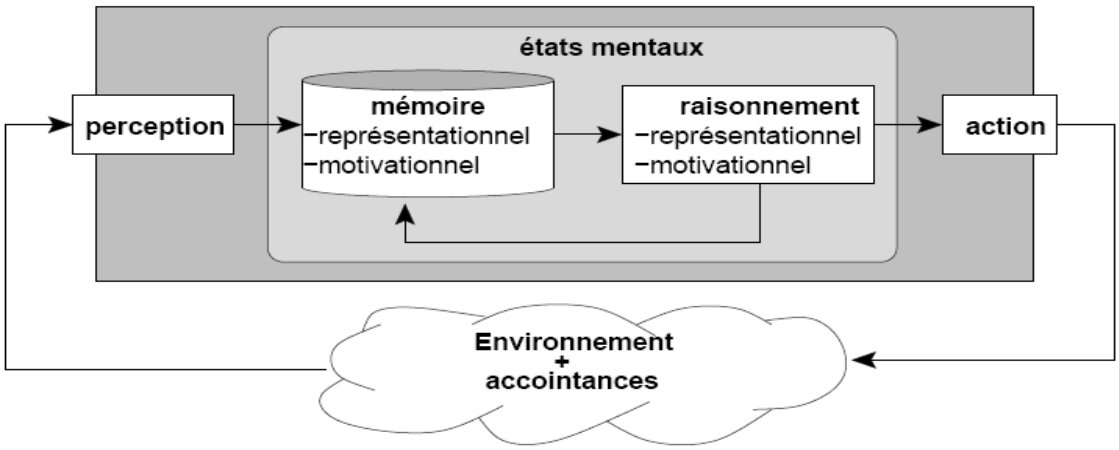


Fig.1.1 Agents Cognitifs

Ainsi, l'école réactive prétend qu'un système globalement intelligent ne nécessite nullement l'intelligence individuelle de ses agents. De simples mécanismes de réaction aux évènements, ne prenant en compte ni l'explicitation des buts *(conduites le plus souvent "réflexives", par opposition aux conduites générales des agents cognitifs)* ni le mécanisme de planification, peuvent en effet résoudre des problèmes qualifiés de complexes. C'est dans ce type de SMA que l'on assiste à l'émergence de processus d'auto-organisation parfois spectaculaires, l'un des exemples les plus frappants étant celui es colonies de fourmis artificielles.

Notons que les SMA constitués uniquement d'agents réactifs possèdent un grand nombre d'agents.

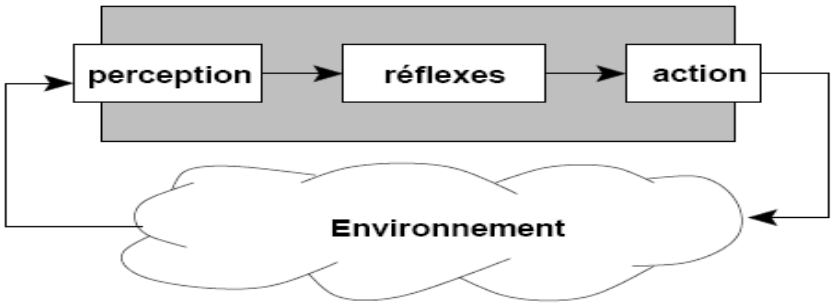


Fig.1.2 Agent réactif

Quant à l'*agent hybride*, il possède des composants réactifs et composants cognitifs pour garantir un raisonnement de qualité. C'est un agent où le concepteur tente de concilier, dans une même architecture logicielle, des activités réflexes, où l'agent réagit directement à des stimuli extérieurs en ignorant, notamment, l'historique de la perception (avec une activité de délibération (ou cognitive), où l'agent planifie à plus long terme son comportement). Traditionnellement, en intelligence artificielle distribuée, les approches réactives et cognitives s'opposent, mais peuvent être vues comme des cas limitent d'agents adaptés aux niveaux des ressources. Ceci à cause des choix particuliers de conception ont été effectués, ces systèmes sont alors optimisés pour des classes particulières de domaines d'applications et ne peuvent pas s'adapter à de nouveaux besoins.
Notons que ces différences de conception concernent la nature algorithmique des agents utilisés. Ainsi, on trouvera d'un côté l'école cognitive *(pour l'intelligence artificielle distribuée)*, de l'autre, celle qui intéressera plus directement notre travail: l'école réactive *(pour la vie artificielle)*.

1.2.2. L'environnement

L'environnement d'un SMA est le contexte dans lequel les agents vont évoluer. Il fournit un support commun aux actions des agents, permettant ainsi l'interaction dans le système, et il constitue une source d'information à laquelle les agents peuvent accéder au travers de leur perception. Dans le cas d'agents réels tells les robots, l'environnement est le milieu dans lequel les agents sont plongés, ce milieu étant gouverné par les lois physiques naturelles. Dans le cas d'agents logiciels, un environnement artificiel possédant ses propres lois d'évolution doit être défini. Il est généralement considéré qu'un environnement est composé d'entités actives *(les agents)*, d'entités passives *(les objets)* et de ressources qui peuvent être produites et consommées. Sans environnement il n'y a pas d'action ou de perception possible, on ne peut donc pas parler d'agent sans parler d'environnement. Ainsi, dans le cas où leurs objectifs individuels deviennent plus complexes, les agents doivent raisonner selon leur environnement afin de répondre à ces objectifs. Cependant, ceci ne reflète pas comment les agents peuvent

atteindre leurs objectifs individuels. Certains objectifs sont relativement simples et nécessitent des agents purement réactifs tels que les caméras de surveillance qui commencent à enregistrer sur requête, les détecteurs de fumée qui déclenche une alarme dès les premiers signes de feu. Ainsi, l'environnement est le contexte dans lequel les agents évoluent. Un certain nombre de propriétés pour l'environnement ont été définies et classées. En effet, l'environnement peut être soit accessible ou inaccessible, déterministe ou non déterministe, statique ou dynamique et discret ou continu. Ainsi, un environnement est accessible lorsque, pour un agent, il est possible d'obtenir la totalité des informations qui concernent son état. Dans le cas contraire il est inaccessible. Si l'environnement est déterministe, une même action aura toujours le même effet et sans aucune incertitude quant à son résultat. Dans le cas contraire il est non déterministe. L'environnement est statique s'il ne change que sous l'impulsion des actions effectuées par les agents. Dans le cas contraire, l'environnement définit aussi ses propres processus d'évolution et son état peut être modifié sans qu'aucune action ne soit intervenue. Cet environnement est discret lorsqu'il existe un nombre fixé de perceptions et d'actions possibles sur lui. Dans le cas contraire il est continu.

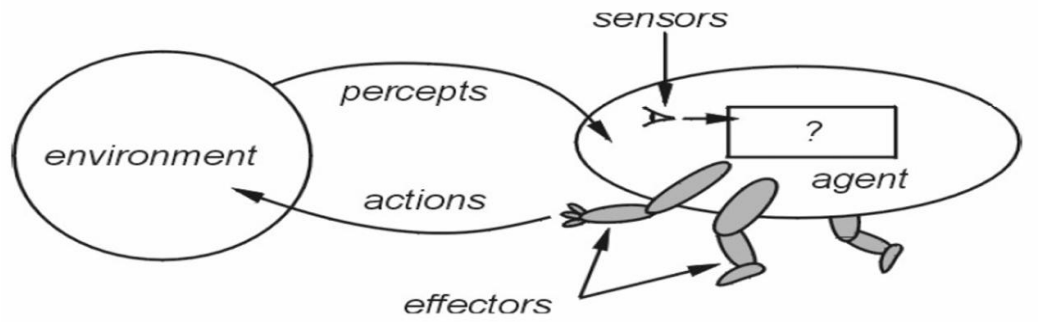


Fig.1.3 Interaction entre un agent et son environnement

En résumé, les classes générales les plus complexes des environnements sont celles qui ne sont pas accessibles, non déterministe, dynamique et continue. Les environnements qui présentent ces caractéristiques sont souvent désignés comme ouverts.

1.3. Les systèmes multi-agents

On appelle système multi-agent (ou SMA), un système composé d'éléments suivants:

1. Un environnement E, c'est-à-dire, un espace disposant généralement d'une métrique.
2. Un ensemble d'objets O. Ces objets sont situés, c'est-à-dire que, pour tout objet, il est possible, à un moment donné, d'associer une position dans E. Ces objets sont passifs, c'est-à-dire qu'ils peuvent être perçus, créés, détruits et modifiés par les agents.
3. Un ensemble A d'agents qui sont des objets particuliers (A ⊆ O), lesquels représentent les entités actives du système.
4. Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux.
5. Un ensemble d'opérations O*p* permettant aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O.
6. Des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde `a cette tentative de

D'après Jennings et al. [2], *"Un système multi-agent est un réseau couplé d'agents qui travaillent ensemble pour résoudre des problèmes qui dépassent les capacités ou la connaissance des agents individuels"(*Fig.1.4)

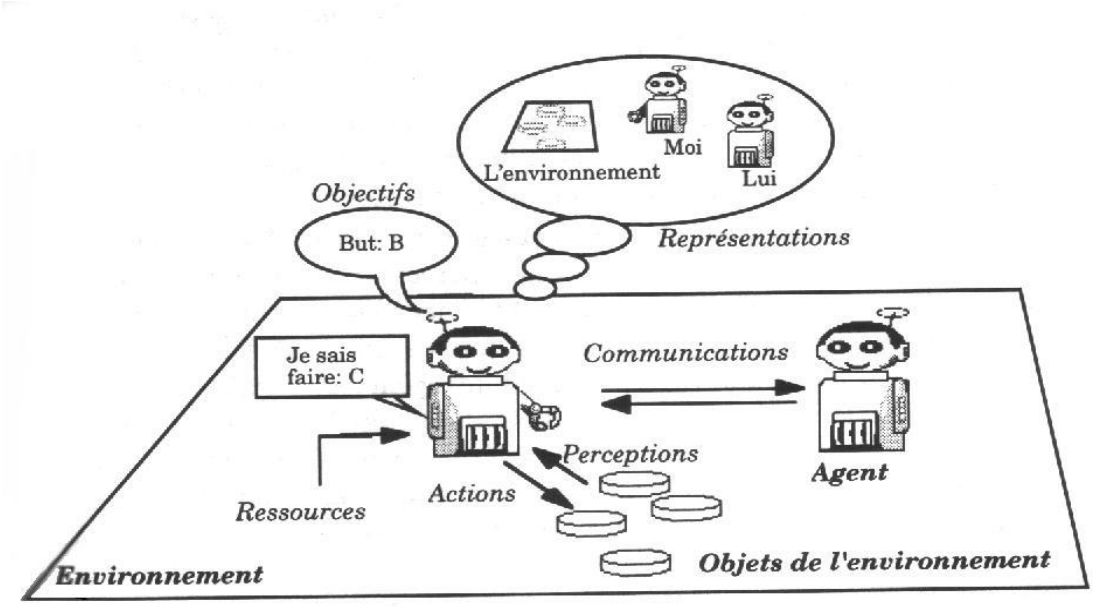


Fig.1.4 Représentation d'un système multi-agents d'après Ferber [1].

Le simple fait de placer un ensemble d'agents dans un même environnement n'est pas suffisant pour définir un SMA. Les différents agents doivent être en mesure d'interagir et de se comprendre mutuellement afin de pouvoir se coordonner et éventuellement coopérer. L'étude des mécanismes d'interaction est donc primordiale dans la conception d'un SMA.

1.3.1. Interactions dans les systèmes multi-agents

Une interaction est une mise en relation dynamique de deux ou plusieurs agents par le biais d'un ensemble d'actions réciproques. Les interactions s'expriment ainsi à partir d'une série d'actions dont les conséquences exercent en retour une influence sur le comportement futur des agents. Les agents interagissent le long d'une suite d'évènements pendant lesquels les agents sont d'une certaine manière en contact les uns avec les autres, que ce contact soit direct ou qu'il s'effectue par l'intermédiaire d'un autre agent ou de l'environnement.

Les cas d'interactions sont nombreux et divers: l'aide d'un robot par un autre, l'échange données entre serveurs informatiques, la mise en commun des compétences de plusieurs spécialistes pour porter un diagnostic, l'utilisation d'une imprimante par deux programmes simultanément, la répartition des charges sur plusieurs processeurs, la collision de deux véhicules sont autant d'exemples de situations d'interactions. Les interactions sont non seulement la conséquence d'actions effectuées par plusieurs agents en même temps, mais aussi l'élément nécessaire à la constitution d'organisations sociales. C'est par les échanges qu'ils entretiennent, par les engagements qui les lient, par l'influence qu'ils exercent les uns sur les autres que les agents sont des entités sociales et que de nouvelles fonctionnalités peuvent émerger de ces systèmes d'actions mutuelles. Les groupes sont donc à la fois les résultats d'interactions et les lieux privilégiés dans lesquels s'accomplissent les interactions.

C'est pourquoi il est généralement impossible d'analyser des organisations sociales sans tenir compte des interactions entre leurs membres.

D'après [1], la notion d'interaction suppose:

1. La présence d'agents capables d'agir et/ou de communiquer.
2. Des situations susceptibles de servir de point de rencontre entre agents: collaboration, déplacement de véhicules amenant à une collision, utilisation de ressources limitées, régulation de la cohésion d'un groupe.
3. Des éléments dynamiques permettant des relations locales et temporaires entre agents : communication, choc, champ attractif ou répulsif, etc.
4. Un certain "jeu" dans les relations entre les agents leur permettant à la fois d'être en relation, mais aussi de pouvoir se séparer de cette relation, c'est à-dire de disposer d'une certaine autonomie. Si des agents sont totalement liés par un couplage fixe, leur interaction devient rigide et ils n'interagissent plus au sens plein du terme.

L'interaction est à la base de la constitution d'organisations [2] L'éthologie, qui s'intéresse aux comportements animaux et humains , montre que l'apparition des formes cognitives est le produit d'une série de comportements complexes dans lesquels l'échange et l'interaction avec des individus de sa propre espèce jouent un rôle essentiel dans le développement d'un être vivant [3][4][5]. Lorsque cet échange n'existe pas, les individus ne peuvent s'accomplir totalement et restent dans un état "primitif".

La leçon la plus importante dans ce point et peut être même la plus importante dans ce chapitre est que lorsqu'on on parle de SMA, il est absolument important de comprendre le type d'interaction qui prend place entre les agents [6].

1.3.1.1. Les situations d'interactions

Selon Ferber [1], on appellera situation d'interaction un ensemble de comportements résultant du regroupement d'agents qui doivent agir pour satisfaire leurs objectifs en tenant compte des contraintes provenant des ressources plus ou moins limitées dont ils disposent et de leurs compétences individuelles. Ainsi, une situation d'interaction permet de

décrire des types d'interactions en les reliant aux éléments qui la composent.

Ferber identifie trois critères pour classifier les principales situations d'interaction :

1- les objectifs ou intentions des agents :
   Les buts ou objectifs des agents dans un environnement peuvent être compatible ou incompatible.
2- les relations qu'ils entretiennent envers les ressources qu'ils possèdent :
   Les buts sont souvent liés aux ressources : lorsque deux agents veulent accéder en même temps à une même ressource, un conflit peu prendre place et la coordination peut être nécessaire afin de résoudre le problème.
3- les moyens (ou compétences) dont ils disposent pour parvenir à leurs buts : Afin d'atteindre leurs but les agents ont besoin d'être doté de compétences et de moyens .Certains agents ont des compétences insuffisantes et ont besoin des compétences des autres agents afin d'atteindre leurs buts.

1.3.1.2. Types d'interactions

En se basant sur les trois précédents critères (la nature des buts, l'accès aux ressources et les compétences des agents), une typologie des situations d'interactions peut être faite selon la figure 1.5.

On peut distinguer plusieurs situations d'interactions : La situation d'*indépendance,* d'*encombrement,* de *collaboration coordonnée,* de *compétition individuelle pure*, de *conflits individuels pour des ressources*, de *compétition collective pure etc.*

La situation d'*indépendance* ne pose aucun problème d'interaction du point de vue multi agents et se résume à la simple juxtaposition des actions des agents pris indépendamment sans qu'il y ait effectivement d'interaction [1]. Il s'agit d'une situation qui ne réclame aucune interaction, de point de vue communication. Nous rappelons que nous nous intéressons dans ce travail aux interactions basées sur la communication entre agents.

Buts	Ressources	Compétences	Types de situation	Remarques
compatibles	suffisantes	suffisantes	*indépendance*	situation d'indifférence
compatibles	suffisantes	insuffisantes	*collaboration simple*	situations de
compatibles	insuffisantes	suffisantes	*emcombrement*	coopérations
compatibles	insuffisantes	insuffisantes	*collaboration coordonnée*	
incompatibles	suffisantes	suffisantes	*compétition individuelle pure*	
incompatibles	suffisantes	insuffisantes	*compétition collective pure*	situations
incompatibles	insuffisantes	suffisantes	*conflits individuels pour des ressources*	d'antagonismes
incompatibles	insuffisantes	insuffisantes	*conflits collectifs pour des ressources*	

Fig.1.5. Classification des situations d'interactions

Dans la situation d'*encombrement* les agents se gênent mutuellement dans l'accomplissement de leurs tâches, cela est dû à l'insuffisance des ressources. Puisque les buts sont compatibles, les agents essayent de coordonner leurs actions. Comme son nom l'indique, la situation de *collaboration coordonnée* représente une situation où les agents doivent collaborer pour combler leur manque de compétences. En plus de la collaboration, les agents doivent coordonner leurs actions à cause de l'insuffisance des ressources. C'est la plus complexe des situations de collaboration. La situation de *compétition individuelle pure*, ne pose pas de problème sur les ressources ou les compétences des agents. Cela ne crée pas de problèmes spécifiques d'interaction liés à ce type de situation.

Dans la situation de *conflits individuels pour des ressources*, chaque agent essaye d'acquérir les ressources pour pouvoir atteindre ces buts. Cette

situation pousse les agents à négocier l'accès aux ressources. On parle dans ce cas de protocoles de négociation. Dans la situation de *compétition collective pure*, il faut créer des coalitions et des groupes d'agents ayant des objectifs compatibles, et ramener ainsi le système à une situation de *compétition individuelle pure*. L'interaction entre les agents d'un même groupe est *collaborative simple*. Nous constatons que cette situation est une composition de la situation de *compétition individuelle pure* et celle de *collaborative simple*. Le même raisonnement est vrai pour la situation de *conflits collectifs des ressources*, qui peut être considérée comme une combinaison de la *collaboration coordonnée* et de la situation des *conflits individuels pour des ressources*.

En résumé, nous retenons trois grandes classes de situations d'interaction : les situations de *coordination*, les situations de *coopération* (ou de collaboration), et les situations de *résolution de conflit*.

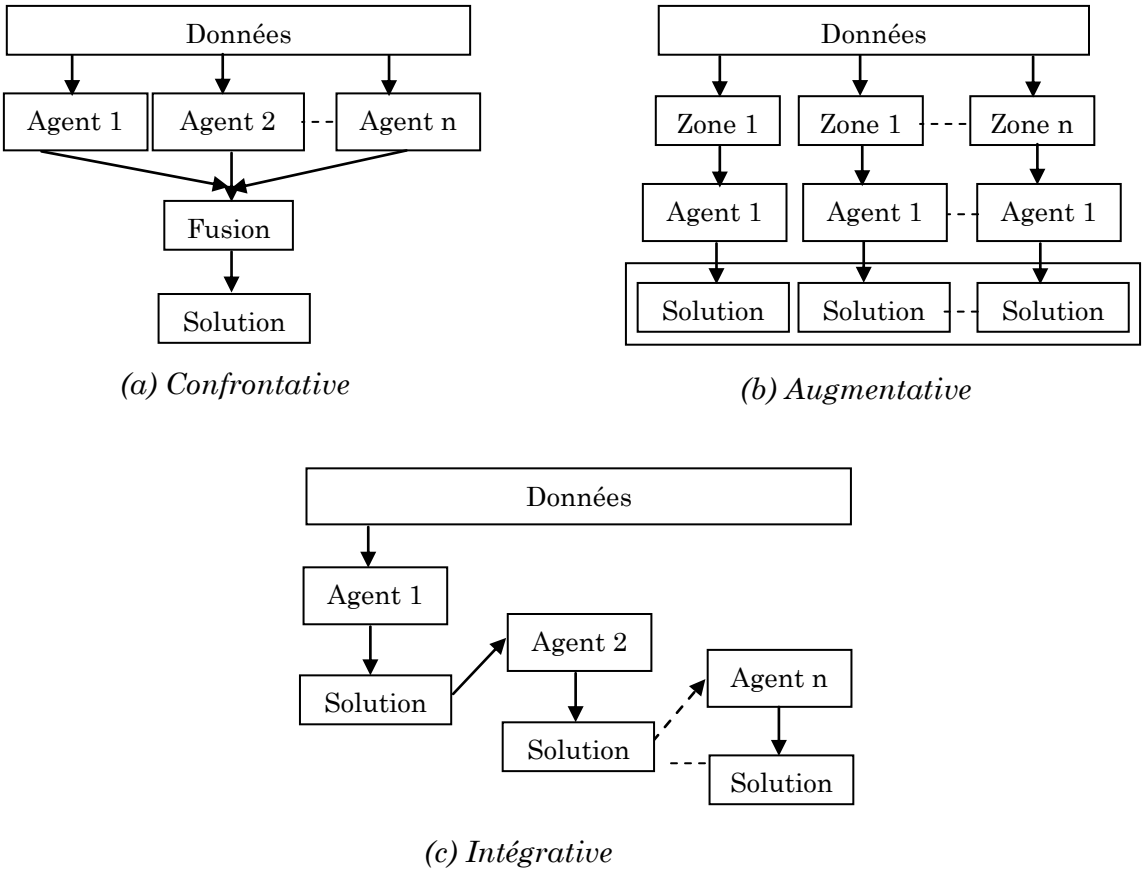
1.3.1.3. Coopération

L'interaction entre agents représente donc l'une des plus importantes facettes d'un SMA et elle englobe la coopération, la coordination, la négociation, la collaboration, etc. La coopération est une caractéristique très importante dans les SMA. En effet, une résolution distribuée d'un problème est le résultat de l'interaction coopérative entre les différents agents [3]. La coopération est une technique qui combine plusieurs sources d'information ou modalités de traitement pour accomplir une tâche. Il existe de multiples manières d'envisager la coopération dans les approches Multi-Agent. *J. Hoc* dans [7] et [8], propose de distinguer trois formes de coopération Multi Agent (Fig.1.6) : la coopération *confrontative*, la coopération *augmentative* et la coopération *intégrative.*

La coopération *confrontative* est celle selon laquelle, une tâche est exécutée par plusieurs agents de spécialités différentes, œuvrant de manière concurrente sur le même ensemble de données. Le résultat étant obtenu par fusion. La coopération *augmentative* est celle selon laquelle, une tâche est répartie sur une collection d'agents similaires, œuvrant de manière concurrente sur des sous ensembles disjoints de données (zones). La

solution étant obtenue sous la forme d'un ensemble de solutions locales. Quant à la coopération *intégrative,* une tâche est décomposée en sous tâches, accomplies par des agents de spécialités différentes et œuvrant de façon coordonnée. La solution étant obtenue au terme de leur exécution.

La *coordination* correspond au contrôle social ou collectif du système. Le but de la coordination est d'organiser l'activité des agents (coopération) en procédant généralement en deux étapes à savoir le choix d'un plan d'exécution suivi par la synchronisation des agents de ce plan qui est centralisé ou distribué. Dans la coordination *centralisée,* le coordinateur est responsable de la bonne exécution. Il agit suivant son plan, envoie des ordres d'actions aux agents esclaves et attend un accusé de réception ou bien le résultat d'exécution d'une action pour ordonner l'action suivante. On parle de coordination *distribuée*, plusieurs agents décident de coopérer. Chaque agent choisit un protocole d'interaction adapté à la coopération qu'il souhaite mettre en œuvre. Cette approche vise à augmenter l'autonomie décisionnelle des agents, ce qui améliore la flexibilité et l'ouverture du système. Ainsi, la *coordination* permet d'éviter les situations de conflits par la négociation pour les agents antagonistes, ayant des buts et des objectifs contradictoires. Elle permet aussi d'améliorer l'efficacité de chacun des agents par la coopération pour les agents non antagonistes. Pour les agents antagonistes, la négociation permet à plusieurs agents d'arriver à un accord mutuel pour entreprendre une action donnée d'une certaine manière. Ceci est effectué grâce à un processus de communication, d'échange d'informations et de concessions mutuelles comme la relaxation de buts initiaux ou l'usage de mensonges ou de menaces. La *négociation* peut être considérée comme une méthode de résolution de conflits et de recherche de consensus. Dans le cas d'agents non antagonistes, il s'agit d'une coopération entre ces agents pour se satisfaire mutuellement sans se gêner. Cette coopération, initiée par un échange d'information est souvent associée à la notion de collaboration qui est une forme d'interaction qui étudie la manière de répartir le travail et, par conséquent, l'allocation de tâches entre plusieurs agents.



(a) Confrontative

(b) Augmentative

(c) Intégrative

Fig.1.6. Les Différentes formes de coopération Multi-Agent.

1.3.1.4. Communication

En ce qui concerne la communication dans les SMA celle-ci est capitale afin que les agents puisse coopérer ,se coordonner et réaliser des taches qui dépasserait de loin celles qu'un agent à lui seul pourrait faire en effet il serait difficile de concevoir un système ou les agents sont censés coopérer, sans qu'il puisse s'échanger des informations entre eux ou encore envoyer des requêtes, sans communication l'agent stagnera dans un état primitif , autrement dit il sera isolé, refermé, sourd, muet etc.

Dans les systèmes cognitifs les communications se font par *envoi de messages* asynchrones suivant les protocoles de communication bien définis. Les connaissances et les mécanismes de traitement pour la résolution d'un problème sont distribués dans les agents. La communication

s'établit, alors point par point, ou par diffusion, alors que dans les systèmes réactifs, la communication résulte de la diffusion d'un signal ou en laissant des traces de leur présence dans l'environnement qui peut être perçus par d'autres agents. Ce mode de communication s'applique à des domaines comme l'éthologie ou la robotique. Les agents peuvent alors communiquer *par mémoire partagée* qui nécessite une structure de données commune à l'ensemble des agents, comprenant tous les éléments nécessaires à la résolution d'un problème. Les agents déposent et lisent les informations sur cette zone de données communes, comme dans le cas du "*tableau noir*" d'Erman et al. [6]. Ce dernier étant une zone de travail commune, dévolue au passage d'informations entre les différents agents.

1.3.2. Approches de modélisation d'un système multi-agent

Plusieurs travaux se sont concentrés, ces dernières années, sur l'approche de modélisation et de conception d'un SMA. Différentes approches ont été proposées. Parmi les approches proposées, on distingue **approches centrées tâche** et **Les approches centrées modèle.** Dans la première, l'organisation est hiérarchique prédéfinie. Les agents des niveaux supérieurs adressent des requêtes aux agents des niveaux inférieurs. Le rôle des agents est prédéfini et les relations sont fixées. La coopération est généralement à caractère intégratif, les agents des niveaux supérieurs intègrent les résultats des agents des niveaux inférieurs. Ces architectures sont constituées de peu d'agents cognitifs. Parmi ces approches on trouve le système MAVI qui est une variante du projet européen VAP [7] et MESSIE-II [8]. Dans l'**approche centrée modèle,** l'organisation est du type variable hiérarchique prédéfinie. Les agents du modèle disposent de la connaissance opératoire pour piloter les tâches et pour coopérer suivant le lien existant entre eux. Le rôle respectif des agents est prédéfini mais les relations entre eux varient en fonction des reconnaissances effectuées. Deux modes de coopération sont utilisés, la coopération intégrative, car le niveau supérieure intègre le résultat de ces sous partie et la coopération confrontative, car l'utilisation des cohérences spatiales mutuelles tendent à confirmer ou à infirmer la certitude des hypothèses. Les architectures centrées modèle disposent de

quelques dizaines d'agents cognitifs. Parmi ces approches en trouve le système développé par Hanson et Riseman [9] et le système proposé par Yannai et Deguchi [10]. Nous pouvons aussi citer l'approche AALAADIN [11] [12] et l'approche VOYELLES [13]. Dans l'approche *AALAADIN,* le modèle organisationnel est basé sur l'association de trois concepts clés : l'agent, le groupe et le rôle utilisés simultanément pour décrire des organisations concrètes d'agents (fig.1.6). Ainsi, l'agent est une entité autonome communicante qui joue des rôles au sein de différents groupes. Le groupe est un terme générique qui qualifie une communauté d'agents en relation par interaction, par partage d'un environnement, par un but ou une ontologie commune, etc. Le rôle est une représentation abstraite d'une fonction, d'un service ou de l'identification d'un agent au sein d'un groupe particulier.

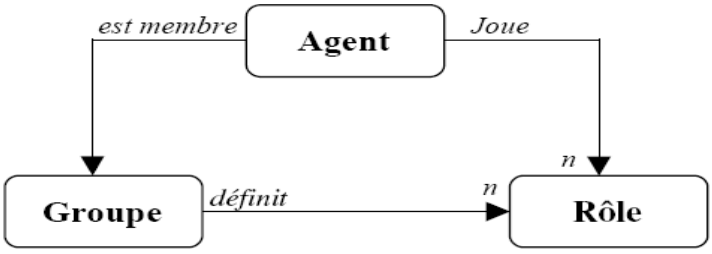


Fig.1.6. Le modèle organisationnel d'AALAADIN [8]

Notons que, dans le modèle AGR, un même agent peut jouer différents rôles et il participe alors à des interactions spécifiques à chacun de ses rôles. De même, un même rôle peut être joué par plusieurs agents.

L'approche VOYELLES, proposée par Yves Demazeau [13], consiste à considérer que l'analyse, le design, l'implémentation et le déploiement d'un système multi-agents peuvent être étudiés en fonction des quatre composantes fondamentales suivantes : Agents, Environnement, Interaction et Organisation (les voyelles A, E, I, O). Ainsi, la composante *Agent* représente l'ensemble des fonctionnalités de l'agent. Elle consiste à décrire l'architecture interne d'un agent à savoir, les mécanismes qu'il met en œuvre pour raisonner et produire son comportement, ses représentations de la connaissance, ses buts, etc. La composante *Environnement* est le milieu

dans lequel sont plongés les agents et qui représente l'ensemble des éléments grâce auxquels les entités (agents) vont pouvoir agir ensemble, être mises en relation. La composante *Interaction* concerne les infrastructures, les langages et les protocoles d'interactions entre agents, depuis de simples interactions physiques à des interactions langagières par actes de langage. La *composante Organisation* représente les moyens utilisés pour structurer l'ensemble des entités.

Cette approche permet un jeu d'écriture par associativité afin d'exprimer le type d'approche choisie pour le développement d'un système. C'est ainsi que, à l'informatique, il a été attribué une approche (((*A*+*E*) +*I*) +*O*) dans laquelle les agents et l'environnement sont les priorités de développement (approche centrée agent/environnement).

1.4. Application des systèmes multi-agents

Les domaines d'application des SMA sont nombreux et ils sont utilisés dans plusieurs catégories d'applications comme la résolution de problèmes au sens large, la robotique distribuée, la simulation multi-agent, etc.

Pour la *résolution de problèmes* au sens large, elle concerne en fait toutes les situations dans lesquelles des agents logiciels accomplissent des taches utiles aux êtres humains. La résolution concerne la résolution distribuée de problèmes et la résolution de problèmes distribués. En résolution distribuée de problèmes, l'expertise globale est distribuée parmi l'ensemble des agents. Les agents aux compétences complémentaires coopèrent entre eux à la poursuite d'un objectif commun, tel qu'un diagnostic médical, la conception d'un produit industriel [14], l'acquisition de connaissances, le diagnostic de réseaux [15], la reconnaissance de formes [16], le système de contrôle et de surveillance d'un réseau de télécommunications [17], etc. En résolution de problèmes distribués, le problème est lui-même distribué et les agents peuvent avoir des compétences semblables. Il n'y a pas seulement distribution de l'expertise mais aussi distribution du domaine. Il s'agit d'applications telles que l'analyse, l'identification, le diagnostic et la

commande de systèmes physiquement répartis pour lesquels il est difficile d'avoir une vision totalement centralisée.

Quant à la *simulation multi-agent*, elle est fondée sur l'idée qu'il est possible de représenter sous forme informatique le comportement des entités qui agissent dans le monde et qu'il est ainsi possible de représenter un phénomène comme le résultat des interactions d'un ensemble d'agents disposant de leur propre autonomie opératoire. L'intérêt de ces simulations est de pouvoir considérer aussi bien des paramètres quantitatifs, c'est-à-dire, des paramètres numériques que qualitatifs (des comportements individuels faisant éventuellement appel à des raisonnements stratégiques). Enfin, les SMA permettent la modélisation de situations complexes dont les structures globales émergentes des interactions entre individus, c'est-à-dire de faire surgir des structures du niveau macro à partir de modélisations du niveau micro. Un des simulateurs les plus connus est le simulateur SIMPOP [18] qui modélise la dynamique de l'évolution d'un système de villes et, plus particulièrement, la genèse, le développement et la concentration des fonctions urbaines à différents niveaux durant une longue période de temps (environ 2000 ans).

En ce qui concerne la *robotique distribuée,* elle porte sur la réalisation d'un ensemble de robots qui coopèrent pour accomplir une mission. La robotique distribuée utilise des agents concrets qui se déplacent dans un environnement réel.

1.5. Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur les concepts d'agent, d'environnement, d'interaction et de communications. Cela nous a permis de voir que chacun de ces concepts étaient nécessaires pour la conception d'un SMA. Cependant, cette étude n'est pas exhaustive. D'autres caractéristiques des SMA et du comportement des agents qui n'ont pas été citées. Comme l'apprentissage qui a pour but l'amélioration des performances du système en tenant compte des ressources et des

compétences dont il dispose. Dans notre étude, il s'agit de combiner l'approche SMA avec l'approche lignes de produits logiciels ou « Product Lines » qui fera l'objet du prochain chapitre.

Deuxième chapitre

Software Product Lines

2.1. Introduction

Introduite à la fin des années 1960 pour répondre à la croissance de la complexité des logiciels, la discipline «Génie Logiciel » dont le but est de proposer des solutions et des méthodologies pour le développement des logiciels a vu son champ évoluer de manière considérable pour couvrir les nouvelles problématiques introduites par l'extension de la pénétration de l'informatique dans l'industrie, créant ainsi le besoin d'améliorer la qualité et la productivité relative au développement et à l'entretien des systèmes d'information. Ainsi, la réutilisation s'est imposée progressivement comme une solution méthodologique et technologique pour le développement de systèmes logiciels complexes. De ce fait, plusieurs auteurs en Génie Logiciel ont vu, dans la réutilisation, l'occasion d'améliorer la qualité et la productivité relatives au développement et à l'entretien d'un système d'information. Aujourd'hui, un des principaux défis de la communauté de chercheurs et des industries est de trouver les concepts, mécanismes et langages adéquats pour une meilleure réutilisation et configuration en masse des systèmes logiciels. Parmi les derniers de ces défis, l'ingénierie des

lignes de produits logiciels ou Software Product Lines (SPL) qui est une approche qui va vers une vision de modélisation et de développement dans laquelle l'objectif envisagé n'est plus l'obtention d'un seul système logiciel, mais plutôt de concevoir et développer une ligne ou une famille de logiciels qui prend en compte des facteurs de variation et permet de minimiser les coûts et les temps de réalisation. Ces facteurs de variation peuvent être techniques, commerciaux et culturels. Pour les facteurs techniques et commerciaux, il s'agit respectivement de l'utilisation d'une variété de matériels associés aux logiciels, de la création de plusieurs versions allant d'une version limitée à une version complète. Pour les facteurs culturels, il s'agit, entre autre, de logiciels destinés à plusieurs pays [19].

Notons que cette notion de PL encore appelé plus communément Product Lines (PL) en anglais, n'est pas totalement nouvelle puisque David Parnas, dès 1976, a étudié dans [20] les familles de programmes ou Program Family et a défini la famille de programmes comme étant "*un ensemble de programmes dont les propriétés communes sont si nombreuses que c'est avantageux d'étudier d'abord ses propriétés communes avant d'analyser les membres individuels de l'ensemble*". Cependant, les SPL n'ont émergé comme une approche à part entière du génie logiciel que ces dernières années. Cette approche a été appliquée avec succès dans plusieurs secteurs. La majorité des succès concerne les familles de produits logiciels embarqués dans plusieurs types d'appareils électroniques tels que les téléphones mobiles, les scanners médicaux, les automobiles, la les télévisions, les satellites, les imprimantes, etc.

2.2. La réutilisation en logiciel

Les entreprises « logiciels » cherchent, depuis des années, à améliorer leur productivité et la qualité de leurs produits de façon à faire face à des développements de plus en plus contraints économiquement et complexes techniquement. C'est ainsi que l'idée de la réutilisation logicielle s'est imposée depuis longtemps en reprenant des éléments logiciels existant pour produire de nouveaux logiciels ce qui permet, théoriquement, de réduire

significativement le temps et l'effort nécessaires pour développer des applications logicielles. Pour sa part, Krueger donne, au sens large, la définition suivante de la réutilisation « *Software reuse is the process of creating software systems from existing software artifacts rather than building them from scratch. Typically, reuse involves the selection, specialization and integration of artifacts, although different reuse techniques may emphasize or de-emphasize certain of these* » [22].

Cette définition met en lumière la diversité de la réutilisation. Certes, la réutilisation concerne la reprise de code logiciel, structurée ou non, pour implémenter une fonction ou une application. Il peut s'agir, par exemple, d'une bibliothèque de logiciels qui peut être réutilisée à tout moment lors de nouveaux développements. Cependant, la réutilisation du logiciel ne se limite pas à la seule réutilisation de code. Elle concerne, en effet, tous les artefacts produits lors d'une première réalisation logicielle. Ces artefacts contiennent du code, mais aussi des connaissances liées, par exemple, à la conception, à l'architecture, aux conditions et contextes d'exécution, aux cas de tests, etc. La réutilisation peut ainsi être mise en œuvre tout au long du cycle de vie d'un développement logiciel. Cependant, elle ne constitue pas une idée nouvellement utilisée dans la production des systèmes logiciels. Elle l'a déjà été comme stratégie de production à des fins précises comme la diminution des coûts de développement et l'amélioration de la qualité [20] [23] sachant que les anciennes stratégies de réutilisation s'intéressaient plutôt à la réutilisation des pièces de codes relativement petites par exemple les librairies qui contiennent des algorithmes, modules, objets ou composants. Le problème à ce niveau consiste à localiser les pièces à réutiliser et à les intégrer avec un effort inférieur à celui de leur création à partir de zéro. Cependant, cet objectif reste toutefois difficile à réaliser, car l'effort de réutilisation demeure souvent plus important que la création à partir de zéro [27]. L'idée du concept des SPL a ensuite offert plus de facilité pour la réutilisation puisqu'il ne s'agit plus de construire des systèmes uniques au cas par cas, mais, plutôt, de viser, dès le début, une famille de

systèmes qui appartiennent au même domaine et qui représentent des similarités [24].

2.3. Software Product Line SPL

Product Lines ou PL consiste en un ensemble de produits représentant des similarités. La définition suivante a été proposée par Clements et Northrop [26] qui donne une idée plus précise sur les SPL. « *Une SPL est un ensemble de systèmes qui partagent et gèrent un ensemble de propriétés communes, satisfaisant des besoins spécifiques pour un domaine particulier et développés de manière contrôlée à partir d'un ensemble commun d'éléments réutilisables* ». Cette définition caractérise les produits, les membres de la SPL par un ensemble de propriétés communes (similarité) et aussi par leurs différences (variabilité). Rappelons que la **variabilité** est le concept clé dans la SPL. Elle permet le développement des produits configurés par la réutilisation des artéfacts prédéfinis et ajustables. Elle est modélisée au niveau de la phase d'ingénierie de domaine pour représenter les différences entre les produits planifiés. Sa définition est la suivante : « *La **variabilité** regroupe l'ensemble des hypothèses montrant comment les produits, membres de la ligne de produits, diffèrent* » [25]. Quant à la **similarité,** tout comme la variabilité, elle constitue un concept central dans les SPL. Cependant, sa gestion demande moins d'efforts que celui des variabilités. Elle est définie comme suit : « *La **similarité** regroupe l'ensemble des hypothèses qui sont vraies pour tous les produits, membres de la ligne de produits* » [25].

2.3.1. Product Lines dans l'industrie

Fondamentalement, l'approche des Product Lines propose d'adapter les principes du fordisme au développement logiciel. Notons que le fordisme désigne le mode d'organisation du travail mis en place par l'industriel américain H. Ford (1863-1947) dans ses usines d'automobiles de Détroit, notamment pour la production de la Ford T, noire, à partir de 1907 [42]. L'idée est de transposer les principes de fabrication des voitures au développement des logiciels embarqués dans celles-ci. Sauf exception, les

véhicules que nous conduisons à l'heure actuelle ne sont plus produits manuellement. Les processus de fabrication ont été drastiquement rationalisés et ces véhicules sont produits à la chaîne, essentiellement afin de maximiser la productivité et de minimiser les coûts. Différents modèles de voitures sortent de mêmes chaînes de montage en utilisant les mêmes châssis, les mêmes moteurs, les mêmes plans de tests, ce qui présente plusieurs avantages. Parmi ces avantages, on peut citer la diminution des efforts liés à l'entretien ou à la maintenance, l'augmentation de la sécurité des véhicules étant donné que chaque pièce est testée au préalable dans de nombreuses différentes situations en suivant des plans de test ayant déjà fait leurs preuves. Les Product Lines permettent également de capitaliser la connaissance. En effet, les ouvriers, par exemple, fabriquant des véhicules à Gent, à Tokyo ou au Brésil utiliseront les mêmes techniques, seront potentiellement confrontés aux mêmes problèmes et envisageront donc les mêmes solutions. Enfin, les risques et les coûts liés à l'élaboration de nouveaux modèles seront réduits. Les ingénieurs pourront se concentrer surtout sur les innovations à haute valeur ajoutée et sur une plus grande personnalisation des véhicules.

Ainsi, l'approche PL a été adoptée depuis sa naissance dans l'industrie. Le SEI (Software Engineering Institute) a publié plusieurs expériences industrielles prouvant sa réussite [28]. Dans [29] [30], il est montré comment Nokia a opté pour l'approche PL pour gérer la diversité des logiciels des téléphones mobiles. Maccari et al. estiment que pour maintenir le marché, Nokia introduit sur le marché entre 30 et 40 nouveaux produits par an, ce qui rend leur production à partir de zéro très difficile. En plus de cette donnée, Nokia doit aussi répondre à plusieurs facteurs de variation entre ses produits tels que la langue de l'interface utilisateur. En effet, les produits de Nokia supportent approximativement 60 langues [30], chaque langue a ses propres particularités comme l'écriture de gauche à droite pour les langues basées sur les caractères latins et de droite à gauche pour certaines langues orientales comme l'arabe ou le chinois qui a aussi ses propres caractéristiques. Les produits de Nokia doivent aussi être compatibles avec

les différents standards de communication numérique telle que GSM 900, GSM 1900, TDMA, etc. [30].

Aujourd'hui, la plupart des industries d'automobile, de bicyclettes, de matériels électroniques, etc. utilisent cette approche et leur problème de productivité est totalement résolu. Certaines d'entre elles, participent même à l'amélioration de cette approche en adhérant au Software Product Line Hall of Fame. Parmi ces industries, on peut citer Boeing, Bosch Group, CelsiusTech Systems AB, Cummins Inc, Ericsson AXE, FISCAN, General Motors Powertrain (GMPT), Hewlett Packard, HomeAway, LSI Logic, Lucent, Market Maker, Nokia, Philips (Medical Systems), Philips (Software for Television Sets), Philips (Telecommunication Switching System), Salion Inc, Siemens (syngo.via), Toshiba et U.S. Naval Research Laboratory [31].

2.3.2. Avantage des Software Product Lines.

En plus de la réutilisation, l'utilisation des SPL s'appuie aussi sur la configuration en masse qui est reconnue comme une des principales caractéristiques des SPL [27]. Elle a été définie par Stan Davis dans *« Future Perfect »* [34] comme *« La production à grande échelle est très efficace pour répondre aux besoins individuels des clients ».*

SPL s'appuie ainsi sur la configuration en masse des produits logiciels en se basant sur les éléments communs qui sont créés une seule fois et qui sont réutilisés dans tous les systèmes d'une SPL. Les éléments optionnels, quant à eux, ils ont varié pour obtenir, à la fin, plusieurs produits où chacun est adapté aux besoins d'une catégorie spécifique d'utilisateurs. Par exemple, dans l'industrie de l'automobile, certains véhicules sont conçus pour le déplacement d'une seule personne alors que d'autres sont conçus pour le déplacement de plusieurs personnes. Ainsi, SPL renforce la production en masse des systèmes logiciels adaptés aux besoins des utilisateurs. Ceci fait que le temps de mise sur le marché et les coûts de développement sont réduits pendant que la qualité des produits s'améliore. La figure 2.1 montre le gain de temps pour la mise sur le marché avec et sans utilisation des SPL. Le temps de mise sur le marché pour un produit unique reste constant. Pour la SPL, le temps de construction des éléments communs à tous les produits

est important. Il est cependant compensé par la réutilisation des éléments communs.

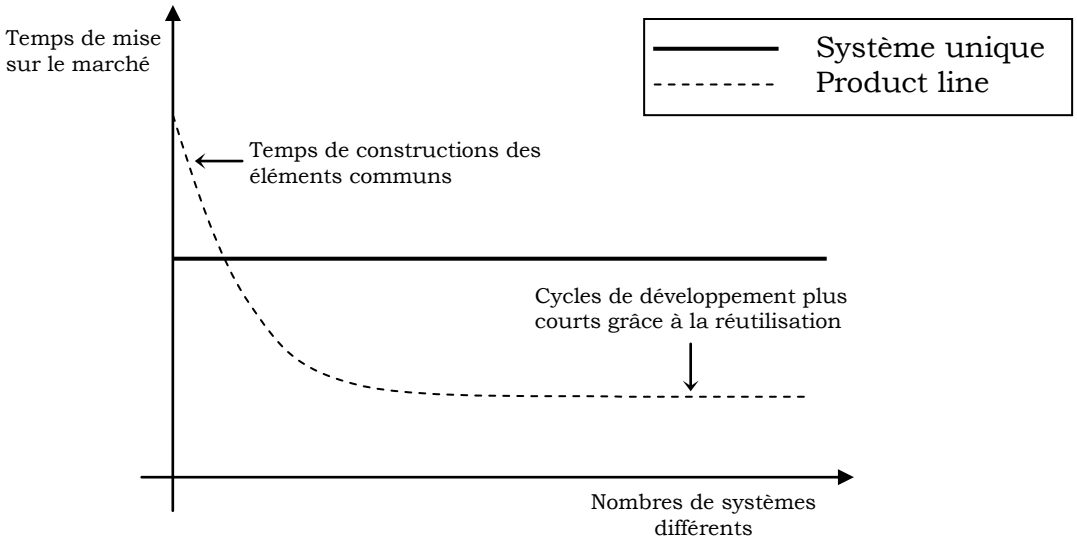


Fig.2.1 Temps de mise sur le marché avec et sans l'ingénierie de lignes de produits

Les succès dus à l'utilisation des SPL sont nombreux et touchent plusieurs domaines comme l'industrie de l'automobile, les systèmes avioniques et la téléphonie. Citons à titre d'exemple Nokia qui arrive à produire de 25 à 30 modèles de téléphones différents par an grâce à l'utilisation du concept des PL [26].

2.3.3. La variabilité

La variabilité dans les SPL a, aussi, été définie dans [19] comme le « *regroupement des caractéristiques qui différencient les produits de la même famille* ». Ce concept est lié à celui de « point de variation » qui permet d'identifier la partie du système où une variation va être introduite dans la famille de produits, donnant naissance à un nouveau membre ou produit dans la ligne. Dans [40], variabilité dans les SPL a encore été définie d'un point de variation comme « *la description de l'ensemble des différences entre les systèmes finaux d'une famille de produits* ». L'activité d'identifier, représenter, exploiter, implémenter et faire évoluer la variabilité dans une ligne de produits est appelée « gestion de la variabilité ». Puisque toute l'architecture des Product Lines et ses propriétés dépendent de la gestion de la variabilité, cette activité constitue l'enjeu principal lors de la conception d'une famille de produits [41]. Pour mieux mettre en évidence la notion de

SPL, les points de variation et les variantes, un exemple de ces concepts est donné à la figure 2.2.

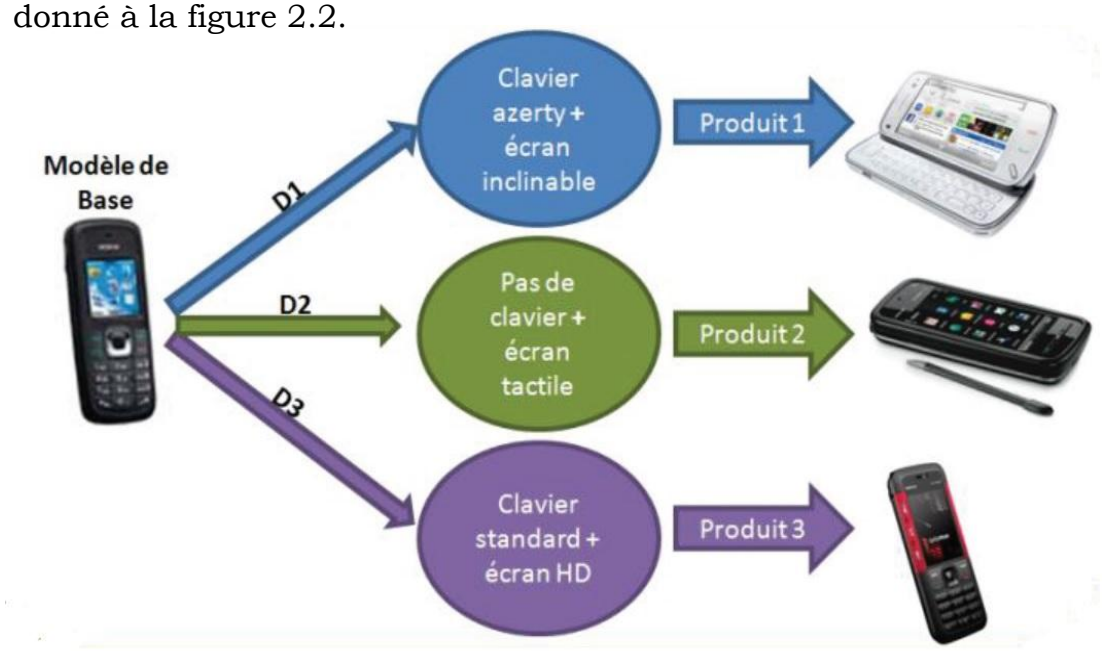


Fig.2.2 Exemple d'une PL

Cette Product Line contient un nombre de composants et de fonctionnalités de base, communes à tous les téléphones de la famille qui sont représentées par le téléphone simple qui se trouve à gauche de la figure. Ces fonctionnalités de base peuvent être, par exemple, l'envoi de SMS, le module d'appels, la gestion des contacts, etc. Ensuite, il y a trois décisions de conception ou configurations différentes qui mélangent les options disponibles en termes de type d'écran et type de clavier. Finalement, dans la partie droite de la figure, nous pouvons observer trois produits dérivés à partir de notre PL. Dans cet exemple nous pouvons identifier deux points de variation, le clavier et l'écran, avec ses variantes {azerty, pas de clavier, standard} et {inclinable, tactile, HD}.

2.3.4. Analyse des besoins (Feature Model)

Rares sont les éléments logiciels directement réutilisables. Avant de pouvoir réutiliser un élément logiciel, il est nécessaire d'identifier sous quelles conditions celui-ci peut être effectivement réutilisé et comment. Plus l'élément pourra être réutilisé dans des contextes différents, moins son

utilisation sera soumise à des conditions et plus sa réutilisabilité augmentera.

De manière simplifiée, chaque **produit** appartenant à une PL détermine un contexte particulier. Chaque produit se compose d'un ensemble d'éléments logiciels représentés par des caractéristiques ou « *features* ». Chaque caractéristique regroupe les exigences devant être satisfaites par les produits logiciels finaux intégrant cette caractéristique. L'objectif d'une caractéristique ou *feature* est de délimiter un ensemble d'*exigences* fortement connexes et directement réutilisables par différents produits finaux. Les *feature*s sont successivement décomposées en sous-*features* jusqu'à l'obtention des *features* terminales. Idéalement, chaque *feature* terminale est associée à un *élément logiciel réutilisable* (composant, service, etc.) implémentant les exigences déterminées par la *feature* correspondante. En outre, ces *features* permettent de distinguer les produits les uns des autres. L'analyse de la **variabilité** permet d'identifier ces *features*, de spécifier les contraintes les reliant et d'expliciter les alternatives offertes lors de la sélection (réutilisation) des *features*. Cependant, de nombreuses sources de variation existent et peuvent apparaître durant toutes les phases du développement logiciel. Le « *Feature Model* » a été défini dans [49] par Ziadi comme une notation standard pour décrire la variabilité dans les lignes de produits. Les figures 2.3, 2.4, 2.5, 2.6 et 2.7 montrent différentes notations des « *features* ».

Feature Obligatoire : Les caractéristiques communes à tous les produits.

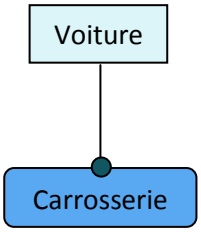


Fig.2.3 Notation standard pour définir une Feature Obligatoire.

Feature Optionnelle : Une caractéristique présente dans seulement certains produits.

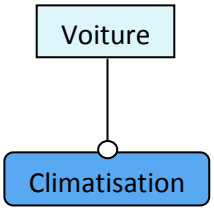


Fig.2.4 Notation standard pour définir une *Feature* Optionnelle

Composition des Features :

OR : choix

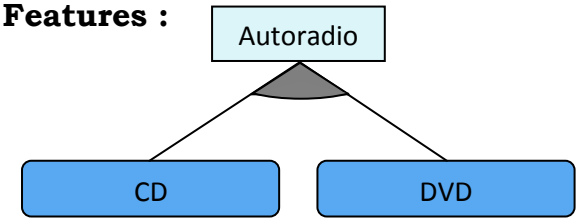


Fig.2.5 Notation standard pour définir le point de variation de type OR

XOR : choix exclusif

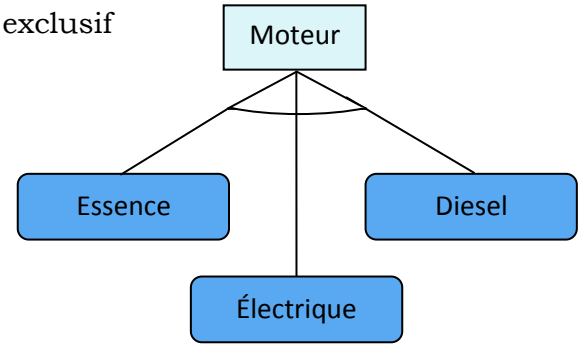


Fig.2.6 Notation standard pour définir le point de variation de type XOR

Contraintes de cohérence : Des dépendances de présence (ou d'exclusion) entre features.

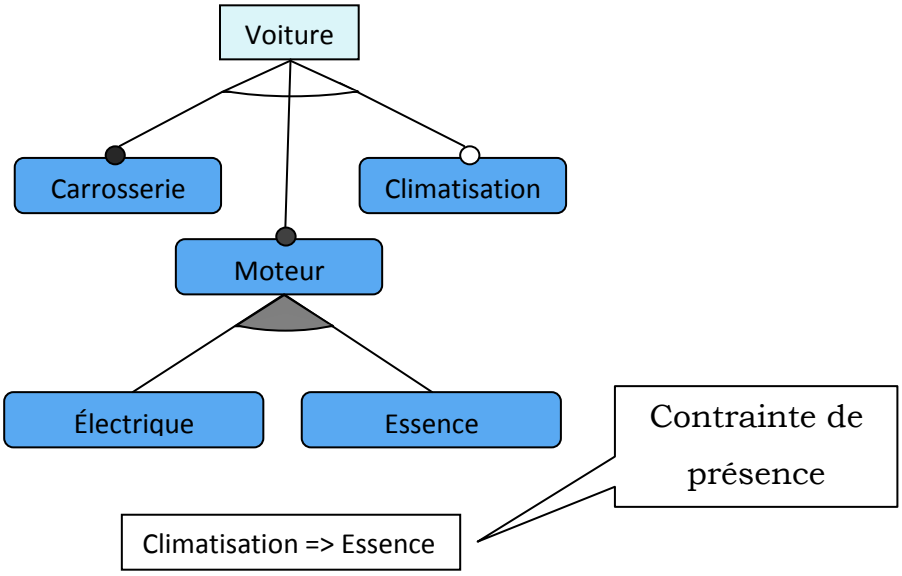


Fig.2.7 Notation standard pour définir une contrainte de présence

Le *Feature* Model sera étudié dans les détails à la section 2.3.5.1

2.3.5. L'ingénierie de domaine et d'application

L'objectif des SPL est de promouvoir une réutilisation logicielle effective et systématique. Pour ce faire, la mise en œuvre d'une PL est divisée en deux processus de développement parallèles. Le premier, appelé ingénierie du domaine ou « *Domain Engineering* » a pour but le développement des artéfacts réutilisables. C'est un processus de développement *pour la réutilisation* en ce sens qu'il prépare les éléments qui seront plus tard réutilisés. Le second processus, appelé ingénierie des applications ou « *Application Engineering* » a pour but de construire de nouvelles applications spécifiques. C'est un processus de développement *par la réutilisation* puisqu'il se base sur les artéfacts logiciels réutilisables développés lors de l'ingénierie du domaine [32]. La figure 2.8 illustre ces deux processus de développement en précisant certaines de leurs activités.

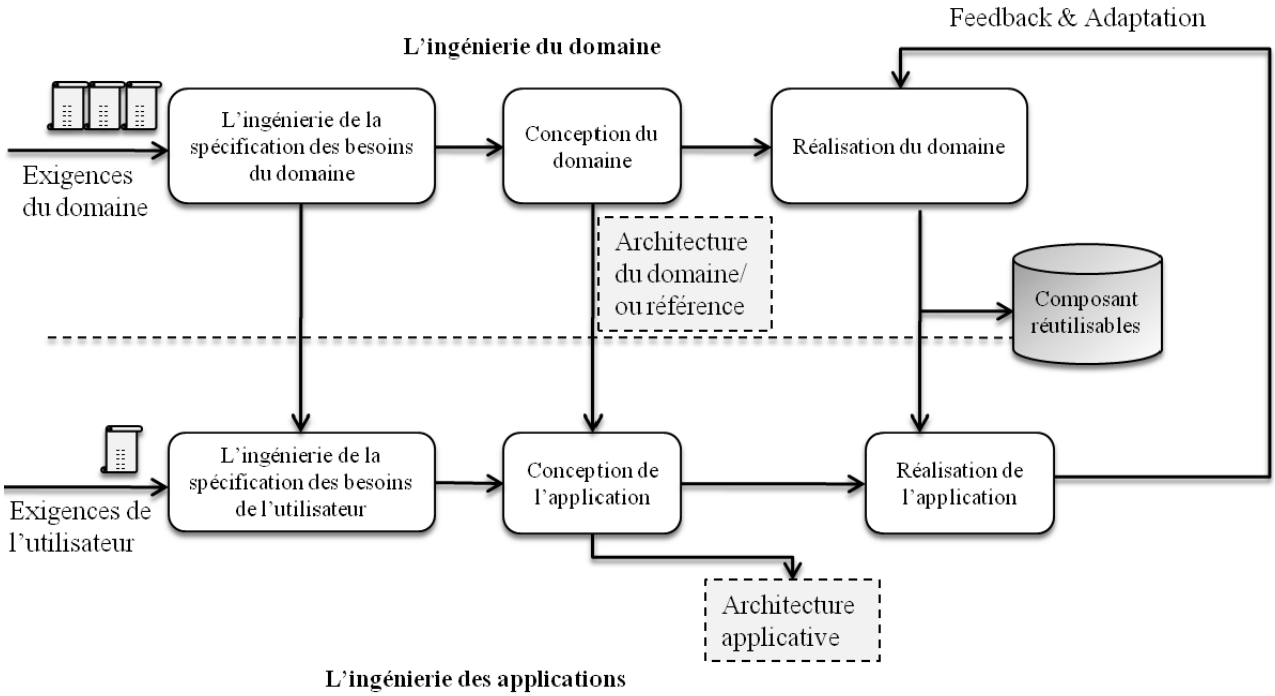


Fig.2.8 Modèle des deux cycles de vie d'une SPL

Le but de l'ingénierie du domaine est de mettre en place une plate-forme de réutilisation. Celle-ci comprend tous les artéfacts logiciels qui peuvent être partagés par des produits appartenant à la ligne de produits. Les artéfacts de différentes natures. Ils couvrent toutes les activités d'une production logicielle allant de l'analyse des besoins du domaine, de la conception

jusqu'à la réalisation et les phases de test. Ils peuvent correspondre à des documents (*une spécification textuelle d'exigence*), des modèles (une *architecture spécifiée dans un outil adapté)*, du code, etc. Ils doivent tous être testés et leur fiabilité doit être avérée. C'est ainsi que l'ingénierie du domaine peut être la base de la production de produits de haute qualité. Enfin, la plate-forme comprend également des stratégies et des procédures de production systématique des produits individuels.

Le but de l'ingénierie des applications est de développer les produits ou systèmes finaux s'appuyant sur la plate-forme établie lors de l'ingénierie du domaine. Dans cet esprit, la construction d'un produit logiciel spécifique est un processus de configuration et d'assemblage d'artéfacts réutilisables. Les artéfacts réutilisables possèdent en effet des parties variables qui doivent être configurées lors d'une réutilisation. Cela permet, par exemple, de ne retenir que certaines parties des exigences d'un domaine ou d'adapter l'architecture de référence aux besoins spécifiques d'un produit. La gestion des variabilités (les différences entre produits) doit être gérée et planifiée. L'ingénierie du domaine fournit des outils cela a été indiqué précédemment. De nombreux mécanismes ont été proposés dans [35] et [36] pour utiliser des points de variation explicites de manière à configurer ou à paramétrer des artéfacts lors du processus de la dérivation de produits.

SPL comporte ces deux activités entrecroisées qui se complémentent et qui interagissent depuis la construction de la PL jusqu'à la construction des produits finaux.

2.3.5.1. Ingénierie de Domaine

L'ingénierie du domaine consiste à développer et construire les assets qui seront réutilisés pour la construction de produits. Notons qu'un asset est un élément qui permet de développer un logiciel, par exemple un document de spécification, modèle, code, etc. Ainsi, il s'agit donc d'un développement pour la réutilisation. Son but est d'étudier le domaine de la SPL et d'identifier les similarités et les variabilités entre les produits. Dans ce premier niveau, nous distinguons trois activités [35] [36] [37] : l'analyse, la conception et l'implantation du domaine. La figure 2.9 présente ces

principales activités ainsi que leurs liens en termes d'entrées requises et de résultats fournis. Cette figure met également en évidence la dépendance entre les processus domaine et applicatif. Le développement d'un nouveau produit requiert parfois l'implantation d'un nouvel artéfact pour répondre à certains besoins spécifiques. Dans ce cas, ce nouvel artéfact peut intégrer la plate-forme de la ligne de produits. Il s'agit d'un feedback du processus applicatif vers le processus de gestion du domaine.

Nous détaillons dans les sections suivantes les tâches principales, à savoir l'analyse du domaine (A), la conception du domaine (B) et l'implantation du domaine (C).

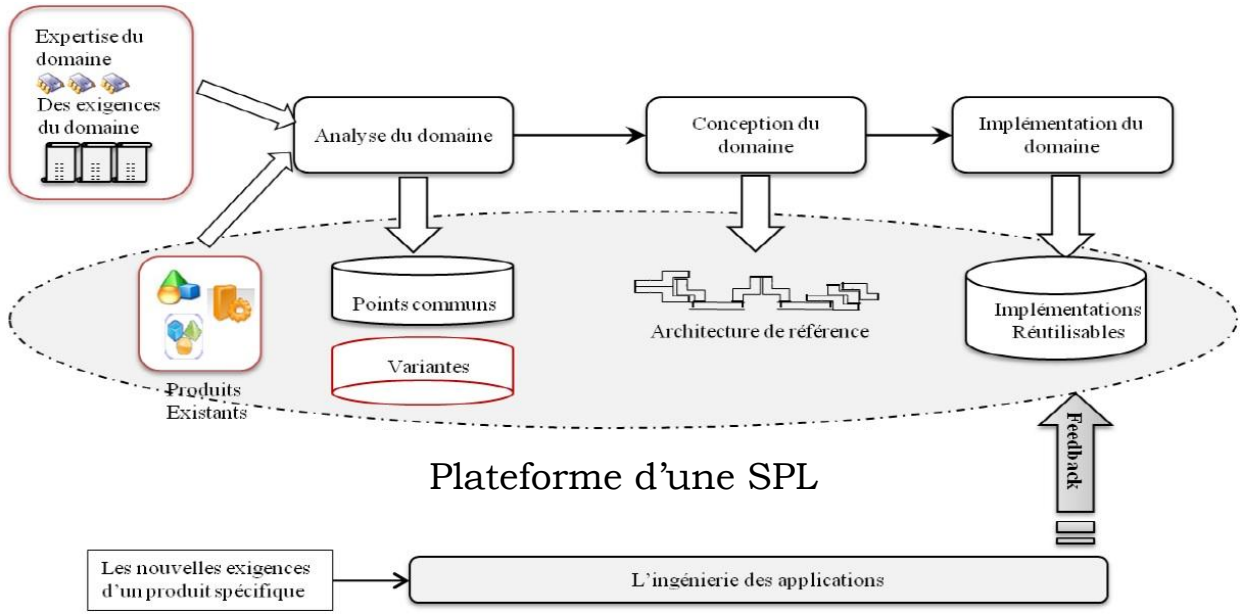


Fig.2.9 Processus de développement de l'ingénierie des domaines

A. *L'analyse du domaine*

De façon générale, l'analyse du domaine métier est une activité visant à définir un domaine et à capturer et formaliser les connaissances dans ce domaine. Il s'agit ainsi d'une activité de transformation de connaissances métier vers des représentations informatiques. Cette activité inclut la définition du domaine en lui-même, la collecte d'informations, le tri et le regroupement des informations, la modélisation des informations. La modélisation produit une description normalisée des connaissances du

domaine et, si possible, un ensemble de procédures systématiques afin de les implémenter [43]. L'analyse du domaine s'appuie grandement sur les éléments existants dans le domaine, tel que des matériels, des logiciels ou des documents. Les produits existants dans un domaine sont donc clairement des entrées pour l'analyse. Les expériences et compétences des experts du domaine (techniques et métier) représentent également des ressources implicites qui impactent fortement le résultat de cette activité. L'étude des exigences du domaine est une partie fondamentale de l'analyse. Elle doit identifier les fonctionnalités du domaine, mais aussi les contraintes, les règles métier, les niveaux de qualité attendus. Les exigences sont généralement considérées comme des informations acquises.

Comme nous l'avons indiqué précédemment, certains aspects sont partagés par tous les produits, c'est-à-dire, ils sont nécessaires pour tous les produits appartenant au portefeuille de la ligne de produits. Ces aspects qui sont les similarités et les variabilités apparaissent naturellement dans les différents artéfacts réutilisables. Il convient donc de fournir des mécanismes de configuration des artéfacts réutilisables pour agir sur les points de variations. La capture des points de variation n'est pas le but final. Il faut en effet identifier et classifier les liaisons possibles entre les variantes de façon à complètement illustrer les solutions alternatives dans le domaine. Cela conduit à la définition d'ensemble de variantes en relation. Chaque ensemble est un point de variation. Au niveau du domaine, les différences entre produits d'une même ligne s'expriment à l'aide des points de variation au niveau du modèle de spécification des exigences. Cela permet d'expliciter les différences entre produits à un haut niveau d'abstraction, souvent dans un langage proche des considérations métier. Une étape fondamentale de l'analyse du domaine est de déterminer le périmètre fonctionnel. Il s'agit de la portée de la ligne de produits [38]. Celle-ci définit explicitement l'ensemble des produits à étudier et implicitement les futurs produits qui pourront entrer dans la PL. Au cours de cette étape, il faut trouver un bon compromis entre coût, effort et qualité de développement de l'ingénierie des domaines et de l'ingénierie des applications. Plus précisément, l'activité déterminant la

portée d'une PL est réalisée à trois niveaux différents en termes de la granularité de la réutilisation logicielle [44]. Il s'agit de :

- *Déterminer le domaine* : le but ici est d'analyser systématiquement les besoins (plutôt les fonctions) sur un segment de marché. Cela permet de guider les décisions d'investissement sur une ligne de produits dans le cadre du marché analysé.
- *Déterminer le portefeuille de la PL* : le but ici est d'identifier les produits à considérer dans la ligne de produits et leurs fonctionnalités supportées par la plate-forme de la ligne de produits. Ceci dépend en partie des besoins du marché.
- *Déterminer le portefeuille des artéfacts réutilisables* : le but ici est de sélectionner les artéfacts abstraits (comme la spécification des besoins) et concrets (comme des composants implémentés) servant de base à la ligne de produits dans un domaine spécifique.

La définition de la portée d'une PL influence plusieurs d'activités au sein des deux processus du développement de la PL. Une mauvaise définition de la portée conduira à l'échec de la construction de la ligne de produits. Les deux cas extrêmes suivants doivent être évités :

- *Définition d'un cadre trop limité* : de nombreux points communs sont définis dans le troisième niveau (le portefeuille des artéfacts réutilisables). Par contre, peu de variabilités sont identifiées parmi les produits de la famille. Les artéfacts réutilisables (comme l'architecture de référence) manquent alors de flexibilité afin d'adapter des besoins spécifiques croissants au cours de nouveaux développements.
- *Définition d'un cadre trop large* : de nombreux points variables sont déterminés dans l'ensemble des artéfacts réutilisables. Les produits inclus dans le portefeuille de la ligne de produits sont très variés. La construction d'un nouveau produit spécifique est donc très difficile, car elle ne repose pas sur suffisamment de points communs. Les développeurs ne peuvent profiter des avantages d'une réutilisation planifiée.

En résumé, le résultat de la définition et de la modélisation du domaine est formalisé et documenté comme une spécification contenant les éléments fondamentaux ci-dessous :

- *le vocabulaire du domaine* : celui-ci définit les notions du domaine exprimant le comportement fonctionnel et/ou les attributs non fonctionnels ainsi que les contraintes.
- *L'ensemble des produits concernés du domaine* : la portée de la PL identifie le périmètre du domaine, le portefeuille des artéfacts réutilisables, les produits et fonctionnalités qui appartiennent au domaine et ceux qui n'y appartiennent pas.
- *l'identification des points communs et variables* : les points communs permettent de capitaliser sur des développements alors que les variantes permettent la configuration pour différencier les applications/produits spécifiques.

L'analyse d'un domaine est une activité plus complexe que la « simple » analyse des besoins lors du développement d'une application individuelle. Cette différence n'est pas toujours perceptible à première vue, car le périmètre du domaine dans le cadre de l'ingénierie des domaines est en général flou au démarrage et, souvent, trop limité. L'analyse du domaine est ainsi considérée comme un processus progressif et relativement non structuré d'apprentissage et d'acquisition d'expérience qui permet au fur et à mesure de clarifier les concepts du domaine jusqu'à ce qu'ils deviennent suffisamment concrets pour pouvoir être transformés en artéfacts réutilisables. De nombreuses méthodes ont été proposées pour guider l'analyse de domaine. Elles servent à découvrir et exploiter des points communs et variables de façon systématique lors de la modélisation du domaine, notamment au sein de méthodes telles que FODA (*Feature-Oriented Domain Analysis*) [38], SCV (*Scope, Commonality and Variability*) ou FAST (*Family-Oriented Abstraction, Specification, and Translation*) [45], la plus connue est la méthode FODA.

Le domaine dans FODA est décrit dans un modèle de caractéristiques spécifié sous forme d'arbre dont les nœuds représentent les caractéristiques

de domaine et les arcs spécifient des liens de composition entre les caractéristiques. FODA distingue trois catégories de caractéristiques : les caractéristiques obligatoires pour tous les produits membres de la PL, les caractéristiques optionnelles qui sont présentes seulement dans certains produits et les caractéristiques alternatives. La figure 2.10 montre un exemple de modèle de caractéristique FODA d'une ligne de PL de voitures [37]. Chaque caractéristique dans le diagramme correspond à un concept du domaine. Les caractéristiques obligatoires sont représentées par des rectangles avec des cercles pleins, tandis que les caractéristiques optionnelles sont représentées par des rectangles avec des cercles vides. La caractéristique *Climatisation* dans le modèle FODA de la figure 2.10 est optionnelle. Il existe trois types de moteurs dans une PL de voitures : électrique, essence ou diesel. Ceci est spécifié par une caractéristique alternative *Moteur* avec trois caractéristiques variantes *Électrique*, *Essence*, et *Diesel*. Une caractéristique alternative est représentée par un arc de cercle à travers les arcs des caractéristiques variantes.

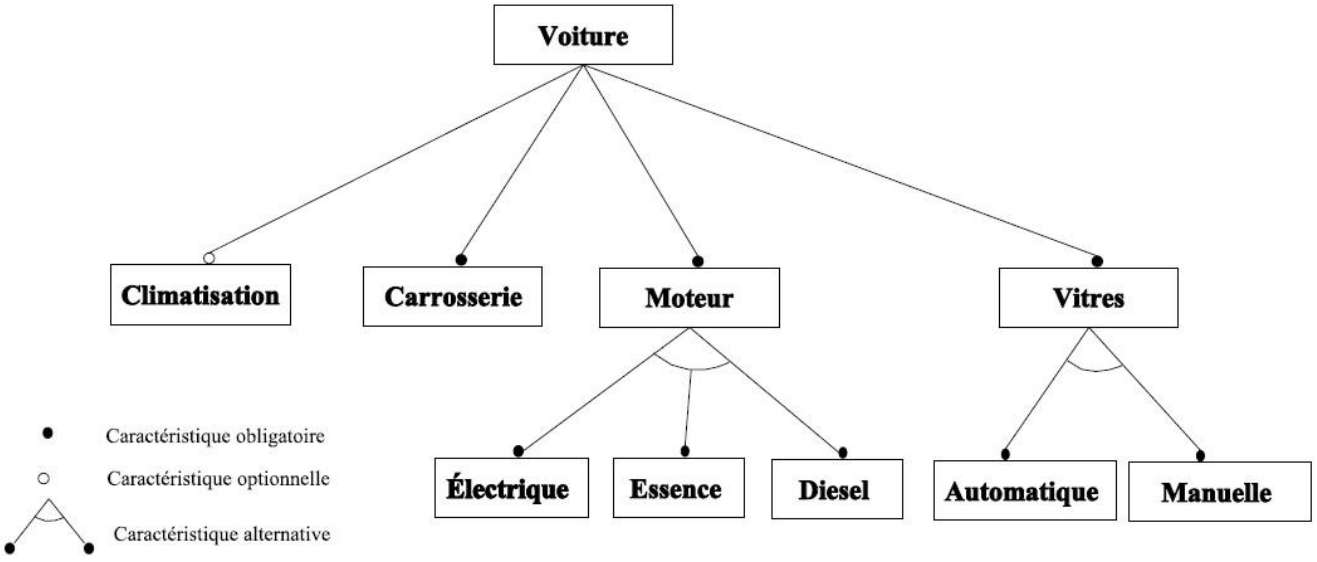


Fig.2.10 Exemple d'un diagramme de caractéristiques FODA

B. *Conception de domaine et l'architecture de référence*

Comme L'illustré la figure 2.11, la conception domaine apparaît à la suite de l'analyse du domaine et repose sur les résultats produits. Son but est de concevoir les éléments logiciels qui feront partie de la plate-forme de la

PL et qui seront donc réutilisables. Deux types d'éléments particulièrement importants sont définis à ce niveau : l'architecture de référence et les composants logiciels pouvant être réutilisés au sein de la PL. Ces composants sont en général configurables ou extensibles, ce qui est une façon d'apporter des points de variation.

L'architecture de référence définit de façon abstraite les composants structurels des produits et leurs relations. Nous l'avons montrée précédemment que l'architecture de référence est un élément clé dans une plate-forme de PL. Généralement, les experts du domaine se focalisent en premier lieu sur ce point. Ils s'appuient sur l'ensemble des modules communs identifiés lors de l'analyse du domaine pour obtenir une première structuration de l'architecture. Cette structuration met en évidence les composants importants et leurs relations. Ensuite, l'architecture est successivement affinée de façon à faire apparaître des points de variation explicites. La création d'un point de variabilité est une décision importante. En effet, rappelons qu'un trop grand nombre de points de variabilité conduit à une ligne de produit inefficace, car demandant trop d'effort lors de la phase d'ingénierie applicative. Dans certains cas, l'architecture de référence permet d'automatiser le développement des produits finaux en définissant de façon systématique comment les artéfacts réutilisables doivent être assemblés. Toutes les architectures des produits spécifiques dans le portefeuille de produits doivent être « conformes » à l'architecture de référence de la ligne de produits.

De façon générale, la *variabilité* [46] est une capacité des artéfacts logiciels leur permettant d'être étendus, modifiés, personnalisés ou configurés sur mesure pour répondre à des besoins particuliers. Au niveau d'une architecture de référence, un *point de variation* [46] identifie précisément en endroit dans l'architecture où des changements/variations peuvent apparaître dans le temps. Un point de variation concrétise une décision de conception retardée. Afin d'obtenir un produit particulier dans une PL, il est nécessaire de prendre cette décision de conception. Un point de variation peut être caractérisé par un ou plusieurs choix appelés des *variantes*. Les

variantes représentent des choix architecturaux possibles sur le point de variation (Fig.2.11). On peut envisager deux situations

- *Le point de variation est « ouvert ».* Dans ce cas, il est possible lors de l'ingénierie applicative d'ajouter une nouvelle variante ou de modifier des variantes existantes.
- *Le point de variation est « fermé ».* Dans ce cas, il n'est pas possible de sortir des choix proposés par le point de variation.

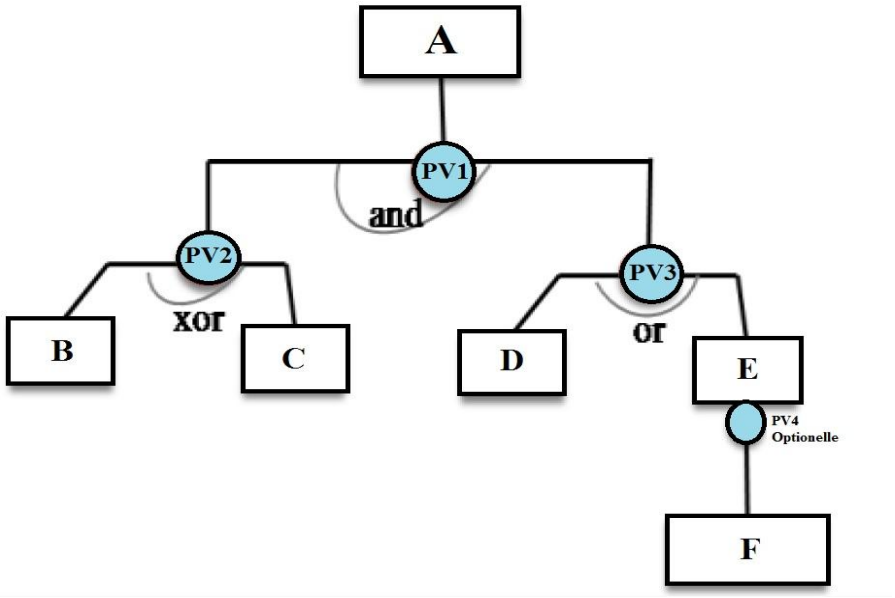


Fig.2.11 Schéma des dépendances des points de variation

Les points de variation apparaissent souvent sur les liaisons entre composants. Ces liaisons peuvent mettre en jeu deux ou plusieurs composants. Dans ce cas, les points de variation peuvent prendre les formes suivantes :

- *Optionnel* : la liaison architecturale ainsi définie n'est pas indispensable pour le système. Par exemple, l'utilisation d'un composant spécifique permettant l'envoi de messages sécurisés peut être optionnelle.
- *Alternatif ou « XOR »* : la liaison architecturale dans ce cas est nécessaire. Le système spécifie un ensemble de composants possibles à relier. Par exemple, on peut spécifier qu'un composant de visualisation est nécessaire, mais qu'il peut prendre la forme d'une télévision ou d'un téléphone portable. Les deux équipements

électroniques fournissent un ensemble de fonctionnalités informatiques communes.

- *Multiples ou « OR »* : la liaison architecturale dans ce cas est également nécessaire. Le système spécifie un ensemble de composants possibles à relier et plusieurs variantes peuvent être sélectionnées à la fois. Par exemple, un système peut inclure un composant de messagerie électronique, un composant de traitement de SMS, un composant de liaison à un fax comme mode de communication entre de différents systèmes ou applications logicielles.

Lorsqu'une variante est choisie sur un point de variation, on dit qu'une décision de conception est prise à ce point de variation. Dans la grande majorité des cas, cette décision est prise lors de la conception d'un produit et n'est pas remise en cause ensuite, notamment durant à l'exécution. Un des apports de ce mémoire est justement de retarder la prise de décision jusqu'à l'exécution.

La résolution d'un point de variation n'est pas toujours indépendante des autres points de variation de l'architecture. En effet, des dépendances entre points de variabilité peuvent exister. En particulier [40], on peut rencontrer les cas suivants :

- La sélection d'une variante donnée à un point de variation entraîne la sélection obligatoire d'une variante spécifique sur un autre point de variation (peut être pour le même point de variation).
- La sélection d'une variante donnée à un point de variation entraîne l'impossibilité de choisir certaines variantes sur un autre point de variation (peut être pour le même point de variation).

L'architecture de référence de la PL a ainsi pour but d'intégrer explicitement des points de variations, caractérisés par d'éventuelles variantes et des liens entre ces points de variation. La gestion de cette variabilité est une problématique plus complexe. Le mécanisme de gestion de variabilités de PL demande de considérer de multiples aspects (*c.-à-d.* la conception des variabilités, l'implantation de variabilités ainsi que la maintenance et l'évolution de la plate-forme de l'ingénierie des domaines). L'introduction

d'un mécanisme de gestion des variabilités représente une forte différence entre les approches de SPL et les autres approches de réutilisations conventionnelles.

C. L'im*plémentation du domaine*

L'implantation de domaine consiste à implanter l'architecture générique définie dans la conception de domaine sous forme de composants qui vont être réutilisés dans l'ingénierie d'application pour la construction de chaque produit. Il est important que ces composants soient adaptables à différents contextes d'utilisation. Il existe plusieurs façons de créer, lors de l'ingénierie du domaine, des composants adaptables ou extensibles. Les plus connues sont les techniques suivantes :

- *Utilisation de l'héritage en orienté objet*. Un composant peut être implanté sous la forme d'un *framework* orienté objet, permettant des modifications ou des ajouts de comportement afin de s'adapter à des besoins spécifiques. Comme pour les architectures de référence, des variantes peuvent être mises à disposition des développeurs et, dans ce cas, les points d'extension peuvent être ouverts ou fermés selon la latitude offerte au développeur.
- *Utilisation d'une interface de configuration*. Un composant peut, dans sa logique interne, prévoir des comportements différents et fournir une interface permettant de choisir le comportement souhaité. On peut également mettre à disposition des développeurs un fichier de configuration pour décrire les variantes souhaitées et, éventuellement, les liens entre les variations. Un mécanisme similaire est la paramétrisation lors des appels des procédures ou des fonctions.
- *Configuration au moment de la compilation*. Les compilateurs peuvent fournir des mécanismes pour adapter un composant lors de la compilation en utilisant un préprocesseur et des macros. Par exemple, un « *makefiles* » peut compiler un composant dans les plusieurs variantes binaires, ou sélectionner différents binaires pour résoudre des dépendances lors de la création d'applications exécutables.

- *Utilisation d'un générateur de code*. Le principe d'un générateur de code est de prendre des spécifications en entrée et de générer un module logiciel, un composant dans ce cas, répondant aux spécifications. Le langage de spécification est plus ou moins ouvert, laissant une place plus ou moins grande aux variations dans le comportement du composant généré. Dans certains cas, le générateur de code devient très compliqué et son développement ainsi que sa maintenance sont donc très coûteux.
- *Utilisation de mécanismes d'extension*. L'adaptation d'un composant peut être définie par addition dynamique de code comme c'est le cas, par exemple, avec les mécanismes de type « *plug-ins* ». Les *plug-ins* sont parfois communs, ou peuvent aussi servir à une application spécifique (il s'agit des variantes).

Ainsi, l'activité d'ingénierie du domaine fournit donc les éléments réutilisables du domaine. Nous nous sommes particulièrement intéressés aux notions de composants adaptables et d'architecture de référence. Cette phase d'ingénierie du domaine peut également fournir un environnement de spécification d'architecture de produits individuels utilisable durant la phase suivante, à savoir, la phase d'ingénierie applicative. Un tel environnement peut s'appuyer sur un langage spécifique au domaine, ce qui facilite grandement le travail des ingénieurs du domaine. Un tel langage porte le nom de langage spécifique au domaine (Domain Specific Language ou DSL) [47].

Pour conclure sur cette partie, nous dirons que le développement de l'ingénierie des domaines est un processus itératif qui vise à la construction des artéfacts réutilisables et des outils qui vont être mis à disposition des développeurs d'applications du domaine. Il est important d'apporter de la flexibilité au niveau des artéfacts réutilisables tout en ne sacrifiant pas la facilité d'utilisation et de maintenance de ces artéfacts. Il est également important de fournir des procédures, des guides, permettant d'étendre ou d'adapter les artéfacts réutilisables. Sans de tels supports, le danger est de retomber dans les travers des anciennes méthodes de la réutilisation.

2.3.5.2. Ingénierie d'application

L'ingénierie des applications est le processus de création et de mise à disposition des logiciels de la PL. Elle consiste à utiliser les résultats de l'ingénierie de domaine pour la construction, appelée aussi dérivation, d'un produit particulier. Il s'agit d'un développement par la réutilisation comme le montre la figure 2.8 précédente. Les résultats de l'ingénierie de domaine (*les modèles de caractéristiques, l'architecture générique, et les composants*) contiennent de la variabilité. La dérivation d'un produit particulier a donc besoin de décisions (ou des choix) associées à ces points de variation. La notion de modèle de décision [40] est utilisée pour capturer et enregistrer les décisions nécessaires à la dérivation de produits.

Comme dans l'ingénierie du domaine, la mise en œuvre de l'ingénierie des applications est divisée en trois activités principales. Ces trois activités du développement sont l'analyse d'application spécifique (A), la conception d'applications spécifiques (B) et l'implémentation d'applications logicielles spécifiques (C). La figure 2.12 illustre de processus et met en évidence les entrées requises correspondant à chaque activité du développement ainsi que le résultat apporté par chaque activité.

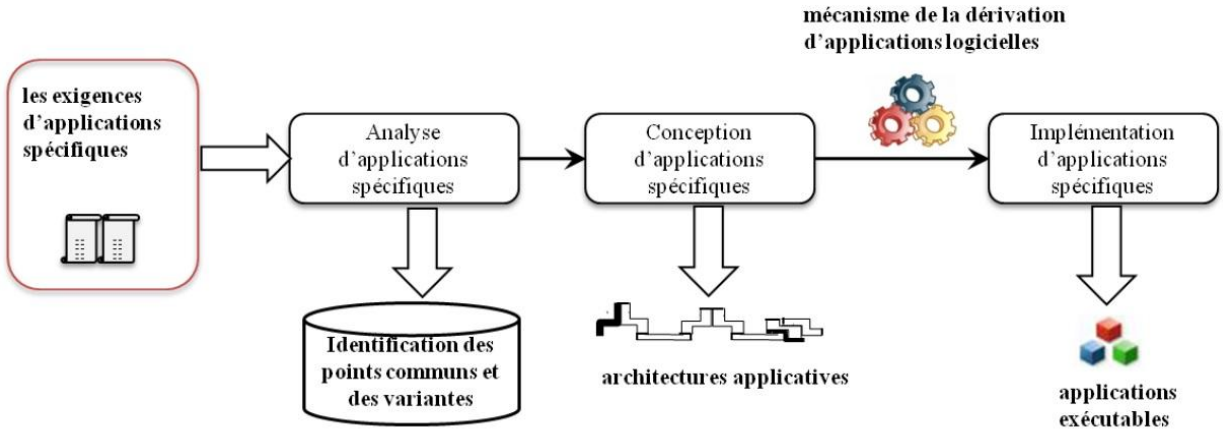


Fig.2.12 Processus de développement de l'ingénierie des applications

Nous présentons en détail ces trois activités majeures de l'ingénierie des applications.

A. *Analyse des exigences d'applications (application spécifique)*

L'analyse des exigences d'applications désigne les exigences spécifiques aux produits souhaités. Ces exigences ont un impact direct sur les artéfacts du domaine qui seront réutilisés dans la construction du produit. Les exigences spécifiques [27] sont obtenues à partir des interactions avec les différents intervenants du développement de produits, tels que des utilisateurs finaux, les manageurs de produits, les clients ou les techniciens de maintenance. Cette étape d'analyse est essentielle et doit apparaître explicitement. C'est elle, en effet, qui met en évidence les besoins spécifiques et justifie les futurs travaux d'adaptation ou de développement.

On voit apparaître trois types d'exigences :

- *Les exigences qui sont incluses dans les exigences du domaine répertoriées lors de l'ingénierie du domaine.* Celles-ci sont connues et sont généralement reliés à des décisions architecturales et d'implantation des composants.
- *Les exigences spécifiques qui sont prévues dans l'analyse du domaine sous forme de variabilités*. Celles-ci donnent lieu à des configurations particulières de l'architecture, via les points de variabilité, et des composants réutilisables.
- *Les exigences spécifiques qui n'ont pas été prévues dans l'analyse du domaine.* Ces exigences posent généralement problème. Elles constituent des demandes ou des contraintes qui donnent lieu à des configurations particulières, non guidées, des artéfacts réutilisables ou à de nouvelles implantations.

La différence entre les exigences prévues et les exigences inattendues doit être soigneusement étudiée. Il s'agit, en effet, de bien vérifier que la portée de la ligne de produits est bien respectée et de ne pas se lancer dans des développements coûteux non réutilisables. Les nouvelles exigences [27] demandent de modifier et d'étendre les fonctionnalités des artéfacts réutilisables. Ces évolutions doivent également être analysées pour décider si elles doivent intégrer la plate-forme de la ligne de produits. Il est également

possible de reporter la mise en place des nouvelles exigences spécifiques à l'aide de l'implémentation de l'application sur la version prochaine.

B. *Conception d'application et l'implémentation*

A partir du résultat de l'activité de l'analyse des exigences spécifiques, l'architecture du produit visé est dérivée de l'architecture de référence conçue lors de l'ingénierie du domaine. Les points de variation définis et planifiés dans l'architecture de référence sont instanciés et configurés durant cette étape du développement pour créer une architecture applicative. Celle-ci est constituée de concepts abstraits issus de l'architecture de référence. Il faut ensuite faire la correspondance avec les artéfacts d'implémentation, typiquement des artéfacts logiciels. Des outils peuvent être fournis pour faciliter et guider la gestion et la résolution des points de variation et le traitement de certains aspects des points communs. Les architectures des produits individuels se différencient à l'aide des points de variation planifiés dans l'architecture de référence. Rappelons que ces points de variation déterminent les endroits précis où des variations sont anticipées et que chaque point de variation représente une décision de conception retardée. Dès qu'une décision est prise lors du processus d'instanciation de l'architecture de référence, la variabilité correspondante est retirée.

Le moment exact où les décisions de conception retardées sont prises est variable. Ce moment, généralement appelé « *Binding Time* », peut survenir tout au long du cycle de vie du développement d'une application spécifique. Si nous reprenons la classification proposée par [48], une décision de conception peut s'effectuer :

- *Au moment de la décision* de réutiliser un composant donné. Lorsqu'un composant est choisi pour être intégré dans une architecture, il est de suite adapté au contexte d'utilisation.
- *Au moment du développement* de l'application spécifique, c'est-à-dire lors de l'assemblage des composants au sein de l'architecture.
- *Au moment de l'instanciation* du code source, juste avant la compilation.

- *Au moment de la compilation.*
- *Au moment de l'emballage* (ou « packaging »), c'est-à-dire lors de l'assemblage de code binaire et des exécutables.
- *Au moment de la personnalisation coté* client, c'est-à-dire lors du processus d'implémentation chez un client spécifique.
- *Au moment de l'installation.*
- *Au moment du démarrage.*
- *Au moment de l'exécution.*

Pour un même produit spécifique, il est possible d'utiliser plusieurs *binding times* afin de traiter les différentes décisions de conception retardées. Cela permet de prendre certaines décisions plutôt au début du cycle de vie, lors de la conception architecturale par exemple, et de prendre d'autres décisions plus tard, jusqu'au l'exécution. Cette approche permet d'intégrer les compétences des différentes personnes intervenant dans un projet. Certaines décisions, par exemple, peuvent être prises par développeur assemblant divers composants métier alors que d'autres décisions peuvent être prises par des utilisateurs finaux pouvant configurer l'application au cours de l'exécution. Lorsque l'on utilise plusieurs « *Binding Times* », le processus de développement devient très incrémental. Les décisions prises à certains points de variations deviennent les entrées d'artéfacts logiciels partiellement instanciés par rapport à la prochaine phase de production.

Il faut néanmoins reconnaitre que, dans la plupart des cas, les décisions se prennent assez tôt. Le «*Binding Times* » correspond rarement à l'exécution dans les PL actuelles.

Bien que les décisions de conception se prennent tout au long du cycle de développement, toutes les décisions doivent être conservées et tracées dans le cadre de la ligne de produits. Certaines exigences spécifiques, qui ne pas inclues dans la spécification des exigences du domaine, donnent lieu à des modifications d'artéfacts existants ou à de nouveaux développements. Ces implémentations produisent des artéfacts logiciels potentiellement réutilisables, qui seront évalués et, éventuellement, intégrés à la plate-forme de la ligne de produits. Cela correspond à la notion de « *feedback* » au sein

d'une ligne de produit. Les artéfacts réutilisables communs sont donc complétés par le biais des implémentations de variantes.

La modélisation des SPL facilite ainsi la réutilisation des concepts manipulés. Plusieurs approches dirigées par les modèles s'intéressent au développement et à la maintenance des SPL. Cependant, la modélisation demeure une tâche complexe voir irréalisable pour les SPL complexes et à très grande échelle. C'est ainsi qu'est venue l'idée de distribuer la tâche de modélisation sur plusieurs équipes, chacune travaille sur un modèle partiel. Chaque fragment de modèle représente une vue partielle du modèle global de la ligne de produits qui doit être obtenu par composition des fragments de modèles qui le constituent.

La composition de modèles de SPL renforce la réutilisation et permet d'obtenir de nouveaux modèles de SPL. Cependant cette tâche est loin d'être facile ou évidente, d'autant plus qu'il s'agit de modèles intégrant des éléments variables. La variabilité est donc un point principal à gérer durant la composition de ces modèles, d'où l'intérêt d'investir dans des travaux de recherche à ce sujet. La sous section suivante présente le rôle joué par la variabilité dans le contexte des SPL.

2.4. La gestion de la variabilité

La gestion de la variabilité joue un rôle essentiel afin de déterminer dans quel contexte, sous quelles conditions et comment une *feature* peut être réutilisée de manière optimale. La gestion de la variabilité est décomposée en trois activités principales :

- *L'identification de la variabilité* qui détermine :
    - les *features* terminales qui permettent de distinguer les produits logiciels appartenant à la même ligne de produits (variabilités).
    - les contraintes qui existent entre ces *features*.
    - où, comment et pourquoi ces variabilités peuvent apparaître (points de variation). Les points de variation correspondent souvent aux *features* non terminales qui facilitent la structuration des *features* terminales.

- *L'implémentation de la variabilité* qui détermine quels mécanismes (héritage, paramétrage, configuration, génération, template instanciation, plugins, etc.) peuvent être utilisés afin de la retranscrire au niveau du code ou de l'architecture.
- *L'évolution de la variabilité* qui indique comment éviter d'introduire des conflits ou de créer des interactions inattendues entre *features* lorsque de nouveaux points de variation ou de nouvelles variabilités apparaissent.

La gestion de la variabilité se complexifie très rapidement (exponentiellement) lorsque le nombre de *features* augmente. C'est pourquoi différents outils logiciels ont été développés afin de faciliter la gestion de la variabilité durant tout le processus de développement. Nous citons : Gears[1], pure, variants[2], GenArch[3] et FeatureIDE[4]. Ces outils offrent principalement trois types de fonctionnalités :

- Des environnements de développement spécifiques aux SPL.
- Des outils de configuration et de gestion du changement destinés aux lignes de produits logiciels.
- Des outils de tests et de vérification adaptés aux SPL.

Un exemple sur FeatureIDE est donné dans la section suivante.

2.5. Exemple de l'éditeur Notepad avec Eclipse Feature IDE

Nous avons le code source d'une application implémentant l'éditeur « Notepad ». Il possède des fonctionnalités classiques :

- Edition
- Ouvrir
- Copier/Coller/Couper (On nommera cette fonctionnalité « Couper »)
- Recherche

[1] http://www.biglever.com
[2] http://www.pure-systems.com/Variant_Management.49.0.html
[3] http://www.inf.puc-rio.br/~ecirilo/genarch/index.html
[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

- Annuler

La question qui se pose est : comment modifier le code source de cette application pour produire plusieurs versions :

- Version 1 : Notepad sans Copier/Coller sans Recherche sans Annuler
- Version 2 : Notepad avec Copier/Coller avec Recherche sans Annuler
- Version 3 : Notepad sans Copier/Coller avec Recherche avec Annuler

Prendre la version originale et enlever "manuellement" le fragment de code concernant les fonctionnalités Copier/Coller, Recherche et Annuler reste une solution. Mais, là, il s'agit de trois features et de quelque version. Imaginez ce que va endurer, par exemple, une industrie de téléphone mobile. Plusieurs modèles de téléphone, trop d'exigences des utilisateurs à travers le monde, plusieurs versions de la même application … La gestion de cette variabilité logicielle ne sera plus aussi facile. C'est la qu'entre en jeu le concept des SPL. La figure 2.12 suivante nous propose un aperçu de l'outil logiciel FeatureIDE dans l'environnement de développement Eclipse.

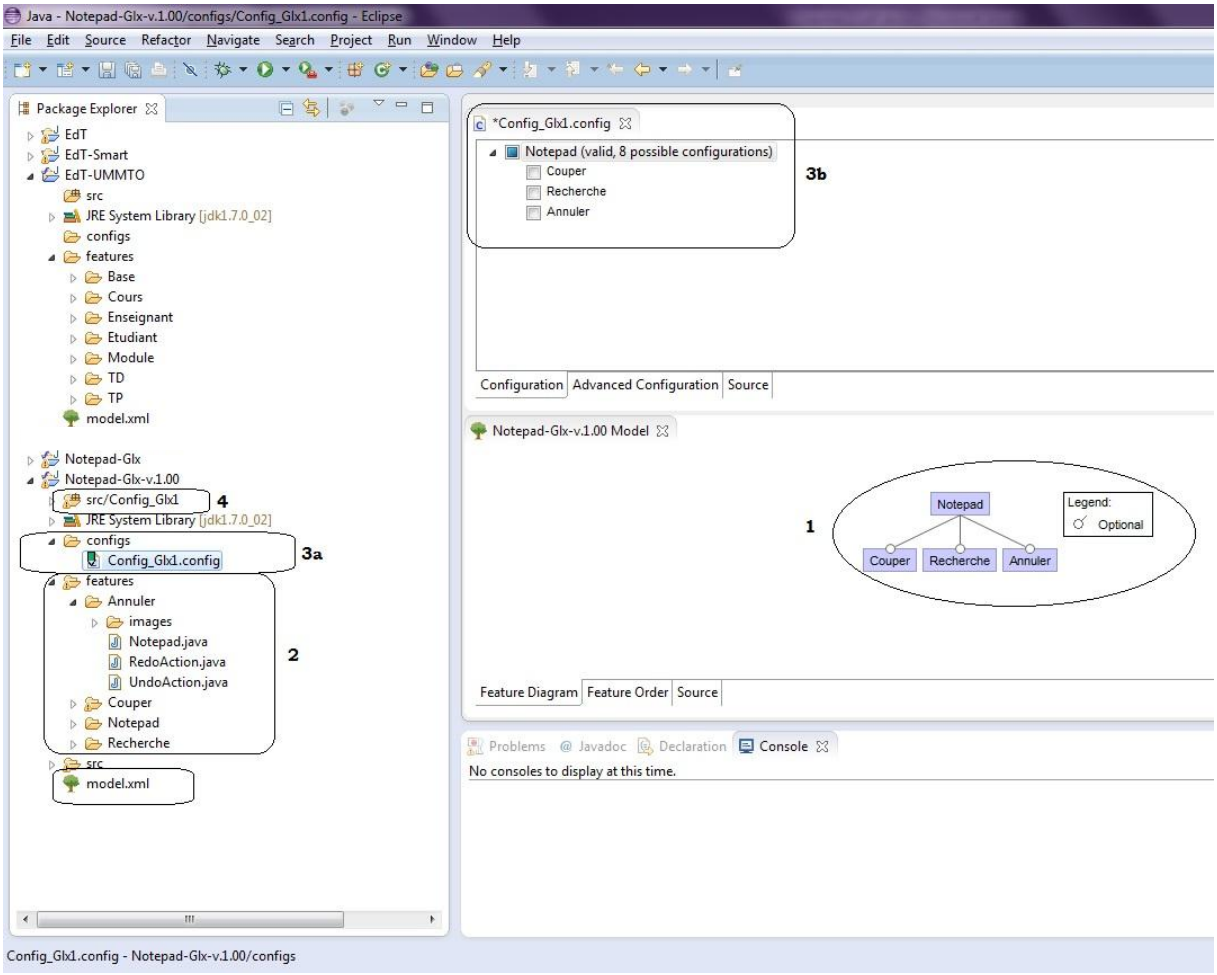


Fig.2.13 FeatureIDE dans Eclipse

1. La représentation du Feature Model avec FeatureIDE.
2. Le répertoire features qui contient les artéfacts de code.
3. La création des configurations de produits de votre choix :
   a. Le répertoire réservé aux configurations souhaitées.
   b. La fenêtre d'édition d'une configuration (ici, 8 configurations possibles).
4. Dossier source du produit configuré après compilation.

2.6. Conclusion

L'approche des lignes de produits logiciels ou Software Product Lines, est très prometteuses en termes d'économie et de réduction des coûts de développement logiciel. Malheureusement, elle était principalement réservée à certains domaines possédant déjà une grande expérience des Product Lines « non logicielles », tels que les télécoms ou la construction automobile. Récemment, ces principes ont aussi été appliqués dans d'autres domaines tels que, l'E.gouvernement [50], l'E.commerce [51] ou l'E.banking [26]. En outre des chercheurs ont fait appel aux avantages des SPL dans d'autres domaines. En ce qui nous concerne, il s'agira du domaine de l'ingénierie logiciel orienté agent, afin d'appliquer les SPL aux SMA et qui fera l'objet de notre dernier chapitre.

Troisième chapitre

Application des Product Lines aux SMA

3.1. Introduction

Le domaine des lignes de produits logiciels (SPL), comme décrit dans le chapitre précèdent, couvre tout le cycle de vie du développement logiciels nécessaires au développement d'une famille de produits dont la dérivation de produits concrets est systématiquement et rapidement faite. D'autre part, l'ingénierie logiciel orienté agent (AOSE) est un nouveau paradigme du génie logiciel qui cherche à appliquer les meilleures pratiques lors du développement de systèmes multi-agents complexes (SMA) en mettant l'accent sur l'utilisation d'agents et organisations (communautés) d'agents en tant que principales abstractions. Malgré un début un peu lent, la technologie agent commence à s'imposer. Les avantages et nécessités de technologies orientées agents et de SMA sont devenu évidents. Malheureusement, les méthodes d'AOSE actuel sont dédiées au développement d'un seul SMA. De toute évidence, beaucoup de SMA utilisent de manière significative les mêmes techniques, adaptations et approches. Le champ est ainsi prêt pour exploiter les avantages des SPL : réduction des coûts, amélioration de la mise sur le marché, et amélioration

de la technologie agents de telle sorte qu'elle puisse être applicable en industrie. En en combinant les deux approches : l'application de la philosophie des SPL pour la construction de SMA cela nous offre tous les avantages des SPL et faire du développement de SMA un développement plus pratique. Ainsi, notre intention dans ce chapitre est double : insister sur l'importance de ce que nous appelons Multi-Agent Systems Product Line (MAS-PL) et présenter les techniques avantageuses de l'approche MAS-PL.

3.2. Ingénierie logiciel orientée agent

Actuellement, une large panoplie de travaux s'intéresse au paradigme orienté agent. Tout autour de ce point, nous exposerons les différentes théories liées à l'application du paradigme orienté agent. Ces théories englobent des modèles, des concepts et des notions ayant été minutieusement étudiées ou appliqués par certaines équipes de recherche [52]. Ces théories visent à être réutilisables et fournissent un référent pour les développeurs.

Les théories orientées agent modifient les procédés de production de logiciels. Afin de mieux appréhender l'application de ces théories, un nouvel axe de recherche appelé Ingénierie logiciel orienté agent ou Agent Oriented Software Engineering (AOSE) en anglais est né. Celui-ci est complètement dédié à la recherche des bonnes pratiques de développement qui aident à l'élaboration d'un système multi-agents. Cet axe de recherche couvre plusieurs aspects [53] liés à la dimension génie logiciel et, plus précisément, aux techniques d'ingénierie. Ceci inclut la recherche de nouvelles méthodes, méthodologies, langages et outils adéquats nécessaires à l'analyse, la conception, l'implémentation, ainsi que la vérification et la validation des systèmes basés sur le paradigme orienté agent. De plus, le paradigme orienté agent est destiné à développer des systèmes complexes ayant une forte interaction, aussi bien entre les composantes du système qu'avec l'environnement. Dans ce contexte, il est nécessaire de pouvoir exprimer et vérifier les propriétés fonctionnelles et non fonctionnelles liées à ces systèmes. Ces propriétés devraient être conservées tout au long du cycle de

vie. Ainsi, dans ce qui suit, nous allons particulièrement nous intéresser aux:

- méthodologies orientées agent qui guident les développeurs durant les phases de développement,
- plates-formes multi-agents qui facilitent l'implémentation et le déploiement des SMA.

3.2.1. Les méthodologies orientées Agent

Les SMA ont montré leur pertinence pour la conception d'applications distribuées (logiquement ou physiquement), complexes et robustes. Le concept d'agent, aujourd'hui, représente un nouveau paradigme pour le développement de logiciels dans lesquels l'agent est un logiciel autonome qui évolue dans un environnement et interagit avec d'autres agents au moyen de langages et de protocoles. Souvent, l'agent est considéré comme un objet « intelligent » ou comme un niveau d'abstraction au-dessus des objets et des composants. Les méthodes de développement orientées objet, au vu des différences entre les objets et les agents, ne sont pas directement applicables au développement des SMA. Il est alors devenu nécessaire d'étendre ou de développer de nouveaux modèles, de nouvelles méthodologies et de nouveaux outils adaptés au concept d'agent.

Après la révolution de la conception et de la programmation orientée objet, nous sommes donc à l'aube d'une nouvelle révolution qui serait celle de la conception et de la programmation orientée agent/interaction/organisation. Le constat concernant la conception de SMA, dans les années 1998-1999, est que le développement des SMA est coûteux en temps à cause de leur complexité, mais aussi, par le fait que pour chaque application, il faut créer le SMA adéquat. En effet, la majorité des applications existant en SMA ont été développées de manière *ad hoc* [54]. Ce foisonnement a conduit en parallèle à de multiples propositions de modèles d'agents, notamment par le rapprochement effectué avec l'approche objet. De là, plusieurs formalismes sont apparus, chacun mettant en avant une représentation de l'agent et de son système. En 1999, se fait donc sentir le besoin de fournir des modèles,

des méthodologies, des plates-formes pour faciliter la prise en compte de la complexité des systèmes à concevoir [55] et pour aider les concepteurs qui ne sont pas nécessairement spécialistes des SMA.

Les grandes familles de travaux concernant les méthodes orientées agent étendent les concepts soit des méthodes orientées objet, soit des méthodes issues de l'intelligence artificielle et plus particulièrement de l'ingénierie des connaissances, soit utilisent les deux [56]. Les premiers travaux sur les méthodologies sont proposés dans [57] [58] [59]. En 2000, cet axe sur les SMA se confirme et les aspects ingénierie des SMA représentent alors le thème de groupes de travail avec le groupe Architecture et Société d'Agents (ASA) de l'Association française d'intelligence artificielle (PRC-GDRI3-AFIA) ou européenne avec les groupes Methodologies and Software Engineering for Agent Systems (MSEAS), d'AgentLink II (2000-2003) et Agent-Oriented Sotware Engineering (AOSE) d'AgentLink III (2003-2005) ainsi que des thèmes de conférences ou de workshops comme AOIS (Agent-Oriented Information Systems), AOM (Agent Oriented Methodologies), AOSE (Agent-Oriented Software Engineering), ESOA (Engineering Self-Organising Applications), ou les JFIADSMA (1999 et 2000). De nombreuses méthodologies sont alors conçues [60] comme, par exemple, ADELFE [61], Gaia [62], INGENIAS [63], MaSE [64], PASSI [65], Prometheus [66], Tropos [67] et Voyelles [68].

Contrairement aux méthodes orientées objet poussées et portées par des industriels, les méthodes orientées agents sont développées et essentiellement utilisées dans le monde académique même si la motivation première de leurs concepteurs était de promouvoir la programmation et le développement orientés agent dans un contexte industriel. Toutefois, bien que n'étant pas moteurs, les industriels sont aussi intéressés par une approche d'ingénierie basée sur leurs exigences qui prennent en compte tout le cycle de vie du logiciel [69] comme en témoignent AML (A*gent Modeling Language*) et ADEM (A*gent Development Methodology*) et le processus de développement associé de Whitestein Technologies [70]. Actuellement, on peut constater que de nombreuses méthodologies ont été développées [60] et

que les travaux s'orientent vers la conception de métamodèles et vers la prise en compte des phases de développement de tests et de déploiement [71]. En effet, l'organisme FIPA (Foundation for Intelligent Physical Agents) œuvre dans le sens d'une standardisation d'une méthode au travers de la définition d'un métamodèle [72]. Les premiers travaux sur la standardisation dans le domaine des agents ont eu pour objectif de faire communiquer des agents développés par différents concepteurs. Pour mettre en œuvre cette interopérabilité, le langage ACL (A*gent Communication Language*) est devenu le langage standard de communication entre les agents. Une architecture et des protocoles ont aussi été développés par la FIPA pour prendre en charge les agents communicants.

La particularité des concepts utilisés dans les SMA rend l'utilisation des méthodologies traditionnelles obsolètes. En effet, ces concepts ont particulièrement élevé le niveau d'abstraction. Ils couvrent une sémantique très large et ont des interrelations très complexes. Ainsi, pour aider le concepteur à développer ce type de système, plusieurs méthodologies orientées agent ont été proposées. Il n'existe pas à l'heure actuelle une méthodologie unifiée pour assister le développement des SMA. Actuellement, les travaux sur la dimension méthodologique prennent en compte divers aspects. Il est difficile dans ce contexte de pouvoir les comparer.

La problématique de la comparaison des méthodes et méthodologies a été souvent soulevée dans la littérature pour aider le concepteur à choisir celles qui sont les plus adaptées au problème donné. Shehory et Sturm ou bien Dam et Winikoff comparent les méthodes suivant quatre ou cinq axes : les concepts manipulés, les notations utilisées, le processus de développement et la pragmatique [73] [74]. Bernon *et al.* [75], utilisent huit critères pour comparer les méthodologies : l'étendue de la couverture du processus, la spécialisation de la méthode à une application, l'architecture d'agent sous-jacente, l'utilisation de notations existantes, le domaine d'application (dynamique ou non), le modèle de rôle, le modèle de l'environnement et l'identification des agents. Plus récemment, Cernuzzi *et al.* [76] se focalise sur les modèles de cycles de vie qui ont été adoptés dans les différentes

méthodologies. Ils ont aussi précisé le degré de recouvrement de chaque phase de développement.

Enfin, nous pouvons aussi citer les travaux de Santos *et al.* [77] qui proposent une table d'évaluation multicritères. Ces travaux peuvent être considérés comme les plus complets. Ils se basent en effet sur 12 critères d'évaluation qui sont :

- la définition du processus,
- l'utilisation d'UML,
- la prise en compte des caractéristiques d'adaptation (l'agent est-il capable de s'adapter à diverses situations en apprenant de son expérience ?), d'apprentissage (l'agent est-il capable d'apprendre à appréhender son environnement ?) et de composition (est-il possible de composer plusieurs agents dans un environnement ?), de mobilité (les agents sont-ils capables de migrer d'une plate-forme à une autre ?),
- la représentation explicite de la structure organisationnelle,
- le traçage des modèles (c'est-à-dire garder la trace les différents modèles développés tout au long du développement),
- la capture des besoins,
- l'identification des agents,
- l'interaction entre les agents,
- le modèle interne des agents,
- la plate-forme d'implémentation,
- la modélisation du processus organisationnel,
- la structure des messages.

3.2.2. Les plates-formes orientées Agent

Tout comme les méthodologies, il existe une multitude de plates-formes multi-agents dédiées à différents modèles d'agent. Les plates-formes fournissent une couche d'abstraction permettant d'implémenter facilement les concepts des SMA. D'un autre côté, elle permet aussi le déploiement de ces systèmes. Ainsi, elles constituent un réceptacle au sein duquel les agents peuvent s'exécuter et évoluer. En effet, les plates-formes sont un

environnement permettant de gérer le cycle de vie des agents et dans lequel les agents ont accès à certains services.
Parallèlement au développement des méthodes, des modèles ont été définis pour aider les concepteurs tels que AGR [78] ou BDI [79]. Pour le développement, les concepteurs disposent de langages orientés agent dont les précurseurs ont été Agent0 et son évolution PLACA ou LALO et METATEM. Des approches impératives (comme JAL1), déclaratives (comme CLAIM [80] ou hybrides (comme 3APL2) ont vu le jour depuis [81], liées ou non à des plates-formes de développement dont les plus célèbres sont Madkit ou JADE. Bien entendu, l'utilisation de langages de programmation classiques (C++, Java, etc.) reste envisageable. Cependant, ces plates-formes ne proposent pas une solution pour faciliter l'utilisation des protocoles d'interaction, à l'exception de la plate-forme JADE. Nous nous sommes ainsi restreints dans notre étude a la plate-forme JADE étant donné que JADE est la seule plate-forme multi-agents à notre connaissance qui propose une bibliothèque de protocoles d'interaction.

La plate-forme JADE (Java Agent Development framework) [82] est certainement celle qui est la plus utilisée par la communauté des SMA. JADE permet de développer et d'exécuter des applications distribuées basées sur le concept d'agents et d'agents mobiles. Elle est compatible à la plate-forme FIPA.
Les agents dans JADE sont implémentés selon les 6 propriétés suivantes :

- Autonomie : les agents ont leur propre thread de contrôle qui leur permet de contrôler leurs actions, de prendre leurs propres décisions afin de réaliser leurs buts mais aussi de contrôler leur cycle de vie.
- Réactivité : les agents peuvent percevoir les évènements de leur environnement et réagissent en fonction de ces évènements,
- Aspects sociaux : les agents exhibent des aspects sociaux qui leur permettent de communiquer et d'interagir entre eux. La communication se fait à travers le passage de messages asynchrones. La communication est considérée comme un type d'actions et peut de

ce fait intégrer un plan d'action. Les messages ont une sémantique et une structure définit par le standard FIPA.

- Dynamicité : les agents ont la possibilité de découvrir dynamiquement d'autres agents et de communiquer avec eux.
- Offre de service : chaque agent offre un ensemble de services. Il peut enregistrer ses services et les modifier. Il a aussi la possibilité de chercher des agents qui offrent les services dont il a besoin.
- Mobilité : les agents dans Jade ont la possibilité de se déplacer. Ils sont implémentés dans des conteneurs et ils peuvent se déplacer.

Il existe une autre plateforme appelée Jadex, qui est basé sur JADE, permettant l'implémentation de l'architecture interne des agents. Jadex [83] [84] prend en compte l'architecture des agents hybrides, c'est-à-dire des agents qui sont à la fois réactifs et proactifs. Pour les agents proactifs, Jadex se base sur le modèle BDI. Jade assure la sécurité en offrant aux applications des systèmes d'authentification qui vérifient les droits d'accès des agents.

Jade n'offre pas de méthodologie. Par contre, plusieurs méthodologies la prennent comme plate-forme cible lors de la génération de code tel que Gaia et PASSI.

L'implémentation de Jade est basée sur Java. La plate-forme peut être répartie sur un ensemble de machines et configurée à distance. La configuration du système peut évoluer dynamiquement puisque la plate-forme supporte la mobilité des agents.

3.3. Application des Product Lines aux SMA

De récentes recherches [85] [86] [87] ont exploré l'intégration des lignes de produits logiciels à l'ingénierie orientée agent. Cette combinaison est appelée les lignes de produits des SMA ou Multi-Agent Systems Product Lines (MAS-PL) [85]. MAS-PL est donc un nouveau domaine de recherche consacré à l'application de la philosophie des PL aux constructions des MAS.

Ces derniers offrent tous les avantages des PL pour le développement de MAS-PL [85].

Notre travail est donc d'étudier le développement d'une famille d'agents logiciels en adoptant des techniques appropriées pour permettre la construction d'agents avec la réutilisation des assets. L'objectif principal de cette étude est d'explorer comment les parties d'une architecture agent peuvent être modularisées et comment celles-ci peuvent être suffisamment génériques pour être réutilisées. Il s'agit donc pour nous d'une modélisation des agents du point de vue intra-agent. Du point de vue interne, un agent est considéré comment étant un système logiciel ayant son propre but. Cependant, peu d'efforts ont été faits en ce qui concerne le processus de l'ingénierie d'application. Rappelons que l'ingénierie d'application consiste à utiliser les résultats de l'ingénierie de domaine pour la construction d'un produit particulier.

3.3.1. Études existantes

Dans cette section, nous présentons les travaux relatifs à notre étude. Elle se compose d'approches qui ont été proposées dans le but de promouvoir la réutilisation du logiciel dans les SMA. Cependant, la plupart des approches sur les SMA n'adoptent pas les réutilisations extensives qui apportent la réduction du temps et des coûts de développement des logiciels. Les premières initiatives, relatives à l'exploitation de la réutilisation en logiciel des SMA, étaient l'utilisation des patrons de conception. Notons que ces études restent toujours au stade de la recherche dans la communauté SMA. Néanmoins, ces études présentent quelques lacunes comme :

1. La plupart des patrons définissent les protocoles au lieu de définir une architecture d'agents, ou du moins, une partie de cette architecture.
2. Certains patrons semblent être plus des frameworks agents que des patrons [88].
3. Des structures organisationnelles globales sont présentées comme des patrons dans [89], mais il y a un désaccord entre la description textuelle du patron et sa mise en œuvre, laissant cette tâche soumise à l'expérience du concepteur.

Des approches récentes ont travaillé sur l'application des SPL aux MAS. Dans [85], une approche a été proposée dans le but de construire une architecture de base d'un MAS-PL basé sur la composition des modèles de rôle. Cependant, l'approche se focalise sur la composition des modèles and non pas comment cette composition est utilisée au niveau de l'implémentation. Dehlinger and Lutz [86] proposent un modèle extensible de spécification des besoins orientés agents pour des systèmes distribués qui prennent en charge la réutilisation en toute sécurité. Cependant, cette approche ne traite que la variabilité à granularité grossière. Nunes et al. [90] proposent le processus de l'ingénierie de domaine pour un MAS-PL et ont fait une étude expérimentale sur les techniques d'implémentation des MAS-PLs [91] Notre étude entre dans l'évolution de cette étude expérimentale. Cette dernière visait la documentation et la modélisation des MAS-PL, mais tout en se focalisant sur les variabilités à granularité grossière. Notons que, dans ce cas, on parle d'inter-agent, c'est-à-dire, les variabilités à granularité grossière sont : agents, capacités, interactions, etc. Les recherches les plus récentes sur les MAS-PL n'ont pas traité les variabilités à granularité fine, c'est-à-dire, les variabilités croyances, buts, plans. Les variabilités à granularité fine sont essentielles lors de l'identification des paramètres ou *features* à partir d'applications héritées. En outre, certaines SPL peuvent bénéficier des variabilités à granularité fine afin de réduire les duplications de code et d'améliorer sa lisibilité [92]. Ainsi, comme nous venons de le préciser plus haut, notre étude suit le travail de Nunes et al. dans [87] et [91]. Celle-ci est basée sur les variabilités à granularité fine. Les problèmes posés lors de cette étude sont :

1. Quels sont les types de variabilités rencontrées pendant le développement de l'SPL.
2. A quel point les modèles d'agents existants sont expressifs.
3. Quels sont les techniques adoptées afin de traiter les variabilités des SPL et comment diffèrent-ils des techniques de l'orienté-objet.

Les différences principales entre cette étude et les autres citées auparavant sont :

1. Dans cette étude, le processus d'ingénierie du domaine se focalise sur comment permettre la documentation de variabilités MAS-PL par l'intégration des approches SMA et PL existants avec extensions
2. Ces études expérimentales se sont limitées aux techniques d'implémentation de deux plates-formes d'agents utilisés JADE et Jadex.

L'étude exploratoire présentée dans ce projet porte sur les différents niveaux de variabilités à granularité, par exemple, les croyances et les paramètres des plans, et cette étude est s'intéresse à l'adoption de nouvelles stratégies visant à améliorer l'architecture de l'agent et pas seulement à l'aide des techniques plates-formes disponibles.

3.3.2. Techniques d'implémentation de variabilité

La plate-forme multi-agent Jadex qui suit l'architecture BDI permet l'implémentation de famille d'agents. Jadex s'appuie sur la programmation d'agents logiciels en XML et le langage de programmation Java. Un agent est défini dans un fichier XML, fichier de définition de l'agent nommé (ADF), qui précise les croyances, buts et plans de l'agent. Un ADF peut également contenir la définition d'autres concepts qui aident à l'implémentation de l'agent, en particulier, les messages qui peuvent être envoyés et reçus. Les plans sont déclarés dans l'ADF, mais leur corps est implémenté dans des classes Java qui héritent de la classe Plan. Afin de permettre l'implémentation des variabilités d'une famille d'agents, nous avons adopté certaines techniques d'implémentation, qui sont décrites ci-dessous.

- *Décomposition de but et modularisation des plans*

La modularisation de la variabilité commence lors de la phase d'analyse, en identifiant les buts et en les décomposant en sous-buts. Lorsque cette décomposition est effectuée, certains buts peuvent être alternatifs ou optionnels. En outre, les plans sont modularisés de manière à ce que chacun d'eux soit obligatoire ou corresponde à une variante unique. La décomposition de but aide à cette modularisation, car plus la granularité des buts sera fine, plus les plans seront spécifiques.

- *Paramétrage des plans*

L'une des caractéristiques de Jadex est de permettre le passage de paramètres aux plans. Un but pourrait être réalisé par plusieurs plans différents. Toutefois, la seule différence qu'il y aura entre ces plans serait un paramètre passé dans un message. Par conséquent, la technique de passage de paramètres à des plans de Jadex permet de réutiliser un seul même plan en fournissant plusieurs paramètres différents.

- *Capacités*

Une capacité [93] est essentiellement un ensemble de plans ou un fragment de la base de connaissances qui est manipulé par ces plans. Ce concept est mis en œuvre par la Plate-forme multi-agent Jadex. Les capacités ont été introduites dans certains SMA comme mécanisme du génie logiciel afin de permettre la modularité et la réutilisation tout en permettant raisonnement de niveau supérieur. Le concept de capacité permet d'encapsuler les croyances, les buts et les plans. Par conséquent, nous avons modularisé ces concepts en une composante, la capacité, qui peut être facilement (dé) liée à l'agent et réutilisée dans d'autres agents.

- *Compilation conditionnelle*

La dernière technique d'implémentation adoptée est la compilation conditionnelle. L'architecture d'une famille d'agents peut avoir des parties optionnelles et alternatives qui n'ont pas été modularisées en code assets spécifiques principalement, car tous les buts, croyances et plans doivent être déclarées dans les ADF. Par exemple, même si un but est réalisé par trois plans différents, c'est-à-dire les classes Java, le plan doit être déclaré dans l'ADF. Par conséquent, les trois plans différents sont déclarés dans l'ADF avec des balises les entourant et indiquant la variante à laquelle se réfère ce fragment de code XML. Grâce à cette information, il est possible d'enlever les fragments qui concernent les variantes non sélectionnées avant la compilation du code. Cette technique est également adoptée dans les fichiers classe Java. Si un but est décomposé en sous-buts, un plan est créé afin de

se débarrasser de l'ensemble des sous-buts, cependant certains de ces sous-buts peuvent être optionnels. Dans ce cas, les balises entourant la dissolution des sous-buts sont introduites dans le code afin de faire la compilation conditionnelle possible.

3.3.3. Leçons apprises

Dans cette section, nous présentons et discutons des leçons tirées de nos études sur les familles d'agents. Pour les construire, nous avons dû adopter des techniques qui prennent en charge la variabilité de domaine et réutilisent les SPL assets pour obtenir différentes configurations de familles d'agents. Nos enseignements sont essentiellement reliés aux points suivants : les types de variabilité que nous avons rencontrés (section 3.3.1), comment les méthodologies orientées agents actuelles sont en mesure de documenter la variabilité et les mécanismes pour permettre la réutilisation de logiciels (section 3.3.2), et les techniques d'implémentation adoptée dans notre étude de cas (section 3.3.3). En outre, nous parlerons également des défis que nous avons rencontrés en utilisant la plate-forme Jadex (section 3.3.4).

3.3.3.1. Types de variabilité

Chaque agent d'un SMA peut être classé sous deux angles différents : (1) en interne comme étant un système logiciel avec son propre but (intra-agent) ; (2) en externe en tant que partie d'une société et interagissant avec d'autres individus (inter-agent). La vue inter-agent, représente les agents qui font partie d'un SMA et leurs interactions. D'autre part, la vue intra-agent représente la structure interne de l'agent. Nous allons, dans ce point, explorer cette vue intra-agent et sa variabilité, autrement dit, notre objectif traitera de variabilité à granularité fine et non pas de variabilité à granularité grossière, telle qu'un agent optionnel.
Dans une famille d'agents, les variabilités suivantes ont été identifiées :

• *Capacités*

Le concept de capacité fournie par Jadex a été utilisé afin de regrouper croyances, buts et plans liés à une préoccupation spécifique, une autre

capacité qui pourrait faire partie d'une famille d'agents est la capacité de négocier, qui regrouperait des concepts afin de fournir à l'agent des moyens qui lui permettront de négocier avec d'autres agents. Par conséquent, les capacités peuvent être optionnelles ou alternatives dans une architecture de famille d'agents.

• *Croyances*

Les croyances des agents, dans l'architecture BDI, influences les deux activités de raisonnement pratique :

1. la délibération est l'activité de trouver les buts que l'agent veut atteindre.
2. l'analyse des fins et moyens est l'activité de décider comment atteindre ces buts.

Par conséquent, la croyance doit faire partie de la connaissance de l'agent si elle participe au moins à une de ces activités. Prenons l'exemple où un agent acheteur a une base de connaissance, d'un magasin, et qui varie selon le type d'achat, livraison à domicile ou achat direct au magasin. La stratégie d'achat est magasin le moins cher ou magasin le plus proche. S'il décide que le produit sera livré et que la stratégie est de choisir le magasin le moins cher, l'agent doit connaître les magasins qui ont le produit en stock en ligne. Si l'agent acheteur choisit le produit directement dans un magasin et que la stratégie est de choisir le magasin le plus proche, l'agent doit connaître les différents magasins qui ont le produit en stock et de leur emplacement. Par conséquent, les croyances d'un agent peuvent également varier. Nous proposerons un aperçu plus détaillé dans le prochain point.

• *Buts*

Les sous-buts permettant la réalisation du but principal peuvent être différents lorsqu'ils traitent avec différentes variantes. Un exemple est de vérifier si un produit est en stock, ce qui sera notre but principal. Ce produit peut être soit directement réalisé par une consultation du stock en ligne (Plan) ou décomposée en deux sous-buts (trouver un magasin pas loin et vérifier si le produit est en stock dans ce magasin). Par conséquent, il y a

deux sous-buts optionnels. Dans ce cas, il n'y a pas de but alternatif étant donné que ce dernier cherche juste à vérifier si un produit est en stock. Cela est une possibilité de variabilité.

• *Plans*

De la même manière qu'un but, qui peut être décomposé en différents ensembles de sous-buts et peut tout aussi bien être atteint par des plans différents. Ainsi, ces différents plans sont alternatifs. En outre, il existe des plans optionnels. Si un but est optionnel, le plan qui permet de l'atteindre est également optionnel.

• *Les paramètres du plan*

Comme cela est indiqué dans le point précédent, le paramétrage des plans fourni par Jadex nous offre un avantage. En effet, il existe des paramètres alternatifs qui sont donnés comme entrée d'un plan. Ainsi, cela nous permet de réutiliser les actions du plan pour implémenter plusieurs variantes différentes.

3.3.3.2. Documentation et modélisation de la variabilité

Plusieurs approches SMA ont été proposées dans le but d'analyser et de concevoir des SMA comme les méthodologies [94] [95] et les langages de modélisation [96] [97]. Cependant, ces approches offrent des modèles fondés sur des concepts qui n'existent pas nécessairement dans la plate-forme d'implémentation. En conséquence, l'effort dépensé pour développer de bons modèles de conception, c'est-à-dire, en tenant compte des principes tels que la modularité et la réutilisation, n'en vaut pas la peine, car ils ne peuvent être reproduits dans le code. Cet écart de compréhension entre les modèles de conception et d'implémentation rend le code plus difficile à comprendre parce que les éléments au niveau de l'implémentation ne correspondent pas directement aux éléments se trouvant au niveau de la conception. Par conséquent, il est également difficile de maintenir et de faire évoluer le code.
À côté, nous avons identifié d'autres lacunes dans les méthodologies orientées agents et langages de modélisation. Elles sont principalement liées

à l'absence de mécanismes permettant de concevoir des éléments réutilisables. Des exemples de tels mécanismes sont les suivants :

- *Réutilisation des buts*

Beaucoup d'approches SMA adoptent le concept de but. Certaines d'entre elles associent les agents à une liste de buts et d'autres représentent les buts comme un arbre, ce qui signifie que les fils d'un but sont les sous-buts qui doivent d'abord être atteints afin d'atteindre ce but, cependant, les sous-buts peuvent être nécessaires pour atteindre plus d'un but. Par conséquent, les buts ont une relation n:n, et non 1:n tel qu'il est exprimé dans un arbre. Une méthodologie qui permet de modéliser cette relation n:n est Tropos, qui est basé sur i * Framework ;

- *Les capacités*

Les capacités sont un mécanisme qui permet de modulariser et de réutiliser un comportement de l'agent spécifique. Il a été introduit par la plateforme JACK, mais quelques approches seulement ont adopté les capacités comme un élément de première classe

- *Paramétrage de plan*

Jadex permet de fournir des entrées aux plans, ce qui permet d'instancier les plans dans différents contextes. Il s'agit d'un mécanisme qui peut être adoptée au niveau de l'implémentation, mais seulement quelques approches de conception orientées agent l'offrent.

Les SMA sont essentiellement inspirés de la nature humaine sous ses différents aspects, telle que l'aspect organisationnel et cognitif. Cela est intéressant, d'un point de vue informatique seulement si l'utilisation d'approches inspiré-humain apporte des avantages en matière de développement de logiciels. Par conséquent, il serait intéressant d'évaluer si les techniques ne suivant pas strictement les modèles humains peuvent être introduites dans les actuelles approches orientées agent afin d'améliorer le développement de logiciels.

3.3.3.3. Techniques d'implémentation et modularisation de variabilités

L'un des avantages majeurs d'AOSE et que ce paradigme permet de décomposer un problème complexe concernant les agents autonomes, qui communiquent entre eux par messages. Une principale différence qu'il y a entre un agent et un objet est que le premier encapsule non seulement des données (son état), mais aussi le processus de sélection de comportements et quand de tels comportements sont nécessaires [98]. Cela permet de construire un système composé d'éléments à forte cohésion et à faible couplage. Néanmoins, ces principes ne sont généralement pas pris en compte lors de la modélisation et de l'implémentation de la structure interne des agents.

L'architecture BDI définit que les agents ont un ensemble de croyances, buts et intentions (plans). Cette structure repose sur des bases philosophiques, qui sont inspirées du raisonnement humain. Toutefois, si un agent a plus d'une responsabilité, les concepts liés à chacun d'eux seront mélangés dans l'architecture de l'agent, ce qui conduit à un code plus difficile à comprendre et à maintenir, par exemple, si un agent acheteur était un utilisateur avec différents services comme, acheter des produits et chercher sur le web. Les croyances, les buts et les plans associés aux services feraient partie de cet agent et il n'y a aucun moyen de savoir à quel service une certaine croyance est nécessaire. La première idée permettant de résoudre ce problème a été l'introduction de la notion de capacité, introduite par la plate-forme JACK. Dans ce cas-là, tous les buts, les croyances et les plans associés au service de l'achat d'un produit seront encapsulés dans une capacité.

Avec la décomposition des buts et la modularisation des plans, les variantes dans une famille d'agents pourraient être modularisées dans des plans simples. Toutefois, étant donné que tous les buts, croyances et plans font partie des agents (ou capacités), et doivent être définis dans des ADF, le code associé aux variantes est réparti partout dans les ADF. Même si la compilation conditionnelle a pu résoudre le problème de gestion de variabilité, cette technique n'est tout de même pas une bonne pratique, car

elle conduit à un code et la configuration des connaissances qui sont difficiles à comprendre et à maintenir.

Dans le contexte du génie logiciel, plusieurs approches ont été proposées afin d'améliorer les architectures logicielles, suivant des principes tels que la dissimulation d'information, l'encapsulation, la réutilisation, la maintenabilité, la forte cohésion et le faible couplage. Un exemple d'une approche qui pourrait être appliquée à une famille d'agents est l'utilisation des patrons de conceptions, la Fabrique Abstraite, dont l'intention est de fournir une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes. Dans les SPL il y a des ensembles de plans alternatifs pour la réalisation d'un ensemble de buts donnés. Les plans alternatifs ont les mêmes prés et post conditions, c'est-à-dire, la même interface en étant différents dans la façon dont ils atteignent un but. Par conséquent c'est exactement le problème que le patron de conception « Fabrique abstraite » résout. Notre objectif est d'instancier une famille de plans selon une variante choisie. Donc l'AOSE devrait prendre une leçon des travaux de recherche qui ont été faits dans le génie logiciel « traditionnel » afin de concevoir et implémenter de meilleures architectures logicielles.

3.3.3.4. L'utilisation de la plate-forme d'agent Jadex

Dans le point précèdent, nous avons introduit la plate-forme Jadex, qui est utilisée pour implémenter des familles d'agents. Dans ce point nous allons parler de l'utilisation de Jadex et des défis rencontrés avec. Le principal avantage de Jadex est qu'elle fournit des concepts de l'architecture BDI pour les développeurs. Ainsi, un agent modélisé utilisant cette architecture peut être directement implémenté sans définir de stratégie quelconque pour implémenter les concepts modélisés. En outre, le concept de capacité qui permet le packaging d'un sous-ensemble de croyances, plans et buts dans un module d'agent est de réutiliser ce module là où il se doit. En conséquence, on peut facilement (dé) lié les capacités aux agents et de les réutiliser.

Comme indiqué précédemment, Jadex définit les agents dans des fichiers XML. Et cela apporte des inconvénients au cours de l'implémentation. La

recherche d'erreurs dans les fichiers XML est une tâche fastidieuse. De plus, les fichiers XML ne sont pas compilés; ce qui fait que compiler une application ne capturera pas les erreurs à l'intérieur. En conséquence, les fichiers XML sont validés uniquement avec leur DTD, qui ne suffit pas pour détecter les fautes de frappe. Par exemple, si un but est référencé dans le fichier XML avec une mauvaise lettre, une erreur se produit uniquement lors de l'exécution et le message est que le fichier XML contient des erreurs. Par conséquence, le développeur doit trouver l'erreur manuellement. Par ailleurs, même si les plans sont des classes Java, les croyances et les paramètres sont récupérés par des méthodes qui retournent un objet de la classe Object, donc il doit y avoir un typecasting lors de l'appel de ces méthodes. Cela conduit, à nouveau, à la capture des erreurs que lors de l'exécution. En outre, l'utilisation de fichiers XML n'est pas appropriée pour l'adoption de techniques de modularisation, comme indiqué ci-dessus. Par conséquent, nous visons à explorer d'autres plates-formes d'agents pour enquêter si elles fournissent des solutions à ces carences en Jadex.

3.4. Conclusion

Les SMA visent à la conception et la modélisation de logiciels complexes et de systèmes distribuées en termes d'abstractions de haut niveau afin de réduire l'écart entre l'espace de problème et l'espace de solution facilitant ainsi la modélisation du système a de telles abstractions de calcul et la communication avec les intervenants. Cependant, ces modèles ne seront pas effectivement adoptés dans l'industrie s'ils ne favorisent pas un *timetomarke*t réduit, des coûts de développement bas et une meilleure qualité. Les travaux de recherche sur la réutilisation des logiciels se sont penchés sur les questions de thèses depuis plusieurs années, néanmoins quelques efforts de recherches ont été effectués dans le cadre de SMA dans ce sens.

Ainsi, nous avons présenté une étude exploratoire du développement d'une famille d'agents à l'aide de l'architecture SPL. Cela permet de dériver des agents personnalisés configurés selon la spécification de l'utilisateur afin

d'entrer dans un SMA existant. Concernant les familles d'agents, nous avons exploré différentes granularités de variabilité, y compris les variabilités à granularités fines, qui sont les croyances, les buts, les plans et les paramètres du plan. Sur la base de notre étude, nous avons présenté et discuté des questions importantes soulevées au cours de son développement. Ces questions sont principalement liées à l'absence de techniques, à la fois ; dans les niveaux de conception et d'implémentation, pour développer des SMA en tenant compte des principes du génie de logiciel traditionnels, tels que la modularité, la réutilisabilité et la maintenabilité. Plusieurs pratiques du génie logiciel ont été proposées afin d'améliorer le développement de logiciels, durant ces dernières décennies, ainsi que les approches orientées agent qui devraient tirer une leçon de leur part.

En tant que futur travail, nous viserons à étudier l'utilisation de ces SPL afin de modifier dynamiquement une configuration d'agents. Dans notre étude, toute variabilité à un temps contraignant de compilation, donc, une fois l'agent dérivé, il ne peut pas changer son comportement pour adopter une configuration avec d'autres variantes. L'adoption de techniques pour développer une architecture SPL résulte de composants à faible couplage, qui peuvent être échangés d'une manière plus facile. Par conséquent, nous croyons que l'utilisation d'architectures SPL améliore le développement d'agents qui peuvent changer leur configuration lors de l'exécution et s'adapter en fonction du contexte actuel.

Conclusion générale

La combinaison de l'ingénierie des Softwares Product Lines (SPL) et de l'ingénierie logiciel orienté agent (AOSE) appelé MAS-PL nous offre de nouvelles perspectives au vu du développement de systèmes complexes distribués. En effet, tout au long de ce présent travail, nous avons vu que, grâce aux différentes méthodologies et plates-formes de l'ingénierie logicielle orientée agent, le développement de SMA peut être d'une véritable complexité. En appliquant les techniques des SPL aux SMA, les coûts en temps dû à la complexité de l'architecture et de son développement ne seront plus une réelle contrainte grâce au principe de gestion de variabilité et de réutilisation de code. De ce fait, pourront être applicables dans l'industrie qui reste une priorité au sein de la communauté des SPL et SMA.

Ainsi, ce travail concernant las applications des SPL aux SMA, nous a permis :

- D'abord, de prendre conscience de la complexité du développement de SMA ainsi que des méthodologies orientées agents existantes qui peuvent montrer leurs limites à ce niveau,
- De découvrir la philosophie de SPL et comprendre son fonctionnement, autrement dit, ses avantages liés à la réutilisation en logiciel.
- D'étudier l'application du principe des SPL aux SMA, ce qui permet de rendre le développement des SMA beaucoup plus pratique, voire révolutionner ce domaine.

Ce thème entre bien dans la spécialité de notre formation et, actuellement, il fait objet d'une forte demande dans le monde industriel. Malgré les diverses difficultés rencontrées, sachant qu'il s'agit d'un thème nouveau pour nous dans le cadre de notre formation et qui est encore au stade de la recherche, nous espérons que notre mémoire soit initiateur en son genre et sera à l'origine de futurs travaux innovant dans le domaine des MAS-PL. Nous souhaiterions aussi qu'il ouvre une nouvelle fenêtre dans le domaine de la recherche au sein du département informatique et auquel nous souhaitons y contribuer.

Références bibliographiques

1. J. FERBER, Les systèmes multi-agents : Vers une intelligence collective, InterEditions, 1995.
2. N.R. Jennings, K. Sycara, M. Wooldridge, A roadmap of agent research and development, Autonomous Agents and Multi-Agent Systems, (1):7–38, 1998.
3. D. McFarland, Dictionnaire du comportement animal, Robert Laffont, 1990.
4. Feyereisen, P., de Lannoy, J.-D.: Gestures and speech: Psychological investigations. Cambridge University Press, New York (1991). B. Cyrulnik, Sous le signe du lien, Hachette, 1989.
5. M. Wooldridge, An Introduction To Multiagent Systems, 2009.
6. L. Erman, F. Hayes-Roth, V. Lesser, D. Reddy, The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, ACM Computing Surveys, 12, 1980.
7. Edgar Morin, La méthode 1 : La nature de la nature, Éditions du Seuil, 1977.
8. GALLAND, S., GRIMAUD, F., BEAUNE, P., AND CAMPAGNE, J.-P. MaMA-S : an introduction to a methodological approach for the simulation of distributed industrial systems. International Journal of Production Economics (special issue) 85 (Mar. 2003), 11–31.
9. COSSENTINO, M., GAUD, N., HILAIRE, V., GALLAND, S., AND KOUKAM, A. ASPECS : an agent-oriented software process for engineering complex systems - how to design agent societies under a holonic perspective. Autonomous Agents and Multi-Agent Systems 2, 2 (Mar. 2010), 260–304.
10. GALLAND, S. Approche multi-agents pour la conception et la construction d'un environnement de simulation en vue de l'˝evaluation des performances des ateliers multi-sites. PhD thesis, éccole Nationale Supérieure des Mines et Université Jean-Monnet, Saint-Etienne, France, Dec. 2001.
11. Chauvet, G. (1998). La vie dans la matière. Flammarion.
12. Thiefaine A., Guessoum Z., Perrot J.-F., Blain G., « Génération de systèmes multiagents à partir de modèles », JFSMA'03, Hermès, p. 107-111, 2003.
13. An Introduction to MultiAgent Systems. John Wiley and Sons, 2002.
14. F. ARLABOSSE, M.-P. GLEIZES et M. OCCELLO : Méthodes de Conception. Dans Systèmes Multi-Agents, volume 29 de Arago, pages 137–171. Editions Tech & Doc, 2004.
15. P. BALL : The Self-Made Tapestry. Oxford University Press, 1998.
16. J.-P. GEORGÉ : Résolution de problèmes par émergence – Étude d'un Environnement de Programmation Émergente. Thèse de doctorat, Université Paul Sabatier - Toulouse III, 2004.
17. H. HAKEN : Synergentics : an Introduction. Springer Verlag, 1978.
18. T. Ziadi, J.M. Jézéquel, Manipulation de Lignes de Produits Logiciels : une approche dirigée par les modèles, IDM'05, Premières Journées sur l'Ingénierie Dirigée par les Modèles, Paris, 30 juin, 1juillet 2005.

19. D.L. Parnas, On the design and development of program families, IEEE Transactions on Software Engineering, SE2(1) :1–9, March 1976.
20. F. van der Linden, Software Product Families in Europe : The Esaps and Café Projects. Software, 19(4) :41–49, July/August 2002.
21. C.W. Krueger, Software Reuse, ACM Computing Surveys, Vol. 24, N°2, pp.131–183, 1992.
22. D. Mcilroy, Mass-produced Software Components, in Proceedings of Software Engineering Concepts and Techniques, Garmisch, Germany, 1969, p. 138-155.
23. R. L. Glass, Frequently forgotten fundamental facts about software engineering, in the Journal of IEEE Software, IEEE, vol. 18, no. 3, p. 112 -111, mai 2001.
24. D. M. Weiss, C. T. R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.
25. P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, 1er éd. Addison Wesley, 2001.
26. K. Pohl, G. Böckle, F. V. D. Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Birkhäuser, 2005.
27. L. Bass, P. Clements, and R. Kazman, Software Architecture in Practices. Addison-Wesley, 1998.
28. A. Maccari and A. Heie, Managing infinite variability, In Jilles van Gurp and Jan Bosch, editors, Software Variability Management Workchop, number IWI preprint 2003-7-01.
29. A. Maccari and A-P. Tuvinen, System family architectures : Current challenges at nokia. In F. van der Linden, editor, IW-SAPF, volume LNCS.
30. Web : http://splc.net/fame.html
31. Jézéquel, Jean-Marc, Perrouin, Gilles, Vers des lignes de produits flexibles : Apports de l'ingénierie dirigée par les modèles à la dérivation de produits, Paris, France : Lavoisier, 2008, Vol. 14. 1262- 1137.
32. T. Ziadi, Hélouët, Loïc, Jézéquel, Jean-Marc, Towards a UML Profile for Software Product Lines, 2004, PFE 2003, LNCS 3014, pp. 129-139.
33. S. Davis, «Future Perfect» 10th Anniversary Edition, Addison-Wesley Pub Co, Harlow, England, 1996.
34. ESAPS project, http ://www.esi.es/esaps/, 1999.
35. CAFE project, http ://www.esi.es/en/projects/cafe/cafe.html, 2001.
36. W.U. Eisenecker and K Czarnecki. Generative Programming : Methods, Tools, and Applications. Addison-Wesley, 2000.
37. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990.
38. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J.Wüst, and J. Zettel. Component-based Product Line Engineering with UML. Component Software Series. 2001.
39. Linden, Frank, Schmid, Klauss et Rommes, Eelco. Software Product Lines in Action. Berlin : Springer-Verlag, 2007. pp. 3-5.
40. Millymaki, Tommi. Variability Management in Software Product Lines. Tampere, Finland : Tampere University of Technology, 2001.
41. Web : http://www.toupie.org/Dictionnaire/Fordisme.htm
42. P. Clements, “On the Importance of Product Line Scope”, in Proceedings of the 4th International Workshop, Software Product-Family Engineering, Bilbao, Spain, pp.70-78, 2001

43. K. Schmid, “Scoping software product lines – an analysis of an emerging technology.”, in Proceedings of the first conference on Software product lines, Denver, Colorado, United States, pp. 513 – 532,2000.
44. J.O. Coplien, D.M. Hoffman and D.M. Weiss, “Commonality and variability in software engineering”, IEEE Software, Vol.15, No.6, pp. 37-45, 1998.
45. J. van Gurp, J. Bosch and M. Svahnberg, “On the Notion of Variability in Software Product Line”, in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, pp.45-54, 2001.
46. M. Voelter with S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, G. Wachsmuth, "DSL Engineering - Designing, Implementing and Using Domain-Specific Languages", dslbook.org.
47. C. W. Krueger, “Variation Management for Software Product Lines”, in Proceedings of the 2nd Software Product Line Conference (SPLC2), San Diego, CA, USA, pp.37-48, 2002
48. T. Ziadi, Les Lignes de Produits Logiciels (Software Product Lines), UPMC/LIP6, 2013.
49. Arnaud Hubaux, Patrick Heymans, David Benavides. 2008. Variability Modelling Challenges from the Trenches of an Open Source Product Line Re-Engineering Project, in Proceedings of the 12th Software Product Lines Conference (SPLC'08), pp. 55-64.
50. J. Greenfield, K. Short, S. Cook, and S. Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. August 2004, Wiley.
51. Rao A. S., Georgeff M. P., BDI-Agents: from Theory to Practice, Rapport n° 56, Australian Artificial Intelligence Institute, Melbourne, Australie, 1995.
52. C. Bernon, M. Cossentino, and J. Pavon. An Overview of Current Trends in European AOSE Research. Informatica (Slovenia), pages 379–390, 2005.
53. Treur J., Report of the First SIGMeeting, Contribution to Methodologies/software engineering SIG, Rapport, First SIG Meeting – AgentLink, 1998.
54. Jennings N. R., « Agent-Oriented Software Engineering », Garijo F. J., Boman M. (dir.), Multi-Agent System Engineering, 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW’99, Valence, Espagne, juin-juillet 1999, vol. 1647 de LNAI, Springer, 1999.
55. Iglesias C., Garijo M., Gonzalez J., «A Survey of Agent-Oriented Methodologies», Intelligent Agents V, ATAL’98, vol. 1555 de LNAI, Springer, 1999.
56. Kinny D., Georgeff M., Rao A., «A Methodology and Modelling Technique for Systems of BDI agents », Van de Velde W., Perram J. W. (dir.), Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a MultiAgent World, vol. 1038 de LNAI, Springer, p. 51-71, 1996.
57. Collinot A., Drogoul A., « Agent Oriented Design of a Soccer Robot Team », TokoroM. Ed., Second International Conference on Multi-Agent Systems (ICMAS’96), Nara,Japan, AAAI Press, p. 41-47, 1996.
58. Brazier F., Dunin-Keplicz B., Jennings N., Treur J., «DESIRE:Modelling Multi- Agent Systems in a Compositional Formal Framework », International Journal of Cooperative Information Systems, vol.6, n° 1, p. 64-94, World Scientific, 1997.
59. Bergenti F., Gleizes M.-P., Zambonelli F. (dir.), Methodologies and Software Engineering for Agent Systems, Kluwer Publishing, 2004.
60. Picard G., Gleizes M.-P., « The ADELFE Methodology – Designing Adaptive Cooperative Multi-Agent Systems », Bergenti F., Gleizes M.-P., Zambonelli F. (dir.), Methodologies and Software Engineering for Agent Systems, Kluwer Publishing, p. 157-176, 2004

61. Cernuzzi L., Juan T., Sterling L., Zambonelli F., « The GaiaMethodology: Basic Concepts and Extensions », Bergenti F., Gleizes M.-P., Zambonelli F. (dir.), Methodologies and Software Engineering for Agent Systems, Kluwer Publishing, p. 69-88, 2004.
62. Gomez Sanz J., Fuentes R., « Agent Oriented System Engineering with INGENIAS », Fourth Iberoamerican Workshop on Multi-Agent Systems, Iberagents'02, 2002.
63. Deloach S., Wood M., Sparkman C., « Multiagent Systems Engineering », International Journal of Software Engineering and Knowledge Engineering, vol.11, n° 3, p. 231-258, World Scientific, 2001.
64. Cossentino M., Potts C., «A CASE tool supported methodology for the design of multi-agent systems », The Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02), 2002.
65. Padgham L., Winikoff M., « Prometheus: A Methodology for Developing Intelligent Agents », Giunchiglia F., Odell J., Weiß G. (dir.), Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologne, Italie, 15 juillet 2002, Revised Papers and Invited Contributions, vol. 2585 de LNCS, Springer, p. 174-185, 2003.
66. Castro J., KolpM., Mylopoulos J., «ARequirements-driven DevelopmentMethodology », Dittrich K., Geppert A., Norrie M. (dir.), Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), vol. 2068 de LNCS, Springer, p. 108-123, 2001.
67. Demazeau Y., VOYELLES, Mémoire d'habilitation à diriger des recherches, INP Grenoble, 2001.
68. Bussman S., Position Paper onAgent-Based Software Engineering, Contribution to Methodologies/software engineering SIG, Rapport, First SIG Meeting – AgentLink, 1998.
69. Cervenka R., Trencanský I., Calisti M., Greenwood D. A. P., « AML: Agent Modeling Language Toward Industry-Grade Agent-Based Modeling», Agent Oriented Software Engineering, p. 31-46, 2004.
70. Garcia A., Pereira de Lucena C., Zambonelli F., Omicini A., Castro J. (dir.), Software Engineering for Large-Scale Multi-Agent Systems, vol. 2603 de LNCS, Springer, 2003.
71. Odell J., NodineM.H., Levy R., «AMetamodel forAgents,Roles, andGroups », Agent Oriented Software Engineering, p. 78-92, 2004.
72. A. Sturm and O. Shehory. A framework for evaluating agentoriented methodologies. In Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS- 2003), volume 3030, pages 94–109, Melbourne, Australia, 2003. Springer-Verlag.
73. K.H. Dam and M. Winikoff. Comparing agent-oriented methodologies. In AAMAS'03, editor, Fifth International Bi- Conference Workshop on Agent-Oriented Information Systems (AOIS-2003), volume 3030, pages 78–93, Melbourne, Australia, 2003. Springer-Verlag.
74. C. BERNON, M.-P. GLEIZES, G. PICARD, and P. GLIZE. The ADELFE Methodology For an Intranet System Design. In Fourth International Bi-Conference Workshop on Agent- Oriented Information Systems (AOIS-2002), volume 57, Toronto, Canada, 2002. CEUR Workshop Proceedings.
75. L. Cernuzzi, M. Cossentino, and F. Zambonelli. Process Models for Agent-based Development. Journal of Engineering Applications of Artificial Intelligence, 18, 2005.

76. D.R. Dos Santos, M. Blois Ribeiro, and R. Melo Bastos. A comparative study of multi-agent systems development methodologies. In Second Workshop on Software Engineering for Agent- Oriented Systems, pages 37–48, Florianopolis, 2006.
77. J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In In Proceedings of the third international conference on multi-agent systems., 1998.
78. Rao A. S., Georgeff M. P., BDI-Agents: from Theory to Practice, Rapport n° 56, Australian Artificial Intelligence Institute, Melbourne, Australie, 1995.
79. El Fallah Seghrouchni A., Suna A., « CLAIM : Un langage de programmation pour des agents autonomes, intelligents et mobiles », Technique et Science Informatiques, vol. 22, n° 4, p. 91-105, 2003.
80. Bordini R., Braubach L., Dastani M., El Fallah Seghrouchni A., Gomez-Sanz J., Leite J., O'Hare G., Pokahr A., Ricci A., «A Survey of Programming Languages and Platforms for Multi-Agent Systems », Informatica, An International Journal of Computing and Informatics, vol.30, n° 1, p. 33-44, 2006.
81. F. Bellifemmine, A. Poggi, and G. Rimassa. JADE - A FIPA compliant agent framework. In 4th International Conferenceand Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, 1999.
82. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex : A bdi reasoning engine, chapter of multi-agent programming. Kluwer Book, Editors : R. Bordini, M. Dastani, J. Dix and A. Seghrouchni, 2005.
83. J. Sudeikat, L. Braubach, A. Pokahr, and W. Lamersdorf. Validation of BDI Agents. In The Fifth International Workshop on Programming Multiagent Systems (PROMAS-2006), 2006.zeazeazeaz azeaze azeazeaze.
84. Pena et al 2006a Pe˜na, J., Hinchey, M.G., Cortés, A.R., Trinidad, P., 2006a. Building the core architecture of a nasa multiagent system product line. In: International Workshop on Agent Oriented Software Engineering , pp. 208–224.
85. Dehlinger, J., Lutz, R.R., 2008. Supporting requirements reuse in multi-agent system product line design and evolution. In: ICSM , pp. 207–216.
86. Nunes, I., de Lucena, C., Kulesza, U., Nunes, C., 2011. On the development of multi-agent systems product lines: a domain engineering process. In: Gleizes, M.-P., Gomez-Sanz, J. (Eds.), Agent-Oriented Software Engineering X. vol. 6038 of Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 125–139.
87. Sabatucci, L., Cossentino, M., and Gaglio, S. (2008). A semantic description for agent design patterns. In AT2AI6.
88. GonzalezPalacios, J. and Luck, M. (2004). A framework for patterns in gaia: A casestudy with organisations. In AOSE'04, pages 174–188.
89. Nunes, I., Lucena, C., Kulesza, U., and Nunes, C. (2009b). On the development of multiagent systems product lines: A domain engineering process. In AOSE'09, pages 109–120.
90. Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I., Garcia, A., and Lucena, C. J. (2009a). Comparing stability of implementation techniques for multiagent system product lines. In CSMR'09, pages 229–232.
91. Kastner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In ICSE '08, pages 311–320, USA. ACM.
92. Nunes, I., 2010. Towards a multi-agent product line development methodology. http://www.inf.puc-rio.br/ionunes/maspl/.
93. Rao, A. S. and Georgeff, M. P. (1995). BDIagents: from theory to practice. In ICMAS'95.

94. Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. ACM Trans. Softw. Eng. Methodol., 12(3):317–370.
95. Cossentino, M. (2005). From requirements to code with the passi methodology. In HendersonSellers, B. and Giorgini, P., editors, Agentoriented Methodologies, pages 79–106. Idea Group Inc.
96. da Silva, V. T., Choren, R., and de Lucena, C. J. P. (2008). Masml: a multiagent system modelling language. IJAOSE, 2(4):382–421.
97. Bauer, B., M¨uller, J. P., and Odell, J. (2001). Agent uml: a formalism for specifying multiagent software systems. In AOSE 2000, pages 91–103.
98. Odell, J. (2002). Objects and agents compared. Journal of Object Technology (JOT), 1(1):41–53.

Fin.