TECHNO 35

Publication périodique de Smals

Juin 2009

Les lignes de produits logiciels

Réutilisation et variabilité

Jean-Christophe est maître et docteur en informatique. Depuis février 2009, il est employé comme consultant à la section Recherches de Smals. Ses principaux sujets de recherche sont la réutilisation logicielle, la gestion des données de référence (Master Data Management), le développement dirigé par les modèles et les langages de modélisation.

Contact: 02 7875816 Jean-Christophe. Trigaux@smals.be

1. Introduction

À la fin du XIXe siècle, Eli Whitney et Henry Ford ont révolutionné la *production industrielle*. Cette révolution se basait notamment sur un changement radical des méthodes de travail. La production de masse est née avec l'automatisation des processus favorisant l'assemblage d'éléments interchangeables sur des lignes de production. Ces changements ont permis à la fois de mutualiser les efforts, d'augmenter la productivité et de réduire drastiquement les coûts.

Jusqu'à présent, la *production de logiciels* n'a pas suivi la même évolution. Pourtant, depuis les années 1990, différentes initiatives ont vu le jour afin de promouvoir de nouvelles méthodes de développement basées sur l'assemblage d'éléments réutilisables et sur la systématisation de la réutilisation logicielle. Dans le cadre du développement logiciel, les principales méthodes qui prônent ces principes ont été regroupées sous un même intitulé : l'approche des « *lignes de produits logiciels* ». Évidemment, la rentabilité de ces méthodes dépend principalement du nombre de similarités partagées entre les logiciels ainsi que du nombre de logiciels à produire. L'objectif est d'arriver à fournir des logiciels conformes aux exigences du client de manière plus rapide, en dépensant moins d'argent et en augmentant la qualité du logiciel.

À l'heure actuelle, cette approche a été appliquée avec succès dans plusieurs secteurs [Birk *et al.*, 2003; Pohl *et al.,* 2005]. La majorité des « success stories » [Northrop, 2002] concerne des familles de produits logiciels embarqués dans plusieurs types d'appareils électroniques tels que les téléphones mobiles, scanners médicaux, automobiles, télévisions, satellites, imprimantes,… Toutefois, ces solutions sont aussi exploitables dans d'autres contextes. Dans le monde de l'e-gouvernement, le logiciel PloneMeeting en est un exemple révélateur. En effet, PloneMeeting n'est pas qu'un simple logiciel, c'est une ligne de produits logiciels supportant l'organisation, la gestion et le suivi de réunions pour différentes organisations [Hubaux *et al.*, 2008].

> *"PloneMeeting takes it power, simplicity and adequation from the way is was built: within PloneGov, a wide community of users, business analysts and programmers joined their efforts to make a product that could easily be shared and deployed in many organizations that have common needs but also specificities."*[1]

[1] http://www.plonegov.org/software/products/plonemeeting

Ce logiciel est relativement similaire pour toutes les organisations car ses fonctionnalités de base demeurent identiques. Néanmoins, une version spécifique du logiciel doit souvent être créée pour chacune d'elle. Le logiciel de départ est donc configuré (paramétrisé) conformément aux exigences propres de l'organisation. Les procédures liées à la gestion des réunions sont généralement différentes et surtout fortement dépendantes de contraintes légales, sociales, politiques et linguistiques particulières. D'autres différences apparaissent aussi bien au niveau de la présentation (interface graphique) que du format des documents et des rapports. Toutes ces différences, aussi appelées **variabilités**, doivent être clairement explicitées afin de faciliter la configuration dudit logiciel.

L'objectif de ce Techno est, d'une part, de familiariser le lecteur avec les concepts et l'approche *des lignes de produits logiciels* et, d'autre part, de souligner l'importance de la gestion de la *variabilité* afin de maximiser la *réutilisation*. La structure de ce document se présente comme suit : dans un premier temps, la réutilisation logicielle est introduite par un bref historique des principales approches traitant ce sujet (Section 2) ; dans un second temps, nous présenterons les concepts fondamentaux d'une des approches les plus récentes : l'ingénierie des « lignes de produits logiciels » (Section 3) en insistant plus particulièrement sur la gestion de la variabilité (Section 4) ; ensuite, nous identifierons les différentes perspectives et défis inhérents à cette approche (Section 5) ; enfin, nous proposerons quelques recommandations afin de favoriser la mise en place et la gestion d'une ligne de produits logiciels (Section 6).

2. Historique

La réutilisation logicielle (*software reuse*) est une préoccupation fondamentale et récurrente depuis l'origine du génie logiciel. Son objectif principal est de réduire les coûts de développement logiciel en favorisant la réutilisation d'éléments logiciels préexistants. En pratique, toutes les méthodes de développement logiciel proposent des mécanismes et des principes facilitant la réutilisation. On en retrouve les bases aussi bien dans les travaux sur les architectures logicielles [Shaw and Garland, 1996] que dans les travaux traitant des « design patterns » [Gamma et al., 1995], « analysis patterns » [Fowler, 1996] ou « frameworks » [Fayad et al., 1999]. Au fil des années, plusieurs méthodes de développement logiciel (Table 1) se sont approprié ces principes et ont identifié :

1. les éléments logiciels à réutiliser,
2. les mécanismes de réutilisation (héritage, paramétrisation, configuration, génération, instanciation de *templates*, *plugins*, …) et
3. les phases du cycle de vie logiciel durant lesquelles ces mécanismes de réutilisation sont exploitables.

Décennie	*Méthode de développement*	*Élément réutilisable*	*Phase dans le cycle de vie*
1970	Procédurale	Modules	Coding
1980	Orienté objet	Classes	Design-Coding
1990	Orienté composant	Composants	Design-Coding
1990	Orienté agent	Agents	Design-Coding
1990	Orienté aspect	Aspects	Design-Coding
2000	Orienté service	Services	Design-Coding

Table 1: Historique de la réutilisation logicielle

Néanmoins, les résultats attendus en termes d'amélioration de la productivité et de réduction des coûts n'étaient pas suffisamment significatifs, malgré la multitude d'approches et d'outils ayant fait leur apparition sur le marché. Malheureusement, le mécanisme de réutilisation de loin le plus usité est toujours l'inépuisable « copier-coller » de morceaux de code. Cette *réutilisation* est limitée et surtout *opportuniste*, c'est-à-dire qu'elle n'a pas été planifiée au préalable. Elle se base principalement sur l'expérience du développeur qui identifie un morceau de code, une librairie, une classe ou un service qui a été développé dans un projet précédent afin de résoudre un problème relativement similaire. Dans la majorité des cas, ce type de réutilisation nécessite des adaptations conséquentes et potentiellement dommageables.

L'approche des *« lignes de produits logiciels »* vise à systématiser la réutilisation tout au long du processus de développement logiciel : de la définition des besoins jusqu'au code final et aux plans de test. Ces principes fondateurs ne sont pas nouveaux et ont été abordés dès 1968, respectivement par McIllroy [McIllroy, 1968] et Parnas [Parnas, 1976]. Néanmoins, l'ingénierie des lignes de produits logiciels [Clements and Northrop, 2001] n'a émergé qu'au début de ce siècle avec l'apparition d'une communauté très active aussi bien dans des projets européens[2] que dans des projets menés par le Software Engineering Institute[3] (SEI) aux Etats-Unis.

3. Concepts fondamentaux

3.1. Les lignes de produits logiciels

Fondamentalement, l'approche des lignes de produits logiciels propose d'adapter les principes du fordisme au développement logiciel. L'idée est de transposer les principes de fabrication de nos 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 des mêmes chaînes de montage en utilisant les mêmes châssis, les mêmes moteurs, les mêmes plans de tests, …

Les avantages d'une approche de type ligne de produits sont nombreux et, dans une certaine mesure, transposables au monde des logiciels. Les coûts et les temps de production se réduisent radicalement. Les efforts liés à l'entretien ou à la maintenance diminuent. La sécurité des véhicules est accrue car chaque pièce aura été testée au préalable dans de nombreuses situations différentes en suivant des plans de test ayant déjà fait leurs preuves. Les lignes de produits permettent également de capitaliser la connaissance. Par exemple, les ouvriers fabriquant des véhicules à Gent, à Tokyo ou à Essen 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 sur les innovations à haute valeur ajoutée et sur une plus grande personnalisation des véhicules.

---

[2] http://www.esi.es/Families/

[3] http://www.sei.cmu.edu/productlines/

D'un point de vue purement logiciel, la problématique est d'éviter de redévelopper à partir de zéro un logiciel pour chaque (nouveau) modèle de voiture. Au contraire, l'objectif est de minimiser les redéveloppements et de réutiliser massivement les travaux de développement, de tests et de maintenance réalisés lors des développements logiciels précédents. À terme, ces différents logiciels sont regroupés dans une ligne de produits logiciels dans laquelle on différencie :

- Les éléments logiciels communs à tous les membres de la ligne de produit, aussi appelés « **similarités** ». Par exemple, les systèmes de gestion du freinage, d'optimisation de la consommation ou d'ABS sont depuis quelques années intégrés dans tous les modèles de voitures.
- Les éléments logiciels variant d'un membre de la ligne de produits à l'autre, aussi appelés « **variabilités** ». Ces variabilités peuvent dépendre de différents facteurs, dont des facteurs techniques (utilisation d'une variété de matériels associés aux logiciels), commerciaux (création de plusieurs versions allant d'une version limitée à une version complète) ou culturels (logiciels destinés à plusieurs pays). Par exemple, les systèmes de GPS, d'aide à la conduite ou d'évitement d'obstacles sont spécifiques à certains modèles de voitures ou aux options sélectionnées par l'acheteur du véhicule lors de sa commande.
- Les **contraintes** existant entre les éléments logiciels. Ces contraintes devront être respectées lorsque ces éléments seront assemblés. Elles ont pour origine principalement des considérations techniques ou des règles imposées par le métier. Un exemple de contrainte technique est la dépendance forte qui existe entre le système d'évitement d'obstacles et le système de détection d'obstacles. Un exemple de contrainte métier est l'obligation légale dans certains pays de ne pas intégrer de systèmes de « cruise control » ou de systèmes d'aide à la conduite pouvant perturber l'attention du conducteur. Les taxes de mise en circulation déterminées en fonction des chevaux fiscaux nécessitent également la configuration des logiciels gérant la limitation de puissance des moteurs en fonction des législations en vigueur dans chaque pays ou région.

Les éléments logiciels sont construits à différents niveaux du cycle de vie logiciel et comprennent notamment des exigences, des schémas de conception (des algorithmes à l'architecture), du code, des programmes de tests, de maintenance, etc. Au final, le développement d'un nouveau logiciel se limite principalement à combiner ces éléments logiciels en respectant les contraintes techniques et métier. En théorie, d'une part, les similarités devraient être directement réutilisées pour tous les membres de la ligne de produits et, d'autre part, les variabilités devraient être sélectionnées et réutilisées en fonction des besoins associés au logiciel. Dans la réalité, l'exercice n'est pas aussi trivial. Généralement, tous les logiciels, même ceux appartenant à une ligne de produits logiciels, possèdent des fonctionnalités qui leur sont complètement spécifiques. Inévitablement, le développement de ces fonctionnalités, aussi appelées « **spécificités** », doit être réalisé. De plus, leur intégration avec les éléments logiciels réutilisables doit être effectuée avec la plus grande précaution afin d'éviter tout conflit ayant potentiellement un impact sur l'ensemble de la ligne de produits.

Les constructeurs automobiles tels que Daimler ou General Motors ne sont pas les seuls à être confrontés à ce type de problématique. Différentes initiatives ont obtenu des résultats probants que ce soit dans le secteur de la téléphonie mobile (Nokia, Ericsson), de l'aéronautique (Boeing, Airbus), des systèmes médicaux (Philips), ou de l'impression (HP). À titre d'exemple, chaque fournisseur de téléphones mobiles lance sur le marché une vingtaine de nouveaux modèles par an. Ces modèles se distinguent par les standards de communication supportés, les différentes langues sélectionnables, les jeux proposés, la taille des écrans,… Dans un secteur aussi évolutif et compétitif, avec une aussi grande diversité

de produits, les fournisseurs veulent à tout prix éviter de redévelopper une fonctionnalité déjà existante dans un modèle précédent. Leur cauchemar est de devoir redévelopper quasiment l'intégralité de leurs logiciels à cause d'un simple changement de taille d'écran.

En résumé, une ligne de produits logiciels regroupe un ensemble de produits appartenant à un même domaine et caractérisé par des éléments logiciels très proches. Un domaine est un secteur de métier ou de technologies ou de connaissances caractérisé par un ensemble de concepts et de terminologies admises par les utilisateurs de ce secteur. Une ligne de produits a pour but la mise en commun des travaux de développement, de tests et de maintenance de ces éléments logiciels communs de façon à réduire les coûts de production et de maintenance, réduire les temps de production et améliorer la qualité.

> ***Définition***
>
> Une **ligne de produits logiciels** est un ensemble de systèmes (1) partageant un ensemble de fonctionnalités communes, (2) satisfaisant des besoins spécifiques pour un domaine particulier et (3) développés de manière contrôlée à partir d'un ensemble commun d'éléments réutilisables. [Clements and Northrop, 2001]

L'objectif est donc de rationaliser le processus de développement d'entreprises développant des systèmes (logiciels et, éventuellement, matériels) fortement similaires. Cette rationalisation est soutenue par un processus d'ingénierie adapté au développement des lignes de produits logiciels. Deux points d'attention essentiels sont à considérer : (1) l'identification des similarités et des variabilités existant dans la ligne de produits et (2) la structuration et le développement d'une base d'éléments réutilisables.

3.2. L'ingénierie des lignes de produits logiciels

Au même titre que le développement logiciel, le développement d'une ligne de produits logiciels est une ingénierie. En conséquence, comme toute ingénierie, la construction, l'utilisation et la maintenance d'une ligne de produits doivent suivre une approche systématique, disciplinée et quantifiable. Du point de vue de l'ingénierie logicielle dite « classique », un processus de développement itératif est préconisé. Il se compose généralement de quatre étapes successives (Figure 1) :

1. ***Application Requirements***. La définition des besoins.
2. ***Application Design***. La spécification de la solution permettant de satisfaire ces besoins.
3. ***Application Coding***. L'implémentation de cette solution.
4. ***Application Testing***. La phase de test de la solution et son évaluation par rapport aux besoins initiaux.

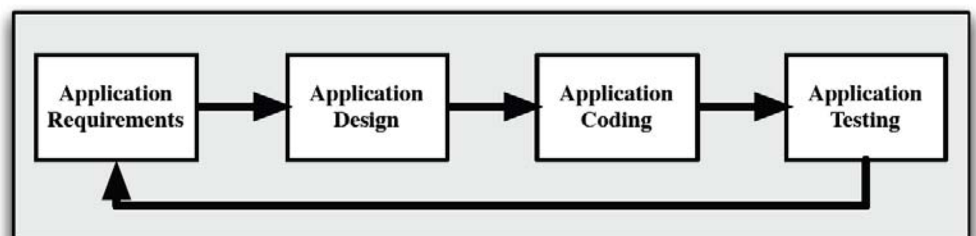

Figure 1: Processus de développement logiciel classique

Du point de vue de l'**ingénierie des lignes de produits logiciels**, le principal changement est le passage d'une *réutilisation opportuniste* et non contrôlée à une *réutilisation systématique* et *transversale*. La réutilisation est dite *systématique* lorsqu'elle est contrôlée et planifiée à l'avance pour un ensemble de logiciels. Néanmoins, la réutilisation ne doit pas se limiter à la phase d'implémentation (coding) mais doit devenir *transversale* à tout le processus de développement logiciel : de la spécification des besoins, en passant par la définition de l'architecture jusqu'aux phases de tests.

Dans cette optique, l'ingénierie des lignes de produits logiciels se différencie significativement de l'ingénierie logicielle dite « classique ». Elle regroupe les activités de développement liées non pas à un logiciel spécifique mais à un ensemble de logiciels appartenant à un domaine particulier. Le processus de développement d'une ligne de produits logiciels est traditionnellement divisé en deux processus distincts : Domain Engineering et Application Engineering (Figure 2).

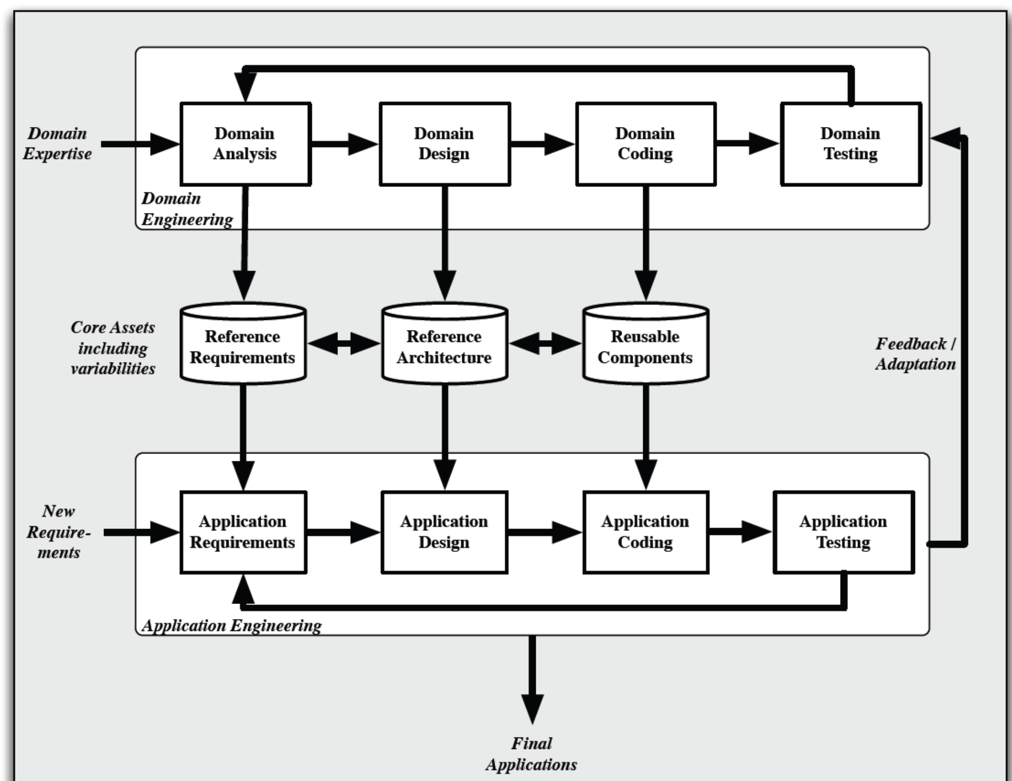

Figure 2: Processus de développement d'une ligne de produits logiciels

Le premier processus « **Domain Engineering** » ou « *Development for reuse* » a pour objectif de construire la base d'éléments réutilisables (« *Core Assets* »). La principale originalité de ce processus est l'identification des membres de la ligne de produits, des similarités existant entre eux et des variabilités permettant de les différencier. Les principaux outputs de ce processus sont (Figure 2) :

- ***Scope***. La définition du *scope* de la ligne de produits consiste souvent en une énumération de noms de produits. La délimitation de ce scope est très importante car un scope trop vaste a tendance à accroître considérablement la complexité et un scope trop restreint ne justifiera pas les investissements consentis pour le développement et la maintenance des éléments réutilisables.

- ***Reference Requirements***. La définition des *exigences de références* comprenant les similarités, les variabilités ainsi que les contraintes existant entres elles.
- ***Reusable Components***. Un ensemble de *composants réutilisables* implémentant chaque similarités et variabilités.
- ***Reference Architecture***. Une *architecture de référence* décrivant comment ces composants doivent être combinés afin de construire le produit final.
- ***Production Plan***. Des *plans de production* documentant la manière dont l'architecture de référence et les composants réutilisables doivent être utilisés afin de produire efficacement les logiciels finaux.

Le second processus « **Application Engineering** » ou « *Development with reuse* » a pour objectif de dériver de manière semi-automatique les logiciels finaux à partir des éléments réutilisables. Ce processus s'aligne parfaitement sur le processus de développement classique (Figure 1). Cependant, chaque étape est facilitée par la réutilisation des outputs du processus précédent (Figure 2) :

- ***Application Requirements***. Durant l'étape de définition des besoins du logiciel final, la réutilisation des exigences de référence permet d'une part de sélectionner les variabilités et d'autre part de se focaliser sur les besoins réellement spécifiques du nouveau logiciel.
- ***Application Design***. Concevoir l'architecture du logiciel consiste à configurer l'architecture de référence suivant les variabilités sélectionnées et éventuellement à l'adapter en fonction des besoins spécifiques.
- ***Application Coding***. L'implémentation se limite principalement à développer de nouveaux composants ou à étendre des composants de référence afin de prendre en compte les besoins spécifiques.
- ***Application Testing.*** De nombreux tests ont déjà été spécifiés et exécutés pour les composants réutilisables. L'objectif principal de cette phase est de s'assurer que les interactions entre les composants réutilisables et les développements spécifiques ne génèrent pas de comportements inattendus.

Ces deux processus doivent parfaitement s'articuler car ils sont fortement interdépendants. De plus, toutes les lignes de produits logiciels doivent évoluer pour répondre aux besoins du marché. Le développement d'un nouveau produit nécessite fréquemment la mise à jour des éléments réutilisables, mise à jour qui impactera l'ensemble de la ligne de produits logiciels. La gestion efficace de ces évolutions n'est envisageable que grâce à des *liens de traçabilité* reliant ces deux processus ainsi que les éléments réutilisables entre eux (Figure 2).

La complexité de ces processus n'est certainement pas négligeable. Elle dépend principalement du nombre de produits à gérer (scope) et surtout des différences qui existent entre eux (variabilité). La gestion efficace et correcte de la variabilité est un facteur de réussite déterminant lors de l'utilisation d'un processus de ce type.

4. La variabilité

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 ligne de produits logiciels détermine un contexte particulier. Chaque produit se compose d'un ensemble d'éléments logiciels représentés par des « ***features*** ». Chaque *feature* regroupe les exigences devant être satisfaites par les produits logiciels finaux intégrant cette *feature*. L'objectif d'une *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 obtenir des *features* terminales. Idéalement, chaque *feature* terminale est associée à un *élément logiciel réutilisable* (composant, service, …) 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 (Figure 1). En effet, les produits peuvent se différencier à la fois par les services qu'ils offrent, les structures de données qu'ils manipulent, les technologies qu'ils utilisent, les flux de contrôles qu'ils doivent respecter, les interactions avec leur propre environnement, leurs objectifs de qualité, de sécurité, … Il est donc primordial de *gérer* efficacement la variabilité (Section 4.1) et plus particulièrement de l'*expliciter* et de la *documenter* (Section 4.2).

4.1. La gestion de la variabilité

La gestion de la variabilité joue donc un rôle essentiel afin de déterminer dans quel contexte, sous quelles conditions et comment une *feature* (et donc l'élément logiciel associé) 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, paramétrisation, configuration, génération, template instanciation, plugins, …) 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. Deux outils très compétitifs sur le marché sont : « Gears »[4] et « pure::variants »[5]. Ces outils offrent principalement trois types de fonctionnalités :

- Des environnements de développement spécifiques aux lignes de produits logiciels.

[4] http://www.biglever.com/)

[5] http://www.pure-systems.com/Variant_Management.49.0.html

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

Néanmoins, même avec les meilleurs outils de gestion de la variabilité, si elle n'a pas été clairement explicitée et documentée dès le départ, sa gestion est clairement impossible.

4.2. La modélisation de la variabilité

La modélisation de la variabilité est une technique utilisée afin d'une part de **documenter** la variabilité et d'autre part de **raisonner** sur la variabilité. Ses principaux objectifs sont (1) d'expliciter la variabilité dès les premières phases du projet et (2) de diminuer la complexité liée à la gestion de la variabilité tout au long du processus de développement. De nombreux langages permettent de décrire graphiquement la variabilité. Le premier langage spécifiquement dédié à la modélisation de la variabilité a été proposé par Kang [Kang *et al.*, 1990]. Ce langage de modélisation définit différentes constructions (graphiques et textuelles) afin de modéliser les *features* et les relations existant entre elles au moyen d'un modèle particulier appelé « **Feature Diagram** ». Ce langage se voulait avant tout minimal et simple d'utilisation en comparaison avec d'autres langages de modélisation plus complexes tels que UML[6] ou BPMN[7].

Prenons l'exemple d'une ligne de produits de téléphones mobiles. Il suffit d'aller sur le site Internet de n'importe quel constructeur pour identifier les produits et les *features* propres à chaque produit. Cette information est souvent décrite par produit sous forme tabulaire. D'autres formes sont possibles. Dans notre cas, nous utilisons un Feature Diagram (Figure 3) afin de modéliser de manière plus compacte cette ligne de produits ainsi que sa variabilité.

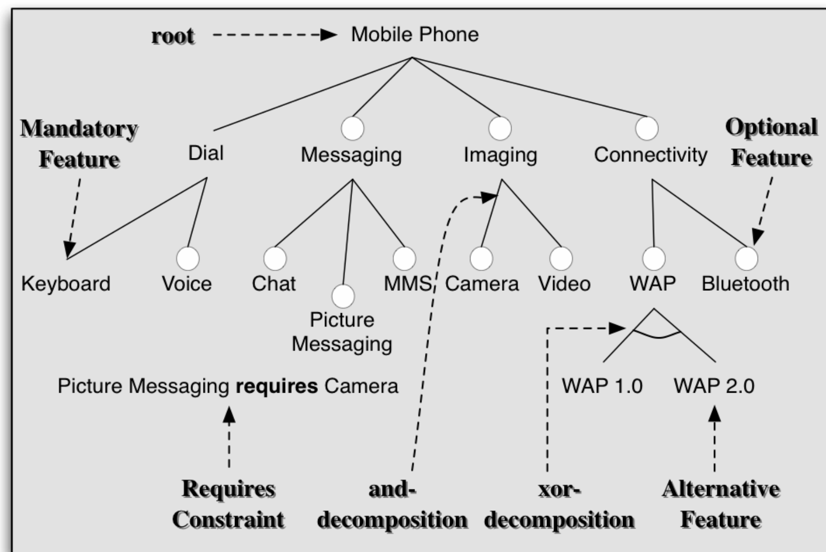

Figure 3: Feature Diagram: Mobile Phone

[6] http://www.uml.org/

[7] http://www.bpmn.org/

Un Feature Diagram se présente généralement sous la forme d'un arbre composé de nœuds reliés par des arêtes. Chaque nœud correspond à une *feature* et chaque arête correspond à une relation entre deux *features*. Différents types de *features* existent :

- La **root**. Un Feature Diagram est caractérisé et identifié par une *feature* spécifique appelée la racine (*root*). Cette root détermine le point d'entrée du diagramme qui est l'unique nœud ne possédant pas de parent. Dans notre exemple (Figure 3), la root est la *feature* « Mobile Phone » située au sommet de l'arbre et qui représente la ligne de produits.
- Les **optional *features***. Si une *feature* est optionnelle (*optional*) alors si un de ses parents est sélectionné elle ne l'est pas nécessairement. Graphiquement, une *feature* optionnelle est décorée par un cercle vide. Dans notre exemple (Figure 3), la *feature* « Bluetooth » est considérée comme optionnelle.
- Les **mandatory *features***. Si une *feature* est obligatoire (*mandatory*) alors si un de ses parents est sélectionné elle est dès lors aussi sélectionnée. La *root* est par définition toujours obligatoire. Graphiquement, une *feature* obligatoire n'est pas décorée par un cercle vide. Dans notre exemple (Figure 3), la *feature* « Keyboard » est considérée comme obligatoire.

Ces features sont reliées par différents types de relations:

- Les **décompositions**. Les *features* peuvent être décomposées en sous-*features* suivant différents types de décompositions. Ces décompositions définissent des contraintes entre des *features* partageant le même parent.
  - *And*-decomposition. Cette décomposition signifie que si le parent est sélectionné alors ses enfants le sont aussi. Dans notre exemple (Figure 3), la *feature* « Imaging » est décomposée en deux sous-*features* : « Camera » et « Video ». Cette décomposition est de type « and-decomposition » représentée graphiquement par des arêtes ayant pour origine la *feature* parent (« Imaging ») et pour destination les *features* enfants (« Camera », « Video »).
  - *Xor*-decomposition. Cette décomposition signifie que si le parent est sélectionné alors un et un seul de ses enfants peut être sélectionné. Dans notre exemple (Figure 3), la *feature* « WAP » est décomposée en deux sous-*features* : « WAP 1.0 » et « WAP 2.0 ». Cette décomposition est de type « xor-decomposition » représentée graphiquement par des arêtes reliées par un arc de cercle. Chaque arête ayant pour origine la *feature* parent (« Imaging ») et pour destination ses *features* enfants (« WAP 1.0 », « WAP 2.0 »).
  - *Or*-decomposition. Cette décomposition signifie que si le parent est sélectionné alors au moins un de ses enfants peut être sélectionné.
- Les **contraintes**. Dans certains cas, on veut pouvoir exprimer des contraintes entre des *features* qui ne partagent pas le même parent. Ces contraintes peuvent être établies entre toutes les *features* appartenant au même Feature Diagram. C'est pourquoi elles sont représentées de manière textuelle afin d'éviter tout surchargement réduisant la lisibilité du modèle. Les deux contraintes les plus utilisées sont :
  - *Requires*. La contrainte *requires* entre deux *features* (*f1 requires f2*) permet d'exprimer que si la *feature f1* est sélectionnée alors *f2* doit nécessairement l'être, mais pas inversement. Dans notre exemple (Figure 3), la sélection de la *feature* « Picture Messaging » impose la sélection de la *feature* « Camera ».

- *Excludes.* La contrainte *excludes* entre deux *features* (*f1 excludes f2*) permet d'exprimer que si une des deux *features* est sélectionnée alors l'autre *feature* ne peut pas l'être en même temps pour le même produit.

En pratique, un Feature Diagram décrit la majorité des produits développés par l'entreprise et comprend des milliers de *features* et de contraintes. Ce modèle devient donc stratégique pour l'entreprise. Néanmoins, sa taille engendre des problèmes d'évolutivité et de complexité. À partir d'un Feature Diagram réel, c'est-à-dire un Feature Diagram contenant des centaines de features et de contraintes entre features, il est pratiquement impossible de manuellement (1) lister tous les produits qui peuvent être générés ou (2) garantir que toutes les contraintes qui ont été exprimées ne sont pas conflictuelles. Une solution est donc de faire comprendre ces modèles aux ordinateurs, de leur donner la possibilité de les interpréter et ensuite d'utiliser leur puissance de calcul afin de **raisonner** dessus. Les outils informatiques supportant les langages de Feature Diagrams sont essentiellement basés sur des SAT-Solvers ou sur des systèmes de résolution de contraintes. Ils nous permettent notamment de :

- Vérifier que les Feature Diagrams produits sont corrects par rapport aux conventions graphiques et aux méta-modèles définis pour le langage.
- Lister l'ensemble des produits respectant les contraintes imposées par le Feature Diagram et appartenant donc à la ligne de produits modélisée par celui-ci.
- Vérifier qu'il existe au moins un produit qui satisfasse aux contraintes.
- Vérifier qu'un produit donné satisfait les contraintes.
- Identifier des *features* mortes, c'est-à-dire des *features* qui n'apparaissent plus dans aucun produit. Dans certains cas, il est sans doute nécessaire de relaxer certaines contraintes ou d'éliminer certaines *features* mortes.
- Identifier des *features* communes, c'est-à-dire des *features* qui apparaissent dans tous les produits. Dans certains cas, il est sans doute nécessaire d'agréger plusieurs *features* communes.
- Générer du code, configurer un produit final ou assembler semi-automatiquement des composants ou des services.

Dans notre exemple (Figure 3), l'utilisation de ces outils révèle que la ligne de produits est composée de 288 produits, malgré qu'elle ne contienne que 16 *features*. Elle ne possède pas de *features* mortes mais trois *features* communes (« Keyboard », « Dial » et « Mobile Phone »). Avec l'introduction d'une nouvelle contrainte fictive de la forme « Dial » *excludes* « Chat », nous remarquons l'apparition d'une *feature* morte (« Chat ») et la diminution du nombre de produits par un facteur deux.

Actuellement, la plupart des outils de modélisation de la variabilité sont principalement dédiés à la recherche. Cependant, des avancées significatives ont été réalisées notamment avec des **outils de modélisation**[8] basés sur Eclipse et EMF et des **outils d'analyse**[9] des Feature Diagrams.

[8] http://gsd.uwaterloo.ca/projects/fmp-plugin/

[9] http://www.isa.us.es/fama/

5. Les perspectives et défis

Malgré leurs nombreux avantages, les lignes de produits logiciels ne constituent pas la solution miracle à tous les problèmes de productivité et de réutilisation. Dans cette section, nous distinguons les différentes perspectives et défis inhérents à cette approche.

5.1. Les perspectives

Les principaux avantages associés à une approche de type ligne de produits logiciels concernent principalement :

- **L'augmentation de la productivité[10] et la diminution des coûts de développement et de maintenance des produits finaux.** Les développeurs ont la possibilité de se baser sur des exigences uniformisées et cohérentes et de réutiliser des composants spécifiquement conçus afin de maximiser la réutilisation. De plus, lorsque les produits sont mis en production leur maintenance est beaucoup mieux maîtrisée. À titre d'exemple, des résultats significatifs directement liés à l'application des principes des lignes de produits logiciels ont été obtenus dans différents secteurs [Baas et al., 2003] :
  - Nokia a constaté une augmentation du nombre de modèles de téléphones mobiles produits et introduits sur le marché de 4 à plus ou moins 25-30 modèles par an.
  - Cummins Inc. a constaté que le temps de développement des logiciels contrôlant ses moteurs diesel a diminué de un an à une semaine.
  - Motorola a constaté, pour ses beepers, une augmentation de la productivité de sa ligne de produits de 400%.
  - Hewlett-Packard a constaté, pour ses logiciels d'impression, une diminution des temps de production d'un facteur sept et une augmentation de sa productivité d'un facteur six.
- **L'augmentation de la qualité des produits finaux.** Les développeurs réutilisent des composants de meilleure qualité en suivant des standards de programmation communs à toute la ligne de produits. On évite ainsi la duplication de code et on favorise un meilleur contrôle de la qualité logicielle.

5.2. Les défis

Une ligne de produits logiciels ainsi que les bénéfices qui y sont associés n'apparaissent pas comme par magie. Des efforts conséquents doivent être consentis et des défis importants relevés:

- L'investissement initial, nécessaire à la mise en place d'une telle approche, est relativement important et, malheureusement, le retour sur investissement n'est pas immédiat. Dans la majorité des cas, ces deux facteurs poussent les petites et moyennes entreprises à abandonner ce type d'approche.
  - L'investissement initial est considérable car, même si le projet peut être conduit de manière itérative, il est toujours nécessaire d'avoir une vue

[10] Différentes approches et techniques existent afin d'augmenter la productivité. Pour plus d'information, veuillez consulter le rapport de recherche : Verhoging van de productiviteit in Software Engineering, Groebbens Adelbert, Oktober 2007, Smals, sectie Onderzoek.

d'ensemble non pas sur un logiciel mais sur un ensemble de logiciels incluant à la fois les logiciels passés, présents et à venir.

- Le retour sur investissement n'est certainement pas immédiat et la rentabilité ne sera vraisemblablement atteinte qu'à moyen ou long terme. Généralement, le coût des premiers logiciels développés à partir de la ligne de produits sera même plus élevé que dans la situation précédente. Il faudra attendre la production d'un certain nombre de logiciels (Break even point) avant de pouvoir générer de réels bénéfices à moyen ou long terme (Figure 4). Préalablement à la mise en place de la ligne de produits logiciels, les coûts et les bénéfices sont pratiquement impossibles à estimer de manière fine. Ce qui a souvent pour conséquence d'empêcher les décideurs d'évaluer les budgets nécessaires et d'établir des plannings raisonnables.

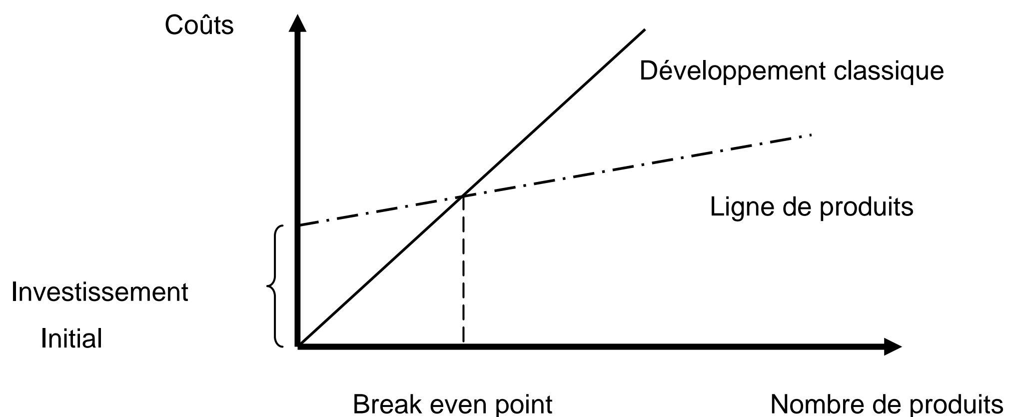

Figure 4: Rentabilité des lignes de produits logiciels [Pohl et al., 2005]

- La manière de concevoir et de développer une ligne de produits logiciels implique une évolution radicale des mentalités et des changements organisationnels importants. Chaque acteur doit être convaincu des bénéfices potentiels à la fois pour lui-même et pour l'organisation dans son ensemble. Force est de constater que ce changement de mentalité est généralement plus facile à entreprendre dans des entreprises développant des logiciels embarqués dans des appareils électroniques. En effet, ces entreprises appliquent depuis des années les principes des lignes de produits au niveau de la conception physique de leurs appareils qui résultent de la composition de différents modules électroniques réutilisables. En revanche, l'acceptation et l'application des principes des lignes de produits logiciels sont plus délicates dans des entreprises développant des systèmes d'information plus classiques. Ces difficultés se justifient par, d'une part, les exigences clients beaucoup plus contraignantes et, d'autre part, le principe de réutilisation qui n'est pas encore entré dans les mœurs.

- Une attention toute particulière doit être portée sur l'évolution et la gestion du changement dans les lignes de produits logiciels. Gérer l'évolution d'un logiciel est déjà complexe. Gérer l'évolution d'un ensemble de logiciels dont la cohérence doit être préservée est extrêmement complexe. Tout changement portant sur un élément logiciel partagé par plusieurs produits finaux doit être contrôlé méticuleusement. Tous les impacts potentiels sur les produits finaux et sur leurs éléments logiciels respectifs doivent être envisagés avant de mettre ces changements en production.

De ce fait, la flexibilité d'une ligne de produits logiciels est souvent réduite afin d'améliorer son efficacité.

- La quantité d'information à rassembler afin d'établir une nouvelle ligne de produits est souvent conséquente. La difficulté consiste à identifier les personnes et les documents permettant de recueillir suffisamment d'information afin de comparer ces produits, d'identifier leurs similarités et leurs variabilités. Dans la majorité des cas, il est nécessaire (1) de clarifier et de compléter la documentation des produits existants, (2) d'entamer une réingénierie et (3) d'anticiper les différentes avancées et innovations envisageables pour les nouvelles mises en production.

6. Recommandations

La mise en place et la gestion d'une ligne de produits logiciels sont des tâches complexes confrontées à des difficultés tant organisationnelles que techniques. Dans cette section, nous proposons quelques recommandations et bonnes pratiques afin de faciliter ces tâches et ainsi d'augmenter les chances de réussite d'un projet de cette envergure.

- **Think Big, Start Small.** La mise en place d'une ligne de produits logiciels doit être réalisée de manière progressive et itérative. La première étape consiste à définir le « *scope* » et l'« architecture » générale de la ligne de produits. Tous les produits contenus dans le scope ne pourront pas être intégrés en même temps dans la nouvelle ligne de produits. Le choix du premier produit à (re)développer est donc primordial. Nous conseillons de tenir compte des critères de sélection suivants :
  - Choisir un produit « standard » contenant peu de variabilités.
  - S'assurer de la maîtrise et de la qualité de ce produit.
  - S'assurer de l'adéquation de ce produit avec l'architecture définie pour la ligne de produits.

  Un seul produit ne constitue pas une véritable ligne de produits. L'étape suivante consiste donc à intégrer les autres produits en respectant ou en faisant évoluer l'architecture de départ. L'objectif est de compléter petit à petit le produit « standard » avec de nouvelles variabilités. Chaque variabilité pouvant être ajoutée ou retirée de manière relativement indépendante. On débute donc avec une ligne de produit minimale qui évolue et se complexifie au fur et à mesure.

- **Pensez en terme de variabilité.** L'un des principaux facteurs de succès d'une ligne de produits logiciels est la gestion de sa variabilité. Trop souvent, la réutilisation a été synonyme d'échec dans les projets de développements informatiques car la dimension variabilité était complètement éludée. La variabilité doit être explicitée et documentée dès les premières phases du processus de développement. De la même manière que des modèles et des outils sont utilisés pour analyser, communiquer et valider des processus métiers (BPM), il est primordial de modéliser la variabilité (Section 4.2) et de maintenir à jour ces modèles en utilisant des outils adéquats.

- **Développez vos core assets et assemblez vos produits.** La gestion d'une ligne de produits logiciels doit distinguer trois activités essentielles (Figure 5) :
  - Le *développement des core assets*. Une équipe prend en charge le développement et la maintenance de la base d'éléments réutilisables.

Son objectif principal est d'assurer sa cohérence et de favoriser sa réutilisation pour le développement des produits finaux.

- L'*assemblage des produits finaux*. Une autre équipe prend en charge le développement et la maintenance des produits finaux. Sa principale tâche est de sélectionner et d'assembler correctement les éléments réutilisables. Dans une certaine mesure, des développements spécifiques peuvent encore être nécessaires.
- La *gestion de la ligne de produits logiciels*. La gestion d'une ligne de produits logiciels nécessite une implication forte du management. L'objectif principal est d'assurer la synchronisation et la bonne réalisation des activités de développement des core assets et des produits finaux. Les objectifs de ceux-ci pouvant être divergents.

Figure 5: La gestion d'une ligne de produits logiciels [Clements and Northrop, 2001]

- **Communiquez.** La mise en place et la gestion d'une ligne de produits logiciels transforment une grande partie de l'organisation. Chaque acteur doit avoir pris connaissance des impacts sur ses méthodes de travail. Une collaboration étroite est nécessaire afin de mutualiser les efforts et d'assurer la qualité, la cohérence et la réutilisabilité des core assets transversalement à toute la ligne de produits.

7. Conclusion

L'approche des lignes de produits logiciels est très prometteuse en terme d'économie d'échelle et de réduction drastique des coûts de développement logiciel. Malheureusement, elle était principalement réservée à certains domaines possédant déjà une grande expérience des lignes de produits « non logiciels » tels que les télécoms ou la construction automobile. Récemment, ces principes ont aussi été appliqués dans d'autres domaines tels que l'e-gouvernement [Hubaux *et al.*, 2008], l'e-commerce [Greenfield *et al.*, 2004] ou l'e-banking [Clements and Northrop, 2001]. Néanmoins, les principaux freins à la mise en place de ces

principes demeurent : le manque de gestion de la variabilité existant entre les différents produits logiciels, le nombre insuffisant de produits logiciels partageant suffisamment de similarités, le travail de réingénierie à entreprendre et les problèmes organisationnels liés à la résistance aux changements.

8. Bibliographie

[Baas et al., 2003] Baas, L., Clements, P. C., Kazman, R., 2003. Software Architecture in Practice, Second Edition. SEI Series in Software Engineering. Addison-Wesley.

[Birk et al., 2003] Birk, Andreas, Heller, Gerald, John, Isabel, Schmid, Klaus, von der Massen, Thomas, Muller, Klaus. December 2003. Product Line Engineering, the State of the Practice. Software. IEEE Volume 20, Issue 6, pp. 52-60.

[Clements and Northrop, 2001] Clements, P. C., Northrop, L., Aug. 2001. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

[Cohen, 2002] Cohen, S., Sep. 2002. Product Line State of the Practice Report. Technical Note CMU/SEI-2002-TN-017, Software Engineering Institute, Carnegie Mellon University.

[Fayad et al., 1999] Mohamed Fayad, Ralph Johnson. November 1999. Domain Specific Application Frameworks, John Wiley & Sons Inc.

[Fowler, 1996] Martin Fowler. Oct. 1996. Analysis Patterns. Addison-Wesley.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. January 1995. Design Patterns. Addison-Wesley.

[Greenfield et al., 2004] J. Greenfield, K. Short, S. Cook, and S. Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. August 2004, Wiley.

[Hubaux et al., 2008] 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.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., Nov. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

[McIllroy, 1968] McIlroy, D., 1968. Mass-Produced Software Components. In: Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany. pp. 88-98.

[Northrop, 2002] Northrop, Linda M. August 2002. SEI's Software Product Line Tenets. Software, IEEE Volume 19, Issue 4, pp. 32-40.

[Parnas, 1976] Parnas, D., Mar. 1976. On the Design and Development of Program Families. IEEE Transactions on Software Engineering SE-2 (1), 1-9.

[Pohl *et al.*, 2005] Pohl, K., Bockle, G., van der Linden, F., July 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer.

[Shaw and Garland, 1996] Mary Shaw. 1996. Software Architectures, David Garland.