

N° d'ordre : 3623

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LILLE 1
MENTION INFORMATIQUE

Stéphane BONNET

Une démarche dirigée par les modèles pour la personnalisation des applications embarquées dans les cartes à puce

Sous la direction du Professeur Jean-Marc GEIB

Soutenue publiquement le 26 Mai 2005

Jury :

Président	:	Pierre Paradinas, Professeur	CNAM, Paris
Rapporteurs	:	Jean-Marc Jézéquel, Professeur	IRISA, Rennes
		Guy Bernard, Professeur	INT, Evry
Directeur	:	Jean-Marc Geib, Professeur	LIFL, Université de Lille 1
Co-encadrants	:	Olivier Potonniée, Ingénieur de recherche	GEMPLUS, La Ciotat
		Raphaël Marvie, Maître de conférences	LIFL, Université de Lille 1

Laboratoire d'Informatique Fondamentale de Lille
UMR CNRS 8022, Bâtiment M3,
59655 Villeneuve d'Ascq Cédex, France.

Gemplus, Systems Research Labs
La Vigie, Avenue du Jujubier, Z.I Athelia IV,
13705 La Ciotat Cédex, France.

À mon père

Remerciements

Je tiens à remercier en premier lieu les personnes qui sont à l'origine de cette thèse. Un immense merci donc à Monsieur le Professeur Jean-Marc Geib à la fois pour m'avoir fait l'honneur d'accepter la direction de ma thèse et pour être parvenu ensuite, à chacune de nos rencontres, à donner un nouveau souffle à mes recherches. Un grand merci également à Monsieur le Professeur Pierre Paradinas, pour m'avoir fait confiance en me permettant d'intégrer le laboratoire de recherche de Gemplus. Exerçant aujourd'hui de nouvelles fonctions dans un contexte différent, il a confirmé son intérêt pour mes travaux en acceptant de présider le Jury de ma thèse.

Je remercie chaleureusement Messieurs les Professeurs Jean-Marc Jézéquel et Guy Bernard pour avoir accepté de rapporter cette thèse. Leurs commentaires particulièrement avisés m'ont aidé à prendre davantage de recul sur mes travaux et mes résultats.

Je tiens à adresser à Olivier Potonniée et Raphaël Marvie mes plus vifs remerciements pour leur importante contribution à l'encadrement de mes recherches. La disponibilité quotidienne d'Olivier et l'intérêt constant qu'il a porté à mon travail ont permis la mise en œuvre d'un partage de connaissances, récurrent et mutuel, source primordiale de motivation. Raphaël a su insuffler à mes recherches et à mes articles un caractère académique et une rigueur qui leur faisaient parfois défaut. Leurs qualités humaines et leur sympathie font d'Olivier et de Raphaël deux personnes avec lesquelles j'ai pris un grand plaisir à partager un bout de chemin. Je n'oublie pas non plus que mon document de thèse doit énormément à leurs relectures approfondies.

En remerciant Louis Grégoire et David Naccache, je tiens à exprimer ma gratitude envers Gemplus et plus particulièrement envers tous les membres du laboratoire de recherche. Je ne citerai pas ici les noms de tous les 'gemplussiens' et 'gemplussiennes' que j'ai pris du plaisir à côtoyer, aussi bien sur mon lieu de travail qu'au dehors, et envers qui j'ai beaucoup de reconnaissance. J'ai néanmoins une pensée spéciale pour Antoine. Le parallélisme de nos expériences a souvent été réconfortant.

Ces remerciements ne seraient pas complets sans une pensée pour toutes les personnes que j'ai croisées au cours de ces dernières années et qui m'ont encouragé d'une manière ou d'une autre.

Enfin, je tiens à remercier mes amis proches, ma famille, mon frère Xavier, et surtout, mes parents. Cette thèse, je la dois à leur soutien indéfectible et à l'exemple qu'ils m'ont donné. Merci.

Table des matières

Table des figures

Première partie

Contexte

Chapitre 1

Introduction

La carte à puce est un support informatique dont la taille minimale et le niveau élevé de sécurité sont deux des caractéristiques principales. L'aspect personnel inhérent à la carte à puce est très fort. Chaque carte est la représentante d'un individu particulier, son porteur, au sein d'une infrastructure informatique. Si elle joue notamment un rôle clef dans la téléphonie mobile ou les systèmes bancaires de paiement, la carte à puce est de plus en plus utilisée dans des domaines émergents comme ceux de la santé ou de l'identité. Les évolutions matérielles et logicielles des systèmes des cartes à puce favorisent le développement d'applications embarquées de plus en plus évoluées, et par conséquent, permettent d'envisager à court terme une diversification des utilisations de la carte.

Les plates-formes logicielles des cartes à puce de prochaine génération offrent des possibilités accrues en terme de programmation, de coopération entre applications embarquées et d'intégration dans le reste des infrastructures informatiques. Cette évolution du logiciel embarqué motive une réflexion sur la modernisation du processus de production des cartes. La complexité de ce processus réside essentiellement dans la mise en œuvre des niveaux de configuration successifs liés au domaine d'application, à l'organisation émettrice ou encore au porteur final. Dans ce contexte, nous nous intéressons à la personnalisation des applications embarquées, c'est-à-dire à l'adaptation du contenu applicatif de la carte à son utilisateur final.

La personnalisation est la dernière étape de la chaîne de fabrication des cartes. Aujourd'hui, elle consiste typiquement à charger dans la carte un ensemble de données

personnelles telles que des informations d'état civil ou des numéros de comptes bancaires. L'intégration améliorée des cartes de prochaine génération dans les infrastructures informatiques traditionnelles ainsi que la diversification prévisible des applications embarquées motivent un enrichissement du processus de personnalisation. Ce dernier doit par exemple supporter une adaptation à la fois du contenu et du comportement des logiciels embarqués au profil du porteur final. Cette évolution de la personnalisation s'inscrit dans une tendance plus générale de l'informatique moderne qui voit ses acteurs évoluer progressivement vers une fourniture ou une production sur mesure de services, de contenus ou de logiciels.

Les objectifs de nos travaux de recherche sont de concevoir et expérimenter, d'une part, des mécanismes permettant la mise en œuvre d'une personnalisation plus riche des applications embarquées, et d'autre part, une solution pour l'optimisation du processus industriel de personnalisation. Notre travail consiste donc à préciser notre vision de la personnalisation, élaborer une méthodologie pour la réaliser puis mettre en œuvre son automatisation.

Notre proposition consiste à nous appuyer sur les techniques récentes destinées à l'automatisation de la construction de logiciels. Sur la base du développement génératif et des approches dirigées par les modèles, nous élaborons une solution visant à produire de manière semi automatisée le code des applications embarquées personnalisées. Les contributions de notre travail de recherche concernent le domaine de la personnalisation, mais aussi, dans une moindre mesure, celui des techniques de modélisation.

La suite de ce chapitre d'introduction est organisée comme suit.

- La section 1.1 définit la personnalisation et introduit ses concepts de base.
- La section 1.2 présente la carte à puce ainsi que ses domaines d'application les plus courants.
- La section 1.3 précise les particularités techniques de la fabrication et de la configuration des cartes à puce.
- La section 1.4 détaille le mécanisme de personnalisation actuelle des cartes à puce et présente les rôles potentiels de la carte dans différents schémas de personnalisation.
- La section 1.5 présente les motivations et objectifs du travail de recherche.
- La section 1.6 détaille l'organisation de ce document.

1.1 Introduction à la personnalisation

Littéralement, la personnalisation peut se définir comme le processus qui permet de rendre une chose personnelle, c'est-à-dire propre à un individu spécifique. Dans [1], Riecker définit la personnalisation au sein des environnements numériques de la façon suivante :

> *« Personalization is about building customer loyalty by building a meaningful one-to-one relationship »*

La personnalisation est donc l'action de fidéliser le client en établissant avec lui une relation sérieuse et individualisée. Comme l'illustre la Figure 1-1, la personnalisation dans le contexte informatique est le processus qui consiste à façonner un service, un logiciel, ou à adapter un contenu de manière à le faire correspondre à la spécificité d'un individu donné. La personnalisation a pris son véritable essor il y a quelques années, alors que la population utilisatrice des technologies de l'information commençait à croître significativement. Face à la multiplication des portails Internet et des sites de commerce électronique, la prise en compte de la spécificité des individus s'est imposée comme une préoccupation majeure des commerçants et autres fournisseurs de services ou de contenu.

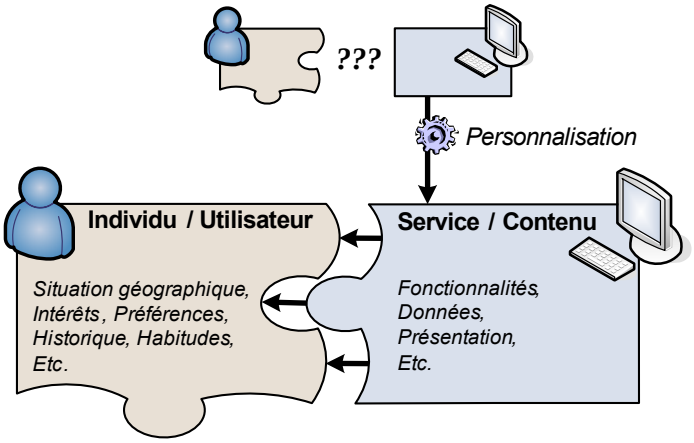


Figure 1-1 : Personnalisation de service / contenu

L'intérêt croissant porté à la personnalisation s'explique notamment par le besoin qu'ont les acteurs des technologies de l'information de différencier leurs produits pour affronter la forte compétition. Si la satisfaction finale de l'utilisateur est le motivation affichée de ce processus, la personnalisation est surtout un moyen de vendre plus, ou plus précisément, de vendre mieux [2]. Deux aspects sont essentiels dans la personnalisation : l'acquisition de données personnelles permettant de mieux connaître l'utilisateur et le façonnage du service ou contenu fourni, sur la base de cette meilleure connaissance.

Dans un premier temps, il est d'abord nécessaire de disposer d'informations sur l'utilisateur qui soient significatives dans le contexte du service offert. En général, les solutions de personnalisation reposent sur l'utilisation de *profils*. Un profil est une collection organisée de données relatives à un individu, sa structure est compréhensible par les mécanismes d'adaptation de service et contenu. Plusieurs stratégies permettent la collecte des informations à stocker dans le profil.

- *Stratégie explicite*. Le profil est renseigné manuellement par le biais d'une action de l'utilisateur qui prend lui-même la décision de divulguer des informations le concernant, par le biais de formulaires ou questionnaires par exemple.
- *Stratégie implicite*. Le profil est renseigné automatiquement par le biais de mécanismes d'analyse comportementale, reposant sur des techniques de filtrages individuels ou collaboratifs. Ce type de stratégie est principalement utilisé pour les systèmes de

recommandation, sur les sites de commerce en ligne notamment. Le *filtrage individuel* se base sur le comportement antérieur spécifique d'un individu donné – achat de produits, visualisation de pages web. Au cours du temps, le profil devient de plus en plus fidèle à l'individu qu'il représente. Le *filtrage collaboratif* [3] consiste à comparer le profil d'un individu donné avec celui d'autres utilisateurs pour estimer un degré de similitude permettant d'inférer des enrichissements.

La deuxième phase de la personnalisation consiste à sélectionner un contenu, ajuster les fonctionnalités d'un service ou recommander un produit, pour chacun des utilisateurs en fonction de son profil. Parmi les applications actuelles de la personnalisation, on peut citer des sites commerciaux comme *Amazon* [4] ou des portails Internet de programmes de télévision comme *PTV* [5]. La Figure 1-2 schématise l'architecture générale utilisée dans les solutions de personnalisation. Le *mécanisme de personnalisation* exploite un *profil utilisateur* pour adapter un *service générique* et en décliner un *service personnalisé*.

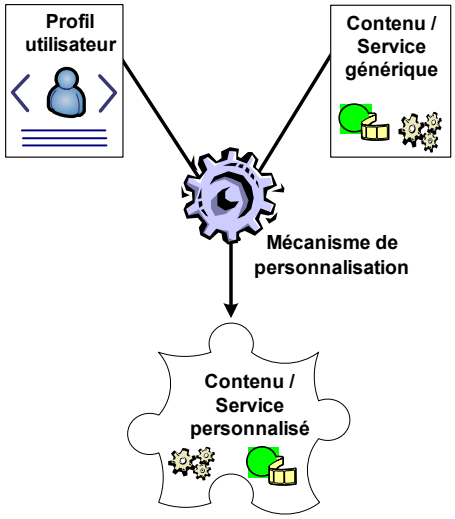


Figure 1-2 : Mécanisme de personnalisation

Que la collecte des informations soit explicite ou implicite, le mécanisme d'adaptation du service doit être imperceptible du point de vue de l'utilisateur : l'automatisation de ce processus est ce qui différencie la personnalisation de la configuration. Cette dernière requiert une implication explicite de l'utilisateur, qui réalise lui-même l'adaptation du service qu'on lui propose. C'est le cas notamment de certains portails d'information, comme *MyYahoo!* [6], qui offrent la possibilité à l'utilisateur de choisir lui-même une interface graphique spécifique, des thèmes d'information, etc. Souvent, l'utilisateur renonce à adapter explicitement le service ou le contenu qui lui est proposé, par manque de temps ou par découragement devant un processus trop complexe [7].

Si la personnalisation suscite un intérêt croissant dans le contexte de l'Internet et du commerce électronique depuis quelques années seulement, elle est depuis plus de quinze ans une préoccupation majeure dans le monde de la carte à puce : utilisée comme élément de commodité ou d'authentification, chaque carte à puce déployée est le support

d'informations spécifiques d'un individu donné. La section suivante présente l'aspect personnel des cartes à puce.

1.2 La carte à puce

1.2.1 Présentation

La carte à puce est un support informatique portable, sécuritaire et intelligent utilisé dans les gestes du quotidien, pour retirer de l'argent, effectuer un paiement, téléphoner, etc.

Une carte à puce est un rectangle en plastique de la taille d'une carte de crédit traditionnelle dans lequel sont incrustés un micromodule et une puce. La puce est un minuscule circuit intégré, réalisé en silicium, relié par des filaments aux contacts d'un micromodule implanté dans l'épaisseur de la carte. Le micromodule est visible sur le recto de la carte, au contraire de la puce qu'il loge. Les principales caractéristiques des cartes sont standardisées par la famille de standards et protocoles internationaux ISO 7816 (*International Standard Organisation*) [8].

Il existe plusieurs familles de cartes à puce. Alors que les basiques cartes à mémoire sont des systèmes permettant de décrémenter des unités stockées au préalable – principe de la carte à jetons –, les différentes cartes à microprocesseur sont dotées de capacité de traitement. Le déploiement massif de la carte à puce a débuté au milieu des années 1980 d'abord avec les cartes téléphoniques prépayées, puis avec les premières cartes bancaires.

Il existe actuellement trois grandes familles de cartes à microprocesseurs évoluées. Les cartes *mono applicatives* se caractérisent par une application embarquée unique figée dans la ROM. Le système d'exploitation et le code applicatif ne sont pas dissociés, ils forment un monolithe. Les cartes *dossiers portables* ont pour objectif principal le stockage d'informations liées à leur porteur et fournissent des mécanismes de structuration plus ou moins évolués, des cartes fichiers [9] aux cartes bases de données [10] [11]. Enfin, les cartes *multi-applicatives* sont basées sur trois principes fondamentaux : *(i)* la coexistence de plusieurs services, *(ii)* l'évolution du contenu applicatif par le biais de retraits et d'ajouts de services et *(iii)* la coopération possible entre les différents services embarqués. L'évolution des systèmes embarqués a considérablement élargi les domaines d'utilisation de la carte. Dans la suite, nous nous concentrons sur les cartes à microprocesseur, et plus particulièrement sur les cartes multi-applicatives.

1.2.2 Domaines d'application

Sa portabilité, sa puissance de calcul et sa capacité de stockage d'informations font de la carte à microprocesseur un outil idéal pour garantir une identification fiable et confidentielle sur les réseaux ouverts et géographiquement distribués. Durant les quinze dernières années, les trois principaux domaines d'utilisation de la carte ont été les applications de paiement, la téléphonie mobile et la sécurisation de divers types de réseaux. Plus récemment, les nouveaux besoins de sécurité, de confidentialité et de commodité

générés par le fort développement des technologies de l'information justifient le déploiement de solutions à base de cartes à puce

Dans le domaine des applications de paiement, la carte bancaire joue un rôle majeur au cœur des systèmes interbancaires de compensation. L'utilisation de la carte à puce vise d'une part à diminuer la fraude en rendant la falsification des cartes difficile, et d'autre part à rendre plus commodes les transactions bancaires, aussi bien pour les banques que pour les commerçants et clients. Les spécifications internationales EMV (*Europay MasterCard Visa)* assurent l'interopérabilité entre n'importe quel terminal de paiement et les cartes de n'importe quelle banque émettrice [12]. Par exemple, dans le cas d'une transaction hors ligne – c'est-à-dire pour les transactions d'un montant inférieur à un certain plafond – le certificat porté par la carte et la présentation du code PIN (*Personal Identification Number*) assurent la validité de la transaction, sans requérir l'établissement d'un contact avec la banque émettrice de la carte. En marge du déploiement massif des cartes bancaires de paiement, d'autres applications commerciales reposent également sur la carte à puce en l'utilisant comme support pour stocker des informations (programmes de fidélité, de promotions, etc.) ou de l'argent (porte-monnaie électronique du type *Moneo* [13], *Proton* [14], etc.).

La téléphonie mobile constitue indiscutablement le domaine dans lequel le plus grand nombre de cartes à puce a été déployé à ce jour. Les réseaux GSM (*Global System for Mobile Telecommunications*), d'abord introduits en 1993 en Europe et aujourd'hui étendus au monde entier, reposent sur l'utilisation des cartes SIM (*Subscriber Identity Module*) pour la sécurisation de l'accès au service. Son format *plug-in*[1] permet à la carte SIM d'être logée à l'intérieur du terminal mobile. Les caractéristiques de la carte SIM ainsi que son interface avec les terminaux mobiles sont standardisées dans la norme GSM [15]. La carte SIM permet *(i)* d'authentifier l'usager indépendamment du terminal utilisé, *(ii)* d'encrypter les transmissions de voix et de données afin d'en assurer la confidentialité, *(iii)* de stocker les données personnelles du porteur de la carte comme le répertoire téléphonique, les abonnements spécifiques, etc.

Dans le domaine de la sécurisation des réseaux, on peut citer par exemple la gestion du contrôle d'accès sur les réseaux internes d'entreprises, le décryptage et la validation des services dans la télévision à péage, ou encore la billettique électronique [16]. Dans ce dernier contexte, la carte à puce permet non seulement de réduire la fraude tout en apportant une certaine commodité pour les usagers. Parallèlement, elle fournit également aux structures émettrices des cartes des moyens pour contrôler l'activité sur leurs réseaux : dans les transports publics par exemple, des informations précises sur les trajets des usagers – destinations, fréquences, horaires – peuvent être extraites puis analysées.

Enfin, le développement global des **technologies de l'information** élargit les domaines d'utilisation de la carte à puce. Dans le domaine de la santé, la carte Vitale [17] en France permet depuis 1998 de simplifier les échanges entre les professionnels de la santé, les régimes d'assurance maladie et les individus. En Europe, la future carte de santé est

[1] Contrairement à celle du support plastique des cartes ISO 7816, la taille du support des cartes SIM est réduite au minimum. Ce format, nommé *plug-in,* est standardisé dans les spécifications GSM.

spécifiée par le consortium NetC@rds [18]. Aujourd'hui, les possibilités technologiques offertes par les cartes à puce – avec notamment l'utilisation de la biométrie pour la gestion des authentifications – fournissent aux gouvernements les outils pour une gestion pratique et sécurisée de l'identité des citoyens. Les premières solutions de passeports et cartes d'identité électroniques ont d'ores et déjà été déployées [19].

Son format et ses caractéristiques font donc de la carte à puce un support très adapté à la représentation de l'individu au sein des environnements numériques. Au-delà de la sécurité qu'elle garantit, la carte assure un niveau de confidentialité essentiel. L'individu porteur de la carte est le propriétaire de ses données personnelles. L'étape de personnalisation est pour cette raison une composante majeure du processus de fabrication des cartes.

1.3 Processus de production des cartes à puce

L'industrialisation de la production des cartes à puce est complexe, en raison *(i)* du besoin de coopération entre différents acteurs, *(ii)* des différents niveaux de configuration requis et *(iii)* des spécificités techniques liées à la fabrication. Cette section présente le processus de fabrication des cartes puis s'attarde sur les mécanismes de personnalisation.

1.3.1 Acteurs

Dans le cycle de vie d'une carte à puce, on peut globalement distinguer cinq intervenants majeurs, chacun jouant un rôle particulier dans le processus de production.

- L'*émetteur* de la carte est le décideur du contenu de la carte et le responsable de son déploiement. Typiquement, l'émetteur est un opérateur de téléphonie mobile, un organisme bancaire, gouvernemental, etc.
- L'*encarteur* conçoit, selon les critères définis l'émetteur, le système d'exploitation de la carte et les applications embarquées. Il réalise la configuration des cartes. Responsable de l'industrialisation, il assemble les différents éléments de la carte (câblage de la puce aux contacts électriques du micromodule, intégration de ces éléments dans le support plastique, etc.). Alors que le masquage du système est confié au *fabricant de semi-conducteur*, l'encarteur charge lui-même les applications dans la mémoire non volatile réinscriptible.
- Le *fabricant de semi-conducteur* (ou *fondeur*) est en charge de la conception matérielle de la puce. Notamment, il réalise sur ordre de l'encarteur l'étape de masquage. Cette étape, qui consiste à écrire le système de base de la carte dans la ROM, peut prendre entre 5 et 8 semaines.
- Le *fournisseur de service*. Optionnellement, un tiers fournisseur de service peut utiliser la plate-forme définie par l'émetteur pour proposer des services spécifiques.
- L'*utilisateur final*. La carte est en quelque sorte la représentation logique au sein des infrastructures déployées par l'émetteur de la personne physique porteuse de la carte.

1.3.2 Étapes de configuration

Une caractéristique majeure de l'industrie de la carte à puce réside dans les niveaux successifs de configuration nécessaires avant l'émission des cartes. L'architecture logicielle de la carte à microprocesseur est structurée en couches : gestion du matériel, machine virtuelle, librairies, applications (voir section 2.2.2). Les différentes étapes de configuration affectent successivement ou simultanément chacune des couches logicielles.

La Figure 1-3 montre que les principales étapes de configuration sont réalisées par rapport :

- *au domaine d'application*, avec notamment une première configuration du système d'exploitation de la carte et l'installation d'applications ou de librairies spécifiques (avec par exemple le chargement d'une plate-forme *SIM Application Toolkit*[2] [20] dans le contexte de la téléphonie),
- *aux émetteurs*, avec par exemple l'installation d'applications *SIM Toolkit* spécifiques.
- *aux utilisateurs finaux*, avec le chargement de données personnelles dans chaque carte (certificats, nom, numéro de compte, etc.).

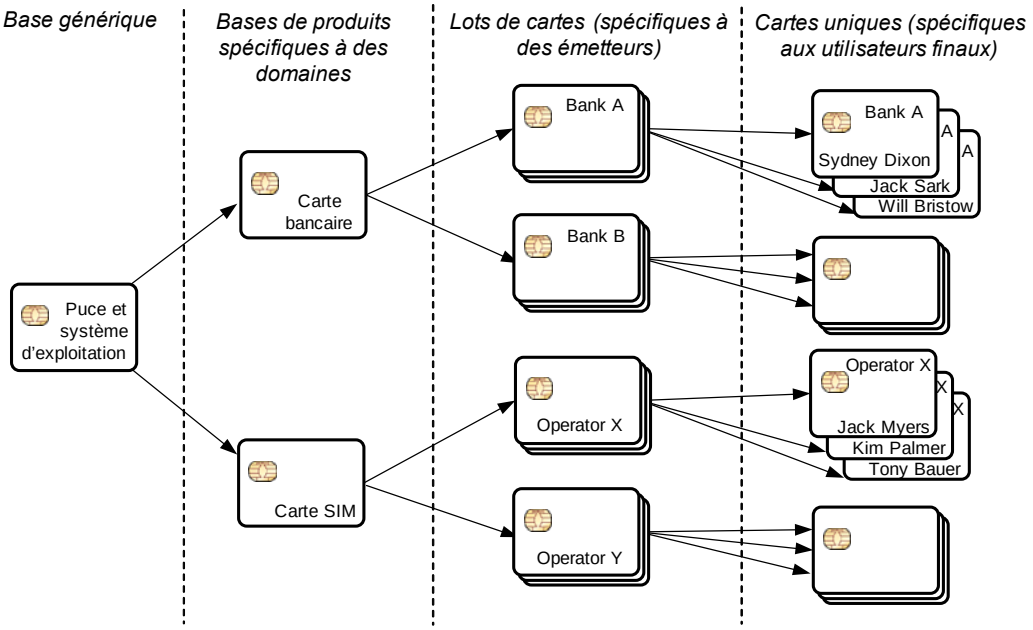


Figure 1-3 : Niveaux successifs de configuration des cartes

Afin de réduire les délais de mise sur le marché, l'encarteur doit anticiper les besoins des émetteurs, et prendre le risque de commander des masques au fabricant de silicium avant même de recevoir des commandes. Considérant les cinq à huit semaines de délai requis par

[2] Une plateforme *SIM Application Toolkit* est une interface donnant à la carte à puce un rôle proactif dans ses interactions avec le téléphone mobile dans lequel elle est insérée. Les applications *SIM Toolkit* permettent à la carte d'utiliser le téléphone mobile comme interface pour ses propres applications embarquées.

la phase de masquage chez le fondeur, il est essentiel pour l'encarteur de préparer des masques offrant un maximum de possibilités de configuration, même après la production physique de la puce. Une grande part de la configuration de la carte est réalisée lors de l'écriture en mémoire non volatile.

Plusieurs étapes sont nécessaires pour réaliser la configuration d'une carte. Les ajustements du système d'exploitation et le chargement des applications requises par l'émetteur sont d'abord effectués sur une carte prototype, appelée carte maître. La carte maître est une carte proche de la version finale, mais elle ne contient aucune information personnelle. Elle est en quelque sorte le dénominateur commun d'un lot de cartes destinées à un émetteur donné. L'image de la mémoire non volatile de la carte maître est extraite pour être répliquée via un protocole de copie rapide sur des cartes clones. L'ensemble des cartes clones forme un lot.

L'étape de personnalisation consiste à charger dans chaque carte les données personnelles de son utilisateur final, et, plus rarement, à charger le code d'applications spécifiques. Pour des raisons de sécurité, certains secrets comme les codes PIN, les certificats ou les clefs de cryptographie doivent être créés à l'intérieur de la carte et ne jamais en être extraits. Il est donc nécessaire d'envoyer des commandes spécifiques à chacune des cartes, ce qui rend le processus de personnalisation extrêmement coûteux en temps.

Une fois la carte produite, l'émetteur se charge de son déploiement vers les utilisateurs finaux. A partir de ce moment là, il dispose encore de moyens pour ajouter ou supprimer à distance des applications embarquées. C'est le principe de *post-issuance*. Les mises à jour postérieures au déploiement sont effectuées de manière transparente soit lorsque la carte est insérée dans un lecteur – un terminal bancaire de paiement par exemple –, soit par le biais de communications OTA[3] (*Over The Air*) dans le cas de la téléphonie mobile.

1.3.3 Vers une modernisation du processus

Les évolutions matérielles et logicielles des cartes imposent une réflexion sur les procédés actuels de production :

- *D'un point de vue méthodologique.* Les procédés par lesquels les informations de configuration sont transmises de l'émetteur à l'encarteur, puis traduites par l'encarteur dans une forme compréhensible par les machines de production sont relativement basiques. Alors que les plates-formes logicielles des cartes de prochaine génération marquent une réelle évolution architecturale, il devient nécessaire de moderniser les procédés de configuration. Par exemple, la mise en oeuvre d'une véritable ligne de produits logiciels pour cartes à puce faciliterait le dialogue entre l'émetteur de cartes et l'encarteur, et donc la définition des configurations spécifiques. Associée à un ensemble d'outils adaptés, une telle architecture pourrait permettre par exemple de réaliser des simulations, des estimations de coûts et délais, et surtout, d'automatiser certaines étapes.

[3] OTA est un standard pour la transmission et la réception d'information relatives aux applications dans les systèmes de télécommunications mobiles.

- *D'un point de vue technique*. Le coût de fabrication d'une carte est fortement dépendant du temps consacré à sa personnalisation : l'étape de chargement des données personnelles, très lente, conditionne le débit de la chaîne de production. Alors que les quantités de mémoire volatile embarquées dans les cartes à puce augmentent, la quantité de données personnelles à charger dans cette mémoire devient également plus importante. Afin de ne pas ralentir la chaîne de production, il est important de chercher à optimiser cette étape de manière à ce que la durée du chargement reste inférieure à un certain plafond de quelques secondes [21].

Dans le contexte de cette réflexion globale sur les procédés de configuration des cartes à microprocesseur, nous nous intéressons plus particulièrement à l'étape de personnalisation.

1.4 Personnalisation et cartes à puce

1.4.1 Personnalisation actuelle des cartes

Alors que la Figure 1-2 introduit les concepts généraux de *profil utilisateur*, de *service / contenu générique*, de *mécanisme de personnalisation* et de *service / contenu personnalisé*, la Figure 1-4 situe ces notions dans le contexte de la personnalisation de la carte à puce.

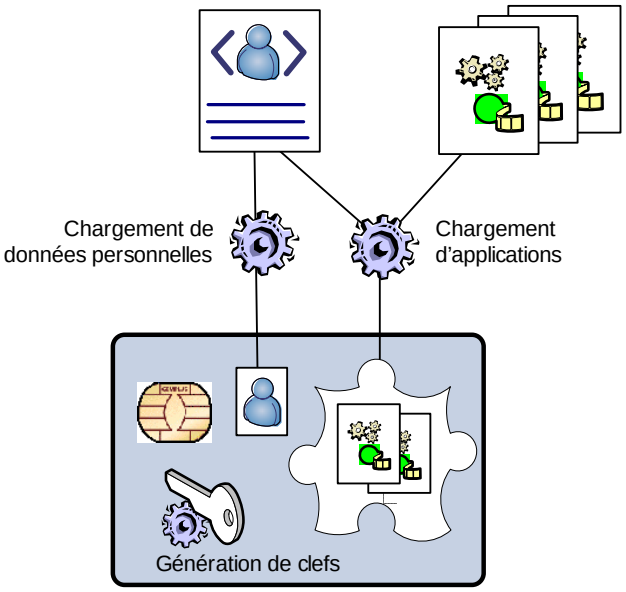


Figure 1-4 : Personnalisation de la carte à puce

En marge de la génération embarquée de clefs de cryptographie, des adaptations de deux types sont réalisées aujourd'hui :

- Le *chargement de données personnelles*, généralement effectué à la fin de l'étape de fabrication des cartes. Les données utilisateur telles que le nom ou le numéro de compte sont chargées successivement sur chacune des instances d'un lot de cartes clones (voir section 1.3.2).

- Le *chargement d'applications*, qui reste aujourd'hui plutôt une préoccupation de *post-issuance*. Dans le contexte de la téléphonie mobile notamment, l'opérateur peut personnaliser à distance la carte SIM d'un utilisateur en fonction des modifications apportées à son abonnement. La mise à jour se fait par le biais de commandes empaquetées dans des SMS[4] (*Short Message Service*) envoyés à la carte via le terminal mobile. Généralement, la personnalisation réalisée OTA consiste à charger des applications du type *SIM Toolkit.*

Dans les deux cas, le *mécanisme de personnalisation* est basique, puisqu'il ne s'agit que d'une simple copie dans la mémoire non volatile de contenu – données personnelles ou code d'applications à embarquer – stocké à l'extérieur de la carte. Cette copie est réalisée par le biais de commandes APDU (*Application Protocol Data Unit*), définies par le standard ISO 7816 [11]. Par analogie avec le mécanisme présenté par la Figure 1-2, l'adaptation du *service / contenu générique* correspond en fait à une sélection des applications à embarquer. Dans le cas de la téléphonie mobile, cette sélection est définie par l'abonnement de l'utilisateur. L'ensemble des applications installées sur la carte pour un utilisateur donné forme le *service / contenu personnalisé*.

Dans un contexte où l'évolution matérielle des puces et des plates-formes logicielles embarquées diversifie et rend plus riches les applications embarquées, les besoins de personnalisation deviennent plus importants. Dès lors, une réflexion sur la mise en œuvre de nouveaux mécanismes s'impose.

1.4.2 Topologie des solutions de personnalisation

Ses atouts de mobilité, de sécurité et de confidentialité destinent naturellement la carte à jouer un rôle clef dans les mécanismes de personnalisation sur l'Internet, dans la téléphonie mobile ou dans d'autres environnements émergents. La Figure 1-5 introduit différentes topologies d'architectures de personnalisation à base de cartes à puce, ou pour la carte à puce.

(a) La carte à puce support du profil de l'utilisateur.

Les notions de personnalisation et de protection de la confidentialité peuvent se compromettre mutuellement : si la personnalisation se base sur la collecte et l'exploitation de données personnelles, l'utilisateur est lui en général sensible à leur protection [22]. La décentralisation du stockage des profils est une option technologique offrant à l'utilisateur davantage de contrôle sur ses informations personnelles. Dans [23] par exemple, une solution proposée pour le stockage côté client consiste à enregistrer et chiffrer les données personnelles dans des *cookies*[5]. Cette solution restreint l'intérêt du profil à l'équipement sur lequel il est stocké.

L'utilisation de la carte à puce comme support du profil permet de mettre en œuvre des solutions décentralisées, sécurisées, et indépendantes des équipements utilisés [24].

[4] SMS est un service permettant d'envoyer des messages d'une longueur maximale de 160 caractères aux téléphones GSM.

[5] Artefact permettant à un serveur Web de déposer des informations sur une machine cliente. Chaque fois que l'utilisateur requiert une page de ce serveur Web, le contenu du *cookie* est transmis.

L'utilisateur peut définir quand, par qui, et dans quelle mesure ses informations personnelles sont utilisées.

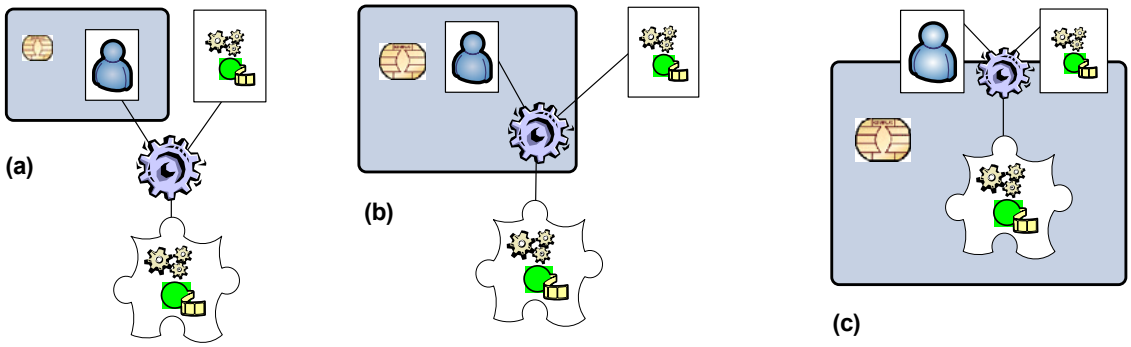


Figure 1-5 : Topologies des solutions de personnalisation

(b) La carte à puce support du profil utilisateur et du mécanisme de personnalisation.

Dans l'architecture **(a)**, le profil est stocké sur la carte, mais des données en sont extraites temporairement lors de l'exécution du processus de personnalisation. Or grâce à ses capacités de traitement, la carte à puce peut également héberger le processus d'adaptation de contenu ou de service. Les informations personnelles n'ayant plus à être extraites de la carte, le respect de la confidentialité est renforcé. Cette architecture constitue notamment la base des solutions suivantes :

- *Système de recommandation de films* dans le contexte de la télévision interactive [25]. La principale originalité de cette expérimentation est de proposer une approche qui s'appuie à la fois sur le filtrage collaboratif et le filtrage de contenu, tout en garantissant la confidentialité des données et donc l'anonymat des utilisateurs.
- *Césure* [26]. Cette approche utilise la carte à puce comme pilote du déploiement d'applications à base de composants. L'utilisateur peut alors retrouver son environnement personnel lorsqu'il passe d'une plate-forme d'exécution à une autre – un ordinateur portable, un PDA, un téléphone mobile, etc.

(c) Mécanisme de personnalisation embarquée.

Dans les configurations **(a)** et **(b)**, la carte est un outil pour le déploiement de solutions à grande échelle. Elle est utilisée comme élément de commodité ou de sécurité pour la personnalisation de services ou de contenus délivrés sur des supports multiples tels que l'Internet, les terminaux de télévision numérique ou les équipements mobiles du type téléphone ou PDA. Or la carte à puce devenant un support plus puissant et mieux connecté, il est possible de l'imaginer non plus comme un outil de personnalisation, mais comme le support des services personnalisés eux-mêmes. L'architecture décrite par la figure **(c)** marque une réelle évolution par rapport aux mécanismes de personnalisation actuellement utilisés dans le monde de la carte (Figure 1-4). Dans cette configuration, la carte peut être indifféremment le support de l'ensemble ou d'une partie du profil utilisateur, de

l'ensemble ou d'une partie du mécanisme de personnalisation et surtout, du service personnalisé.

1.5 Propositions pour la personnalisation des cartes à puce

1.5.1 Motivations et objectifs

Depuis l'apparition des cartes multi-applicatives, les plates-formes logicielles ouvertes constituent la majorité des cartes déployées. La séparation des couches matérielle, système et applicative signifie qu'en théorie, n'importe quel fournisseur de service est aujourd'hui susceptible de développer des applications pour la carte à puce. Dans la pratique, les faibles ressources offertes par les cartes et leur modèle de programmation freinent considérablement l'intérêt des fournisseurs d'applications autres que l'encarteur ou l'émetteur. Pour cette raison, la diversité des applications effectivement déployées est limitée. Ces applications étant le plus souvent pauvres en terme de fonctionnalités, les besoins de personnalisation restent basiques.

Ce paysage est en train d'évoluer. Grâce aux améliorations matérielles proposées par les fabricants de silicium – augmentation de la puissance des microprocesseurs et de la quantité de mémoire embarquée –, les plates-formes logicielles des cartes à puce de prochaine génération :

- *Autorisent des modèles de programmation évolués*, affranchis des limitations actuelles les plus contraignantes. Typiquement les cartes actuelles ne peuvent fonctionner qu'en mode serveur, si bien qu'il est difficile de faire jouer un rôle proactif à la carte. Les systèmes des futures cartes laisseront les développeurs libres de choisir ou de définir leurs modèles de programmation.
- *Permettent une meilleure intégration de la carte dans son environnement*, grâce à l'utilisation de protocoles de communication standard. Les protocoles définis par les standards ISO 7816 [8] sont obsolètes et isolent la carte du reste des infrastructures informatiques. Les prochaines générations de cartes supporteront le protocole TCP/IP (*Transmission Control Protocol / Internet Protocol*), standard *de facto* sur les réseaux actuels.

L'évolution des plates-formes logicielles ouvre de nouvelles opportunités d'utilisation des cartes à puce. Aujourd'hui, des téléphones mobiles aux assistants personnels en passant par les lecteurs numériques de musique ou autres périphériques, les supports informatiques deviennent de plus en plus mobiles, dédiés et performants. Le paysage numérique change, les comportements des personnes évoluent. Dans ce contexte de nomadisme, la carte à puce a le potentiel pour jouer un rôle clef comme support des services personnels. Cette perspective est une motivation essentielle du travail de recherche exposé dans ce mémoire de thèse.

Pour assumer ce rôle clef, la carte ne peut se contenter d'être une plate-forme supportant le chargement de données personnelles ou d'applications *SIM Toolkit.* Elle doit au contraire être au cœur des solutions de personnalisation, en permettant notamment une véritable

adaptation des applications embarquées. Il est donc nécessaire de moderniser les procédés de personnalisation, et de parvenir à élaborer une approche qui puisse être utilisable par différents acteurs du monde de la carte à puce – émetteur, encarteur, fournisseurs de service, commerçants électroniques, etc.

L'objectif de ce travail de recherche est de fournir aux développeurs d'applications pour cartes à puce les moyens méthodologiques et techniques pour automatiser efficacement la configuration de leurs produits selon la spécificité de chacun des utilisateurs. Nous nous concentrons exclusivement sur les mécanismes par lesquels les données personnelles peuvent être intégrées dans le processus de personnalisation du logiciel embarqué, sans attacher d'importance particulière à la façon dont ces données sont collectées.

Cette approche pour la personnalisation dans le contexte de la carte à puce doit :

- *Etre indépendante des plates-formes matérielles et logicielles embarquées*. Si elle a pour vocation d'être partagée entre plusieurs acteurs du monde de la carte, la réalisation de la personnalisation doit s'abstraire dans la mesure du possible des considérations spécifiques liées aux technologies utilisées.
- *Etre applicable à la fois lors de la production et après le déploiement des cartes.* La personnalisation d'une application embarquée doit pouvoir se faire aussi bien dans le contexte de la fabrication de la carte que plus tard au sein d'une infrastructure distribuée.
- *Etre basée sur des outils et méthodes de développement standard.* Les architectures logicielles des cartes tendent à se rapprocher des modèles de programmation standard. Il est nécessaire de profiter de cette évolution pour baser l'approche sur des notions essentielles de l'ingénierie logicielle telles que celles de modularité ou de réutilisation.
- *Promouvoir une automatisation maximale des procédés.* L'automatisation des processus de personnalisation permet non seulement un gain en efficacité, mais réduit également le risque d'erreur.

1.5.2 Propositions

Deux considérations méthodologiques majeures guident notre réflexion. La première est le respect du principe de séparation des préoccupations. Notre proposition est de considérer simplement la personnalisation comme une préoccupation de la conception d'une application parmi d'autres. La seconde est de veiller à l'optimisation du temps passé à la personnalisation de chaque carte sur la chaîne de fabrication. Notre proposition est de séparer, au cœur même du processus de personnalisation, ce qui est réellement spécifique à chaque utilisateur de ce qui ne l'est pas.

Nous proposons de décomposer le processus de personnalisation en deux étapes. L'application originale est d'abord transformée en application personnalisable, par le biais d'artefacts de programmation ajoutés dans le code. Cette version personnalisable est ensuite confrontée aux données spécifiques des utilisateurs, structurées dans des profils, afin que des versions personnalisées soient déclinées. Nous distinguons donc trois types de code : le code fonctionnel original, le code personnalisable et le code personnalisé.

Sur la base de cette décomposition, notre deuxième proposition consiste à mettre en œuvre un processus génératif permettant d'automatiser la production du code personnalisable et du code personnalisé. Afin de rendre l'approche indépendante de l'environnement technologique sous-jacent, nous nous reposons sur les concepts du développement dirigé par les modèles.

Pour la mise en œuvre du processus génératif nous avons, d'une part, défini un ensemble de méta-modèles pour exprimer les préoccupations de personnalisation, et d'autre part, implémenté des transformations de modèles. En particulier, nous proposons une approche pour répondre à une problématique de tissage de modèle.

1.6 Organisation du document

Ce mémoire de thèse est organisé de façon à décrire la démarche adoptée pour atteindre nos objectifs. La succession des différents chapitres est donc calquée sur le raisonnement mené : implémentation d'une solution concrète puis élévation du niveau d'abstraction pour généraliser et automatiser l'approche. L'ensemble du mémoire s'appuie sur un cas d'étude particulier, une application Java embarquée typique introduite dans le Chapitre 2.

Chapitre 2 – Personnalisation des applications embarquées**.** Le but de ce chapitre est d'introduire le concept de personnalisation logicielle et de préciser le contexte de nos travaux de recherche. Après avoir décrit les spécificités matérielles et logicielles des cartes à puce, ce chapitre décrit concrètement la mise ne œuvre d'une approche pour la personnalisation d'applications embarquées Java. L'approche, qui repose sur les concepts de code personnalisé et de code personnalisable, constitue la base de notre réflexion pour automatiser le processus de personnalisation. Ce chapitre introduit l'application *LoyaltyManager* qui est utilisée comme cas d'étude.

Chapitre 3 – Approches pour l'automatisation de la construction de logiciels**.** Ce chapitre présente certaines approches technologiques pour l'automatisation de la construction de logiciels – lignes de produits logiciels, programmation orientée aspects, approches génératives, approches dirigées par les modèles – et évalue leur pertinence par rapport à la problématique de personnalisation des applications embarquées. Le chapitre conclut sur une esquisse d'approche générative dirigée par les modèles pour la personnalisation des applications embarquées.

Chapitre 4 – Outils du développement logiciel dirigé par les modèles. Le but de ce chapitre est de présenter certains outils disponibles actuellement pour la mise en œuvre d'approches dirigées par les modèles, et plus particulièrement de celle pour la personnalisation. Ce chapitre se base principalement sur les spécifications de l'OMG (*Object Management Group*) et décrit l'expression des modèles par le biais du MOF et leur exploitation par le biais de XMI et JMI. Ce chapitre présente enfin une taxonomie des transformations de modèles.

Chapitre 5 – Définition du processus génératif de personnalisation**.** Ce chapitre décrit la mise en œuvre d'une approche générative dirigée par les modèles pour la personnalisation des applications embarquées. Sur la base de l'état de l'art présenté dans les chapitres

précédents, cette approche élève le niveau d'abstraction de la solution décrite dans le Chapitre 2. Elle définit les abstractions de personnalisation par le biais de méta-modèles et spécifie les projections permettant de produire les modèles nécessaires à la génération du code personnalisable et du code personnalisé. L'ensemble du processus est illustré par la personnalisation de l'application *LoyaltyManager*.

Chapitre 6 – Implémentation des transformations de modèles**.** La mise en œuvre du processus génératif de personnalisation requiert l'implémentation de deux types de transformations de modèle : une transformation basique de type 1-1 et une transformation plus complexe du type 2-1. Ce chapitre décrit l'implémentation d'un cadre de conception pour l'implémentation en Java de ces transformations et l'illustre son utilisation dans le contexte du cas d'étude *LoyaltyManager*. Ce chapitre décrit et illustre également une proposition de solution pour le tissage de modèles.

Chapitre 7 – Conclusion**.** Ce chapitre fournit un résumé du travail de recherche. Il en souligne les principales contributions et limites. Enfin, la perspective d'une ligne de produit pour la configuration globale des cartes à puce est évoquée.

Chapitre 2

Personnalisation des applications embarquées dans les cartes à puce

---

Les systèmes logiciels pour cartes à puce ont considérablement évolué depuis le déploiement des premières cartes. Ils sont aujourd'hui comparables, dans une certaine mesure, aux environnements logiciels offerts par les divers terminaux mobiles du type téléphone ou assistants personnels digitaux.

Par ses propriétés de portabilité et de sécurité, la carte à puce est naturellement amenée à jouer un rôle clef dans la fourniture de services personnels au sein des environnements émergent de l'informatique nomade. Le but de ce chapitre est d'introduire le concept de personnalisation logicielle et de préciser le contexte de notre réflexion.

Ce chapitre décrit concrètement la mise en œuvre d'une approche pour la personnalisation d'applications embarquées Java. Cette approche constitue la base d'une réflexion pour l'élaboration d'une méthode pour la personnalisation automatisée des applications embarquées.

Ce chapitre est organisé comme suit :

- La section 2.1 présente la notion de personnalisation logicielle.
- La section 2.2 décrit les spécificités matérielles et logicielles des cartes à puce.

- La section 2.3 se concentre sur l'évolution des plates-formes ouvertes et introduit la prochaine génération de *Java Card*.
- La section 2.4 présente l'application embarquée *LoyaltyManager* qui constitue notre cas d'étude. Les principes de base de ce gestionnaire de fidélité embarqué sont détaillés, avant que la personnalisation à réaliser ne soit explicitée.
- La section 2.5 présente une approche pour la personnalisation d'applications Java embarquées, en introduisant notamment les concepts de code personnalisable et de code personnalisé. La mise en œuvre est illustrée avec l'application *LoyaltyManager*.
- Enfin, la section 2.6 résume notre approche et introduit les directions de recherche suivies.

2.1 Personnalisation de logiciels

Aujourd'hui, la sélection de contenu et la mise en œuvre de systèmes de recommandation constituent les objectifs principaux des recherches académiques ou industrielles menées sur le thème de la personnalisation. Dans le contexte de la carte à puce, nous nous intéressons plus particulièrement à la personnalisation des applications embarquées, c'est-à-dire à la personnalisation de logiciels. De manière générale, le rapport entre le logiciel et son utilisateur final est aujourd'hui un rapport dans lequel le second s'adapte au premier. La personnalisation logicielle est le mécanisme qui permet d'aller vers un certain degré d'automatisation du processus inverse, c'est-à-dire l'adaptation du logiciel à son utilisateur final.

Il est intéressant ici de faire une comparaison entre l'industrie du logiciel et celle de la confection de vêtements. Dans le cas de la confection industrielle, un vêtement est fabriqué en plusieurs dimensions prédéfinies, l'acheteur final choisissant celle qui lui convient le mieux en faisant le plus souvent un compromis par rapport à la taille ou à la longueur. À l'opposé, la confection de vêtement sur mesure consiste à prendre d'abord les mensurations du porteur final pour fabriquer un vêtement dont la taille et la longueur sont ajustées. La fabrication sur mesure est évidemment nettement plus coûteuse.

Dans l'industrie du logiciel, les programmes sont en général disponibles dans un nombre très restreint de déclinaisons. On parle alors d'approche du type *one-size-fits-all*. L'utilisateur final choisit la déclinaison qui se rapproche le plus de ses besoins, puis s'adapte à elle. Au final, l'utilisateur n'utilise qu'une partie des fonctionnalités alors que dans le même temps, certaines autres sont susceptibles de lui manquer. Développer un logiciel sur mesure signifie que la spécificité et les attentes de l'utilisateur final sont prises en compte au cours de la phase de conception. Le défi de la personnalisation logicielle est de parvenir à mettre en œuvre une production sur mesure sans compromettre l'efficacité de la production de masse.

Personnaliser un logiciel consiste à industrialiser sa configuration, c'est-à-dire à tendre vers l'automatisation de celle-ci. Cela concerne par exemple la sélection des fonctionnalités à implémenter, l'affectation de valeurs à des propriétés, le déploiement, l'installation ou

encore à en préciser les propriétés non fonctionnelles. Notre objectif est de fournir les moyens nécessaires à la mise en œuvre de la personnalisation logicielle des applications pour carte à puce.

Avant l'introduction d'un cas d'étude et la description concrète d'une approche pour la personnalisation, la section suivante présente le contexte technique de nos travaux par le biais d'une description des caractéristiques matérielles et logicielles de la carte à puce.

2.2 Présentation technique de la carte à puce

2.2.1 Architecture matérielle

La carte à microprocesseur peut être considérée comme un ordinateur miniature, bénéficiant d'un niveau de sécurité très élevé. Le minimalisme de ses ressources rend nécessaire son intégration systématique dans un système réparti via un terminal dédié, typiquement un lecteur de carte bancaire ou un téléphone mobile. Le circuit intégré qui compose la puce est composé d'une ou plusieurs unités de traitements, de différents types de mémoire et d'un support de communication.

Les **unités de traitement** sont le microprocesseur et le co-processeur cryptographique. Les microprocesseurs les plus évolués sont des architectures RISC (*Reduced Instruction Set Computer*) de 16 ou 32 bits. Les co-processeurs sont dédiés aux calculs cryptographiques complexes, comme les chiffrements et déchiffrements DES (*Data Encryption Standard,* [27]) ou RSA (*Rivest Shamir Adleman,* [28]). Les co-processeurs sont optionnels, en raison du coût supplémentaire qu'ils induisent.

Plusieurs mémoires sont disponibles sur une carte à puce. Chacune possède des caractéristiques particulières qui la destinent à un rôle bien défini.

- La RAM (*Random Access Memory*), entre 128 et 4096 octets. Très rapide en lecture et écriture, elle est principalement utilisée comme mémoire de travail. La valeur des données qu'elle stocke n'étant pas maintenue lorsque la carte n'est plus sous tension, la RAM est une mémoire non persistante. Le rapport entre la quantité de données et la surface du silicium occupée est très élevé, ce qui en fait une mémoire chère.
- La ROM (*Read Only Memory*), entre 32 et 128 Ko. Mémoire persistante, non reprogrammable, qui contient en général le système de base de la carte écrit en C ou en langage d'assemblage. L'écriture des données dans la ROM est appelée le masquage[6]. Cette étape est réalisée par le fournisseur de silicium. Le délai de production d'environ 8 semaines justifie qu'une attention particulière soit portée au masque. La ROM est une mémoire rentable.
- La Flash / EEPROM (Electrical Erasable Programmable Read Only Memory), entre 4 et 64 Ko. Mémoire non volatile, reprogrammable avec un temps d'accès en lecture

[6] L'opération de *masquage* consiste à exposer des surfaces sélectionnées d'un semi-conducteur à une source de lumière, de sorte à enclencher un processus de polymérisation du silicium.

comparable à celui offert par la RAM, mais un temps d'écriture très pénalisant. L'EEPROM joue d'une certaine façon le rôle de disque dur de la carte.

Les échanges de données entre la carte et son environnement extérieur nécessitent un **support de communication**. Dans le cas d'une liaison série, la communication se fait par signaux électriques échangés par le biais des contacts du micromodule. Les taux de transfert sont compris entre 9600 et 192000 Bauds (*bits par seconde*). Dans le cas d'une liaison sans fil, appelée aussi sans contact, la communication se fait via des signaux RF (*Radio Frequency*). La bande passante dépend à la fois de l'antenne embarquée dans la carte et de la puissance du flux magnétique émis par le lecteur.

2.2.2 Architecture logicielle

Les architectures logicielles des cartes à puce ont considérablement évolué depuis les premiers déploiements de cartes au début des années 1980. Depuis les modèles monolithiques jusqu'à l'apparition des plates-formes ouvertes, les changements apportés à l'architecture des systèmes embarqués sont caractérisés par une séparation progressive entre le système d'exploitation et les applications. Quatre générations d'architectures logicielles des cartes se sont succédé [29].

- *Première et deuxième générations (1981 et 1985)* : les architectures sont monolithiques, elles ne distinguent pas les applications du système d'exploitation. La deuxième génération diffère de la première dans le sens où elle permet la factorisation de certains modules logiciels relatifs à la gestion matérielle.
- *Troisième génération (1992)* : lorsque, pour une application donnée, la question n'a plus été « *quels modules réutiliser ?* » mais « *quelle est la plate-forme qui convient le mieux à l'application ?*», une première réelle structuration en couches s'est mise en place, avec notamment une isolation des applications et du système d'exploitation. Les premières cartes SIM ont été basées sur ce type d'architecture.
- *Quatrième génération (1996)* : les architectures de quatrième génération sont généralement basées sur des machines virtuelles et sur des ensembles d'interfaces de programmation. Le principal apport de ces architectures est d'offrir la possibilité de charger des applications après le déploiement, en *post-issuance*. Avec à ce jour environ 600 millions de cartes émises l'implémentant, *Java Card* est la spécification de système pour carte à puce la plus importante [30].

2.3 Evolution des plates-formes logicielles embarquées

2.3.1 Architecture des plates-formes multi-applicatives

Bien qu'offrant le plus souvent des environnements de programmation limités en raison de la faible capacité des ressources offertes par les cartes, ces architectures logicielles ouvertes permettent une séparation claire entre le système d'exploitation d'une part et les applications d'autre part. La Figure 2-1 présente l'architecture typique d'une carte – selon

les contextes, la répartition des composants logiciels entre la ROM et l'EEPROM peut être différente.

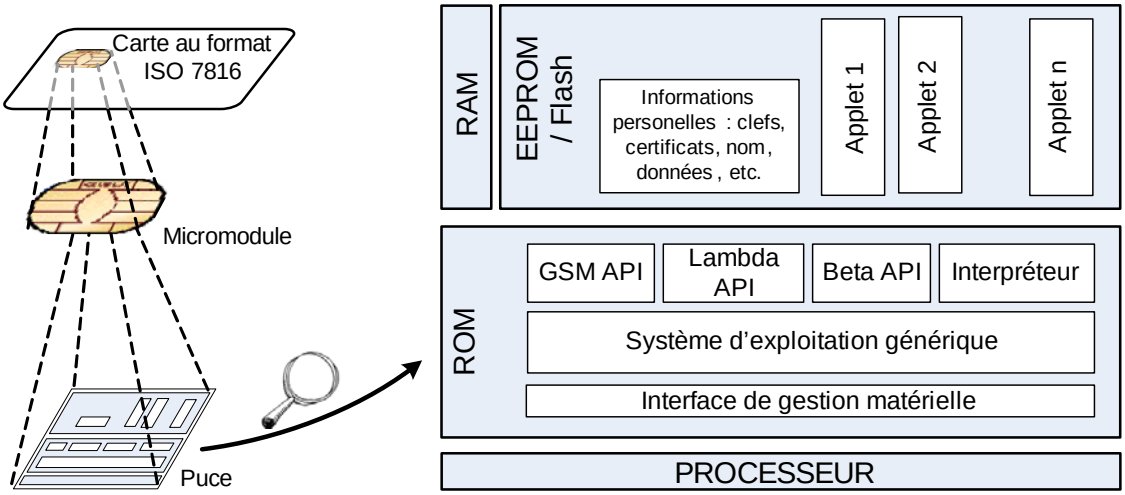


Figure 2-1 : Architecture typique d'une carte à microprocesseur multi-applicative

Parmi les cartes multi-applicatives, on peut citer :

- *MultOS* [31], une plate-forme ouverte autorisant le chargement sûr d'applications par des protocoles cryptographiques,
- *Java Card* [30], une machine virtuelle introduite en 1996 supportant un sous-ensemble du langage Java [32], définie par le Java Card Forum [33], un consortium regroupant entre autres les principaux encarteurs, fabricants de silicium, et Sun Microsystems.
- *Smartcard.Net* [34], une machine virtuelle apparue en 2003 définissant un sous-ensemble de Microsoft .NET [35] et supportant donc le langage intermédiaire CLI (*Common Language Infrastructure,* [36]) de Microsoft.

Dans la suite, nous nous concentrons sur *Java Card* et ses évolutions.

2.3.2 Plate-forme Java Card 2.x

La spécification de *Java Card 2.0* fin 1997 est le résultat du regroupement des principaux fabricants de cartes à puce et de *Sun Microsystems* au sein du *Java Card Forum* [33], une entité chargée de promouvoir l'utilisation de Java [37] pour le développement des applications embarquées dans les cartes à puce. Comme le montre la Figure 2-2, la spécification *Java Card* définit un sous-ensemble du langage Java. Cette spécification est basée sur la configuration minimale suivante : composant 8 bits, doté de 24 Ko de ROM, de 512 octets de RAM et de 16 Ko d'EEPROM.

Supporté	Non supporté
- Notion d'applet Java Card - `boolean, byte, short, int` - `Object` - Tableaux à une dimension - Allocation dynamique - Paquetages - Traitement des exceptions - Mécanismes d'héritage	- Notions d'applets et d'applications standard - `float, double, long` - `char, String` - Tableaux à n dimensions - *Ramasse-miettes*[7] - Désallocation explicite Chargement dynamique de classes - *Multi-threading*[8]

Figure 2-2 : Comparaison Java / Java Card 2.x

Le rôle d'une JVM (*Java Virtual Machine,* [38]) classique est de vérifier le *bytecode*[9] des classes compilées, de charger dynamiquement les classes vérifiées, puis d'interpréter le *bytecode*. La caractéristique principale de la machine virtuelle *Java Card* est de pallier les faibles ressources disponibles en distribuant ces fonctions à la fois à l'extérieur et à l'intérieur de la carte. La partie extériorisée de la machine virtuelle est le *Convertisseur Java Card*. Son rôle est de vérifier et transformer les fichier *class* – format standard des fichiers binaires Java – en fichiers au format allégé *cap*. Les fichiers *cap* sont ensuite chargés dans la carte pour y être interprétés par la partie embarquée de la machine virtuelle.

Le modèle d'application utilisé est celui des *applets Java Card*. Une plate-forme *Java Card* est conçue pour héberger plusieurs applets, chacune dotée d'un identifiant unique. Typiquement, les applets sont utilisées comme serveurs pour les applications clientes localisées sur les terminaux dans lesquels la carte est insérée. Leur rôle se limite à traiter des commandes APDUs et à retourner des APDUs de réponse. Les applets ont un cycle de vie long : une fois chargée dans la carte, une applet reste disponible, passant simplement de l'état *sélectionnée* à l'état *non sélectionnée* sur commande de l'application cliente.

L'architecture particulière de la machine virtuelle, le modèle d'application peu évolué ainsi que les protocoles de communication archaïques rendent la plate-forme *Java Card 2.x* complexe et difficile à appréhender pour les développeurs Java traditionnels [39]. Néanmoins, grâce aux évolutions du matériel, certaines des limitations majeures de *Java Card 2.x* sont sur le point d'être dépassées.

2.3.3 Plate-forme Java Card de prochaine génération

Au cours de ces dernières années, les plates-formes Java se sont considérablement développées. Java n'est plus confiné aux environnements bureautiques standard, mais est

[7] *Garbage-collector* dans le vocabulaire anglo-saxon.
[8] Le *multi-threading* est un mécanisme permettant de partager un processeur entre plusieurs fils, conçu de manière à minimiser le temps requis pour passer d'un fil à l'autre. Un 'fil' diffère d'une tâche dans le sens où un 'fil' partage davantage de son environnement avec les autres 'fils' que ne le font les tâches entre elles.
[9] Le *bytecode* est une forme intermédiaire située entre le code source et le code exécutable natif.

maintenant utilisé aussi bien sur le marché des serveurs avec J2EE (*Java 2 Enterprise Edition,* [40]) que sur celui des plates-formes embarquées avec J2ME (*Java 2 Micro Edition* [41]). Cette diversification des plates-formes a apporté à Java des enrichissements technologiques dont la carte à puce peut tirer profit. Par exemple, la gestion du long cycle de vie des applications embarquées peut s'inspirer des techniques utilisées du côté des serveurs, où ce type d'application est très commun. De même, la définition de sous-ensembles minimaux de Java, par le biais des configurations J2ME, est également intéressante à l'heure où des choix fonctionnels doivent être faits.

En travaillant sur une profonde révision des spécifications de *Java Card 3.0*, le *Java Card Forum* construit les bases de la prochaine génération de plates-formes logicielles pour cartes à microprocesseur. La volonté des fabricants et des émetteurs de cartes est aujourd'hui de profiter de la généralisation des microprocesseurs 32 bits RISC et de l'augmentation des quantités de mémoire embarquées pour aligner au maximum *Java Card* avec les autres plates-formes Java standard. L'architecture logicielle des cartes de prochaine génération semble se dessiner autour des caractéristiques suivantes.

Utilisation de composants et mécanismes Java standard. L'augmentation des capacités matérielles des cartes permet l'abandon du format de fichier binaire *cap* de *Java Card 2.x* pour utiliser le format *class* standard. Cette évolution supprime l'étape contraignante de conversion hors carte, rendant ainsi le processus de déploiement d'applications embarquées plus classique. Le chargement du code applicatif dans la carte se faisant par le biais de fichiers *class*, chaque implémenteur de machine virtuelle devient libre de définir ses propres procédés embarqués de conversion ou d'optimisation [42]. Dans le même esprit d'interopérabilité, la mise à disposition d'APIs moins restrictives permet de profiter des multiples composants et mécanismes Java disponibles aujourd'hui pour les plates-formes Java.

Gestion automatisée de la mémoire. Le *ramasse-miettes* est un outil essentiel des machines virtuelles Java standard. Il permet l'allocation transparente de mémoire aux objets Java quand ils sont créés et gère automatiquement le compactage, les déplacements et libérations de ressources mémoires. L'absence de *ramasse-miettes* dans les spécifications de *Java Card 2.x* contraint le développeur d'applications complexes à concevoir ses propres routines de gestion de la mémoire. Aujourd'hui, l'intégration d'un *ramasse-miettes* dans les systèmes logiciels pour carte à puce est fondamentale.

Intégration dans les réseaux distribués. Les protocoles de communication spécifiques à la carte définis par la norme ISO 7816 [8] sont obsolètes : asynchrones et semi-duplex[10], ils confinent la carte à un rôle d'esclave qui ne fait que répondre à des commandes APDU. Dans la version 2.2 de *Java Card*, l'implémentation de RMI (*Remote Method Invocation*) pour la gestion des accès distants a constitué une première avancée vers une meilleure intégration de la carte dans les infrastructures distribuées. Aujourd'hui, le support du protocole TCP/IP par la carte permet de s'adresser à la carte à puce comme à n'importe quel autre support informatique [43].

[10] Une connection semi-duplex est une communication qui fonctionne dans les deux sens, mais alternativement

Support de modèles d'applications multiples. Le support de multiples modèles d'applications non prédéfinis est une caractéristique importante des plates-formes Java. Au contraire de *Java Card 2.x* qui ne proposait qu'un modèle figé, les systèmes Java embarqués de prochaine génération autorisent différents modèles [44], comme les applications client / serveur RMI, les *Java Servlets*[11] – particulièrement intéressants dans le contexte de la carte à puce pour créer des interfaces graphiques aux applications embarquées sous la forme de pages Web – ou encore les applications indépendantes[12].

Configuration système. Afin de répondre à la complexité de la configuration des cartes à puce (section 1.3.2), la conception du système d'exploitation et de la machine virtuelle est modulaire et accompagnée d'outils de configuration. Ainsi, plusieurs versions sont déclinables selon les contextes.

L'ensemble des évolutions décrites ici sont motivées par la modernisation des processus de développement et déploiement d'applications pour cartes à puce, dans le but de faire de la carte un support d'exécution moins atypique. Face au développement considérable de l'informatique nomade, la carte à puce est désormais armée pour jouer un rôle clef comme support des services personnels. La section suivante introduit un cas d'étude pour la personnalisation d'applications embarquées.

2.4 Cas d'étude : l'application embarquée LoyaltyManager

2.4.1 Présentation

Alors que la carte à puce est utilisée comme porte monnaie électronique depuis plus de quinze ans (section 1.2.2), le concept des micro paiements électroniques prend aujourd'hui une nouvelle dimension [45] [46]. Désormais, plusieurs rôles peuvent être confiés simultanément à la carte à puce, comme par exemple dans le système *Octopus*, mis en place à Hong Kong en 1997 [47]. À une plus grande échelle, l'opérateur de téléphonie japonais *NTT DoCoMo* concrétise le concept du commerce mobile en déployant l'*i-Mode FeliCa Mobile Wallet*[13] [48]. La carte à microprocesseur intégrée dans le terminal mobile fait office à la fois de porte-monnaie électronique, de porteur d'identité, de porteur de billets électroniques, etc. La carte dispose d'une antenne qui lui permet d'envoyer et recevoir des signaux RF de façon autonome – la carte reste fonctionnelle lorsque le téléphone mobile est éteint. Ce service, introduit en Juillet 2004, est déjà utilisé entre autres par diverses compagnies ferroviaires et aériennes.

De manière générale, la carte à puce constitue dans la téléphonie mobile le point unique de convergence physique entre l'utilisateur, l'opérateur et les fournisseurs de services. Elle est donc un support très adapté pour le déploiement des services personnels. L'application fictive que nous utilisons ici comme cas d'étude est le *LoyaltyManager,* un gestionnaire de

[11] Un *Java Servlet* est un programme Java du type serveur, souvent utilisé pour générer du contenu Web dynamiquement.
[12] *Standalone* dans le vocabulaire anglo-saxon.
[13] *i-Mode* est une plateforme permettant l'accès continu à l'Internet depuis les terminaux mobiles.

fidélité embarqué. Exécuté sur la carte SIM – de prochaine génération –, le *LoyaltyManager* maintient des données de fidélité pour le compte des partenaires commerciaux participants. Les paiements étant réalisés par le biais du téléphone et de la carte, le *LoyaltyManager* dispose des informations nécessaires pour enrichir après chaque transaction effectuée la situation de fidélité correspondante.

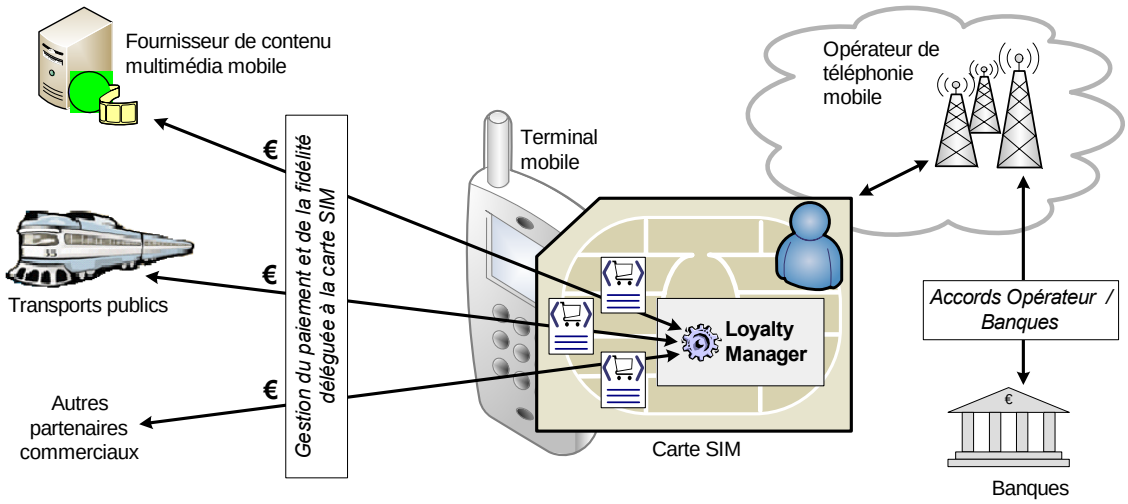


Figure 2-3 : Application embarquée LoyaltyManager

La Figure 2-3 montre la position centrale du *LoyaltyManager* au sein d'une infrastructure de commerce mobile. Les partenaires commerciaux peuvent être des acteurs de la téléphonie mobile comme des fournisseurs de contenu multimédia pour terminaux mobiles – vidéo, musique, informations, services de vidéo conférence, etc. – ou des entités plus communes telles qu'une compagnie de transports en commun par exemple.

Si l'application *LoyaltyManager* n'est qu'un prototype servant de base à notre réflexion, la récente mise en œuvre d'infrastructures de commerce mobile comme l'*i-mode FeliCa Mobile Wallet* la rend très crédible : par exemple, l'utilisation de service au moment de l'embarquement dans un aéroport permet à l'utilisateur avec certaines compagnies aériennes de cumuler davantage de Miles[14].

2.4.2 Fonctionnement et personnalisation

Concrètement, l'application embarquée *LoyaltyManager* se base sur les concepts suivants :

- Le *LoyaltyManager* gère plusieurs porte-monnaie électroniques, et plusieurs partenaires commerciaux.
- Chaque porte-monnaie est opéré pour le compte d'un seul gestionnaire (banque, association de commerçants, etc.) et peut être utilisé chez plusieurs partenaires commerciaux.

[14] Les Miles, du système de mesure anglais, sont les unités couramment utilisées dans les programmes de fidélisation de la clientèle proposés par les compagnies aériennes.

- Un partenaire commercial n'accepte qu'un type de porte-monnaie.
- Un porte-monnaie maintient, pour chaque partenaire commercial, un compte de fidélité.
- Il existe trois statuts de fidélité : bronze, argent, or.
- Chaque statut nécessite un certain nombre de points de fidélité.
- Chaque statut donne droit à un certain pourcentage de réduction lors des achats.
- Les points de fidélité sont acquis lors des transactions, et sont associés à une date d'expiration (ce qui implique que le statut d'un client chez un partenaire commercial peut se dégrader avec le temps et l'absence d'achats).

La façon dont les porte-monnaie électroniques sont rechargés n'est pas essentielle dans le cadre de nos expérimentations. L'opérateur téléphonique peut par exemple jouer le rôle d'intermédiaire auprès des banques et créditer les porte-monnaie par le biais de virements bancaires. On peut également imaginer que l'utilisateur puisse recharger lui-même ses porte-monnaie en utilisant des bornes spécifiques. La Figure 2-4 illustre le fonctionnement de l'application *LoyaltyManager* en détaillant le scénario d'un achat.

Lorsque le client porteur de la carte initie un processus d'achat, le porte-monnaie utilisé se charge de récupérer le statut du client dans le compte de fidélité qu'il maintient pour ce partenaire. Sur la base de ce statut, il effectue la réduction correspondante, rétribue le partenaire commercial, et met à jour le compte de fidélité concerné en le créditant de points de fidélité.

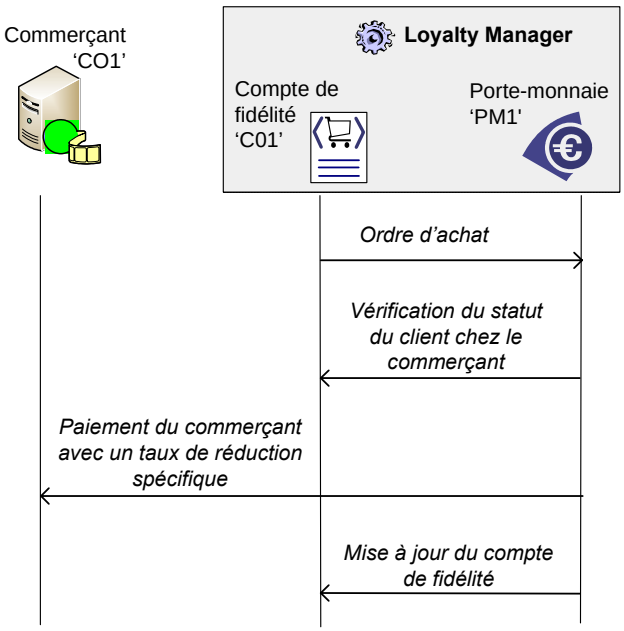


Figure 2-4 : Fonctionnement du LoyaltyManager

Le *LoyaltyManager* est une application simple, qui repose donc sur un ensemble limité de concepts. Notre objectif est de parvenir à personnaliser le déploiement de l'application sur les cartes à puce. Pour un utilisateur final donné, il s'agit de :

- Créer ses porte-monnaie électroniques, créer les comptes de fidélité correspondant à ses commerçants habituels
- Établir les relations entre les porte-monnaie et les comptes de fidélité
- Initialiser chacun des comptes de fidélité

Cette application étant développée en Java, l'instanciation de versions personnalisées consiste à affecter des valeurs spécifiques à certains attributs, à créer des objets – porte-monnaie et comptes de fidélité – et à établir des connections entre eux. Le but de la section suivante est d'investiguer une approche pour mettre en œuvre la personnalisation d'une application Java.

2.5 Implémentation d'applications personnalisées

Cette section présente une approche permettant d'instancier des versions personnalisées d'applications Java dans le contexte de la carte à puce. Après une introduction des principes fondateurs de l'approche, cette section décrit le processus et sa mise en œuvre dans le cas de l'application *LoyaltyManager*.

2.5.1 Considérations méthodologiques

L'objectif de l'approche exposée ici est de permettre la configuration de parties fonctionnelles d'une application embarquée. Deux considérations principales guident notre démarche :

- *L'aspect systématique de l'approche*. Comme nous l'avons expliqué dans la section 1.5.1, notre ambition est de fournir une méthodologie pour la personnalisation qui soit partagée entre tous les acteurs du monde de la carte à puce. Pour cette raison, notre approche doit être indépendante de l'application à personnaliser. Cela nous conduit à faire de la séparation des préoccupations un fondement de notre démarche.
- *L'efficacité du processus*. Dans la section 1.3.3, nous avons évoqué les défis liés à la modernisation du processus de configuration des cartes. En particulier, l'optimisation du temps consacré à la personnalisation de chaque carte sur la chaîne de fabrication est essentielle. Elle nous amène à introduire les notions de code personnalisable et de code personnalisé.

La séparation des préoccupations[15] [49, 50] est un concept mis en avant depuis de nombreuses années dans l'ingénierie des logiciels. Cette approche permet de considérer le logiciel non plus dans sa globalité, mais comme un ensemble de sous parties relatives aux différentes motivations des concepteurs. Cette décomposition réduit la complexité de la

[15] *Separation of concerns* dans le vocabulaire anglo-saxon.

conception logicielle en permettant de traiter des éléments plus ciblés et donc plus facilement manipulables. La séparation des préoccupations est la motivation de plusieurs paradigmes d'ingénierie logicielle comme la programmation par aspects [51], les filtres de composition [52], la programmation par sujets [53] et la séparation multidimensionnelle des préoccupations [54] [55]. Rendre notre approche indépendante de l'application à configurer nous conduit à considérer la personnalisation comme une préoccupation particulière. Le concepteur de l'application n'a plus besoin de gérer lui-même les mécanismes de personnalisation : il peut se concentrer sur le coeur fonctionnel de son application.

Dans la section 1.3.2 décrivant le processus de fabrication d'une carte, nous avons montré que l'optimisation de l'étape de personnalisation est obtenue par le biais d'une décomposition en deux phases : une première qui consiste à dupliquer sur les cartes clones les éléments qui ne sont pas spécifiques à un utilisateur et une deuxième qui consiste à compléter ces cartes clones successivement avec les informations personnelles de chacun des porteurs finaux. Cette séparation de ce qui est commun et de ce qui est spécifique aux utilisateurs finaux nous conduit à introduire les concepts de *code personnalisable* et de *code personnalisé*. L'installation d'une application Java embarquée dans une carte à puce consistant à charger l'ensemble des classes compilées dans la machine virtuelle, on peut alors optimiser ce chargement en dissociant le chargement des classes personnalisables et celui des classes personnalisées.

Etant données la séparation des préoccupations et la distinction de ce qui est spécifique ou non aux utilisateurs finaux, notre approche repose sur trois types de code.

- *Code fonctionnel original*. Il s'agit du code de l'application avant que la préoccupation de personnalisation ne soit prise en considération. Ce code définit la logique de l'application et implémente ses fonctionnalités.
- *Code personnalisable*. Il est le résultat de l'ajout dans le code original d'artefacts de personnalisation. Ces artefacts sont des morceaux de code qui fournissent les outils pour intervenir sur la création des classes et l'affectation de valeurs à leurs attributs. L'ensemble des artefacts ajouté à une classe constitue une *fabrique de personnalisation*. Transformer le code original en code personnalisable est la première phase de l'intégration de la préoccupation de personnalisation. Les structures utilisées dans les fabriques de personnalisation sont valables quelles que soient les applications. Le code personnalisable n'est pas spécifique aux utilisateurs.
- *Code personnalisé*. Ce code est spécifique à un utilisateur donné puisqu'il est écrit en fonction de ses informations personnelles. Il est en fait constitué d'un ensemble d'appels aux méthodes mises à disposition par le biais des fabriques de personnalisation. On appelle ce code personnalisé l'*activateur*, il est appelé lors de l'amorce[16] de l'application.

[16] Dans le vocabulaire anglo-saxon, on parle du *bootstrap* de l'application.

2.5.2 Mise en œuvre en Java

La Figure 2-5 illustre ces concepts dans le cas d'une application Java. Le code d'une application est composé d'un certain nombre de classes dont certaines seulement doivent être configurées. Le code original de ces classes à configurer est transformé en code personnalisable par l'ajout de classes internes[17] statiques[18], appelées `Personalizer`. Chaque classe `Personalizer` comporte d'une part des méthodes permettant de fabriquer des instances de sa classe de référence, et d'autre part des méthodes permettant l'affectation de valeurs aux champs, publics ou privés, de cette classe. Cette solution se rapproche du patron de conception *fabrique*[19] [56]. Le code personnalisé est inséré ultérieurement dans la classe d'amorce de l'application – typiquement la classe contenant la méthode `main` – sous la forme, par exemple, d'une méthode `personalize`. Cette méthode `personalize` utilise les diverses classes `Personalizer` pour contrôler l'instanciation des classes à configurer.

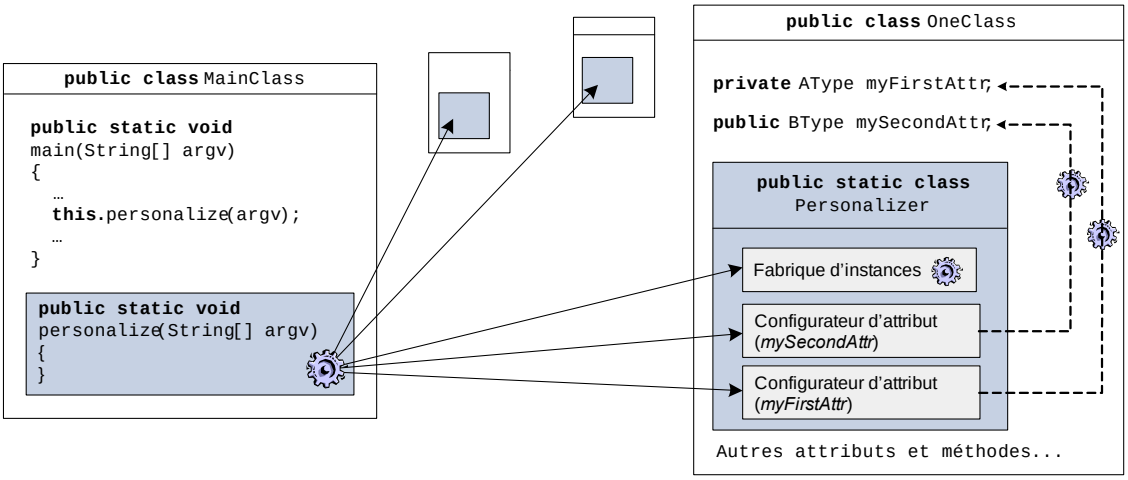


Figure 2-5 : Concepts de code personnalisable et code personnalisé

Nous illustrons ici ces notions dans le contexte de l'application *LoyaltyManager* présentée dans la section 2.4. Le code Java de cette application basique comporte un nombre restreint de classes, dont `Purse` (porte-monnaie), `LoyaltyAccount` (compte de fidélité) et `Main` (amorce de l'application). Une partie de la personnalisation de cette application embarquée consiste à créer pour chaque utilisateur final un certain nombre de porte-monnaie et de comptes de fidélité, chacun avec des paramètres spécifiques. Les classes `Purse` et `LoyaltyAccount` sont donc des classes qui requièrent une personnalisation. Chacune des deux est enrichie d'une classe interne `Personalizer` qui la rend personnalisable. La Figure 2-6 illustre la classe interne `Personalizer` dans le cas de la classe `Purse`. Cette dernière contient notamment trois champs, `Balance`, `Operator` et `LoyaltyAccounts` auxquels

[17] Dans le langage Java, on parle d'*inner class*.
[18] Les champs, classes et méthodes statiques n'appartiennent pas à des instances particulières.
[19] Dans le vocabulaire anglo-saxon, ce patron est appelé *Factory Method*

doivent être affectées des valeurs propres à chaque utilisateur final. La classe statique interne `Personalizer` est principalement composée :

- d'une méthode de création d'instances de la classe `Purse`,
- d'une structure de données contenant chacune de ces instances,
- des méthodes `setBalance`, `setOperator` et `addLoyaltyAccount` permettant d'affecter respectivement des valeurs spécifiques aux champs `Balance`, `Manager` et `LoyaltyAccounts`. Cette classe fournit les outils nécessaires à l'activateur pour gérer les instanciations relatives à la classe `Purse`.

```
package com.gemplus.srl.loyaltymanager;
import what.is.needed.*;

public class PurseImpl implements Purse {

  // champs - code fonctionnel original
  private float Balance;
  private String Operator;
  private Hashtable LoyaltyAccounts;
  // . . . autres champs non détaillés ici

  // méthodes - code fonctionnel original . . .
  // non détaillées ici

  public static class Personalizer {
      static Hashtable instances = new Hashtable();
      public static Purse create(Object key) {
          Purse instance = new PurseImpl();
              instances.put(key, instance);
              return instance;
      }
      public static void setBalance (Purse inst, float value) {
          ((PurseImpl)inst). Balance = value;
      }
      public static void setOperator (Purse inst, String value){
          ((PurseImpl)inst).Operator = value;
      }
      public static void addLoyaltyAccount (Purse instance, LoyaltyAccount f){
          ((PurseImpl)inst).LoyaltyAccounts.add(f);
      }
      // autres méthodes non détaillées ici
  }
}
```

Figure 2-6 : Classe Personalizer interne de la classe Purse

Dans l'implémentation de l'application *LoyaltyManager*, la classe `Main` joue le rôle de classe d'amorce. L'activateur, par le biais duquel la personnalisation va être effectuée, est donc implémenté sous la forme de la méthode `personalize` dans la classe `Main`. Il existe donc autant de versions de cette classe `Main` qu'il y a d'utilisateurs finaux. La Figure 2-7 détaille, pour l'utilisatrice *Sydney Bauer*, une partie de la méthode `personalize`.

Deux porte-monnaie sont créés pour cette utilisatrice, par le biais de la méthode statique `create` détaillée dans la Figure 2-6. Les porte-monnaie sont gérés par *LocalShops* et *Transportation*, ils sont initialisés avec respectivement 475 et 60 euros. La création des comptes de fidélité n'est pas détaillée ici. Une fois l'ensemble des instances créées, le rôle

de l'activateur est d'établir les liaisons entre instances. Ici, le porte-monnaie *Transportation* gère des comptes de fidélité pour les deux partenaires commerciaux *TaxiCompany* et *TramwayCompany*.

```
public static void personalize ( String[] argv ) {

  // activateur pour Sydney Bauer
  // instanciations et configurations des porte-monnaie
  Purse LocalShops_Purse = PurseImpl.Personalizer.create("LocalShops");
  PurseImpl.Personalizer.setOperator(LocalShops_Purse,"LocalShops");
  PurseImpl.Personalizer.setBalance(LocalShops_Purse,475);
  Purse Transportation_Purse = PurseImpl.Personalizer.create("Transportation");
  PurseImpl.Personalizer.setManager(Transportation_Purse,"Transportation");
  PurseImpl.Personalizer.setBalance(Transportation_Purse,60);

  // instanciations and configurations des comptes de fidélité
  // . . .   (non détaillées ici) . . .

  // gestion des liaisons
  PurseImpl.Personalizer.addLoyaltyAccount
      (Transportation_Purse,TaxiCompany_LoyaltyAccount);
  PurseImpl.Personalizer.addLoyaltyAccount
      (Transportation_Purse,TramwayCompany_LoyaltyAccount);
  // . . . autres liaisons non détaillées ici . . .
}
```

Figure 2-7 : Exemple d'activateur pour l'application LoyaltyManager

2.5.3 Déploiement des applications personnalisées

Le déploiement d'une application sur la carte à microprocesseur consiste à charger les classes personnalisables puis à exécuter le code de l'activateur. Physiquement, l'activateur peut être localisé sur la carte ou à l'extérieur selon le contexte du déploiement – dans le cadre de la chaîne de fabrication ou en *post-issuance*.

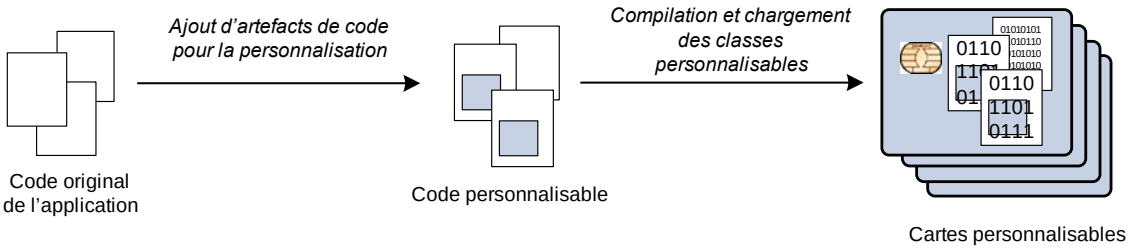


Figure 2-8 : Chargement des classes personnalisables

Dans le contexte du processus de fabrication des cartes, le chargement du code personnalisable, illustré par la Figure 2-8, peut être optimisé. Puisque les classes personnalisables ne diffèrent pas d'un utilisateur à un autre, leur chargement peut se faire lors de l'étape de duplication de l'EEPROM de la carte maître vers les cartes clones (section 1.3.2). Cette possibilité assure le critère d'efficacité évoqué dans les

considérations méthodologiques pour l'élaboration de l'approche. Dans le cas d'une installation de l'application en *post-issuance*, les classes compilées peuvent être chargées à distance, par le biais d'une plate-forme OTA par exemple.

Une fois les classes personnalisables chargées, l'application doit être amorcée avec les activateurs de chacun des utilisateurs. Cette étape, qui constitue le cœur du processus de personnalisation, peut être réalisée à l'intérieur ou à l'extérieur de la carte suivant le contexte.

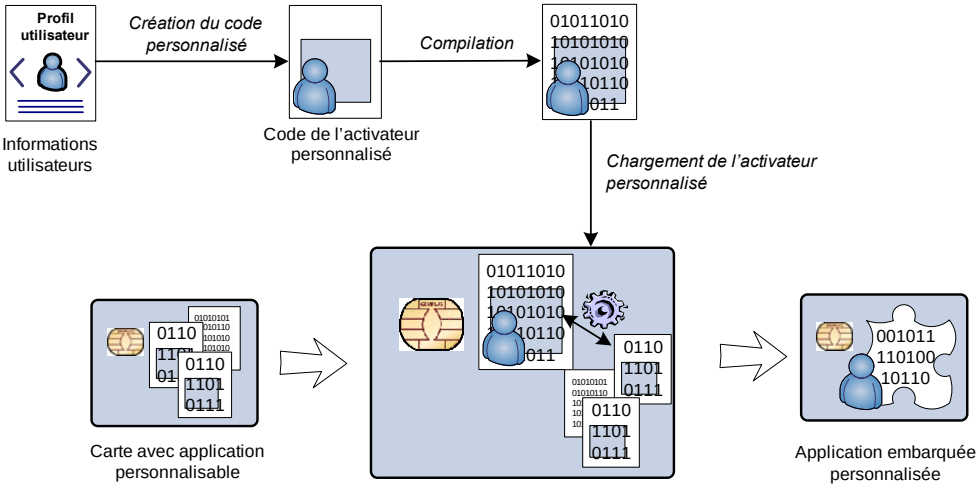


Figure 2-9 : Personnalisation dans la carte

Chargement de l'activateur dans la carte. Le mécanisme complet de personnalisation est exécuté dans la carte elle-même (Figure 2-9). Dans cette configuration, la classe d'amorce contient un appel local vers la méthode `personalize`. Cette solution est notamment intéressante pour une personnalisation en *post-issuance* : l'internalisation complète du processus de personnalisation garantit une sécurité optimale dans un environnement potentiellement hostile.

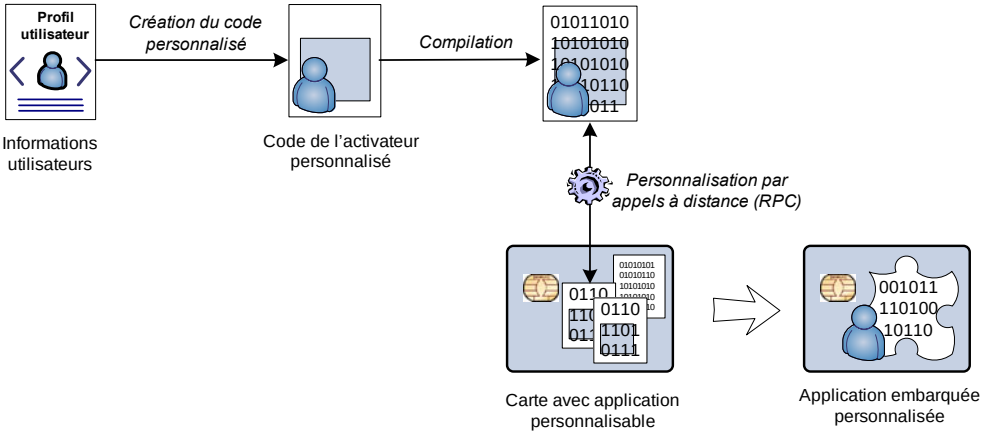


Figure 2-10 : Personnalisation par RPC

Activateur stocké à l'extérieur de la carte et utilisation de RPC (*Remote Procedure Call*). Cette solution est particulièrement adaptée à un environnement clos et protégé tel que celui d'une machine de production. Typiquement, l'activateur est géré par un logiciel d'industrialisation de personnalisation, le mécanisme de personnalisation se faisant par le biais d'appels RPC (Figure 2-10).

L'approche de personnalisation des applications Java destinées à la carte à puce décrite ici met en œuvre des mécanismes plus puissants et complexes que ceux utilisés actuellement. Si nous avons détaillé les aspects techniques relatifs à la personnalisation du code Java d'une application, nous n'avons pas encore abordé la problématique qui consiste à mettre en oeuvre cette approche à grande échelle. En particulier, nous n'avons pas encore parlé de mécanisme permettant d'exploiter efficacement les données personnelles pour obtenir le code de l'activateur de chacun des utilisateurs finaux.

2.6 Synthèse

La nécessité d'une personnalisation plus riche des applications embarquées est une conséquence, d'une part, de l'évolution des architectures matérielles et logicielles des cartes à puce et, d'autre part, de l'augmentation du nombre de fournisseurs susceptibles de délivrer des services au travers des cartes. Ce chapitre a présenté les caractéristiques des prochaines générations de systèmes logiciels qui vont, entre autres, offrir des environnements de programmation beaucoup moins contraints et faire de la carte un support d'exécution mieux intégré dans son environnement informatique.

Alors que le Chapitre 1 a présenté globalement le contexte de nos travaux de recherche, ce chapitre a explicité notre vision de la personnalisation de logiciels, et plus précisément de la personnalisation d'applications pour cartes à puce. De manière générale, produire un logiciel personnalisé consiste à intégrer dès sa conception la spécificité de son utilisateur final. Dans le contexte des applications pour cartes à puce, et notamment des applications Java, cela se traduit par exemple par la définition de fonctionnalités devant être supportées, par la configuration des instanciations d'objets ou encore par l'établissement de liaisons entre instances.

Dans ce chapitre, nous avons présenté une approche pour l'implémentation d'applications embarquées personnalisées. Afin d'intégrer les contraintes techniques liées aux procédés de fabrication des cartes et de respecter le principe de séparation des préoccupations, nous avons introduit pour les langages à objets la distinction entre les concepts de code personnalisé et de code personnalisable. Le code personnalisable est le résultat de l'ajout au code fonctionnel original d'artefacts pour la personnalisation. Ce code est commun à tous les utilisateurs. Le code personnalisé est écrit en fonction des données personnelles d'un utilisateur donné. La séparation entre le code personnalisable et le code personnalisé prend tout son sens lors de la fabrication des cartes, lorsque l'on cherche à minimiser le temps passé à spécialiser chaque carte sur la chaîne de production : le code personnalisable permet de factoriser une partie des données à charger dans la carte.

Nous avons illustré la mise en œuvre de cette approche avec l'implémentation en Java de l'application *LoyaltyManager*, qui sera réutilisée dans la suite comme cas d'étude. Le *LoyaltyManager*, que nous avons décrit précisément, est une application typique gérant un ensemble de porte-monnaie et de comptes de fidélité.

Notre objectif étant de fournir aux développeurs d'applications pour cartes à puce les moyens de réaliser une personnalisation à grande échelle de leurs produits, il est nécessaire d'automatiser autant que possible le processus de personnalisation. Cette automatisation constituant la problématique majeure de nos travaux de recherche, la suite de ce mémoire de thèse se concentre sur la démarche permettant de la réaliser. Nous cherchons donc à automatiser la production du code personnalisable et du code personnalisé.

Dans le chapitre suivant, nous nous intéressons aux approches utilisées aujourd'hui pour automatiser la construction de logiciels. Chacune des solutions technologiques est évaluée par rapport à notre problématique.

Deuxième partie

État de l'art

Chapitre 3

Approches pour l'automatisation de la construction de logiciels

Le chapitre précédent a décrit une approche pour la personnalisation des applications Java pour cartes à puce. Cette approche, qui intègre certaines considérations méthodologiques liées au contexte particulier des cartes et de leur fabrication, repose sur les concepts de code personnalisable et de code personnalisé. Nous avons présenté dans la section 2.5.2 des constructions Java permettant d'implémenter ces concepts. Notre objectif est de fournir aux développeurs d'applications pour cartes à puce les moyens méthodologiques et techniques pour réaliser une personnalisation à grande échelle. Ceci implique l'automatisation du processus qui permet d'obtenir des déclinaisons personnalisées de leurs applications à partir des données utilisateurs.

Notre problématique dépasse la thématique de la personnalisation dans le sens où elle relève également de l'automatisation de la construction de logiciels. Pour cette raison, nous nous intéressons dans ce chapitre aux différentes approches de l'ingénierie logicielle susceptibles de nous aider à générer le code personnalisé des applications embarquées. Le but de ce chapitre est de présenter certaines technologies actuelles et d'évaluer leur pertinence par rapport à notre problématique.

Ce chapitre est organisé comme suit :

- La section 3.1 présente une vue d'ensemble des lignes de produits logiciels. La personnalisation pouvant être considérée dans une certaine mesure comme une forme de configuration de masse, il nous semble important de nous intéresser aux mécanismes au cœur des lignes de produits, et notamment à ceux permettant de gérer la variabilité des produits.
- La section 3.2 présente la programmation orientée aspects. Notre ambition est de traiter la personnalisation comme une préoccupation particulière, et de minimiser son impact sur le reste du développement de l'application. Nous nous intéressons donc à la programmation orientée aspects, qui place justement la séparation des préoccupations au cœur du développement logiciel.
- La section 3.3 présente deux approches génératives, qui suscitent depuis quelques années un intérêt grandissant aussi bien du monde de l'industrie que du monde académique. Puisque notre problématique concerne l'automatisation d'un processus de développement de logiciel, les concepts véhiculés par ces approches nous semblent particulièrement pertinents.
- La section 3.4 conclut ce chapitre.

3.1 Lignes de produits logiciels

Présent dans l'industrie traditionnelle depuis très longtemps[20], le concept des lignes de produits est relativement récent dans l'industrie logicielle. Cette section présente les aspects technologiques liés à la variabilité des produits après avoir introduit leurs motivations et principes. Les corrélations entre les lignes de produits et la personnalisation sont établies.

3.1.1 Motivations

Le paysage logiciel ne cesse de se diversifier et de s'enrichir. Aujourd'hui, la capacité d'un fournisseur à décliner efficacement des versions différentes d'un produit ou d'une offre selon les besoins particuliers d'un client ou la spécificité d'une plate-forme d'exécution est devenue essentielle. Dans cette perspective, les éditeurs de logiciels se doivent donc d'optimiser l'effort requis pour créer ou adapter un produit, afin de minimiser la période entre le début de son développement et sa disponibilité sur le marché[21].

Le gain de productivité est une motivation première des lignes de produits logiciels. L'idée fondamentale de cette approche est de définir des familles de systèmes logiciels en capitalisant sur les similarités entre différents produits d'une même domaine [57, 58]. Ainsi, sur la base d'un ensemble d'artefacts implémentant ces similarités, des mécanismes de variabilité peuvent être mis en œuvre pour décliner efficacement des configurations différentes.

[20] Le succès des lignes d'assemblage dans l'industrie automobile date de 1908, lorsque Henry Ford a commencé à les utiliser pour la production du Model T.
[21] *Time-to-market* dans le vocabulaire anglo-saxon.

La réutilisation, thème récurrent de l'industrie du logiciel, est au cœur des lignes de produits. Malgré de multiples paradigmes tels que les cadres de conception orientés objet[22] [59] ou l'ingénierie à base de composants [60], réutiliser des artefacts logiciels dans différents systèmes ou produits développés au sein d'une même organisation est généralement difficile. Deux caractéristiques essentielles distinguent les lignes de produits des paradigmes précédents. D'une part, les lignes de produits ne reposent pas sur une approche uniquement centrée sur l'aspect technologique de la réutilisation mais sur une approche globale qui inclut également les préoccupations organisationnelles et commerciales [61]. D'autre part, la réutilisation est planifiée plutôt qu'opportuniste [62].

3.1.2 Principes

Dans [63], Clements et Northrop définissent les lignes de produits logiciels de la façon suivante :

> *« A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. »*

Selon cette définition, une ligne de produits logiciels regroupe donc un ensemble de produits d'un domaine particulier qui partagent suffisamment de caractéristiques ou de fonctionnalités pour être dérivés d'un même ensemble d'artefacts logiciels de base.

La Figure 3-1 schématise une ligne de produits selon la structure globale proposée par le *Software Engineering Institute*. Cette structure, expliquée dans [63] et mise à jour régulièrement par le biais du site Internet [64], identifie les différentes problématiques et pratiques relatives à la mise en œuvre d'une ligne de produits au sein d'une organisation. Trois activités essentielles sont explicitées.

- *Développement des artefacts logiciels de base (ou ingénierie du domaine)*. Il s'agit de fixer clairement les objectifs de la ligne de produits : quelles sont les fonctionnalités supportées, les plates-formes visées, etc. Sur la base de cette analyse, le développement des artefacts logiciels de base peut commencer.
- *Développement de produits (ou ingénierie applicative)*. Le développement d'un produit consiste à assembler et configurer plusieurs artefacts de base dans le but de répondre à une attente particulière d'un marché ou d'un client.
- *Gestion de la ligne de produits*. La gestion consiste à attribuer des ressources aux différentes activités, à coordonner et superviser les acteurs et les résultats de chacune de ces activités.

[22] *Object-oriented frameworks* dans le vocabulaire anglo-saxon.

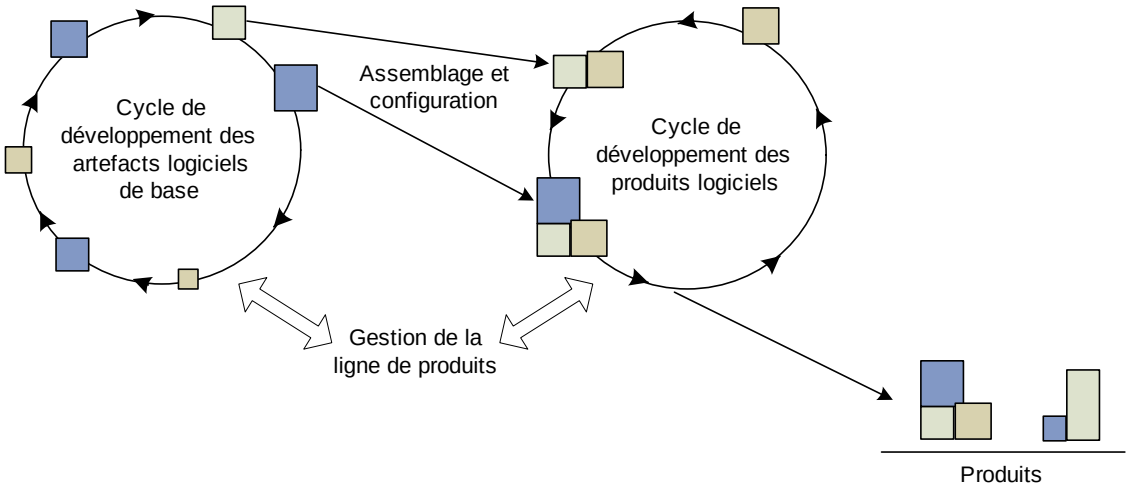


Figure 3-1 : Structure globale d'une ligne de produits

Chacune de ces trois activités est itérative. Par exemple, le développement d'un produit sur la base d'un ensemble d'artefacts de base peut entraîner des retours sur ces artefacts et motiver des ajustements de ceux-ci. De même, la découverte d'une nouvelle similarité non détectée précédemment entre deux produits peut engendrer la création d'un nouvel artefact de base.

Différentes méthodologies sont aujourd'hui disponibles pour supporter la mise en oeuvre des lignes de produits. On peut notamment citer COPA [65], FAST [66], FORM [67], KobrA [68] et QADA [69]. Chacune de ces méthodologies couvre un ensemble plus ou moins exhaustif d'aspects liés aux lignes de produits, comme le fait apparaître la comparaison réalisée dans [70].

Une ligne de produits repose sur une architecture logicielle qui définit ses propriétés structurelles, c'est-à-dire la manière dont les artefacts de base sont développés et assemblés. Le choix d'une architecture fixe les technologies sous-jacentes à la ligne de produits. Une préoccupation majeure de la conception des architectures est la mise en œuvre de mécanismes pour supporter la variabilité des produits.

3.1.3 Variabilité et personnalisation

3.1.3.1 La variabilité dans les lignes de produits logiciels

La variabilité d'un système ou d'un artefact logiciel est sa capacité à être modifié, adapté ou configuré pour être utilisé dans un contexte spécifique [71]. Le concept de variabilité correspond à des décisions de conception retardées : un point de variation dans un système définit un endroit pour lequel le choix entre plusieurs alternatives n'est pas fixé instantanément mais lié à une prise de décision ultérieure. Ainsi, les membres d'une ligne de produits peuvent être assimilés aux feuilles d'un arbre de décision dont chaque nœud correspond à une prise de décision comme par exemple le choix ou la configuration particulière d'un artefact de base. La notion de variabilité s'applique à la totalité du cycle de vie du système, de la conception à l'exécution.

La gestion de la variabilité constitue la problématique architecturale majeure de la construction d'une ligne de produits. Dans [71, 61], cette gestion est décomposée en quatre tâches principales :

- *Identification des points de variabilité*. La définition des points de variabilité est liée à l'analyse de ce qui diffère d'un produit à l'autre. Chaque caractéristique d'un système devient un point de standardisation ou un point de spécialisation au sein de la ligne de produits.
- *Collection des variants*. Chaque point de variation est associé à un ensemble d'alternatives que l'on appelle les variants. La collection peut être finie ou extensible.
- *Sélection des mécanismes appropriés*. Le mécanisme utilisé pour un point de variabilité dépend (*i*) du niveau d'abstraction auquel le point de variabilité est introduit, (*ii*) du moment auquel un variant donné doit être lié au système et (*iii*) de la manière dont de nouveaux variants peuvent, ou non, être ajoutés.
- *Liaison du système aux variants*. Du point de vue du système, la liaison peut être externe ou interne. Dans le cas d'une liaison externe, le choix est réalisé par le concepteur ou l'utilisateur qui réalise un choix motivé. Dans le cas d'une liaison interne, le système dispose de suffisamment d'informations pour se lier automatiquement à un variant spécifique.

La variabilité d'une ligne de produits logiciels peut être représentée de différentes manières. La modélisation des caractéristiques[23] consiste à représenter la variabilité sous la forme d'arbres de caractéristiques obligatoires, optionnelles ou alternatives. Cette approche a été introduite en 1990 par Kang et al. dans la méthode FODA [72] puis raffinée et étendue ultérieurement entre autres dans [67, 71, 73]. Ce type d'approche permet une modélisation indépendante des mécanismes de variabilité eux-mêmes.

D'autres approches s'appuient sur le standard de modélisation UML (*Unified Modeling Language*, [74, 75]), mais en font une utilisation différente. La méthode KobrA [68] utilise un stéréotype spécifique pour spécifier qu'un composant est un variant. Ziadi et al. dans [76, 77, 78] proposent un ensemble d'extensions pour modéliser à la fois les aspects dynamiques et statiques. Ces extensions sont définies au travers d'un profil UML. Au contraire de KobrA, cette approche autorise un certain degré d'automatisation de la dérivation des produits. L'approche décrite par Clauß dans [79] repose également sur UML, et introduit encore d'autres notations. Enfin, des travaux récents s'appuient sur les approches dirigées par les modèles (voir section 3.3.2) et la MDA (Model Driven Architecture, [80]) en particulier pour organiser et automatiser la construction de lignes de produits [81, 82].

D'un point de vue technique, l'implémentation de la variabilité repose sur l'utilisation de concepts et mécanismes classiques de l'ingénierie logicielle. Jacobson et al. dans [83] ainsi que Svahnberg et Bosch dans [84] recensent différents mécanismes en précisant d'une part le niveau d'abstraction auquel ils sont applicables, et d'autre part quels types de variation

[23] *Feature modeling* dans le vocabulaire anglo-saxon.

ils permettent. Ces mécanismes, liés notamment aux techniques de compilation, de programmation ou de construction d'architectures incluent des concepts tels que l'héritage, la configuration, le paramétrage, les patrons[24] ou encore la génération de code. Dans [85], Anastasopoulos et Gacek élargissent ces listes de mécanismes en introduisant des concepts tels que le chargement dynamique de classes en Java, la réflexion ou la programmation orientée aspects.

3.1.3.2 La personnalisation, une forme de variabilité

Dans l'industrie traditionnelle du logiciel, un produit est empaqueté dans une unité logique, puis distribué massivement par le biais de téléchargements sur l'Internet ou de réplication sur des supports physiques tels que les cédéroms. Dans l'industrie de la carte à puce, chaque carte produite est unique : sur la chaîne de fabrication, une carte donnée ne contient pas tout à fait les mêmes informations que la carte précédente ni que la suivante. Si cette particularité distingue l'industrie de la carte à puce de celle traditionnelle du logiciel, elle la rapproche d'autres types d'industries comme celle de l'automobile.

Au cours du XX^{ème} siècle, l'industrie automobile a évolué pour produire à une échelle toujours plus grande des véhicules de plus en plus différents. Cette industrie a appris à configurer, assembler et standardiser des composants pour parvenir à une plus grande automatisation de la production. Aujourd'hui, les chaînes d'assemblage ont atteint un tel degré de maturité que chaque véhicule sur une chaîne est différent de celui qui le précède et de celui qui le suit. La couleur, l'intérieur, les options ou la motorisation sont définis lors de la commande par le client chez le concessionnaire. Cette diversité des automobiles fabriquées signifie que tous les composants particuliers d'une voiture donnée doivent être au bon endroit, au bon moment pour être intégrés sur la chaîne de production. L'industrie automobile a donc su mettre en œuvre des processus pour réaliser à une grande échelle une configuration de masse extrêmement aboutie.

L'analogie avec notre problématique de personnalisation de logiciels pour cartes à puce est évidente. Nous cherchons à évoluer d'une personnalisation basique, dans laquelle la différence entre chaque carte est minime, à une personnalisation beaucoup plus riche avec une réelle particularisation du logiciel embarqué. Les lignes de produits étant la transposition des méthodes de l'industrie traditionnelle à l'industrie du logiciel, l'exemple de l'automobile nous permet d'envisager la variabilité comme un moyen de réalisation de la personnalisation. Le choix entre plusieurs variants pouvant, par exemple, être défini en fonction des profils utilisateurs.

Cependant, le grain de configuration actuel dans les lignes de produits logiciels est encore beaucoup trop gros pour que ces dernières supportent la personnalisation telle que nous l'avons décrite dans les sections 1.1 et 2.1. Alors que l'objectif visé au travers des lignes de produits est de décliner différents logiciels à partir d'une même base, le nôtre est de décliner un nombre très important de configurations d'un seul logiciel. À une échelle différente, ces deux problématiques sont donc similaires.

[24] *Templates* dans le vocabulaire anglo-saxon.

Il est intéressant de constater que le concept des lignes de produits logiciels est particulièrement pertinent dans le contexte plus général de la configuration des cartes à puce – qui englobe d'autres préoccupations que la personnalisation.

3.2 Programmation orientée aspects (AOP)

3.2.1 Motivations

Le besoin de ne traiter qu'un seul problème important à la fois a été identifié par Dijkstra [50] comme le principe de *séparation des préoccupations*. Les différentes préoccupations des concepteurs d'un système apparaissent comme des critères fondamentaux pour organiser et décomposer l'analyse, la conception et l'implémentation du système en un ensemble d'éléments compréhensibles et plus facilement manipulables. Ainsi, un des bons principes de l'ingénierie logicielle depuis près de trente ans est d'essayer de programmer chaque problème important de manière localisée. Cette approche permet d'une part de comprendre plus facilement comment un problème est traité, et d'autre part de mieux analyser, améliorer, corriger, réutiliser ou maintenir la solution à ce problème.

La plupart des notations d'analyse et de conception ainsi que la plupart des langages de programmation fournissent des constructions pour structurer les descriptions de systèmes comme des compositions d'unités modulaires de granularité plus fine. Beaucoup sont concentrées sur la recherche et la composition d'unités fonctionnelles, exprimées par le biais d'objets, de modules, de procédures, etc. [51]. Cependant certains problèmes sont très difficiles, voire impossibles, à exprimer de manière localisée avec ces seules abstractions traditionnelles. Des préoccupations telles que le contrôle de la sécurité, de la persistance, des transactions peuvent avoir une incidence transversale sur des ensembles divers de composants fonctionnels. Il en résulte donc des fragments de code disséminés et enchevêtrés à divers endroits, comme le présente la Figure 3-2.

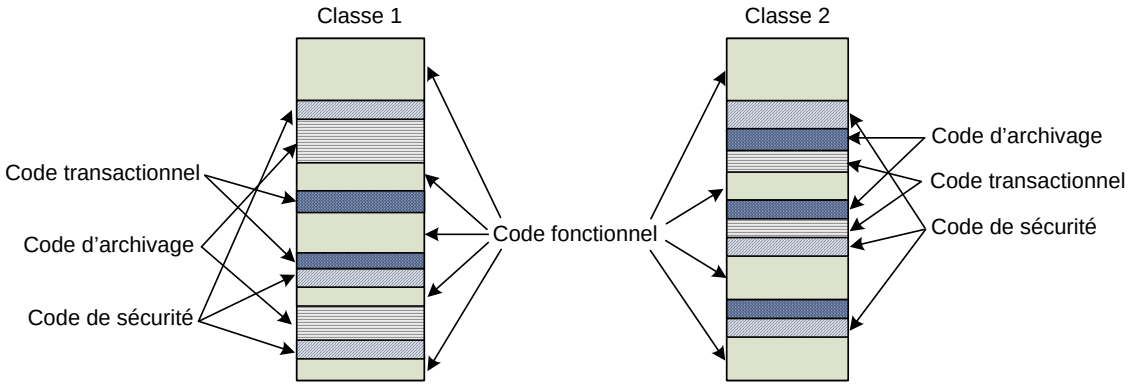


Figure 3-2 : Programmation traditionnelle (enchevêtrement des aspects)

Cette dernière observation est au cœur du principe de *programmation orientée aspects (*AOP)[25] [51] [86] introduit en 1997 par des chercheurs du Xerox PARC (*Palo Alto Research Center*). L'AOP a pour but de fournir les méthodes et techniques pour :

- *Décomposer la définition des problèmes* en un ensemble de composants fonctionnels d'une part et un ensemble d'aspects transversaux à ces composants d'autre part.
- *Lier les composants fonctionnels et les aspects* pour produire l'implémentation du système.

3.2.2 Principes

Au contraire de la programmation traditionnelle, l'AOP permet d'isoler la définition de chacun des aspects, comme le présente la Figure 3-3. Des mécanismes de *tissage* permettent d'intégrer l'implémentation de chacun des aspects dans les modules fonctionnels avant la compilation. Cette approche rend plus pratique la gestion des aspects – définition, mise à jour, maintenance, etc.

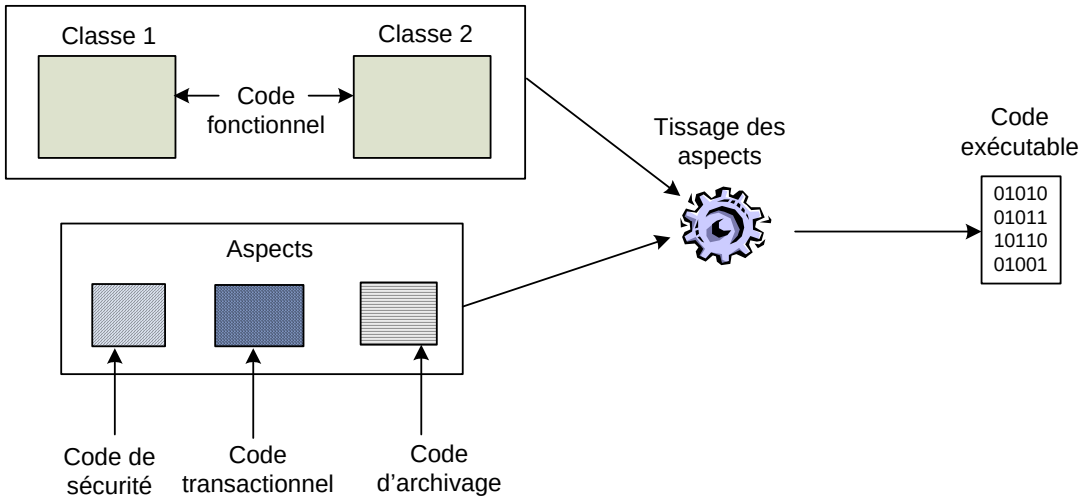


Figure 3-3 : Programmation orientée aspects (AOP)

Réaliser la séparation des préoccupations par le biais de l'AOP requiert la mise en œuvre de méthodes et techniques concrètes pour :

- *Définir quelles préoccupations doivent devenir des aspects*. Une préoccupation qui ne peut être encapsulée dans un module fonctionnel mais qui au contraire affecte de manière transversale plusieurs fonctions du système est susceptible d'être traitée comme un aspect. On peut globalement dissocier deux types d'aspects : ceux qui sont spécifiques au domaine de l'application et ceux qui sont plus généraux comme la sécurité, l'archivage, etc.

[25] *Aspect-Oriented Programming (AOP)* dans le vocabulaire anglo-saxon.

- *Définir les mécanismes de décomposition à utiliser*. Si les aspects ne peuvent être séparés proprement en utilisant les constructions classiques de la programmation, alors il est nécessaire de mettre en œuvre d'autres mécanismes de composition. Ces mécanismes de tissage doivent permettre un couplage libre entre les différentes descriptions partielles du système, et dans la mesure du possible supporter des liaisons de différents types à différents moments – statiques ou dynamiques lors de la conception ou de l'exécution.
- *Définir comment représenter les aspects*. Dans les cas les plus simples l'utilisation des langages conventionnels peut suffire. Dans les autres cas, l'utilisation d'extensions aux langages peut s'avérer nécessaire.

AspectJ™ [87] est un environnement pour l'AOP développé par le Xerox PARC. *AspectJ* est en fait une extension orientée aspects du langage Java, désormais intégrée au projet *Eclipse* [88], qui définit une terminologie pour l'AOP [89].

Join Point. Un point de jonction définit un endroit bien précis dans le flux du programme – appel de méthode, exécution d'une exception ou affectation d'une valeur à un champ par exemple – dans lequel un aspect peut être inséré. Dans le cas d'un appel de méthode, un point de jonction englobe l'ensemble des actions qui composent l'appel à la méthode, débutant après l'évaluation des arguments et finissant par le *return* de la méthode.

Pointcut. Un point de coupe est une collection de certains points de jonction définis dans le flux du programme. Un point de coupe peut potentiellement réunir différents types de points de jonction. Par exemple, le point de coupe `call(void Point.setX(int))` désigne tous les points de jonction qui sont un appel à la méthode de la classe `Point` ayant pour signature void `setX(int)`.

Advice. Ce concept réunit un point d'action et un morceau de code qui sera exécuté à chaque point de jointure identifié par le point d'action. Le code peut être exécuté avant ou après le point de jointure.

AspectJ décrit ses aspects en utilisant des extensions du langage Java. Ces aspects sont traités par le tisseur d'aspects juste avant l'étape de compilation. Le code résultant de cette compilation reste du *bytecode* Java standard. D'autres implémentations de l'approche orientée aspects sont possibles, comme par exemple *Hyper/J*™ [90] d'*IBM Research* qui met en œuvre une séparation multi-dimensionnelle des préoccupations, le projet *JAC* (*Java Aspects Component*) du consortium *ObjectWeb* [91] ou encore une implémentation pour *.Net* [92] qui permet le tissage des aspects durant l'exécution par le biais de mécanismes de réflexion.

3.2.3 Personnalisation et AOP

Dans la section 2.1, nous avons défini la personnalisation logicielle et décrit le fait qu'elle concerne notamment les fonctionnalités à implémenter, les valeurs à affecter aux paramètres, le déploiement, l'installation, la spécification des propriétés non fonctionnelles, etc. Dans la section 2.5.1, nous avons présenté la séparation des

préoccupations comme un fondement méthodologique majeur de notre démarche. L'idée de traiter la personnalisation comme un aspect est donc intéressante.

En fait, l'AOP constitue une alternative à l'approche proposée dans la section 2.5 – notions de code personnalisable et de code personnalisé –, mais pas une solution pour l'automatisation du processus de personnalisation où les modifications à apporter dans le code sont différentes pour chaque utilisateur. Pour réaliser la personnalisation par le biais de l' AOP, il serait en fait nécessaire de mettre en œuvre des mécanismes pour paramétrer l'aspect personnalisation avec les profils utilisateurs. Dès lors, le défi serait de réussir à automatiser ce paramétrage de l'aspect – on pourrait parler de personnalisation d'aspect –, ce qui est à peu près équivalent à la problématique que nous essayons de résoudre actuellement.

Le principal intérêt de notre approche par rapport une utilisation de l'AOP réside dans la minimalisation de ce qui est réellement spécifique à un utilisateur par le biais des concepts de code personnalisable et d'activateur. Ce point est essentiel dans la perspective de l'industrialisation du processus.

3.3 Approches génératives

3.3.1 Programmation générative (GP)

3.3.1.1 Introduction et définitions

La programmation orientée objet constitue indiscutablement une avancée majeure de l'ingénierie logicielle. Cependant, dans le contexte de la conception globale d'un système, la granularité très fine des objets ne permet pas de satisfaire tous les besoins en terme de réutilisabilité. Les cadres de conception orientés objets fournissent un grain de réutilisation plus pertinent, mais ils sont difficiles d'une part à maintenir et d'autre part à intégrer avec d'autres cadres de conceptions. Les patrons de conception capturent certes une connaissance réutilisable mais ne sont pas un moyen d'implémentation.

La tendance actuelle, comme nous l'avons introduit dans la section 3.1.1, est de raisonner non plus en systèmes individuels mais en familles de systèmes, afin de placer concrètement la réutilisation au centre des préoccupations de l'ingénierie logicielle. Dans [93], Czarnecki et Eiseneicker définissent la programmation générative (GP) de la façon suivante :

> *« Generative programming is about modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific language. »*

Selon cette définition, la programmation générative consiste donc à modéliser et implémenter des familles de systèmes de telle manière qu'un système donné puisse être généré automatiquement à partir d'une spécification exprimée par le biais d'un ou plusieurs langages spécifiques, aussi bien graphique que textuel.

Par rapport aux lignes de produits traitées dans la section 3.1, la programmation générative se concentre sur l'automatisation de la création des membres d'une famille de systèmes. La terminologie utilisée dans le contexte de la GP diffère légèrement de celle utilisée dans le contexte des lignes de produits. Plus précisément, alors que les familles de systèmes visent principalement la construction de systèmes à partir d'artefacts de bases, l'ingénierie des lignes de produits ajoute une dimension de gestion des caractéristiques des produits dans une perspective plus commerciale. Ainsi, le *développement des artefacts de base* dans la section 3.1.2 correspond ici à l'*ingénierie du domaine* et celui des *produits logiciels* à l'*ingénierie applicative*.

Le but d'un langage spécifique à un domaine (DSL, *Domain Specific Language*) est de fournir des abstractions permettant de représenter des concepts du domaine qu'il vise. Plusieurs propriétés sont essentielles pour un DSL : *(i)* les concepts qui apparaissent dans un DSL doivent être compréhensibles par les personnes familières avec le domaine visé, *(ii)* le DSL doit fournir une grammaire bien définie permettant de combiner les différents concepts selon un ensemble de règles, *(iii)* une expression correctement formée ne doit pas être ambiguë, de manière à ce que son traitement soit automatisable et *(iv)* un DSL peut indifféremment reposer sur une notation graphique ou textuelle [94].

3.3.1.2 Principes

Le concept clef de la programmation générative est celui de la mise en relation, par le biais de mécanismes de configuration, de projection ou de transformation entre un *espace problème* et un *espace solution*[26] [93]. L'espace problème est un ensemble d'abstractions spécifiques à un domaine – typiquement des DSLs – qui peuvent être utilisées pour spécifier un membre de la famille de systèmes. Le fait que ces abstractions soient spécifiques signifie qu'elles permettent aux concepteurs d'exprimer leurs besoins dans des termes qui sont naturels pour leur domaine. L'espace solution est un ensemble d'abstractions plutôt technologiques ou techniques, qui peuvent être instanciées pour créer l'implémentation du système spécifié en utilisant les abstractions de l'espace problème. La Figure 3-4 présente cette approche.

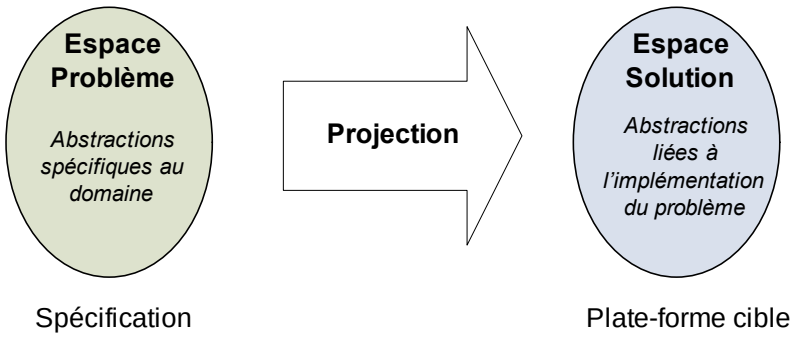


Figure 3-4 : Projection entre un espace problème et un espace solution

[26] *Problem Space* et *Solution Space* dans le vocabulaire anglo-saxon.

Au moins deux perspectives de projections entre les espaces problème et solution sont possibles :

- *Projection du type configuration*. L'espace problème est constitué de concepts spécifiques au domaine et de caractéristiques. L'espace solution est constitué d'un ensemble de composants implémentant ces caractéristiques. La projection entre les espaces problème et solution est en fait une étape de configuration des éléments du premier. L'espace problème et la configuration sont enrichis par la définition de propriétés qui déterminent des combinaisons illégales de caractéristiques, des dépendances entre caractéristiques, de réglages par défaut, de règles de construction, etc.
- *Projection du type transformation*. L'espace problème est représenté par un DSL alors que l'espace solution est basé sur des langages ou architectures d'implémentation. La projection entre les espaces problème et solution se fait par le biais de transformations ou de générateurs.

3.3.1.3 Mise en oeuvre

La GP introduit des concepts relativement généraux dont la mise en œuvre repose sur des technologies diverses. Le choix d'une technologie par rapport à une autre pour implémenter les éléments d'une approche générative dépend du contexte technique ou stratégique. Czarnecki et Eisenecker recensent dans [93] un large éventail de technologies. La Figure 3-5 s'appuie sur leur travail pour présenter de manière non exhaustive certaines catégories de technologies.

Éléments de GP	Catégories de technologies
Espace Problème *(définition de DSLs)*	Langages de programmation, extension des langages de programmation, nouveaux langages textuels, langages graphiques, assistants et configurateurs, etc.
Projections	Configurateurs ou générateurs. Les générateurs peuvent être implémentés en utilisant des systèmes de transformations, des approches à base de patrons, des langages avec support pour la meta-programmation, etc.
Espace Solution *(technologies d'implémentation)*	Technologies orientées objets, architectures à composants, approches orientées aspects, etc.

Figure 3-5 : Catégories de technologies pour la GP

Le terme GP ne désigne donc pas une technologie d'ingénierie logicielle mais plutôt une approche particulière pour rendre la production de logiciels plus automatisée. La section suivante introduit une autre approche de développement logiciel dont le but est d'élever le niveau d'abstraction en utilisant les modèles comme artefacts de développement.

3.3.2 Développement dirigé par les modèles (MDD)

3.3.2.1 Motivations et définitions

Il est communément admis que la spécification d'un système répond à la question « *quoi ?* » et son implémentation répond à la question « *comment ?* ». Ces notions de spécification et d'implémentation sont relatives : si le code source correspond à l'implémentation du point de vue du développeur, ce même code source est une spécification du point de vue du compilateur. En général, le code source est considéré comme l'implémentation car il est le niveau le plus bas restant encore compréhensible. Un des problèmes importants de l'ingénierie logicielle provient du fait que ce code source est souvent l'unique abstraction formelle de la spécification fonctionnelle. En effet, la plupart des méthodes et outils de développement traditionnels ne capture pas, ou alors de manière informelle, l'information au-dessus du code source. Cette information, qui correspond à des étapes successives de raffinages et d'abstractions réalisées par le développeur, est le plus souvent perdue [94].

Une motivation essentielle du développement logiciel dirigé par les modèles (MDD, *Model Driven Development*) est d'élever le niveau d'abstraction pour réduire le fossé entre la problématique d'un système et son implémentation. Pour atteindre cet objectif, le MDD essaie de réduire la différence de sémantique entre les concepts réels d'un domaine et leur matérialisation avec les architectures et technologies de programmation traditionnelles. Dans cette perspective, il considère les modèles – textuels ou graphiques – comme des entités de première classe, capables d'être calculées, transformées ou interprétées par le biais de mécanismes automatisés. Si le précepte « *tout est un objet* » de la programmation orientée objets a conduit à de véritables progrès dans l'ingénierie logicielle ces vingt dernières années, MDD adopte comme précepte de base « *tout est un modèle* » [95] [96].

Dans [97], Bézivin donne la définition suivante d'un modèle :

> *« A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. The answers provided by the model should be the same as those given by the system itself, on the condition that questions are within the domain defined by the general goal of the system. »*

Selon cette définition, un modèle est donc une simplification d'un système construite avec une intention particulière. Le modèle doit pouvoir être utilisé à la place du système pour répondre à certaines questions sur celui-ci. Les réponses fournies par le modèle doivent être identiques à celles qui seraient données par le système lui-même à la condition que ces questions restent dans le domaine du système.

Un modèle est une abstraction de la réalité dans le sens où il ne peut pas représenter tous les aspects de la réalité [98]. Un modèle cache donc certains aspects ou détails du système pour présenter une description simplifiée de certains autres.

Pour que les modèles puissent être considérés comme des entités de première classe, leur interprétation doit être non ambiguë. Les langages de modélisation doivent donc être

définis aussi rigoureusement que le sont les langages de programmation. De manière générale, la syntaxe abstraite d'un langage définit à la fois les éléments qui forment ce langage et les règles de composition de ces éléments. La *méta-modélisation* est une approche qui permet de définir des langages de modélisation. En précisant de manière formelle un ensemble de concepts et leurs relations, un méta-modèle définit la terminologie à utiliser pour définir des modèles [97]. Parce qu'il est impossible d'imaginer un méta-modèle universel, un méta-modèle doit être défini dans un objectif précis, étroitement lié à un domaine et à une utilisation donnés [99].

Les deux concepts clefs du MDD sont la modélisation et la transformation :

- *Modélisation*. La modélisation est l'art de spécifier un système en utilisant des modèles et méta-modèles. Différents aspects ou détails du système peuvent être abstraits dans différents modèles. La réunion de l'ensemble des modèles fournit une représentation abstraite complète du système.
- *Transformation*. Une fois les modèles construits, ils peuvent être traités par le biais d'outils automatiques ou semi-automatiques. Les modèles peuvent être transformés plusieurs fois successivement pour aboutir à l'implémentation des fonctionnalités dont ils sont une abstraction.

3.3.2.2 Le MDD selon l'OMG

L'OMG (*Object Management Group*) est actif depuis de nombreuses années dans le domaine de la modélisation, notamment par le biais du standard UML (*Unified Modeling Language*) [74, 75] qui a établi un ensemble très largement utilisé de concepts, de conventions et de notations graphiques pour la conception orientée objet. En plus d'UML, l'OMG définit d'autres standards de modélisation, dont le MOF (*Meta Object Facility,* [100, 101]) qui est un méta-modèle commun pour l'ensemble des autres spécifications de l'OMG.

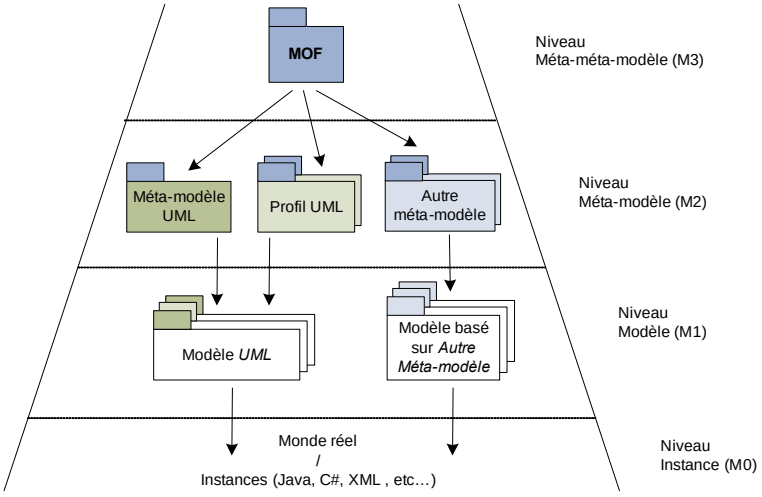


Figure 3-6 : Architecture de modélisation à quatre niveaux

S'appuyant sur une architecture décomposée en quatre niveaux, le MOF fournit un cadre structurant pour définir et exploiter des méta-modèles. La Figure 3-6 présente cette architecture. Au niveau méta-méta-modèle (M3), le MOF – qui suffit à se décrire lui-même – permet de définir des langages de modélisation au niveau M2, celui des méta-modèles. On trouve par exemple à ce niveau M2 le méta-modèle UML ou la définition de profils[27] UML. Les méta-modèles du niveau M2 permettent de définir la sémantique des modèles du niveau M1. Ces modèles constituent des abstractions de la représentation informatisée du monde réel. Le niveau M0 contient les instances.

En 2001, l'OMG a introduit l'approche MDA (*Model Driven Architecture* [80, 102, 103, 104]) pour la spécification des systèmes logiciels par le biais de modèles prônant une séparation claire entre les préoccupations fonctionnelles et technologiques. Ainsi, la MDA distingue les PIMs (*Platform Independant Model*) et les PSMs (*Platform Specific Model*). Tandis que les PIMs définissent la fonctionnalité d'un système en utilisant un langage de modélisation indépendant de toute plate-forme, les PSMs représentent leur projection sur une technologie ou une plate-forme spécifique – architecture à composants particulière, langage de programmation donné, etc.. (Figure 3-7).

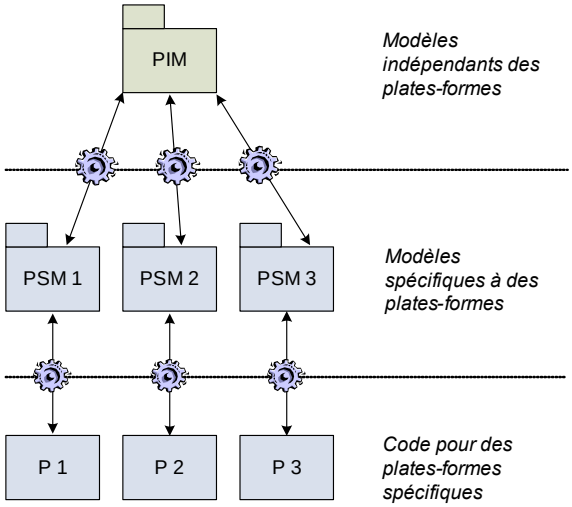


Figure 3-7 : L'approche MDA

L'approche MDA est fortement articulée autour de spécifications de l'OMG. Aux côtés de MOF et d'UML, on retrouve également :

- XMI (*XML Metadata Interchange* [105, 106], voir section 4.2.2) pour la sérialisation et l'échange de modèles,
- CWM (*Common Warehouse Metamodel* [107]) pour la définition, la manipulation et le stockage des données,
- QVT (MOF 2.0 *Query / Views / Transformations* [108], voir section 4.3.3) pour la manipulation et la transformation de modèles.

[27] Un profil UML est une extension du méta-modèle UML.

Les prétentions revendiquées par l'OMG concernant la MDA lors de son apparition en 2001 ont suscité un très grand intérêt à la fois du monde académique et du monde industriel. Initialement, la MDA était très étroitement liée à l'utilisation d'UML [103]. Or l'idée qu'UML puisse être utilisé de manière universelle pour exprimer des PIMs quel que soit le domaine d'application, avant de les projeter sur différentes plates-formes ou technologies, est aujourd'hui considérée peu réaliste [109, 94]. Depuis la fin de 2001, l'OMG a donc évolué dans sa manière de définir la MDA et de présenter cette approche.

En Août 2004, l'OMG a voté une nouvelle définition officielle de MDA qui stipule que « *MDA propose un ensemble non propriétaire de standards spécifiant des technologies interopérables pour mettre en pratique le développement dirigé par les modèles avec des transformations automatisées. […] MDA ne repose pas nécessairement sur* UML*, mais en tant qu'instance de MDD, MDA implique l'utilisation de modèles dans le développement. Ceci signifie qu'au moins un langage de modélisation doit être utilisé. Un langage de modélisation dans le contexte de MDA doit être décrit par le langage MOF, afin que les méta-données soient compréhensibles de manière standardisée, ce qui est une pré condition pour l'implémentation de transformations automatisées.* »

Progressivement, la notion de plate-forme a évolué au cours de ces dernières années. La notion de dépendance par rapport à une plate-forme est en fait une notion relative, qui dépend du niveau d'abstraction dans lequel on se place. Le PSM d'un concepteur peut être le PIM d'un autre.

3.3.2.3 Autres approches pour le MDD

Si la MDA n'est finalement qu'une instance de MDD, sa popularité a démontré la nécessité d'un changement profond des pratiques de développement de logiciels. Un mérite essentiel de MDA, au-delà de toute considération technologique, est d'avoir popularisé l'ingénierie logicielle dirigée par les modèles et d'avoir ouvert la voie à d'autres approches pour le MDD. On peut notamment citer plusieurs initiatives importantes :

- *Le projet européen ModelWare (MODELing Solutions for SoftWARE)* [110]. Ce projet vise à construire au-dessus des spécifications de l'OMG un bus d'interopérabilité sur lequel vont s'échanger non plus du code comme dans les intergiciels classiques mais des modèles, sérialisés dans le format XMI par exemple. Ce projet a débuté en août 2004 pour une durée de deux ans. Il rassemble des acteurs européens, académiques ou industriels.
- *Le MDSD (Model Driven Software Development)* [111, 112]. Le MDSD est défini comme un ensemble de techniques permettant d'appliquer l'approche Agile [113] dans le contexte de développement logiciels industriels de grande échelle. MDSD est notamment étroitement lié au projet open source GMT (*Generative Model Transformer* [114]) dont le but est de construire et d'assembler un ensemble d'outils pour le MDD. GMT est intégré au projet *Eclipse* [88]. GMT se repose fortement sur XMI comme support pour l'interopérabilité entre les différents outils.
- *L'approche d'IBM.* Dans son manifeste pour MDA [115], IBM adhère aux grandes idées de la MDA, tout en insistant sur la notion de *représentation directe* : plus un

concept peut être représenté directement, plus la spécification d'un système est aisée. Cette approche repose sur l'utilisation de DSLs pour manipuler de préférence des concepts spécifiques liés au domaine plutôt que des concepts génériques purement logiciels comme les classes, les composants, les attributs, etc.

Ces différentes approches ont en commun de considérer la notion de DSL comme centrale au MDD. Dans une certaine mesure, cette stratégie rapproche le MDD de la programmation générative (GP).

3.3.3 GP/GSD et MDD comme supports d'une approche pour la personnalisation

3.3.3.1 Complémentarité entre GSD et MDD

Profitant des recherches menées par des communautés très dynamiques, les deux approches se sont considérablement développées au cours des dernières années. Czarnecki décrit dans [116] une vision actualisée du domaine de la programmation générative, qu'il nomme désormais développement logiciel génératif (GSD, *Generative Software Development*). Se basant sur l'idée qu'un espace solution dans un contexte peut être l'espace problème dans un autre, il raffine la GP en introduisant les notions de chaînes de projections, puis celles de projections multiples et de projections alternatives. Ces différents types de projections peuvent être liés par exemple à différentes vues d'un système, ce qui est notamment intéressant dans le cas d'approches orientées aspects. Dans le MDD, des alternatives à l'utilisation d'UML suscitent un intérêt grandissant : la tendance au cours de ces dernières années est à la représentation de modèles par le biais de DSLs légers et appropriés à un contexte donné, plutôt qu'à l'utilisation d'un langage de modélisation universel.

Les approches GSD et MDD ont donc en commun un ensemble de concepts, mais aussi d'objectifs, comme celui d'automatiser les tâches de développement qui ne relèvent pas de la créativité des concepteurs. La différence principale semble être la notion de lignes de produits – ou de familles de programmes suivant la terminologie utilisée – qui est au cœur du GSD, mais pas indispensable dans MDD. En fait, plutôt qu'une comparaison entre ces approches, il est plus intéressant de s'attarder sur leur complémentarité :

- *GSD support du MDD*. L'analyse du domaine et la définition précise des objectifs sont des étapes clef du GSD, notamment pour la conception des DSLs. Cette approche peut être utile dans un contexte MDA par exemple pour la définition des PIMs et PSMs.
- *MDD support du GSD*. Les efforts fournis dans le cadre du MDD pour proposer des cadres de modélisation et des outils de transformation sont très utiles pour mettre réellement en pratique les concepts du GSD. Notamment, il est probable que les outils de modélisation aujourd'hui essentiellement basés sur UML évoluent pour supporter la construction de DSLs.

L'approche récente des *fabriques logicielles (Software Factories,* [94]*)* repose sur cette synergie entre le GSD et le MDD. Les fabriques logicielles définissent la stratégie de Microsoft par rapport au MDD. Selon la définition de Greenfield [94], une fabrique

logicielle est « *une ligne de produits dirigée par les modèles dont les artefacts de production forment un environnement de développement configuré pour supporter le développement rapide des membres de la famille de produits* ». L'idée est de s'appuyer sur des DSLs de petite taille – définis par le biais de techniques de modélisation – et de construire simplement des outils de production autour de ces DSLs. Ces outils de production sont intégrés sous la forme de *plug-in* dans des outils du type Microsoft Visual Studio .NET [117]. Ils sont dès lors disponibles pour les développeurs de produits.

Les travaux récents dans chacune des deux communautés, l'apparition de la notion de fabriques logicielles ainsi que les différents efforts pour la modélisation de la variabilité dans les lignes de produits (section 3.1.3.1) sont autant de signes forts de la convergence entre le développement logiciel génératif et le développement dirigé par les modèles.

3.3.3.2 Une approche générative dirigée par les modèles pour la personnalisation

Notre objectif principal est de fournir aux développeurs d'applications pour cartes à puce les moyens méthodologiques et techniques de réaliser efficacement la configuration automatisée de leurs produits selon la spécificité de chacun des utilisateurs (section 1.5.2). La programmation orientée aspects, le développement génératif associé aux lignes de produits logiciels ainsi que le développement dirigé par les modèles fournissent un ensemble de concepts et de méthodologies sur lequel il est intéressant de s'appuyer pour atteindre cet objectif.

Nous nous proposons ici d'exploiter la complémentarité entre le GSD et le MDD. Le premier nous fournit les concepts permettant de structurer notre approche dirigée par les modèles. Dans la section 2.5, nous avons présenté un mécanisme de personnalisation des applications embarquées Java. Ce mécanisme repose sur les concepts de code personnalisable et de code personnalisé. Dans le vocabulaire du GSD, ces deux notions constituent les abstractions de l'espace solution de notre problématique. Notre problématique consiste donc à spécifier formellement l'espace problème et à mettre en œuvre sa projection (Figure 3-8) pour parvenir à la génération des morceaux de code personnalisable et de code personnalisé.

Les concepts de l'espace problème sont d'une part les descriptions des applications et d'autre part celles des utilisateurs. Notre première tâche consiste à formaliser ces descriptions. Pour ce faire, nous pouvons nous appuyer sur les concepts du MDD, et notamment sur le MOF.

Enfin, la mise en œuvre de la projection entre les espaces problème et solution consiste à concevoir des transformations, ou des chaînes de transformations, pour traduire la mise en relation des modèles d'applications et d'utilisateurs vers les constructions Java associées au code personnalisable et au code personnalisé.

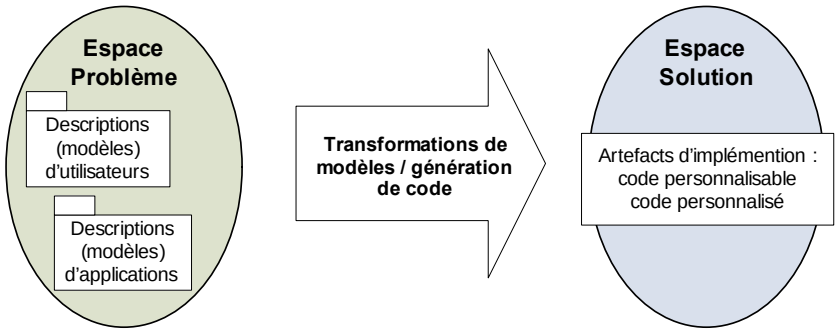


Figure 3-8 : Approche générative pour la personnalisation

3.4 Synthèse

L'objectif de ce chapitre était de réaliser un état de l'art sur les approches de développement logiciel susceptibles de constituer des briques technologiques ou des sources d'inspiration dans le contexte de notre problématique d'automatisation de la personnalisation des applications pour cartes à puce. Nous avons présenté quatre approches pour l'automatisation de la construction de logiciels : les lignes de produits logiciels, la programmation orientée aspect, la programmation générative et le développement dirigé par les modèles.

Ayant comme motivation principale le gain de productivité, les lignes de produits logiciels sont la transposition à l'industrie logicielle des concepts présents dans l'industrie traditionnelle depuis très longtemps. L'idée fondamentale des lignes de produits est de définir des familles de logiciels dans lesquelles *(i)* des artefacts de base implémentent les fonctionnalités communes et *(ii)* des mécanismes de variabilité sont mis en œuvre au-dessus de ces artefacts pour décliner des configurations différentes des logiciels.

La programmation orientée aspect est un paradigme plaçant la séparation des préoccupations au cœur du développement logiciel. L'AOP propose une décomposition des problèmes entre d'un côté, les composants fonctionnels et de l'autre côté, des aspects transversaux à ces composants. L'AOP fournit des mécanismes de tissage permettant de lier les composants fonctionnels et les aspects pour produire l'implémentation. Le principal intérêt de l'AOP est de faciliter le traitement isolé de chacun des aspects.

La notion de programmation générative est étroitement liée à celle de famille de logiciels. La GP se concentre sur l'automatisation de la création des membres d'une famille de logiciels. La GP consiste à mettre oeuvre, par le biais de mécanismes de configuration ou de transformation, la projection des abstractions d'un espace problème sur celles d'un espace solution. Ces concepts, relativement généraux, peuvent être mis en œuvre à l'aide de diverses technologies. Récemment, la notion de GP a été remplacée par celle de développement logiciel génératif (GSD).

Le développement logiciel dirigé par les modèles a deux objectifs majeurs. Le premier est la réduction du fossé d'abstraction entre les concepts réels d'un domaine et leur représentation en termes d'ingénierie logicielle. Le second est l'automatisation des tâches

mécaniques du développement logiciel. À l'origine d'un ensemble de spécifications très complet – UML, MOF, MDA, etc. –, l'OMG joue un rôle prépondérant dans le MDD.

Chacune des technologies présentées dans ce chapitre apporte des idées et des concepts sur lesquels nous nous sommes basés pour imaginer une approche pour la personnalisation des applications embarquées. Le grain de configuration proposé par les lignes de produits logiciels est encore trop gros pour permettre la personnalisation individuelle des produits. Si l'AOP est basée sur des idées intéressantes par rapport à notre volonté de respecter la séparation des préoccupations, elle constitue une solution moins intéressante dans le contexte des cartes à puce que celle décrite dans la section 2.5.1. Le GSD et la MDD non seulement partagent un certain nombre de concepts, mais sont complémentaires. Ces deux approches sont particulièrement intéressantes dans le contexte de notre problématique.

Dans ce chapitre, nous avons finalement proposé une ébauche d'approche générative dirigée par les modèles pour la personnalisation des applications embarquées. Dans le chapitre suivant, nous réalisons un état de l'art des outils du MDD sur lesquels nous nous appuyons pour implémenter cette approche.

Chapitre 4

Outils du développement logiciel dirigé par les modèles

Le processus de développement d'un système logiciel est une suite de créations et de modifications de spécifications ou implémentations représentant des angles de vue ou des niveaux d'abstraction différents. Le degré d'automatisation de ce processus est lié à l'approche adoptée. Dans le cas le plus basique, le concepteur traduit directement les spécifications informelles du système dans un langage de programmation comme Java. La seule étape automatisée dans ce cas est la compilation du code source vers le code exécutable. Alors que les systèmes logiciels deviennent toujours plus riches et complexes, cette approche atteint aujourd'hui ses limites. Une approche plus évoluée consiste à élever le niveau d'abstraction et expliciter formellement les différentes étapes conduisant à l'implémentation finale.

En considérant les modèles comme entités de première classe, le MDD les place au cœur de l'ingénierie logicielle : le rôle qui leur est dévolu n'est plus seulement informatif. La conception d'un système par le biais de modèles consiste d'abord à représenter les différentes parties du système dans des modèles dédiés, puis à mettre en œuvre leur traitement par le biais de chaînes de traitements plus ou moins automatisés permettant d'obtenir l'implémentation finale. Il est nécessaire pour cela de disposer d'outils pour la spécification, la manipulation et la transformation des modèles.

L'objectif de ce chapitre est de donner un aperçu de certains outils disponibles aujourd'hui pour le MDD. Le MOF étant aujourd'hui le cadre de modélisation le plus abouti, la plupart des technologies présentées ici sont plus ou moins liées aux spécifications de l'OMG.

Ce chapitre est organisé comme suit :

- La section 4.1 décrit l'architecture de méta-modélisation à quatre niveaux du MOF et détaille quelques concepts fondamentaux du modèle MOF lui-même. La dernière partie de cette section présente l'évolution des spécifications UML et MOF.
- La section 4.2 se concentre sur les technologies pour l'exploitation des modèles MOF. En particulier, elle décrit la sérialisation des modèles et leur manipulation par le biais d'interfaces de programmation Java.
- La section 4.3 décrit le rôle essentiel des transformations dans le contexte du MDD et présente une taxonomie des différentes approches. Cette section introduit notamment l'initiative de l'OMG pour spécifier un langage standard de définition des transformations de modèles MOF.
- La section 4.4 expose les difficultés posées par la mise en pratique du MDD. Cette section se concentre sur les problèmes liés au manque de méthodologie et sur ceux liés à l'outillage manquant de maturité.
- La section 4.5 résume et conclut le chapitre.

4.1 Expression des modèles : Meta-Object Facility (MOF)

4.1.1 Présentation et définitions

Le développement dirigé par les modèles (MDD) donne à ceux-ci un rôle important dans la chaîne de construction du logiciel. Une fois construit, le modèle subit des transformations successives, assurées par les outils automatisés ou semi-automatisés, pour se rapprocher de l'implémentation. Les modèles sont donc considérés comme des entités de première classe, dont l'interprétation doit être non ambiguë. Il est donc fondamental qu'un modèle soit associé à une sémantique définissant strictement le sens des informations représentées. Cette sémantique est spécifiée par un méta-modèle qui joue le rôle de syntaxe abstraite. Un méta-modèle spécifie les éléments utilisables dans le modèle et les règles selon lesquelles ces éléments peuvent être combinés.

Un intérêt essentiel de l'approche dirigée par les modèles réside dans les possibilités offertes pour la décomposition de l'abstraction des systèmes. Chaque vue ou partie d'un système peut être représentée par un modèle spécifique mettant en relation des concepts directement liés à cette vue. La modélisation globale d'un système implique donc la définition de différents méta-modèles. Puisque ces méta-modèles doivent être mis en relation à un moment donné du processus de conception, une cohérence sémantique et structurelle doit être assurée entre ces différents méta-modèles. Ce rôle incombe aux cadres de méta-modélisation.

Un cadre de méta-modélisation fournit un ensemble de concepts fondamentaux à partir desquels peuvent être définis de nouveaux méta-modèles. Ces concepts forment une syntaxe abstraite commune qui facilite l'interopérabilité entre les méta-modèles. Nous nous concentrons dans la suite sur le cadre de méta-modélisation fournit par l'OMG, le *Meta-Object Facility* (MOF) [100, 101]. Nous aurions également pu choisir de présenter ou d'utiliser le cadre *Eclipse Modeling Framework* (EMF) [118]. Néanmoins, la conception de ce dernier, qui est apparu après le MOF, est influencée par le MOF lui-même. Les deux sont donc relativement similaires d'un point de vue conceptuel, le MOF étant plus riche. Au contraire du MOF, EMF est spécifiquement dédié à une technologie particulière, en l'occurrence le langage Java. Il existe des moyens pour la projection de l'un vers l'autre [119].

Le MOF fournit un méta-modèle commun pour toutes les autres spécifications de l'OMG comme UML ou CWM (Figure 3-6). Dans le contexte du MOF, la terminologie suivante est utilisée :

- Le terme *méta-données* fait référence à des données dont le but est la description d'informations ou d'autres données.
- Le terme *méta-modèle* fait référence au modèle d'un certain type de méta-données, c'est-à-dire de méta-données qui respectent les mêmes règles de structuration et de cohérence.
- Le terme *méta-objet* fait référence à un artefact représentant des méta-données. Un méta-objet peut être un objet abstrait ou spécifique à une technologie. Dans le contexte du MOF, les méta-objets sont les instances de première classe.

4.1.2 Niveaux de modélisation et Modèle MOF

La spécification MOF[28] distingue l'architecture de modélisation MOF et le Modèle MOF [100]. L'architecture conceptuelle MOF se base sur quatre niveaux de méta-modélisation. Dans cette architecture, les éléments de chaque niveau conceptuel décrivent les éléments du niveau inférieur. Le méta-méta-modèle MOF, appelé Modèle MOF, est le langage abstrait qui permet de spécifier des méta-modèles. Le Modèle MOF est situé au plus haut niveau de l'architecture MOF.

4.1.2.1 Architecture de modélisation

La Figure 4-1 présente l'architecture conceptuelle à quatre niveaux sur laquelle le MOF repose. Nous illustrons les différents niveaux de modélisation avec deux exemples. Le premier est un parallèle avec les langages orientés objets à classes. On utilise le classique *dîner des philosophes*. Le second exemple illustre la définition d'un *profil utilisateur* extrêmement simpliste.

- *Le niveau informations (M0) contient les informations* ou données à décrire. Dans l'exemple du profil utilisateur, ce que l'on cherche à décrire est une personne du

[28] Nous nous basons ici sur la version 1.4 de la spécification.

monde réel. Dans celui du dîner des philosophes, on cherche à décrire des instances de philosophes ou de classes.

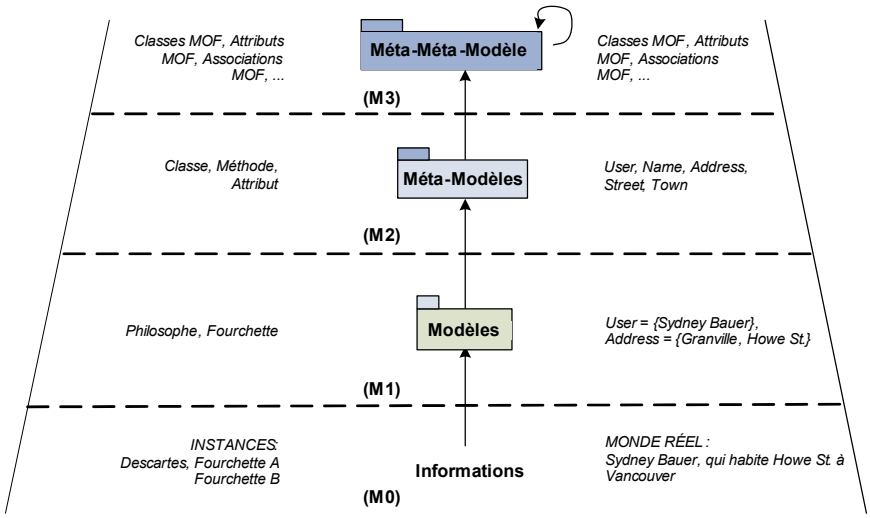


Figure 4-1 : Niveaux de méta-modélisation

- *Le niveau des modèles (M1) contient les méta-données* décrivant les données, ou informations, du niveau M0. Les méta-données sont agrégées sous la forme de modèles. Dans l'exemple du profil utilisateur, le modèle de la personne Sydney Bauer est une abstraction qui spécifie que son nom est Sydney Bauer, que son adresse est Howe St. à Vancouver. Dans l'exemple du dîner des philosophes, ce niveau contient la définition des classes Philosophes et Fourchettes : un philosophe est décrit par deux attributs, une association vers deux fourchettes et deux méthodes, penser et manger.
- *Le niveau des méta-modèles (M2) contient les méta-méta-données* qui définissent la structure et la sémantique des méta-données. Ces informations sont regroupées au sein de méta-modèles. Un méta-modèle est une sorte de langage abstrait, sans syntaxe ni notation, utilisé pour définir différents types de méta-données. Dans le cas du profil utilisateur, le niveau M2 introduit les concepts d'utilisateur (*User*), caractérisé par un nom (*Name*), et d'adresse (*Address*), caractérisée par une ville (*Town*) et une rue (*Street*). Ce méta-modèle définit une certaine vision du monde réel : c'est une abstraction qui ne garde que quelques informations et occulte toutes les autres caractéristiques. Dans l'exemple du dîner des philosophes, le niveau M2 contient la définition de ce qu'est une classe, un attribut, une association. De manière simplifiée, une classe est par exemple définie comme ayant un nom et deux collections, ses attributs et ses méthodes. Un attribut est défini comme ayant un nom et un type.
- *Le niveau du méta-méta-modèle MOF (M3)* définit la structure et la sémantique des méta-méta-données. Le méta-méta-modèle MOF, plus communément appelé Modèle MOF, suffit à se décrire lui-même. Il est orienté objet, et s'appuie sur les mêmes constructions de base qu'UML : classes, attributs, associations, paquets, etc.

4.1.2.2 Le Modèle MOF

Le Modèle MOF est le méta-méta-modèle du cadre de méta-modélisation MOF. Il définit la syntaxe abstraite des méta-modèles du niveau M2. Capable de se décrire lui-même, il repose sur une vingtaine de concepts orientés objets qui sont un sous-ensemble d'UML. Parmi ces concepts, on retrouve notamment les *classes*, les *associations*, les *références*, les *types de données* et les *paquets*. Les classes sont essentiellement composées de collections d'éléments structurels de trois types : *attribut*, *opération*, *référence*. *Une référence* définit une relation avec une autre classe. Une association permet de mettre en relation deux classes d'un méta-modèle. Chaque association contient exactement deux *terminaisons*. Une terminaison est caractérisée entre autres par un nom unique, un type qui est nécessairement une classe, une arité, et un indicateur de navigabilité qui autorise ou non la mise en œuvre de références.

4.1.3 UML 2.0 et MOF 2.0

Au début de l'année 2005, les spécifications d'UML 2.0 et de MOF 2.0 sont en cours de finalisation [75, 101]. Un objectif majeur de ces nouvelles spécifications est la réalisation de l'alignement architectural entre le MOF et UML. La relation entre MOF 2.0 et UML 2.0 est particulière. Les deux reposent sur des constructions élémentaires communes bien qu'étant à différents niveaux de l'architecture de méta-modélisation : UML au niveau M2 est une instance du MOF au niveau M3, mais dans le même temps, MOF est construit sur un sous-ensemble d'UML. La spécification MOF 2.0 distingue deux versions du MOF : *Essential MOF (EMOF)* et *Complete MOF (CMOF)*. EMOF est un sous-ensemble de MOF permettant la définition de méta-modèles simples. Alors qu'EMOF réutilise les constructions du paquet *Basic*, CMOF réutilise le paquet *Constructs* et permet la définition de méta-modèles plus complexes et autorise des possibilités de réflexion plus avancées. Le but de l'OMG à travers cette distinction est de rendre les techniques de modélisation plus accessibles, en offrant une version quelque peu allégée du MOF.

4.2 Exploitation des modèles MOF

4.2.1 Objectifs

Si l'expression des modèles et de leur sémantique peut se faire par le biais du cadre de méta-modélisation fourni par le MOF, leur exploitation se fait par le biais de projections vers des environnements spécifiques. Dans cette section, nous présentons deux types de projections de modèles MOF. La première, *XML Metadata Interchange*, a pour objet la sérialisation des modèles vers un format d'échange standard. La seconde, *Java Metadata Interface*, a pour objet la manipulation des modèles par le biais d'interfaces de programmation. Parce qu'elle ne correspond pas à nos besoins dans le contexte de la personnalisation des cartes à puce, nous ne parlons pas ici de la projection du MOF vers les IDLs (*Interface Definition Language*) de l'OMG, qui sont utilisées dans les

environnements à objets répartis avec CORBA (*Common Object Request Broker and Architecture*).

Afin d'illustrer les projections du MOF, nous utilisons dans les deux sections suivantes l'exemple du méta-modèle très simpliste de profil utilisateur présenté par la Figure 4-2. Ce méta-modèle, conforme au MOF, définit la relation entre un utilisateur et son adresse.

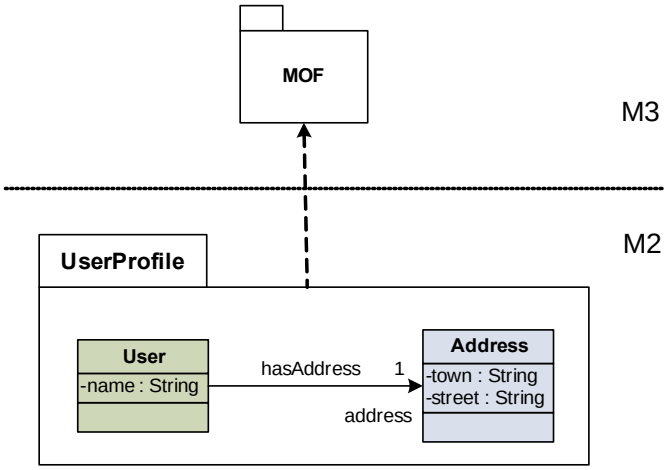


Figure 4-2 : Exemple de méta-modèle MOF

4.2.2 XML Metadata Interchange (XMI)

Afin que les différents outils de modélisation comme les environnements graphiques ou les transformateurs puissent interopérer, une représentation externe standardisée des modèles est nécessaire. À travers la spécification XMI [105, 106], l'OMG définit une projection du MOF vers les langages à marqueurs. Cette projection permet l'expression des modèles sous une forme sérialisée. Concrètement, XMI spécifie comment créer des marqueurs XML [120] pour chacun des concepts définis dans un méta-modèle. XML.

La Figure 4-3 illustre les deux types de projections qui composent XMI :

- *Règles de production de DTD XML (1).* Ces règles unidirectionnelles spécifient comment créer une DTD XML ou un schéma [121] à partir d'un méta-modèle MOF.
- *Règles de sérialisation de documents XML (2).* Ces règles bidirectionnelles spécifient comment encoder des méta-données dans un format XML.

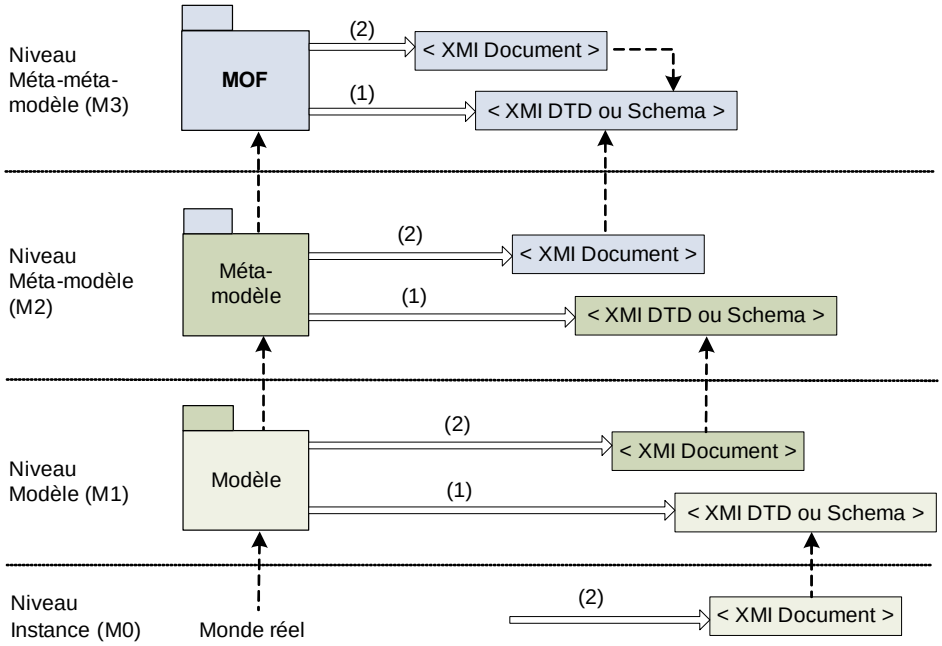


Figure 4-3 : Projections du MOF vers XMI

Si une DTD XML est produite selon *(1)* à partir d'un méta-modèle *mm*, alors cette DTD constitue une spécification syntaxique pour les documents XMI projetés selon *(2)* depuis les modèles *m* conformes au méta-modèle *mm*. Si on prend l'exemple du méta-modèle `UserProfile` illustré par la Figure 4-2, alors la projection *(1)* donne le résultat suivant :

```
<!-- header ... -->
[...]
<!-- _____________________________ Address _____________________________ -->
<!ELEMENT UserProfile::Address::Town (#PCDATA | XMI.reference)*>
<!ELEMENT UserProfile::Address::Street (#PCDATA | XMI.reference)*>
<!ELEMENT UserProfile::Address
          (%UserProfile::AddressProperties;, (XMI.extension*))?>
<!ATTLIST UserProfile::Address
        %XMI.element.att;
        %XMI.link.att;
>
<!-- _____________________________ User _____________________________ -->
<!ELEMENT UserProfile::User::Name (#PCDATA | XMI.reference)*>
<!ELEMENT UserProfile::User::Address (UserProfile::Address)?>
<!ELEMENT UserProfile::User
          (%UserProfile::UserProperties;, (XMI.extension*,
          %UserProfile::UserAssociations;))?>
<!ATTLIST UserProfile::User
        %XMI.element.att;
        %XMI.link.att;
>
<!-- _____________________________ UserProfile _____________________________ -->
<!ELEMENT UserProfile ((UserProfile::Address| UserProfile::User)*)>
<!ATTLIST UserProfile
        %XMI.element.att;
        %XMI.link.att;
>
```

DTD et XMI étant des formats très verbeux, nous ne présentons ici que la partie réellement intéressante, occultant notamment toutes les lignes d'entête. Cette DTD XML définit la structure des instances du méta-modèle `UserProfile`. La projection *(2)* donne le résultat suivant :

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version = '1.2'>
<XMI.content>
<UserProfile:User name="Sydney Bauer">
    <UserProfile:User.address>
        <UserProfile:User.Address town="Vancouver" street="Howe St."/>
    </UserProfile:User.address>
</UserProfile:User>
</XMI.content>
</XMI>
```

La plupart des outils de modélisation supportent aujourd'hui XMI. Néanmoins, les différentes versions à la fois de MOF et de XMI limitent dans une certaine mesure l'interopérabilité réelle entre les outils.

4.2.3 Java Metadata Interface (JMI)

À la manière de la projection XMI qui rend possible l'interopérabilité des méta-données sous la forme de documents, il existe des projections permettant une interopérabilité sous la forme d'interfaces de programmation (APIs). Le standard pour la projection du MOF vers le langage Java a été développé par le JCP (*Java Community Process*) dans la requête de spécification JSR-40[29], sous le nom de *Java Metadata Interface (JMI)* [122, 123]. S'appuyant sur la version 1.4 du MOF, la spécification JMI définit une infrastructure pour la création, l'échange, la découverte, le stockage et l'accès aux méta-données dans les environnements Java.

La spécification JMI comporte trois composants essentiels. *(i)* D'abord, elle définit des règles de projection des méta-modèles MOF vers des APIs Java spécifiques. Les interfaces Java générées pour un méta-modèle donné permettent aux programmes Java clients de créer, de mettre à jour ou d'accéder aux instances du méta-modèle. Les noms et types de méthodes dans ces interfaces reprennent la terminologie du méta-modèle de départ. *(ii)* Les interfaces réflexives de JMI proposent exactement les mêmes fonctionnalités que les interfaces spécifiques, mais leur structure et leur terminologie sont génériques. Ces interfaces sont moins pratiques à utiliser, mais elles permettent par exemple aux applications Java de découvrir et de manipuler des méta-données sans disposer à priori d'informations sur leur structure. *(iii)* Enfin, la spécification JMI définit des mécanismes d'exportation et importation vers des documents XML. La sérialisation des méta-données manipulées en Java par le biais de JMI est basée sur la version 1.2 de la spécification XMI [105].

[29] Le *Java Community Process (JCP)* formalise la participation de la communauté Java internationale dans les spécifications Java. Un participant au JCP peut émettre une requête d'addition ou de modification d'APIs par le biais d'un *Java Specification Request (JSR)*.

Le modèle de méta-données JMI diffère légèrement de celui du MOF. Il repose en fait sur quatre types de méta-objets de niveau M1 : les objets *instance*, les objets *fabrique de classes*, les objets *association* et les objets *paquet*. La Figure 4-4 illustre les correspondances entre le modèle MOF `UserProfile` de la Figure 4-2 et la hiérarchie des interfaces JMI :

- *Chaque paquet représentant un méta-modèle MOF* se traduit en un objet *paquet* JMI héritant de `RefPackage`. Cet objet *paquet* constitue la racine des autres méta-objets JMI crées à partir des concepts du méta-modèle MOF de départ.
- *Chaque classe du méta-modèle MOF* se traduit par deux méta-objets JMI : un objet *fabrique de classe* héritant de `RefClass` et un objet *instance* héritant de `RefObject`. Le rôle d'un objet *fabrique de classe* est de produire les objets *instance* et de jouer le rôle de conteneur pour ces objets *instance*.
- Chaque association du méta-modèle MOF se traduit par un objet *association* héritant de `RefAssociation`. Un objet *association* maintient la collection de liens entre les instances des classes aux extrémités de l'association MOF de départ.

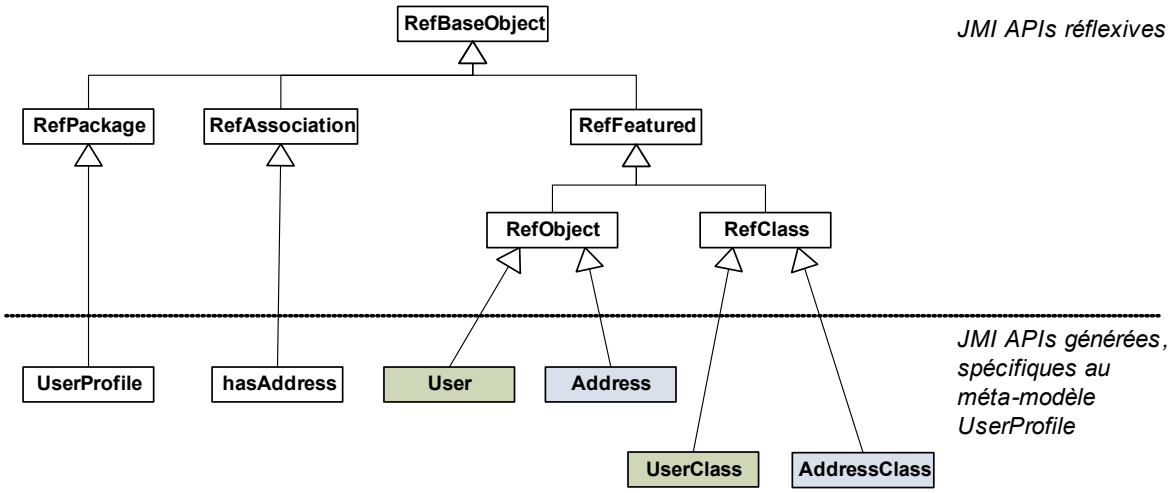


Figure 4-4 : Méta-objets et interfaces JMI

Les morceaux de code suivants illustrent les interfaces JMI spécifiques générées à partir du méta-modèle `UserProfile`. Les classes MOF `User` et `Address` étant relativement similaires, nous ne présentons ici que les interfaces générées pour la première.

Interface des objets *instance* générée pour la classe MOF `User` :

```
public interface User extends javax.jmi.reflect.RefObject {
    public String getName();
    public void setName(String aName);
    public Address getAddress();
    public void setAddress(Address anAddress);
}
```

Interface de l'objet *fabrique de classe* générée pour la classe MOF `User` :

```
public interface UserClass extends javax.jmi.reflect.RefClass {
   public createUser() throws ... ;
   public createUser(String aName) throws ... ;
}
```

Interface de l'objet *association* générée pour l'association MOF `hasAddress` :

```
public interface HasAddress extends javax.jmi.reflect.RefAssociation {
    public Boolean exists(Address a, User u);
    public Address getAddress(User u);
    public boolean add(Address a, User u);
    public boolean remove(Address a, User u);
}
```

Interface de l'objet *paquet* générée pour le paquet MOF `UserProfile` :

```
public interface UserProfilePackage extends
javax.jmi.reflect.RefPackage {
    public UserClass getUser();
    public AddressClass getClass();
    public HasAddress getHasAddress();
}
```

Les racines du graphe d'héritage de la Figure 4-4 forment un groupe d'interfaces génériques prédéfinies qui sont en fait les bases du paquet réflexif de JMI. L'utilisation des interfaces réflexives est particulièrement intéressante pour l'exploration de modèles dont on ne connaît pas la structure. Les interfaces proposent notamment des méthodes pour parcourir les classes, les associations ou les attributs. Le morceau de code suivant compare les syntaxes de création d'instance et d'affectation de valeur à un attribut d'une part en utilisant les interfaces spécifiques générées et d'autre part les interfaces réflexives.

Utilisation des interfaces spécifiques :

```
UserProfilePackage upp = UserProfilePackageImpl.create(...) ;
UserClass uc = upp.getUser() ;
User u = uc.createUser() ;
u.setName('Sydney Bauer') ;
```

Utilisation des interfaces réflexives :

```
RefPackage rp = aUserProfilePackage ;
RefClass rc = rp.refClass('User') ;
RefObject ro = rc.refCreateInstance(null);
ro.refSetValue('name','Sydney Bauer') ;
```

Plusieurs implémentations de JMI sont aujourd'hui disponibles, dont l'implémentation de référence par Unisys [123], celle de Sun Microsystems à travers le projet MDR (*Metadata Repository*) de la plate-forme de développement open source *netBeans* [124], ou celle de *Modfact* à travers ObjectWeb [125].

4.3 Transformations des modèles

4.3.1 Présentation et définitions

Le développement logiciel génératif (GSD) et le développement dirigé par les modèles (MDD) placent les transformations au cœur de l'ingénierie logicielle. Dans le vocabulaire du GSD, le terme *transformation* fait principalement référence à la projection des abstractions de l'espace problème vers celles de l'espace solution. Dans le MDD, le terme *transformation* englobe un ensemble plus exhaustif de traitements effectués sur les modèles. On distingue, entre autres, les raffinements entre différents niveaux d'abstractions, les restructurations ou enrichissements de modèles et les projections d'un méta-modèle vers un autre, etc.

La Figure 4-5 illustre les trois types de transformations que l'on peut distinguer :

- *Les transformations verticales*. La source et la cible d'une transformation verticale sont définies à différents niveaux d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Une transformation qui élève le niveau est appelée une abstraction.
- *Les transformations horizontales*. Une transformation horizontale modifie la représentation source tout en conservant le même niveau d'abstraction. La modification peut être l'ajout, la modification, la suppression ou la restructuration d'informations.
- *Les transformations obliques*. Une transformation oblique combine une transformation horizontale et une verticale. Ce type de transformation est notamment utilisé par les compilateurs, qui effectuent des optimisations du code source avant de générer le code exécutable.

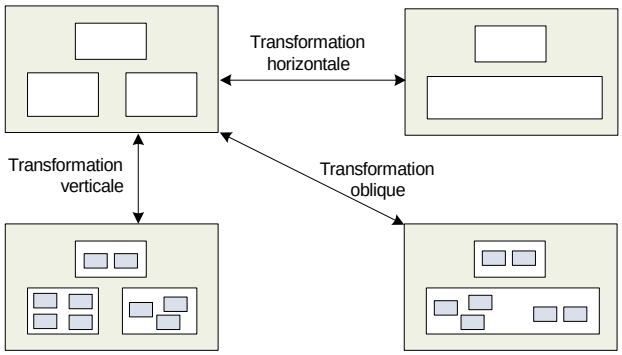


Figure 4-5 : Types de transformations

De manière orthogonale à cette catégorisation, on peut distinguer les transformations de type *modèle vers code* et celles de type *modèle vers modèle*. En fait, même si les premières peuvent être considérées comme un cas particulier des secondes – il suffit de fournir un méta-modèle pour le langage de programmation cible –, la distinction est néanmoins justifiée car le code est le plus souvent généré comme du texte simple.

Les transformations de type *modèle vers code* sont aujourd'hui relativement matures. On distingue deux approches, basées sur les principes *visiteur*[30] ou *patron*[31]. *(i)* Les approches reposant le principe du *visiteur* consistent à traverser le modèle en lui ajoutant des éléments qui réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte. Le projet *Jamda*, destiné à la génération de code Java est un exemple de cette approche [126]. *(ii)* Les approches basées sur le principe des *patrons* sont actuellement les plus courantes. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source. Parmi les outils basés sur cette approche, on peut citer *AndroMDA* [127], un générateur de code qui se repose notamment sur *Velocity* [128] pour l'écriture des patrons.

Si les transformations de type *modèle vers modèle* sont aujourd'hui moins maîtrisées, elles ont beaucoup évolué au cours de ces dernières années, et plus particulièrement depuis l'apparition de MDA [80]. La Figure 4-6 illustre les transformations de modèle dans le contexte du MDD : les transformations sont elles-mêmes considérées comme des modèles. Les modèles source et cible peuvent être des instances du même méta-modèle ou de méta-modèles différents.

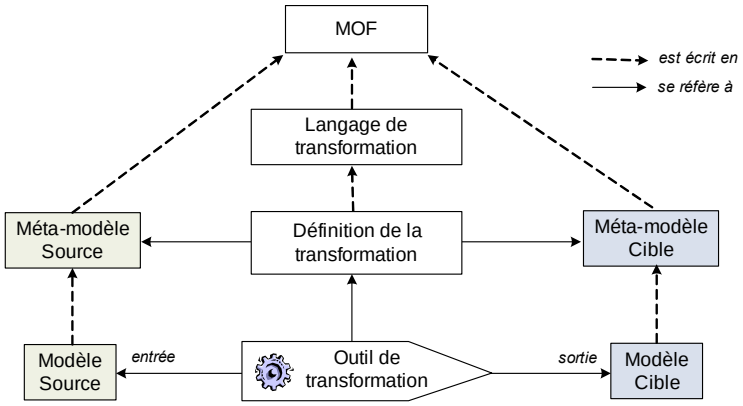


Figure 4-6 : Schéma général d'une transformation de modèle

Dans les sections suivantes, nous nous concentrons sur les transformations de type *modèle vers modèle*, en étudiant d'une part leur structure, et en présentant d'autre part une classification des différentes approches.

4.3.2 Taxonomie des transformations de type *modèle vers modèle*

Dans cette section, nous présentons une vue générale de la structure typique d'une transformation. Nous nous basons ici d'une part sur l'analyse réalisée par Czarnecki et Helsen dans [129] et d'autre part sur celle réalisée par Gerber et al. dans [130].

[30] *Visitor-based approach* dans le vocabulaire anglo-saxon.
[31] *Template-based approach* dans le vocabulaire anglo-saxon.

4.3.2.1 Structure d'une transformation

Une transformation de modèle est principalement caractérisée par la combinaison des éléments suivants : des règles de transformation, une relation entre la source et la cible, un ordonnancement des règles, une organisation des règles, une traçabilité et une direction. Notre but n'étant pas de fournir une description exhaustive des différentes transformations, nous ne détaillons que les caractéristiques qui nous paraissent les plus importantes par rapport à nos besoins dans le contexte de la personnalisation des applications embarquées pour cartes à puce.

- *Règles de transformation*. Une règle de transformation est composée de deux parties : un côté gauche (*LHS, Left Hand Side*) qui accède au modèle source, et un côté droit (*RHS, Right Hand Side*) qui accède au modèle cible. Une règle comporte également une logique, qui exprime des contraintes ou calculs sur les éléments des modèles source et cible. Une logique peut avoir la forme déclarative ou impérative. Une logique déclarative consiste à spécifier des relations entre les éléments du modèle source et des éléments du modèle cible. Une logique impérative correspond le plus souvent, mais pas nécessairement, à l'utilisation de langages de programmation pour manipuler directement les éléments des modèles par le biais d'interfaces dédiées, telles que celles proposées par JMI. Optionnellement, une règle peut être bidirectionnelle, comporter des paramètres, ou encore nécessiter la construction de structures intermédiaires.
- *Relation entre les modèles source et cible*. Pour certains types de transformations, la création d'un nouveau modèle cible est nécessaire. Pour d'autres, la source et la cible sont le même modèle, ce qui revient en fait à une modification de modèle.
- *Organisation des règles*. L'organisation des règles définit comment composer plusieurs règles de transformation. Les règles peuvent être organisées de façon modulaire, avec la notion d'importation. Les règles peuvent également utiliser la réutilisation, par le biais de mécanismes d'héritage entre règles, ou la composition, par le biais d'un ordonnancement explicite. Enfin, les règles peuvent être organisées selon une structure dépendante du modèle source ou du modèle cible. Par exemple, si le modèle cible contient une construction *paquet* qui contient des *classes*, alors la règle de création de *paquets* peut contenir la règle de création des *classes*.
- *Ordonnancement des règles*. Les mécanismes d'ordonnancement déterminent l'ordre dans lequel les règles sont appliquées. Dans le cas d'un ordonnancement implicite, l'algorithme d'ordonnancement est défini par l'outil de transformation. Dans le cas d'un ordonnancement explicite, des mécanismes permettent de spécifier l'ordre d'exécution des règles. Cet ordre d'exécution peut être défini de manière externe ou interne : tandis qu'un mécanisme externe établit une séparation claire entre les règles et la logique d'ordonnancement, un mécanisme interne permet aux règles d'invoquer d'autres règles. Enfin, l'ordonnancement des règles peut se baser également sur des conditions, des itérations ou sur une séparation en plusieurs phases – certaines règles ne pouvant être appliquées que dans certaines phases.

- *Traçabilité*. Les transformations peuvent archiver les corrélations entre les éléments des modèles source et cible. Certaines approches fournissent des mécanismes dédiés pour supporter la traçabilité. Dans les autres cas, le développeur doit implémenter la traçabilité de la même manière qu'il crée n'importe quel autre lien dans un modèle.
- *Direction*. Les transformations peuvent être unidirectionnelles ou bidirectionnelles. Dans le premier cas, le modèle cible est calculé ou mis à jour sur la base du modèle source uniquement. Dans le second cas, une synchronisation entre les modèles source et cible est possible.

Les différentes caractéristiques présentées dans cette section constituent des points de variation qui différencient les différentes approches pour la définition de transformation de type *modèle vers modèle*.

4.3.2.2 Les différentes approches

On peut globalement distinguer cinq types d'approches.

- *Approches par manipulation directe*. Ces approches se basent sur une représentation interne des modèles source et cible, et sur un ensemble d'APIs. Elles sont généralement implémentées comme des cadres structurants orientés objets qui fournissent un ensemble minimal de concepts – sous forme de classes abstraites par exemple. L'implémentation des règles et leur ordonnancement restent à la charge du développeur.
- *Approches relationnelles*. Ces approches sont celles qui utilisent une logique déclarative reposant sur des relations d'ordre mathématique. L'idée de base est de spécifier les relations entre les éléments des modèles source et cible par le biais de contraintes. L'utilisation de la programmation logique[32] est particulièrement adaptée à ce type d'approche. Généralement, les transformations produites sont bidirectionnelles.
- *Approches basées sur les graphes de transformation*. Ces approches, qui exploitent les travaux réalisés sur les transformations de graphes [131], sont similaires aux approches relationnelles dans le sens où elles permettent l'expression des transformations sous une forme déclarative. Néanmoins, les règles ne sont plus définies pour des éléments simples mais pour des fragments de modèles : on parle de *filtrage de motifs*[33]. Les motifs dans le modèle source correspondant à certains critères sont remplacés par d'autres motifs du modèle cible. Les motifs, ou fragments de modèles, sont exprimés soit dans les syntaxes concrètes respectives des modèles soit dans leur syntaxe abstraite.
- *Approches basées sur les structures*. Ces approches distinguent deux phases. La première consiste à créer la structure hiérarchique du modèle cible. La seconde consiste à ajuster les attributs et références dans le modèle cible.

[32] La programmation logique (Prolog par exemple) consiste à définir des règles de logique mathématique plutôt que fournir une succession d'instructions. Elle est souvent utilisée pour des problématiques d'intelligence artificielle.
[33] *Pattern matching* dans le vocabulaire anglo-saxon.

- *Approches hybrides*. Les approches hybrides sont une combinaison des différentes techniques. On peut notamment retrouver des approches utilisant à la fois des règles à logique déclarative et des règles à logique impérative.

De l'ensemble des approches, celles basées sur la manipulation directe sont celles de plus bas niveau. Les approches basées sur les structures conviennent particulièrement aux contextes dans lesquels il existe des correspondances simples de forme 1-1 ou 1-n entre les modèles source et cible. Les approches basées sur les graphes de transformation sont puissantes mais complexes. Comme les approches relationnelles, déclaratives elles aussi, sont moins complexes à mettre en œuvre, elles peuvent constituer un bon compromis.

4.3.3 Vers un langage de transformation pour le MOF

Après l'introduction des principes de la MDA par l'OMG, le rôle des transformations de modèles est rapidement apparu comme essentiel. Le développement dirigé par les modèles repose sur la mise en œuvre de chaînes de manipulations de modèles, ce qui nécessite que les différents outils soient capables d'interopérer. Afin de faire face à la diversité des approches pour la définition de transformations, l'OMG a émis en 2002 le *Request For Proposals*[34] (RFP) *MOF 2.0 Query / Views / Transformations* (QVT). Le but de ce RFP est de standardiser la manipulation et les transformations des modèles MOF [132].

Le but du RFP QVT est de supporter la réalisation de l'approche MDA. Il établit une suite de critères, obligatoires ou optionnels, qui constituent les bases d'un langage de transformation unifié. Les perspectives techniques de la MDA spécifient que les relations entre deux modèles doivent être définies au même niveau d'abstraction que leurs méta-modèles [103]. Dans le contexte de la MDA, tous les modèles sont conformes au MOF. Le RFP requiert la définition d'un langage permettant de représenter les transformations de manière déclarative par le biais du MOF lui-même.

Le RFP QVT distingue d'une part les manipulations actives qui consistent à effectuer des actions sur les modèles, c'est-à-dire les requêtes (*queries*) et les transformations, et d'autre part les manipulations passives, c'est-à-dire les vues (*views*).

- *Une requête* est une expression qui est évaluée sur un modèle. Le résultat d'une requête est soit une sélection d'éléments du modèle source correspondant à certains critères, soit un autre résultat évaluable par le langage de requête utilisé.
- *Une vue* est un modèle complètement dérivé d'un modèle de base. La vue ne peut pas être modifiée séparément du modèle qu'elle représente. Typiquement, le méta-modèle de la vue est différent de celui du modèle de base. Une vue peut être partielle ou complète. Une requête est un type particulier de vue.
- *Une transformation* prend en entrée un modèle et le modifie ou en crée un nouveau. Les différentes caractéristiques des transformations ont été largement décrites dans la

[34] Un *Request For Proposals (RFP)* constitue une sorte de cahier des charges pour une future spécification. Les membres de l'OMG intéressés par cette spécification sont invités à soumettre leurs propositions.

section précédente. Une vue est un type particulier de transformation, pour laquelle le modèle cible ne peut pas être modifié indépendamment du modèle source.

Le RFP QVT a généré de nombreuses propositions, souvent très différentes, au cours des deux dernières années. Les différents participants ont progressivement convergé, pour aujourd'hui proposer une réponse unifiée au RFP [133]. Dans cette réponse, l'approche pour la définition des transformations est une approche hybride, qui distingue les *relations* et les *projections*. *(i)* Une relation est une spécification de relation multidirectionnelle. Les relations ne sont pas exécutables dans le sens où elles ne peuvent être utilisées pour créer ou modifier un modèle. Par contre, elles peuvent servir à vérifier la cohérence entre deux modèles. Les relations sont exprimées de manière complètement déclarative. *(ii)* Une projection est l'implémentation d'une transformation. Au contraire des relations, les projections peuvent être unidirectionnelles. Une projection peut être le raffinement d'une relation. Une projection peut être exprimée de manière impérative.

La spécification MOF 2.0 QVT est aujourd'hui en cours de finalisation. Les premiers outils conformes devraient être disponibles à court terme et les expérimentations à venir vont permettre d'évaluer concrètement le potentiel du langage QVT. En l'absence de standard pour les transformations, la mise en œuvre d'approches génératives dirigées par les modèles au cours de ces dernières années nécessitait soit l'utilisation d'outils de transformations plus ou moins puissants, soit l'implémentation de solutions ad hoc. Dans un contexte donné, l'une ou l'autre de ces alternatives pouvait s'avérer tout à fait satisfaisante. Néanmoins, un langage QVT capable de jouer pour les modèles MOF un rôle similaire à celui que joue XSLT pour les documents XML constituerait assurément une brique technologique majeure pour le MDD.

4.4 Difficultés de mise en œuvre du MDD

Si un certain nombre d'outils concrets pour l'expression et l'exploitation des modèles ont été décrits dans les sections précédentes, la mise en œuvre pratique d'une approche dirigée par les modèles pose encore un certain nombre de difficultés. La méthodologie et l'outillage associés au MDD souffrent encore d'un certain manque de maturité.

4.4.1 Problèmes liés à la méthodologie

La conception d'un système logiciel est un processus complexe qui implique de multiples acteurs ayant chacun une vision du système liée à un rôle ou une compétence spécifique. Dans le MDD, cette complexité se traduit entre autres par une multiplication des méta-modèles. Si le MOF fournit un ensemble de concepts de base pour la définition de ces méta-modèles, il ne définit aucune méthodologie pour guider leur structuration. Plus concrètement, il ne précise pas comment différents méta-modèles peuvent ou doivent être corrélés.

Un de nos objectifs initiaux étant d'isoler les préoccupations liées à la personnalisation, cette section s'attarde sur la séparation des préoccupations comme support méthodologique de méta-modélisation.

4.4.1.1 Modélisation et séparation des préoccupations

La séparation des préoccupations est une approche reconnue pour simplifier la construction de systèmes logiciels. Elle permet de décomposer la conception en éléments compréhensibles et plus facilement manipulables [50]. La section 3.2 a décrit comment la programmation orientée aspects (AOP) permet d'appliquer la séparation des préoccupations au niveau de l'implémentation : chaque préoccupation est encapsulée dans une unité indépendante appelée aspect. Des mécanismes de tissage permettent la composition des différents aspects avec le code fonctionnel.

En tant qu'artefacts d'abstraction offrant la possibilité d'exprimer des visions simplifiées et partielles de la réalité, les modèles sont particulièrement adaptés pour isoler les informations relatives à des préoccupations particulières. À un niveau d'abstraction plus élevé que l'AOP, la séparation des préoccupations constitue donc un support méthodologique possible pour structurer la définition et l'utilisation des méta-modèles [134]. Dans le contexte du MDD, son application comporte deux aspects essentiels :

- *La définition des moyens sémantiques* nécessaires à l'expression des préoccupations. Les concepts propres à une préoccupation donnée doivent être définis et regroupés dans des méta-modèles spécifiques à celle-ci.
- *L'intégration des différents modèles*. L'intégration peut se faire soit en utilisant des constructions du MOF comme l'importation ou l'héritage de paquets, soit en utilisant des mécanismes de tissage basés sur des transformations de modèles.

Par le biais de la MDA, l'OMG propose un ensemble de recommandations et de règles pour structurer la spécification de systèmes logiciels au travers de modèles MOF. Ces règles reposent précisément sur une application basique de la séparation des préoccupations. Elles différencient clairement les préoccupations fonctionnelles et techniques, en distinguant les modèles indépendants vis-à-vis des plates-formes (PIMs) de ceux spécifiques aux plates-formes (PSMs). Cette séparation assure aux modèles une pérennité plus grande dans le sens où le modèle des fonctionnalités du système reste pertinent indépendamment de l'évolution des solutions technologiques de mise en œuvre. Cependant, la dépendance ou non vis-à-vis d'une plate-forme ne constitue pas un critère suffisant pour organiser efficacement la phase de conception d'un système. En particulier, la MDA ne précise rien quant à l'approche à adopter pour structurer les PIMs entre eux, ce qui limite fortement son apport méthodologique.

Il est donc nécessaire d'enrichir la méthodologie proposée par la MDA. Le thème de la séparation des préoccupations dans le MDD suscite un intérêt croissant de la part de la communauté académique. Notamment, les travaux de France et al. [135] ainsi que ceux de Gray et al. [136] contribuent à définir la modélisation *orientée aspects*, en proposant des procédés et outils pour supporter l'expression et la composition d'aspects au niveau des modèles. Les premiers utilisent directement les modèles comme artefacts d'expression des

aspects et étudient les compositions de modèles, tandis que les seconds introduisent un langage spécifique pour la description des propriétés transversales des modèles (les aspects) et fournissent un tisseur de modèles associé à ce langage. Un nombre important de travaux sont articulés autour d'UML comme support direct de l'AOP : évaluation de la pertinence d'UML pour la représentation des aspects, définition d'extensions ou utilisation de profils pour encapsuler les aspects, etc. [137, 138, 139, 140]. Enfin, certaines approches se concentrent spécifiquement sur les mécanismes de tissage : on peut citer par exemple les travaux effectués sur les techniques de transformation de modèles orientées préoccupations [141].

Nous nous intéressons dans la section suivante à une approche s'inscrivant dans cette thématique globale de recherche, la méthodologie CODeX proposée par Marvie [134, 142]. Cette approche se concentre davantage sur la structuration des méta-modèles que sur la problématique de la transversalité des préoccupations.

4.4.1.2 CODeX

CODeX est à l'origine un cadre de travail pour méta-modéliser des langages de description d'architectures logicielles à base de composants (ADLs, *Architecture Description Language*) [ ]. Son but est de fournir des moyens pour la définition d'ADLs par le biais de méta-modèles et pour la structuration des différents ADLs définis afin de permettre la collaboration entre les différents acteurs d'un processus logiciel.

Le cœur de CODeX consiste en un ensemble de règles d'utilisation du MOF pour structurer la définition et l'utilisation des méta-modèles en appliquant la séparation des préoccupations. La méthodologie ainsi définie dépasse le cadre original des descriptions d'architecture et constitue un véritable *patron de méta-modélisation* exportable vers d'autres contextes [134]. Cet aspect est précisément celui qui nous intéresse dans la perspective de la mise en œuvre de notre approche générative pour la personnalisation.

La méthodologie CODeX repose sur une structuration des méta-modèles organisée en trois niveaux : le plan de base, les plans d'annotations et le plan d'intégration. La Figure 5-1 illustre cette approche.

- Le premier niveau contient le *plan de base*. Il représente le vocabulaire commun du domaine, indépendamment de toute préoccupation.
- Le second niveau contient les *plans d'annotations*. Il représente la spécification des préoccupations du domaine. Ces dernières enrichissent le vocabulaire du plan de base par le biais d'annotations. Chaque préoccupation est représentée par un plan spécifique qui contient la définition des concepts qui lui sont propres et leurs relations avec le vocabulaire du domaine.
- Le troisième niveau contient le *plan d'intégration*. Il représente l'intégration des différents plans d'annotations

Pour définir les relations entre les différents méta-modèles, CODeX s'appuie sur les notions d'importation et d'héritage de paquets fournies par le MOF. L'héritage de paquet tel qu'il est défini par le MOF est intéressant car lorsqu'un paquet B hérite d'un paquet A,

tous les concepts contenus dans le paquet A se retrouvent dans le paquet B comme s'ils avaient été définis dans ce dernier.

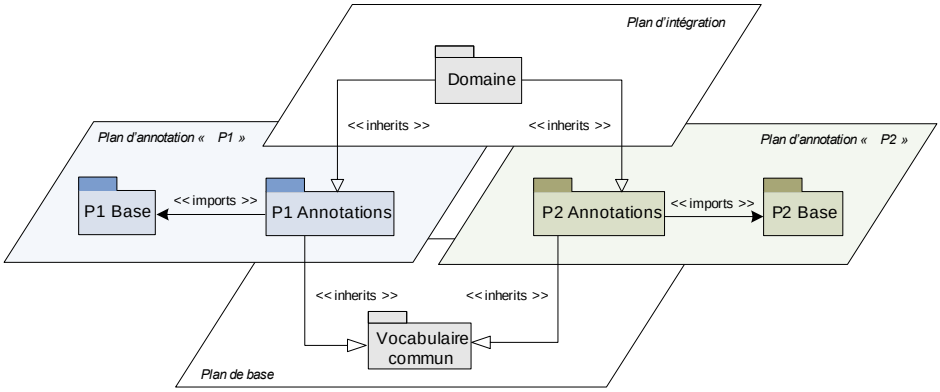


Figure 5-1 : Plans d'annotations CODeX

Le plan d'annotation d'une préoccupation `P1` est composé de deux paquets MOF : `P1Base`, qui est le méta-modèle réunissant les concepts propres à `P1`, et `P1Annotations`, qui définit comment ces concepts sont reliés au vocabulaire commun. Concrètement, `P1Annotations` importe `P1Base`, hérite de `Vocabulaire Commun` et définit un ensemble d'associations entre les éléments de `P1Base` et ceux de `Vocabulaire Commun`. L'intégration des différentes préoccupations se fait également par le biais de l'héritage de paquets : le paquet `Domaine` du plan d'intégration hérite de tous les paquets d'annotations.

4.4.2 Problèmes liés à l'outillage

L'intérêt croissant porté au développement de logiciel génératif et aux techniques de modélisation – en particulier celui porté à la MDA – s'accompagne de multiples travaux de recherche sur les transformations de modèles. Dans la section 4.3.2, nous avons recensé cinq types d'approches utilisées aujourd'hui pour mettre en œuvre des transformations de type *modèle vers modèle*. Parallèlement à cette multiplication des approches, l'OMG a émis le RFP QVT [133] dans le but de standardiser la manipulation et la transformation des modèles MOF (section 4.3.3). Cette spécification étant seulement cours de finalisation, la mise en œuvre d'approches dirigées par des modèles conformes au MOF a jusqu'à aujourd'hui nécessité soit l'utilisation d'outils plus ou moins génériques, soit l'implémentation de solutions ad hoc.

En particulier, la spécification MDA a motivé le développement de divers outils de transformation de modèles vers des environnements spécifiques dont, entre autres, *AndroMDA* [127], *Codagen Architect* [143] et *Arc Styler* [144]. Ces trois outils proposent la projection de modèles UML vers les langages Java, C#, C++, etc. En parallèle de ces outils de génération de code, d'autres outils pour les transformations de type *modèle vers modèle* se développent.

- *MIA Transformations* [145]. Cet outil est complémentaire de *MIA Génération*, destiné à la production de code. Il permet la conception de transformations basées sur des règles d'inférence. Les règles peuvent être enrichies de scripts Java. Cet outil supporte largement UML, mais ne se limite pas à lui : il accepte en entrée le format XMI ou les formats propriétaires des outils majeurs de modélisation.
- *ATL (ATLAS Transformation Language)* [146]. Au sein du projet GMT [114], ATL est une approche hybride : la spécification d'une transformation est déclarative de préférence, mais des constructions impératives permettent la description des projections dont la complexité le requiert. ATL est une réponse au RFP QVT. Une interface graphique pour Eclipse [88] a été définie. Elle repose sur le cadre de modélisation EMF [118].
- *MTL Engine et UMLAUT NG* [147]. Capitalisant sur l'expérience acquise avec le développement d'UMLAUT, un outil dédié à UML, l'équipe *Triskell* travaille sur une nouvelle version destinée à implémenter des transformations de modèles génériques. Pour ce faire, elle fournit un langage de transformation basé sur une structuration orientée objets (MTL) ainsi qu'un compilateur pour ce langage (MTL Engine).

Aucun de ces outils n'a aujourd'hui atteint une maturité qui le place au dessus des autres : chacun repose sur une approche qui lui est propre et qui est plus ou moins adaptée à certains types de problèmes. En attendant le résultat la finalisation de QVT, la possibilité de développer ses propres outils de transformations, notamment en se reposant sur des solutions comme JMI, apparaît comme une alternative possible. L'avantage de solution ad hoc étant de bénéficier d'outils optimisés pour un problème particulier. Cette tendance rejoint le travail réalisé actuellement sur les fabriques logicielles [94].

4.5 Synthèse

La réalisation d'un système dans le contexte du MDD consiste, d'une part, à représenter les différentes parties du système par le biais de modèles spécifiques et, d'autre part, à mettre en œuvre le traitement automatisé ou semi automatisé de ces modèles par le biais de mécanismes de transformation. L'objectif de ce chapitre était de réaliser un panorama des outils que nous avons choisi d'utiliser pour l'implémentation de notre approche générative pour la personnalisation.

Les modèles sont les artefacts par lesquels s'exprime l'abstraction des systèmes. Le sens d'un modèle est défini par son méta-modèle. La conception d'un système requiert la définition de différents méta-modèles qui doivent respecter une cohérence structurelle et sémantique afin de pouvoir être mis en relation. Cette cohérence est assurée au sein des cadres de méta-modélisation qui fournissent des ensembles de concepts fondamentaux à partir desquels peuvent être définis de nouveaux méta-modèles. Dans ce chapitre, nous avons présenté le MOF, fourni par l'OMG. Ce cadre repose sur une architecture en quatre niveaux de modélisation, au-dessus de laquelle on trouve le Modèle MOF, un méta-méta-modèle capable de se décrire lui-même.

Le MDD consiste à exploiter les modèles. Dans ce chapitre, nous nous sommes concentrés sur deux projections des modèles MOF vers des environnements spécifiques. La première, XMI consiste à sérialiser de manière standard les modèles, dans le but de permettre une interopérabilité entre différents outils de modélisation. La seconde, JMI, définit les spécifications d'une infrastructure Java pour la manipulation de modèles MOF, par le biais d'un ensemble d'APIs.

Le développement logiciel génératif (GSD) et le MDD placent les transformations au cœur de l'ingénierie logicielle. Dans ce chapitre, nous avons présenté les différents types de transformations, détaillé leur structure et énuméré les différentes approches de mise en œuvre. Les transformations de type *modèle vers modèle* sont l'objet d'un nombre important de travaux de recherche, particulièrement depuis l'introduction de la MDA. Pour faire face à la diversité des approches, l'OMG travaille sur la standardisation de la manipulation des modèles MOF par le biais de MOF 2.0 QVT.

La mise en œuvre pratique du MDD pose aujourd'hui un certain nombre de difficultés, provenant essentiellement d'un manque de maturité des méthodologies et des outils. Du point de vue méthodologique, la séparation des préoccupations apparaît comme un support particulièrement adapté pour la structuration des méta-modèles. La méthodologie CODeX est un exemple de cadre de modélisation la plaçant au cœur de l'organisation de la modélisation. Du point de vue de l'outillage, la mise en œuvre des transformations de modèles est le point le plus critique. Plusieurs environnements ou ateliers sont disponibles aujourd'hui, aucun ne prend réellement d'ascendant sur les autres. La spécification QVT étant en cours de finalisation, les premiers outils complètement conformes devraient être disponibles à court terme. En attendant, développer ses propres environnements de transformation est une solution alternative intéressante dans certains contextes.

Dans les chapitres suivants, nous présentons notre approche générative dirigée par les modèles pour la personnalisation des applications embarquées. Sa mise en œuvre illustre les problèmes rencontrés dans la mise en pratique du MDD.

Troisième partie

Une démarche dirigée par les modèles pour la personnalisation des applications embarquées dans les cartes à puce

Chapitre 5

Définition du processus génératif de personnalisation

Notre ambition est d'apporter aux fournisseurs d'applications pour cartes à puce les moyens technologiques et méthodologiques nécessaires à la personnalisation de leurs applications. Nous avons présenté dans le Chapitre 2 une solution pour implémenter la personnalisation en intégrant les contraintes spécifiques liées aux procédés de fabrication industrielle des cartes. Dans la perspective d'une généralisation de cette approche et de l'automatisation du processus, nous avons étudié dans le Chapitre 3 différentes techniques d'ingénierie logicielle, dont le développement génératif et le développement dirigé par les modèles.

Ce chapitre décrit la mise en œuvre concrète d'une approche générative dirigée par les modèles pour la personnalisation. Notre approche consiste à élever le niveau d'abstraction de la solution décrite dans la section 2.5.2 et à définir les espaces problème et solution de notre problématique. Les abstractions de chacun des deux espaces sont définies par le biais de méta-modèles spécifiques, tandis que la projection du premier vers le second est le résultat d'une chaîne de manipulations et de transformations de modèles.

Ce chapitre est organisé comme suit.

- La section 5.1 rappelle certains choix méthodologiques et précise les objectifs de notre proposition.

- La section 5.2 présente les abstractions définies pour représenter notre problème. Il s'agit ici de décrire d'une part l'application à personnaliser, et d'autre part, les spécificités des utilisateurs.
- La section 5.3 présente d'une part, les abstractions définies pour modéliser les fabriques de personnalisation et les activateurs, et d'autre part, les processus de génération de code permettant d'obtenir respectivement le code personnalisable et le code personnalisé.
- La section 5.4 décrit les mécanismes de manipulations et de transformations de modèles permettant la projection de l'espace problème vers l'espace solution.
- La section 5.5 présente un résumé de l'approche.

5.1 Objectifs

Dans la section 2.5, nous avons décrit comment le respect de la séparation des préoccupations et l'intégration de contraintes liées à la fabrication industrielle des cartes à puce nous ont conduit à introduire les notions de *code personnalisable* et de *code personnalisé*. Ces concepts permettent d'implémenter une personnalisation riche des applications embarquées.

Les différentes approches pour l'automatisation de la construction de logiciels présentées dans les deux chapitres précédents fournissent un ensemble de briques conceptuelles et technologiques sur lesquelles nous pouvons nous appuyer pour définir une méthodologie de personnalisation permettant d'obtenir de manière semi automatisée le code personnalisable et le code personnalisé. Les concepts du GSD et du MDD nous permettent d'élever le niveau d'abstraction de la solution décrite dans le Chapitre 2.

Nous utilisons les principes du développement logiciel génératif pour structurer une approche dirigée par les modèles en proposant une approche dans laquelle *(i)* l'*espace problème* comporte les descriptions de l'application originale, des utilisateurs et des parties à personnaliser et *(ii)* l'*espace solution* comporte des abstractions pouvant être utilisées pour la génération systématique des fragments de code. La Figure 5-2 décrit une vue d'ensemble de l'approche et explicite neuf problèmes clefs auquel nous devons apporter des solutions.

(1) Représentation de l'application originale. Le modèle original de l'application doit décrire celle-ci d'un point de vue fonctionnel. Typiquement, il peut s'exprimer par le biais d'un diagramme de structure statique.

(2) Formalisation des concepts de personnalisation. Afin d'exprimer quelles parties d'une application peuvent être adaptées et comment elles peuvent l'être, il est nécessaire d'introduire un langage de personnalisation. Ce langage doit permettre la modélisation de l'application personnalisable.

(3) Représentation des profils d'utilisateurs. Le traitement automatisé des informations personnelles est conditionné par une structuration rigoureuse de celles-ci. Si le contenu

du profil utilisateur est différent d'une application à une autre, les concepts pour exprimer ce contenu doivent être génériques. Il est donc nécessaire d'introduire un vocabulaire pour la description de la spécificité des utilisateurs finaux.

(4) (5) Représentation et génération des fabriques de personnalisation. L'expression des fabriques de personnalisation à un niveau plus élevé que celui de l'implémentation permet de décorréler l'approche du langage final d'implémentation. L'obtention du code d'une fabrique à partir de son modèle constitue une nouvelle problématique de programmation générative.

(6) (7) Représentation et génération des activateurs. Comme pour les fabriques de personnalisation, les activateurs doivent être modélisés. L'obtention du code final des activateurs constitue également une autre problématique de programmation générative.

(8) Projection vers les fabriques de personnalisation. Cette projection du type *modèle vers modèle* consiste à prendre comme source le modèle de l'application personnalisable et à en dériver les modèles des fabriques de personnalisation à implémenter.

(9) Projection vers les activateurs. Cette projection est particulière dans le sens où il s'agit d'une projection de type Y, c'est-à-dire comportant deux modèles sources – le modèle de l'application personnalisable et le profil utilisateur. Elle doit être répétée pour chacun des utilisateurs.

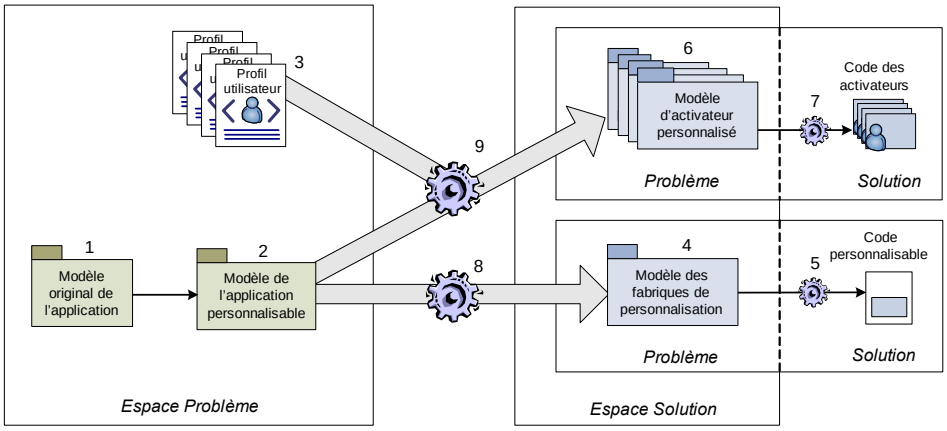


Figure 5-2 : Approche générative pour la personnalisation

La suite de ce chapitre décrit la mise en œuvre de cette approche générative. Nous nous reposons sur les outils du développement dirigé par les modèles que nous avons présentés dans le chapitre précédent. Concrètement, il s'agit de concevoir pour les problèmes *(1) (2) (3) (4) (6)* un ensemble de méta-modèles conformes au MOF et d'implémenter des transformations pour les problèmes *(5) (7) (8) (9)*.

Nous précisons d'abord notre choix de support méthodologique pour l'organisation des méta-modèles avant de nous concentrer successivement sur la définition des deux espaces

problème et solution. L'ensemble de l'approche est illustré par l'application *LoyaltyManager* introduite dans la section 2.4.

Dans le contexte de notre approche générative pour la personnalisation, nous nous appuyons sur les deux premiers niveaux de structuration de la méthodologie CODeX mais nous ne réutilisons pas le plan d'intégration du troisième niveau [148]. Notre but n'est pas d'obtenir un modèle global de l'application personnalisée, mais des modèles spécifiques représentant uniquement les parties personnalisées du code final de l'application. Pour cette raison, nous adoptons une stratégie de traitement du plan d'annotation basée sur les transformations de modèles (voir section 5.4).

5.2 Définition de l'espace problème

Nous présentons dans cette section les abstractions qui vont servir à spécifier notre problématique, c'est-à-dire à exprimer quelles parties d'une application doivent être personnalisées, de quelle manière et avec quelles données. Nous cherchons donc à résoudre les problèmes *(1) (2)* et *(3)* exposés dans la section 5.1. Pour *(1)* et *(2)* nous nous appuyons sur la méthodologie de structuration CODeX : alors que notre vocabulaire commun est un méta-modèle permettant la description fonctionnelle de l'application, nous définissons un plan d'annotation pour la personnalisation. Pour *(3)*, nous organisons dans un méta-modèle spécifique quelques concepts de base permettant la création de profils utilisateurs.

5.2.1 Vocabulaire commun

5.2.1.1 Concepts

Comme nous l'avons expliqué dans la section 2.5, la personnalisation que nous cherchons à réaliser consiste à configurer l'instanciation des applications embarquées. Dans le contexte d'applications implémentées avec des langages à objets, cette configuration se traduit par un paramétrage de la création des objets, de leurs attributs et de leurs liaisons. Le rôle du vocabulaire commun est de permettre la description de la structure logique de l'application.

UML définit un ensemble très riche de concepts pour modéliser la structure statique des applications. Bien que cette solution soit à première vue pertinente, nous ne la retenons pas, pour deux raisons principales :

- *UML est une spécification très riche*, définie pour couvrir un large spectre de contextes. Cette richesse se traduit notamment par une verbosité importante des versions sérialisées des modèles, qui les rend difficile à manipuler sans les outils appropriés.
- *La spécification UML est actuellement dans une phase de transition*. La version 2.0 étant seulement en cours de finalisation, les outils pour son support n'étaient pas prêts

lors des travaux décrits dans ce document. La version 1.5 est stable, mais elle va devenir obsolète à court terme lorsque UML, MOF et QVT seront alignés.

Notre démarche s'inscrivant dans une phase de recherche et d'expérimentation, nous préférons nous concentrer sur l'aspect méthodologique plutôt que sur des considérations techniques comme celles du choix d'outils ou de standards. Pour une meilleure lisibilité de l'approche, des concepts et des transformations à implémenter, nous avons donc choisi de définir un méta-modèle basique, illustré par la Figure 5-3.

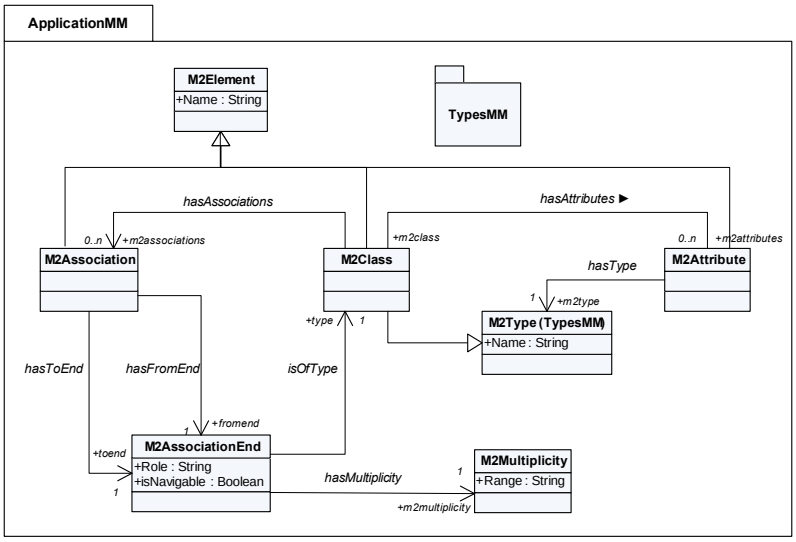


Figure 5-3 : Méta-modèle ApplicationMM

Le méta-modèle `ApplicationMM` est un méta-modèle basique du type classe / attribut / associations. Afin d'éviter les confusions avec les classes et attributs des langages d'implémentation, les noms des concepts définis dans `ApplicationMM` comportent le préfixe `M2`. Il définit une application comme un ensemble de classes (`M2Class`) et d'associations (`M2Association`) entre ces classes. Une classe comporte des attributs (`M2Attribute`), qui ont une valeur (`M2Value`) et un type (`M2Type`). Une association (`M2Association`) entre deux classes est composée de deux terminaisons (`AssociationEnd`), dont le type est celui des classes ainsi reliées. Une multiplicité (`M2Multiplicity`) est affectée à chaque terminaison d'association. Ce méta-modèle n'ayant pas pour prétention de permettre la génération complète de l'application, la notion d'opération notamment est absente. Le méta-modèle `ApplicationMM` importe le méta-modèle `TypesMM` qui définit des types primitifs tels que les entiers, les booléens ou les chaînes de caractères.

5.2.1.2 Cas d'étude : modèle original de l'application LoyaltyManager

La Figure 5-4 illustre l'utilisation du méta-modèle `ApplicationMM` pour spécifier l'application *LoyaltyManager* présentée dans la section 2.4. Puisque les concepts définis

dans `ApplicationMM` sont très proches de ceux d'UML, nous utilisons la notation graphique traditionnelle d'UML.

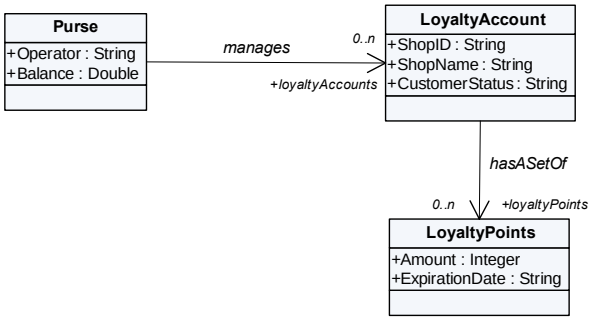


Figure 5-4 : Modèle de l'application LoyaltyManager (notation UML)

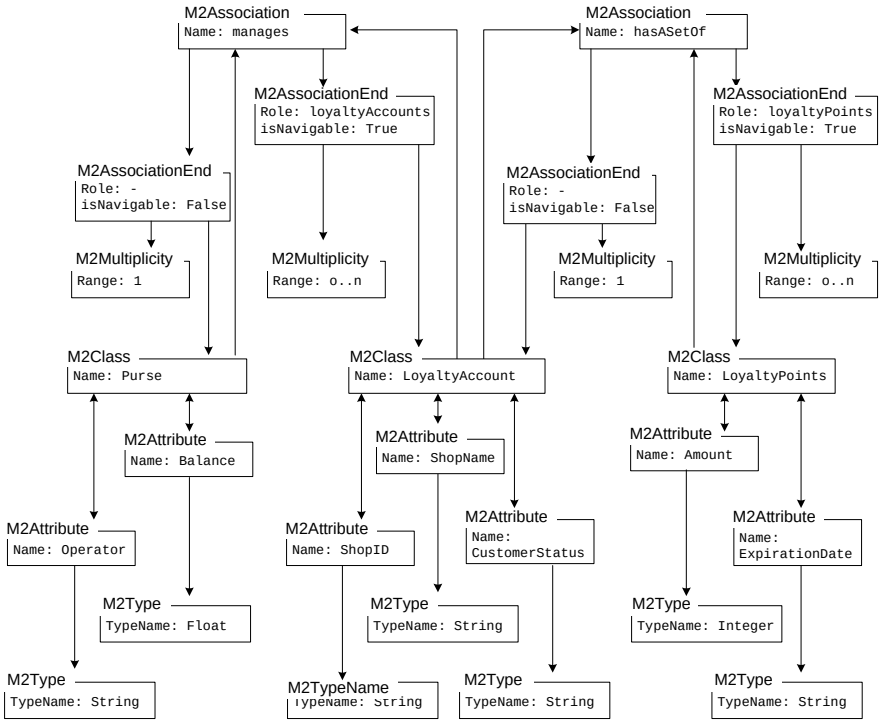


Figure 5-5 : Modèle de l'application LoyaltyManager

Le modèle du *LoyaltyManager* est conforme au fonctionnement décrit dans la section 2.4.2. La Figure 5-5 présente une représentation graphique du modèle. La classe `Purse`, définie pour représenter un porte-monnaie électronique, est caractérisée par deux attributs : le nom du gestionnaire du porte-monnaie (`Operator`) et le solde (`Balance`). La classe `LoyaltyAccount`, définie pour représenter un compte de fidélité, est caractérisée par trois attributs : les identifiants du partenaire commercial chez qui le compte est détenu (`ShopName` et `ShopID`) et le statut du client chez ce partenaire (`CustomerStatus`). Un porte-monnaie pouvant être utilisé chez plusieurs partenaires commerciaux, il est lié à plusieurs

comptes de fidélité. Une association `manages` est donc crée entre `Purse` et `LoyaltyAccount`. La classe `LoyaltyPoints` représente les points de fidélité. Elle est caractérisée par un nombre de points (`Amount`) et une date d'expiration pour ces points (`ExpirationDate`). La classe `LoyaltyAccount` est liée à la classe `LoyaltyPoints` par l'association `hasASetOf`.

Annoter ce modèle pour la personnalisation consiste à déterminer d'une part, pour quelles classes des fabriques de personnalisation devront être crées, et d'autre part, quels attributs et quelles associations ces fabriques devront permettre de configurer.

5.2.2 Plan d'annotation pour la personnalisation

5.2.2.1 Concepts

Le rôle du plan d'annotation pour la personnalisation est de fournir les moyens d'exprimer ce qui est, dans l'application, dépendant de l'utilisateur. Dans les sections 2.5.1 et 2.5.2, nous avons exposé et illustré une approche concrète pour obtenir des instanciations personnalisées d'une application : d'une part, des méthodes permettant d'intervenir sur la création des classes et l'affectation de valeurs aux attributs sont ajoutées au code original, et d'autre part, un activateur exploitant ces méthodes est écrit pour chaque utilisateur. Le plan d'annotation est constitué de deux méta-modèles `PersoBaseMM` et `PersoAnnotationsMM` qui définissent les concepts utilisables comme bases du processus de génération des fabriques de personnalisation.

La Figure 5-6 illustre le plan d'annotation pour la personnalisation. Conformément à la structuration utilisée dans la méthodologie CODeX, ce plan est composé d'un méta-modèle `PersoBaseMM` définissant les concepts de base et d'un méta-modèle `PersoAnnotationsMM` reliant ces concepts à ceux du méta-modèle `ApplicationMM`.

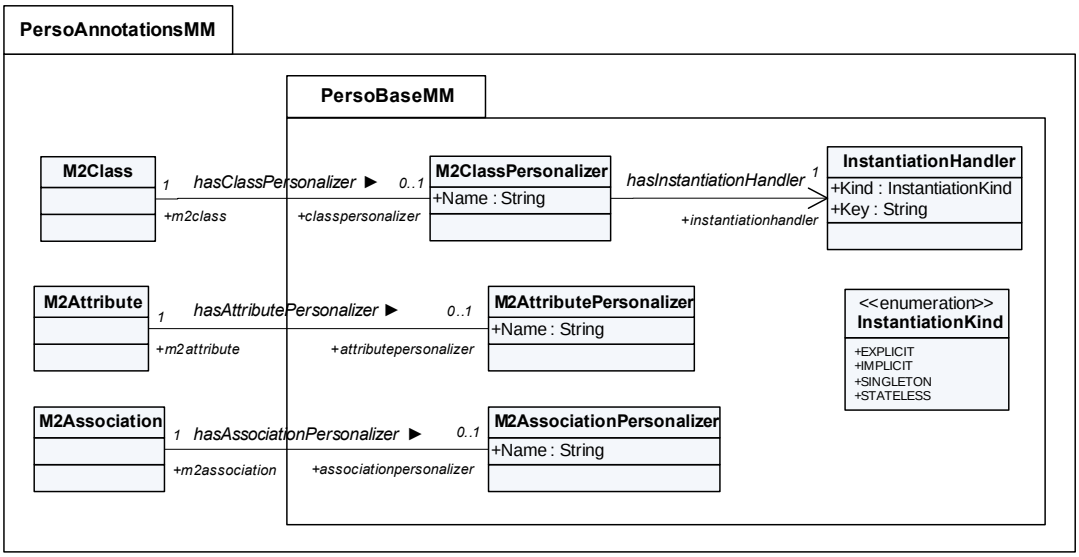


Figure 5-6 : Méta-modèles PersoBaseMM et PersoAnnotationsMM

Personnalisation de classes. Le concept `M2ClassPersonalizer` permet l'annotation du concept `M2Class`. Il signifie que l'instanciation de la classe annotée est gérée au sein du processus de personnalisation. Afin de préciser la façon dont cette instanciation doit être réalisée, un élément `M2ClassPersonalizer` est systématiquement associé à un gestionnaire d'instanciation. Ce gestionnaire, matérialisé par le `M2InstantiationHandler`, spécifie le type d'instanciation de la classe :

- *Explicite*. Chaque instance a une identité propre, caractérisée par un identifiant unique qui peut être la valeur d'un attribut par exemple.

- *Implicite*. Chaque instance a également une identité propre, mais contrairement à une instanciation explicite, l'unicité n'est pas donnée par un identifiant mais par une clef primaire portant sur plusieurs attributs.

- *Singleton.* Il existe une seule et unique instance de la classe par application.

- S*ans état*. Les différentes instances sont interchangeables.

Pour la gestion des attributs, un élément `M2ClassPersonalizer` est potentiellement lié à un ou plusieurs éléments `M2AttributePersonalizer`.

Personnalisation d'attributs. Le concept `M2AttributePersonalizer` permet l'annotation du concept `M2Attribute`. Il signifie que la valeur de l'attribut annoté est dépendante de l'utilisateur final.

Personnalisation d'association. Le concept `M2AssociationPersonalizer` permet l'annotation du concept `M2Association`. Il signifie que l'établissement des liaisons entre des instances des classes impliquées est géré lors du processus de personnalisation.

Afin de faciliter la manipulation des différents éléments lors des transformations, chacune des extrémités des associations entre les concepts hérités de `ApplicationMM` et ceux définis dans `PersoBaseMM` est navigable. Comme la Figure 5-2 l'illustre, le modèle de l'application annoté avec le vocabulaire du plan d'annotation est utilisé à la fois comme base de la génération des fabriques de personnalisation et des activateurs.

5.2.2.2 Cas d'étude : modèle annoté de l'application LoyaltyManager

Dans l'application *LoyaltyManager*, les porte-monnaie et les comptes de fidélité sont propres aux utilisateurs finaux. Il est donc nécessaire de générer des artefacts de personnalisation pour ces deux classes. Pour un porte-monnaie (`Purse`), il s'agit de gérer l'instanciation et de préciser quels sont l'opérateur et le solde. Pour un compte de fidélité (`LoyaltyAccount`), il s'agit de gérer l'instanciation, de définir quel est l'identifiant du partenaire commercial et quel est le statut initial de l'utilisateur final chez ce partenaire. Les relations entre les porte-monnaie et les comptes de fidélité étant également spécifiques aux utilisateurs finaux, l'association `manages` doit également être annotée.

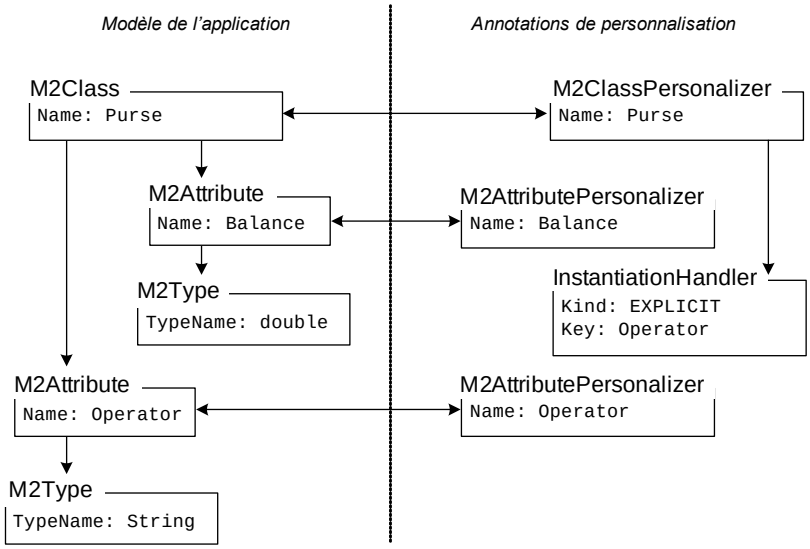


Figure 5-7 : Extrait du modèle annoté de l'application LoyaltyManager

La Figure 5-7 présente un fragment du modèle de l'application annotée. Cet extrait concerne l'annotation de la classe `Purse` : l'instanciation explicite de cette classe, liée au nom de l'opérateur du porte-monnaie, est précisée par l'élément `InstantiationHandler`. L'élément `M2ClassPersonalizer` est lié à deux éléments `M2AttributePersonalizer` pour la définition de l'opérateur et du solde.

Par rapport à la Figure 5-2, nous avons jusqu'ici traités les points *(1)* et *(2)*. La définition de l'espace problème nécessite pour finir de fournir les abstractions nécessaires pour la spécification de profils utilisateurs.

5.2.3 Profil utilisateur

5.2.3.1 Concepts

Comme expliqué dans la section 1.1, la personnalisation nécessite de disposer d'informations sur l'utilisateur qui soient significatives dans le contexte du service offert ou de l'application fournie. Dans la perspective d'une exploitation par un processus automatisé, la structuration de ces données est requise. Ce rôle incombe généralement au profil utilisateur, qui est une collection organisée de données relatives à un individu.

Selon les contextes ou les applications, les données stockées dans un profil utilisateur peuvent être très hétérogènes. Ici, nous introduisons un méta-modèle très simple pour créer des profils utilisateurs éventuellement partageables entre plusieurs applications.

La Figure 5-8 présente le méta-modèle `UserProfileMM`. Il n'introduit qu'un nombre minimal de concepts, suffisamment génériques pour permettre une flexibilité maximale dans la représentation des données. Le méta-modèle explicite une distinction entre les données démographiques et les données applicatives. D'un côté, le concept `Demographics` permet de préciser des informations générales telles que le nom, le prénom ou l'adresse d'un utilisateur. De l'autre côté, les données plus spécifiques aux applications peuvent être

organisées en ensembles (`DataSet`) et sous-ensembles de données matérialisées par des items (`Item`) associés à des valeurs (`ItemValue`). Par le biais de `hasAdditionalItems`, des informations supplémentaires peuvent être associées au concept `Demographics`. Les ensembles de données ainsi que les items sont identifiés par un nom.

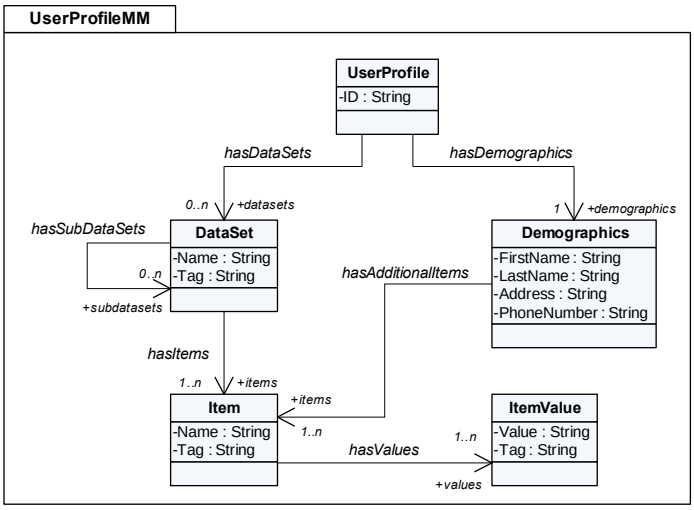


Figure 5-8 : Méta-modèle de profil utilisateur

Dans le cas où un même profil utilisateur est utilisé par plusieurs applications, les données démographiques sont partagées tandis que le concept des ensembles de données fournit un moyen pour séparer les informations relatives à chaque application.

5.2.3.2 Cas d'étude : profil utilisateur pour l'applicationLoyaltyManager

La personnalisation de l'application *LoyaltyManager* consiste à instancier des porte-monnaie et comptes de fidélité particuliers pour chaque utilisateur. La Figure 5-9 présente le profil de l'utilisateur *Sydney Bauer* dont l'identifiant est *SyBa*. À côté des informations démographiques de base, deux ensembles de données sont créés : *Purses* et *Shops*. Les tags des éléments `DataSet` sont utilisés ici pour préciser l'application à laquelle ils sont liés, en l'occurrence *LoyaltyManager*. Pour le premier, nous avons choisi d'utiliser un élément `Item` par porte-monnaie, et de stocker dans des éléments `ItemValue` les informations sur le solde et les comptes de fidélité. Pour le second, un élément `Item` est utilisé pour chaque partenaire commercial. Les spécificités de chaque partenaire sont représentées par le biais d'éléments `ItemValue`. Selon ce profil, *Sydney Bauer* dispose donc de deux porte-monnaie (*WebPurse* et *CityPurse*) utilisés chez trois partenaires commerciaux (*CompStore*, *TransCity* et *WebAuctions*).

Les méta-modèles `ApplicationMM`, `PersoBaseMM`, `PersoAnnotationsMM` et `UserProfileMM` présentés dans cette section permettent une expression rigoureuse des préoccupations liées à la personnalisation d'une application. Dans la suite de ce chapitre, nous présentons d'abord les abstractions définies pour représenter l'espace solution, avant d'expliciter plus

précisément les transformations de modèles permettant la projection de l'espace problème vers celui-ci.

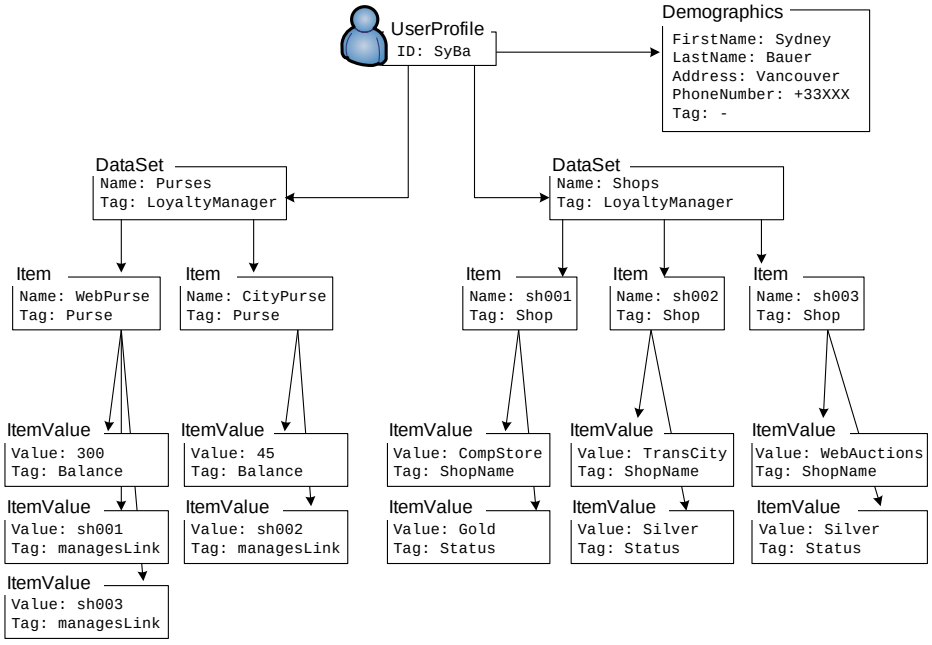


Figure 5-9 : Exemple de profil utilisateur

5.3 Définition de l'espace solution

Dans cette section, nous apportons des réponses aux problèmes *(4) (5)* et *(6) (7)*, exposés dans la section 5.1. Ces problèmes sont une mise en œuvre concrète de la notion de développement logiciel génératif, décrite dans [116] et présentée dans la section 3.3.3 : l'espace solution d'une problématique générative peut constituer l'espace problème d'une autre. Ici, les deux méta-modèles *(4)* et *(6)* respectivement relatifs à la modélisation des fabriques de personnalisation et à celle des activateurs constituent simultanément : *(i)* les abstractions de l'espace problème décrit dans la section précédente et *(ii)* les sources des transformations de type *modèle vers texte* permettant de générer le code final de personnalisation dans un langage spécifique.

5.3.1 Modélisation et production des fabriques de personnalisation

5.3.1.1 Problématique générative

Dans la section 2.5.2, nous avons défini le code personnalisable comme le code original de l'application auquel sont ajoutés des fabriques de personnalisation fournissant les outils pour intervenir sur la création des classes, l'affectation de valeurs aux attributs et l'établissement de liens. Le rôle du méta-modèle `PersonalizationFactoryMM` est de permettre la modélisation des fabriques de personnalisation pour une application donnée. L'intérêt de cette modélisation est double :

- *Elle permet de s'abstraire du langage de programmation cible*. La plupart des applications cartes sont aujourd'hui développées en Java, mais la diversification des machines virtuelles rend probable l'utilisation d'autres langages à moyen terme. La spécification d'un méta-modèle pour les fabriques de personnalisation est un moyen de rendre l'approche applicable à différents langages orientés objets.
- *Elle permet de dissocier la création des fabriques de personnalisation du vocabulaire d'annotation*. Les concepts manipulés dans `PersoBaseMM` et dans `PersonalizationFactoryMM` sont similaires, et une transformation appropriée permettrait de passer directement du modèle annoté au code des fabriques de personnalisation. Néanmoins, la spécification d'un méta-modèle pour ces dernières introduit une étape intermédiaire dans le processus de personnalisation. Cette étape rend possible une évolution séparée des abstractions des espaces problème et solution.

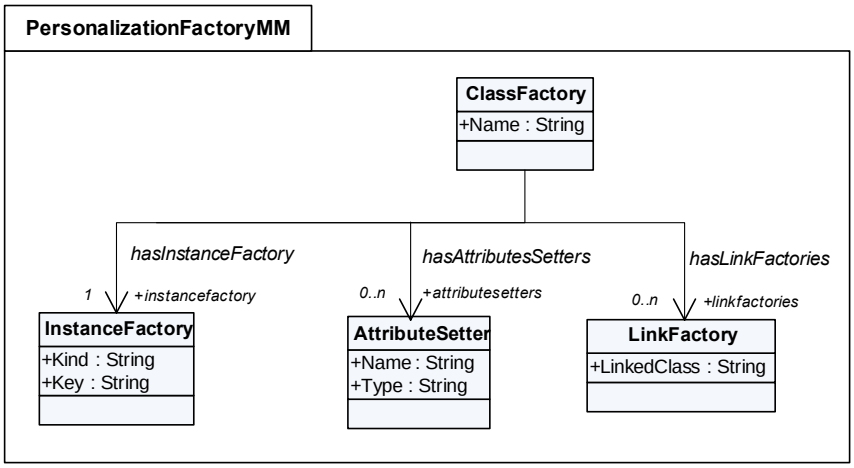


Figure 5-10 : Méta-modèle des fabriques de personnalisation

La Figure 5-10 décrit le méta-modèle `PersonalizationFactoryMM`. Chaque fabrique de personnalisation (`ClassFactory`) est composée d'une fabrique d'instanciation (`InstanceFactory`), d'un ou plusieurs accesseurs d'attributs (`AttributeSetter`) et d'une ou plusieurs fabriques de liens (`LinkFactory`). Tous les éléments d'une fabrique de classe sont accessibles par le biais des associations `hasInstanceFactory`, `hasAttributeSetters` et `hasLinkFactories`.

L'obtention du modèle des fabriques de personnalisation pour une application donnée est le résultat de la transformation *(8)* du modèle annoté pour la personnalisation *(2)*. Ce modèle joue à son tour le rôle de source d'une transformation de type *modèle vers texte* permettant d'obtenir le code final des fabriques de personnalisation. Dans notre contexte, le langage cible est Java. Dans la section 2.5.2, nous avons décrit une approche consistant à coder les fabriques de personnalisation dans des classes internes `Personalizer`. La génération du code de ces classes internes repose sur une simple utilisation de patrons.

Dans le patron illustré par la Figure 5-11, les éléments en **gras** et entourés du symbole **$** marquent les emplacements des valeurs exprimées dans le modèle de la fabrique. Le patron présenté ici est destiné à la génération d'une fabrique de personnalisation dont l'instanciation est explicite. D'autres patrons ont été écrits pour les autres types d'instanciation, mais nous ne les présentons pas ici. Chacune des classes internes générées à partir du modèle des fabriques de personnalisation est ajoutée indépendamment des autres dans sa classe de base.

```
public static class Personalizer {
    static Hastable instances = new Hashtable() ;

    // généré depuis l'élément ClassFactory
    public static $ClassFactory.Name$ create(Object key) {
        $ClassFactory$.Name$ instance = new $ClassFactory$.Name$Impl ();
            instances.put(key, instance);
            return instances;
    }

    // généré depuis l'élément AttributeFactory 'i'
    public static void set$AttributeFactory#i.Name$
        ($ClassFactory$.Name$ inst, $AttributeFactory#i.Type$ value) {
        (($ClassFactory$.Name$Impl)inst).$AttributeFactory#i.Name$ = value;
    }

    // généré depuis l'élément LinkFactory 'i'
    public static void add$LinkFactory#i.LinkedClass$
        ($ClassFactory$.Name$ inst, $LinkFactory.LinkedClass$ linkedinst) {
        (($ClassFactory$.Name$Impl)inst).
        $LinkFactory#i.LinkedClass$s = linkedinst;
    }
}
```

Figure 5-11 : Patron pour le code Java d'une fabrique de personnalisation

5.3.1.2 Cas d'étude : fabrique de personnalisation pour une classe du LoyaltyManager

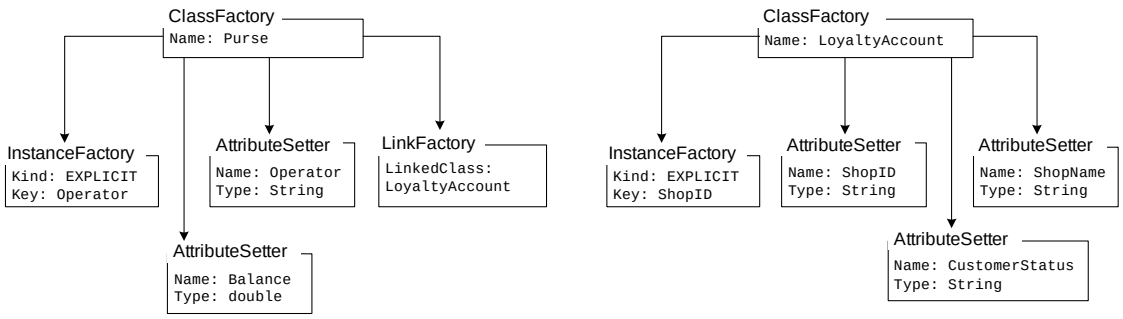


Figure 5-12 : Modèle des fabriques de personnalisation (LoyaltyManager)

La Figure 5-12 présente le modèle des fabriques de personnalisation pour l'application *LoyaltyManager*. La transformation de type *modèle vers texte* basée sur le patron décrit ci-dessus permet d'obtenir le code final des fabriques de personnalisation. Par exemple, la Figure 5-13 présente le code généré Java de la classe interne `Personalizer` devant être intégrée dans la classe `Purse`.

Il est important de noter que le code généré n'est pas totalement indépendant du code original. L'utilisation des patrons repose notamment sur quelques directives que le développeur du code fonctionnel original de l'application doit respecter. Parmi ces directives, on peut noter le fait que nous dissocions l'interface d'une classe (`Purse` par exemple) de son implémentation (`PurseImpl`) ou encore le fait que le nommage des champs doive également respecter certaines contraintes.

```
public static class Personalizer {
    static Hashtable instances = new Hashtable();
    public static Purse create(Object key) {
        Purse instance = new PurseImpl();
            instances.put(key, instance);
            return instance;
    }
    public static void setBalance (Purse inst, double value) {
        ((PurseImpl)inst). Balance = value;
    }
    public static void setOperator (Purse inst, String value){
        ((PurseImpl)inst).Operator = value;
    }
    public static void addLoyaltyAccount (
        Purse inst, LoyaltyAccount linkedinst) {
        ((PurseImpl)inst).LoyaltyAccounts.add(linkedinst);
    }
}
```

Figure 5-13 : Code Java d'une fabrique de personnalisation

5.3.2 Modélisation et production des activateurs personnalisés

5.3.2.1 Problématique générative

Dans la section 2.5.2, nous avons défini le code personnalisé d'une application comme étant une méthode d'activation appelée lors de son amorce. Comme dans le cas des fabriques de personnalisation, modéliser les activateurs permet de rendre l'approche indépendante du langage de programmation cible. La création des modèles d'activateurs pour chacun des utilisateurs est le résultat du processus génératif *(9)* illustré par la Figure 5-2. Ce processus est un tissage entre le modèle annoté de l'application *(2)* et les profils utilisateurs *(3)*. Le méta-modèle `ActivatorMM` constitue donc d'une part, une abstraction de l'espace solution, et d'autre part, la source d'une autre problématique générative, l'implémentation des activateurs dans un langage spécifique.

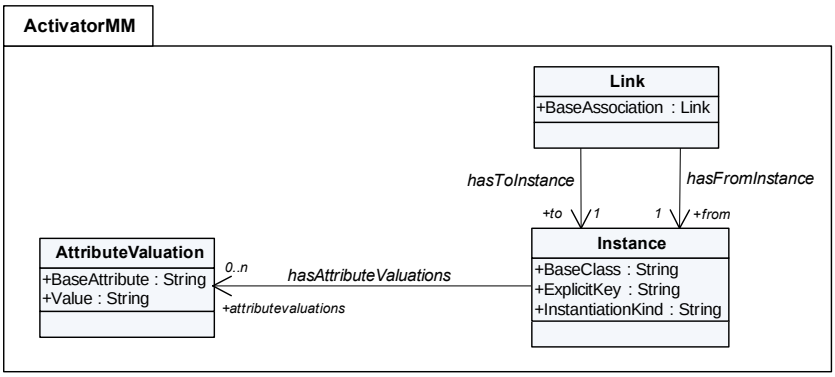


Figure 5-14 : Méta-modèle des activateurs personnalisés

Concrètement, un activateur décrit pour un utilisateur spécifique les instanciations, les affectations de valeurs et les créations de liens entre instances. La Figure 5-14 détaille le méta-modèle `ActivatorMM`. Il est composé de trois concepts basiques : les instances (`Instance`), les affectations (`AttributeValuation`) et les liens (`Link`). Pour la génération du code Java correspondant à un modèle d'activateur, nous nous reposons comme dans le cas des fabriques de personnalisation sur l'utilisation d'un patron. La Figure 5-15 illustre un extrait de ce patron, utilisable pour des instanciations explicites.

```
public static void personalize (String[] argv) {

  // Création d'une instance explicite i
  $Instance#i.BaseClass$
      inst_$Instance#i.ExplicitKey$_$Instance#i.BaseClass$ =
      $Instance#i.BaseClass$Impl.Personalizer.
      create  ($Instance#i.ExplicitKey$);
  // Affectation d'une valeur à un attribut j d'une instance i
  $Instance#i.BaseClass$Impl.Personalizer.
      set$AttributeValuation#j.BaseAttribute$ (
      inst_$Instance#i.ExplicitKey$_$Instance#i.BaseClass$,
      $AttributeValuation#j.Value$);
  // création d'une liaison entre une instance i et une instance k
  $Instance#i.BaseClass$Impl.Personalizer.
      add$Instance#k.BaseClass$ (
      inst_$Instance#i.ExplicitKey$_$Instance#i.BaseClass$,
      inst_$Instance#k.ExplicitKey$_$Instance#k.BaseClass$);
  }
```

Figure 5-15 : Extrait du patron pour la génération du code d'un activateur

5.3.2.2 Cas d'étude : l'activateur de Sydney Bauer pour l'application LoyaltyManager

La Figure 5-16 présente un extrait du modèle de l'activateur pour l'utilisatrice *Sydney Bauer*. La figure se concentre sur les instances de porte-monnaie et ne détaille pas de manière complète les instances des partenaires commerciaux. La Figure 5-17 détaille le code Java généré depuis ce modèle.

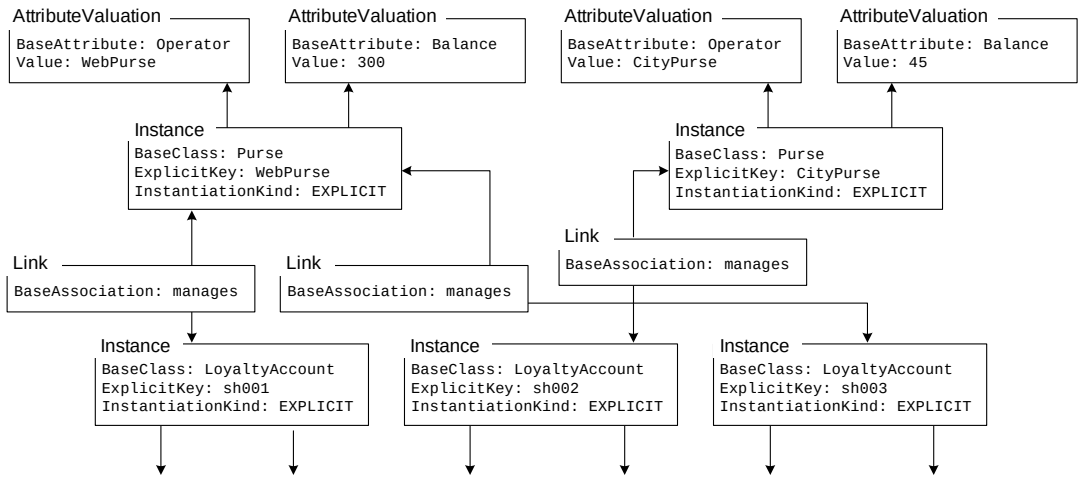


Figure 5-16 : Modèle de l'activateur pour Sydney Bauer

```
public static void personalize ( String[] argv ) {

  // activateur pour Sydney Bauer
  // instanciations et configurations des porte-monnaie
  Purse inst_WebPurse_Purse = PurseImpl.Personalizer.create("WebPurse");
  PurseImpl.Personalizer.setOperator(WebPurse_Purse,"WebPurse");
  PurseImpl.Personalizer.setBalance(WebPurse_Purse,300);
  Purse City_Purse = PurseImpl.Personalizer.create("CityPurse");
  PurseImpl.Personalizer.setOperator(CityPurse_Purse,"CityPurse");
  PurseImpl.Personalizer.setBalance(CityPurse_Purse,45);

  // instanciation et configuration des comptes de fidélité non détaillés ici

  // gestion des liaisons pour les instances de porte-monnaie
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_WebPurse_Purse,inst_sh001_LoyaltyAccount);
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_WebPurse_Purse, inst_sh003_LoyaltyAccount);
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_CityPurse_Purse, inst_sh002_LoyaltyAccount);
  // . . . autres liaisons non détaillées ici . . .
}
```

Figure 5-17 : Extrait du code Java de l'activateur pour Sydney Bauer

Le déploiement d'une application sur une carte à puce consiste à charger les classes personnalisables puis à exécuter le code de l'activateur. Physiquement, le code de l'activateur peut être localisé sur la carte ou à l'extérieur selon le contexte du déploiement – dans le cadre de la chaîne de fabrication ou plus tard en *post-issuance*. Les mécanismes de déploiement sur les cartes à puce ont été présentés dans la section 2.5.3.

5.4 Mécanismes de transformations

Après avoir présenté dans la section 5.2 les abstractions de l'espace problème, nous avons détaillé dans la section 5.3 les différentes abstractions de l'espace solution. Le cœur de l'approche générative pour la personnalisation des applications embarquées réside dans la

projection des abstractions du premier espace vers celles du second, et plus particulièrement dans les deux transformations de type *modèle vers modèle (8)* et *(9)*. La première est une transformation permettant d'obtenir le modèle des fabriques de personnalisation tandis que la seconde permet d'intégrer les données utilisateurs pour obtenir les modèles des activateurs.

5.4.1 Vers le modèle des fabriques de personnalisation

Le processus génératif *(8)* permettant d'obtenir le modèle des fabriques de personnalisation à partir du modèle annoté de l'application est une transformation de type *modèle vers modèle*. Les méta-modèles source et cible étant différents, le but de cette transformation est de mettre en œuvre la projection des concepts du méta-modèle `PersoAnnotationsMM` vers ceux du méta-modèle `PersonalizationFactoryMM`. Une transformation peut être vue comme un ensemble de règles de projection. Ici, la transformation est basée sur un parcours du modèle source et l'application de quatre règles.

Fabriques de classes. Chaque élément `M2ClassPersonalizer` du modèle source requiert la création d'un élément `ClassFactory` dans le modèle cible. L'attribut `Name` du premier est copié dans le second.

Fabriques d'instances. Chaque élément `M2InstantiationHandler` du modèle source requiert la création d'un élément `InstanceFactory` dans le modèle cible. Les attributs du premier sont copiés dans le second. Dans le modèle cible, une association `hasInstanceFactory` est définie pour relier l'élément `InstanceFactory` ainsi créé à l'élément `ClassFactory` correspondant.

Configurateurs d'attributs. Chaque élément `M2AttributePersonalizer` du modèle source requiert la création d'un élément `AttributeSetter` dans le modèle cible. L'attribut `Name` du premier est copié dans le second. L'attribut `Type` est déduit du modèle original de l'application : il prend la valeur de l'attribut `TypeName` appartenant à l'élément `Type` auquel est lié l'élément `M2Attribute` annoté. Dans le modèle cible, une association `hasAttributesSetters` est définie pour lier l'élément `AttributeSetter` ainsi créé à l'élément `ClassFactory` correspondant.

Fabriques d'instances. Chaque élément `M2AssociationPersonalizer` du modèle source requiert la création d'un élément `LinkFactory` dans le modèle cible. L'attribut `Name` du premier est copié dans le second. Une association `hasLinkFactory` est définie pour lier l'élément `LinkFactory` ainsi créé à l'élément `ClassFactory` correspondant.

La transformation décrite ici n'est pas dépendante de l'application à personnaliser. Dans la section 4.3.2, différentes approches pour implémenter les transformations de type *modèle vers modèle* ont été présentées. Pour mettre en œuvre cette transformation, nous optons pour une approche par manipulation directe, c'est-à-dire une approche reposant sur l'utilisation d'APIs de manipulation des modèles. L'implémentation en Java ainsi que la conception d'un cadre structurant générique pour l'implémentation de transformations dans ce langage fait l'objet du Chapitre 6.

5.4.2 Vers les modèle des activateurs personnalisés

Contrairement à la transformation permettant d'obtenir le modèle des fabriques de personnalisation *(8)*, la transformation permettant d'obtenir les modèles d'activateurs *(9)* est spécifique à une application donnée. Plus précisément, elle est dépendante des corrélations entre les éléments annotés pour la personnalisation et la structure des informations stockées dans les profils utilisateurs. Pour cette raison, la transformation *(9)* est en fait une transformation complexe.

La grande majorité des problématiques de transformations de modèles consiste à prendre un modèle comme source pour soit le modifier, soit l'utiliser comme base de la création d'un nouveau modèle. L'obtention d'un modèle d'activateur pour un utilisateur ne rentre pas dans cette catégorie de transformation, dans le sens où elle est le résultat du traitement non pas d'un, mais de deux modèles : le modèle annoté de l'application et le profil de cet utilisateur. La transformation ayant deux modèles sources, on parle de transformation 2-1 – à opposer aux transformations 1-1.

Ce schéma en Y — les deux branches supérieures du Y représentent les deux sources tandis que le pied du Y représente le modèle cible – ne permet d'adopter telles quelles l'une ou l'autre des approches classiques décrites dans la section 4.3.2. La dualité des modèles source impose une stratégie particulière. Dans le Chapitre 6, nous proposons et détaillons une solution pour implémenter ce type de transformation. De manière simplifiée, notre approche consiste à restructurer la problématique pour changer le Y en transformation 1-1 paramétrée. La source de cette transformation 1-1 est le modèle de l'application annoté tandis que le paramétrage est réalisé par rapport au profil utilisateur. Ainsi, la transformation comporte trois règles principales.

Création des instances. Selon les données contenues dans le profil utilisateur conforme au méta-modèle `UserProfileMM`, chaque élément `M2ClassPersonalizer` du modèle source est susceptible d'entraîner la création d'un ou plusieurs éléments `Instance` dans le modèle cible. Dans le cas d'une instanciation de type explicite, la clef est récupérée dans le profil utilisateur.

Configurations des attributs. Chaque élément `M2AttributePersonalizer` du modèle source entraîne la création d'autant d'éléments `AttributeValuation` que d'éléments `Instance` ont été créés pour la classe correspondante. Des associations `hasAttributeValuation` sont définies à ce stade dans le modèle cible. Les valeurs des attributs `Value` des éléments `AttributeValuation` sont récupérées dans le profil utilisateur.

Création des liaisons. Selon les données contenues dans le profil utilisateur, chaque élément `M2AssociationPersonalizer` du modèle source entraîne la création d'un ou plusieurs éléments `Link` et la définition des associations `hasFromEnd` et `hasToEnd` avec les éléments `Instance` correspondants.

Cette transformation est répétée pour chacun des utilisateurs. Les modèles résultats, conformes au méta-modèle `ActivatorMM`, sont utilisés pour générer le code des activateurs.

Cette transformation étant dépendante de l'application à personnaliser, son implémentation est à la charge du concepteur de l'application. Dans le Chapitre 6, nous illustrons l'implémentation de cette transformation 2-1 dans le contexte de notre cas d'étude.

5.5 Synthèse

Notre objectif est de maximiser l'automatisation du processus de personnalisation des applications embarquées pour cartes à puce. Sur les bases des choix techniques et méthodologiques motivés et exprimés dans le Chapitre 2, nous avons cherché dans cette perspective à élever le niveau d'abstraction pour mettre en œuvre un processus génératif. Dans ce chapitre, nous avons d'abord présenté la vision globale d'une approche dirigée par les modèles et identifié un ensemble de problèmes à résoudre pour l'implémenter. Parmi eux, la définition des abstractions des espaces problème et solution :

- *Espace problème* : définition des méta-modèles permettant de modéliser l'application originale (`ApplicationMM`), l'annotation de cette application (`PersoBaseMM` et `PersoAnnotMM`) et la modélisation des profils d'utilisateurs (`UserProfileMM`).
- *Espace solution* : conception des méta-modèles permettant de modéliser les fabriques de personnalisation (`PersonalizationFactoryMM`) ainsi que les activateurs (`ActivatorMM`) et implémentation de sous problématiques génératives permettant d'utiliser ces deux méta-modèles pour la génération du code Java de personnalisation.

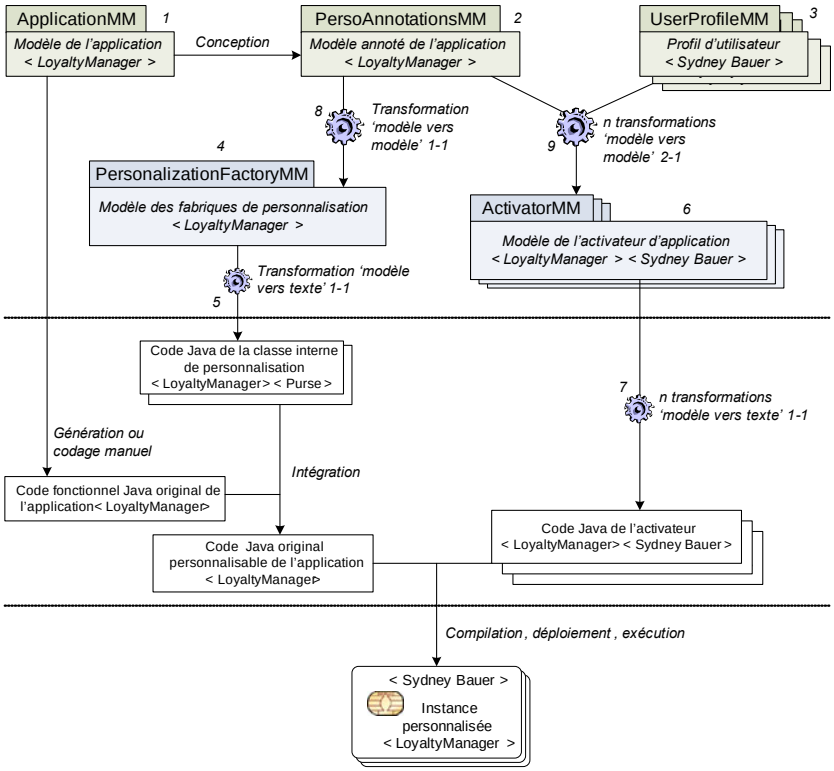


Figure 5-18 : Vue d'ensemble du processus génératif

La Figure 5-18 fournit une vue d'ensemble des solutions proposées pour les différents problèmes identifiés en début de chapitre. Elle constitue une sorte de raffinement de la Figure 5-2. La mise en œuvre et l'expérimentation du processus génératif avec le cas d'étude du *LoyaltyManager* valident et confirment la pertinence des concepts introduits dans la section 2.5. Néanmoins, une phase de consolidation de l'approche est nécessaire pour une utilisation réelle par les fournisseurs d'applications embarquées :

- *L'utilisation d'UML* en lieu et place de notre méta-modèle expérimental `ApplicationMM` peut requérir la création de profils UML pour refléter les méta-modèles de personnalisation.
- Selon les besoins réels des fournisseurs, *un enrichissement des méta-modèles* présentés dans ce chapitre peut s'avérer nécessaire.
- Le *développement d'outils supportant l'approche est souhaitable* : par exemple, un outil graphique pour l'annotation du modèle original, la forme d'une extension à un environnement de modélisation existant, rendrait cette étape plus pratique.

Le cœur du processus génératif réside dans la projection de l'espace problème vers l'espace solution. Nous avons donc présenté de manière informelle les deux transformations de type *modèle vers modèle* qui permettent d'obtenir respectivement le modèle des fabriques de personnalisation et les modèles des activateurs. Dans le chapitre suivant, nous approfondissons le traitement de ces transformations, en nous concentrant *(i)* sur la mise en œuvre d'un cadre de conception pour l'implémentation de transformations en Java et *(ii)* sur une stratégie permettant d'implémenter la transformation 2-1 dont nous avons explicité le besoin.

Chapitre 6

Implémentation des transformations de modèles

Le cœur d'une approche générative pour le développement de logiciel réside dans la projection des abstractions de l'espace problème vers celles de l'espace solution. Dans le cas d'une approche dirigée par les modèles, cette projection est réalisée par le biais de transformations, du type *modèle vers modèle* ou du type *modèle vers texte*. Dans le chapitre précédent, nous avons décrit une approche générative pour la personnalisation des applications embarquées dans les cartes à puce. Si nous avons évoqué le rôle central des deux transformations permettant la création du modèle des fabriques de personnalisation et des modèles des activateurs personnalisés, nous n'avons pas détaillé leur implémentation.

Plusieurs approches sont possibles pour mettre en œuvre une transformation de type *modèle vers modèle*. Dans la section 4.3, nous avons présenté une taxonomie des transformations et énuméré les types de transformation les plus courants. Nous décrivons et illustrons dans ce chapitre la conception d'un cadre structurant pour l'implémentation de transformations de type 1-1 en Java.

Dans le contexte de la personnalisation des applications embarquées, la création des modèles d'activateurs personnalisés requiert la mise en œuvre d'une transformation devant réaliser un tissage entre deux modèles sources – on parle de *transformation 2-1* ou de *transformation en Y*. Dans ce chapitre, nous proposons une solution pour gérer de telles

transformations. Pour implémenter cette solution, nous définissons une extension au cadre structurant pour les transformations 1-1. Nous illustrons l'approche avec le cas d'étude du *LoyaltyManager*.

Ce chapitre est organisé comme suit.

- La section 6.1 justifie le choix d'une approche par manipulation directe et précise les objectifs visés à travers la conception d'un cadre structurant pour l'implémentation des transformations en Java.
- La section 6.2 décrit notre vision de la structure d'une transformation de modèle de type 1-1. Cette vision est inspirée de la taxonomie décrite dans la section 4.3.2.
- La section 6.3 présente l'implémentation du cadre structurant pour les transformations de type 1-1. Cette implémentation repose sur la structure décrite dans la section précédente. La mise en œuvre de ce cadre est illustrée par la transformation permettant d'obtenir le modèle des fabriques de personnalisation.
- La section 6.4 présente une proposition pour la mise en œuvre du tissage de modèle, consistant à raffiner la transformation de type 2-1 en transformation de type 1-1 paramétrée.
- La section 6.5 décrit l'extension apportée au cadre structurant pour supporter le tissage de modèle. Son utilisation est illustrée avec l'implémentation de la transformation permettant d'obtenir les modèles des activateurs personnalisés de l'application *LoyaltyManager*.
- La section 6.6 conclut le chapitre.

6.1 Motivations et objectifs

La section 4.4.2 a exposé le manque de maturité de l'outillage disponible actuellement pour supporter la mise en pratique du MDD. La spécification QVT n'étant pas finalisée, et ne souhaitant pas nous lier avec un outil spécifique, nous avons choisi d'adopter une approche ad hoc, et de n'implémenter que ce dont nous avions effectivement besoin. Le raisonnement derrière ce choix est similaire à celui qui nous a conduit à utiliser un méta-modèle expérimental plutôt qu'UML (section 5.2.1) pour décrire nos applications. Parmi les différentes approches possibles, nous avons retenu celle par manipulation directe – celle de plus bas niveau – qui offre une souplesse intéressante dans le contexte de nos expérimentations.

Une approche par manipulation directe se base sur un accès aux modèles source et cible par le biais d'APIs. Dans la section 4.2.3, nous avons présenté JMI, une spécification pour la manipulation en Java de modèles MOF. Nous nous basons sur cet environnement. Dans le contexte de l'approche générative pour la personnalisation, nous avons deux transformations à implémenter. La première est une transformation générique, réutilisable quelle que soit l'application à personnaliser. La seconde est une transformation fortement dépendante des applications, et reste donc à la charge du concepteur de la personnalisation.

D'une transformation à l'autre, les mécanismes mis en œuvre sont relativement similaires. Pour cette raison, il est intéressant d'opter pour la conception d'un cadre structurant fournissant l'implémentation du moteur et de la structure de la transformation. Cette solution permet au développeur de se concentrer uniquement sur la définition de la logique des règles. Nous basons ce cadre de conception sur la structure de transformation décrite dans la section suivante.

6.2 Structure d'une transformation

Nous avons présenté dans la section 4.3.2 une analyse de la structure typique d'une transformation. Les différentes caractéristiques décrites – règles, relation entre modèle et cible, organisation et ordonnancement des règles, traçabilité, direction – constituent autant de points de variation qui différencient les approches d'implémentation. Sans prétendre décrire une transformation universelle, nous exploitons ici cette analyse pour proposer une structure suffisamment générique pour être utilisée comme base architecturale du cadre de conception Java.

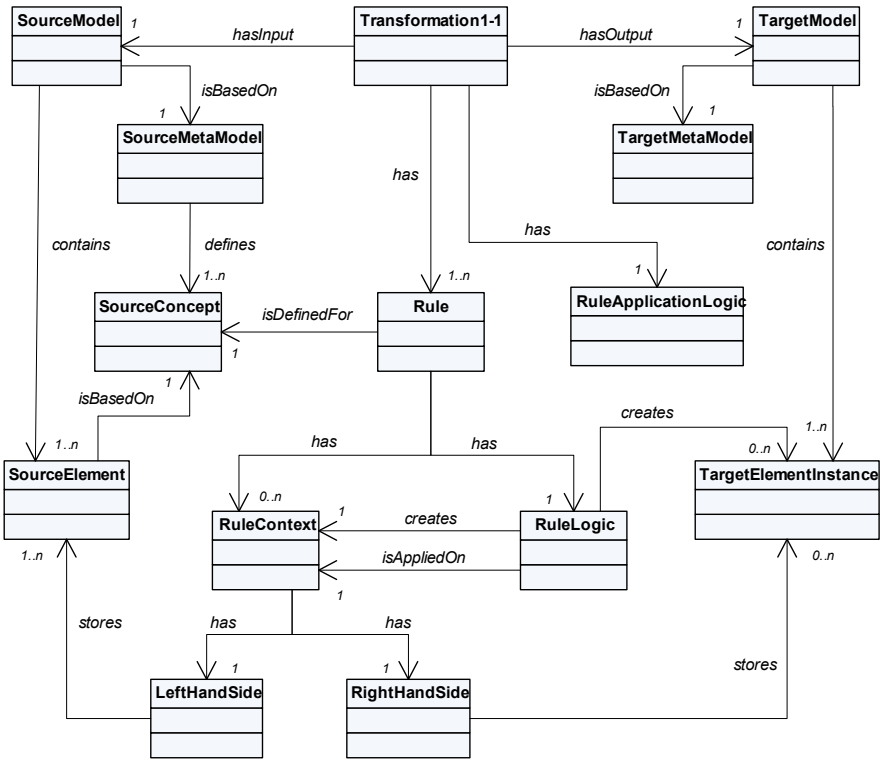


Figure 6-1 : Structure d'une transformation 1-1

La Figure 6-1 décrit la structure d'une transformation 1-1 et précise notre terminologie. Une telle transformation prend comme paramètre d'entrée un modèle source (`SourceModel`) et retourne un modèle cible (`TargetModel`). Ces deux modèles sont chacun basés sur un méta-modèle (`SourceMetaModel` et `TargetMetaModel`). Une transformation est

composée d'un ensemble de règles (`Rule`) et d'une stratégie d'application de ces règles (`RuleApplicationLogic`). Cette stratégie permet de définir dans quel ordre les règles sont appliquées. Une règle est définie pour un concept particulier du méta-modèle source (`SourceConcept`). Une règle est composée :

- *D'un ensemble de contextes* (`RuleContext`). Un contexte est une construction permettant de stocker les données nécessaires pour l'application de la règle à une instance spécifique du concept source pour lequel elle est définie. Un contexte est composé de deux côtés. Le côté gauche (`LeftHandSide`) stocke obligatoirement la référence sur l'élément à transformer et optionnellement d'autres données ou d'autres éléments nécessaires à l'exécution de la logique. Le côté droit (`RightHandSide`) peut être utilisé pour stocker des éléments du modèle cible (`TargetElement`) – nécessaire par exemple pour la création d'associations entre des éléments du modèle cible.
- *D'une logique* (`RuleLogic`) qui définit les traitements à appliquer, et notamment la création d'un ou plusieurs éléments du modèle cible. Les informations et éléments stockés dans les contextes constituent des paramètres pour ces traitements.

L'exécution d'une règle consiste à parcourir le modèle source pour préparer les contextes puis à appliquer la logique à chacun d'eux. Afin de permettre une imbrication des règles, la logique peut également être utilisée pour la création de sous contextes.

Si la notation utilisée dans la Figure 6-1 est calquée sur celle du MOF – sans les rôles et les attributs –, ce « méta-modèle » n'en est pas réellement un dans le sens où il ne permet pas d'exprimer des modèles spécifiant la transformation de manière complète. Néanmoins, il aide à la compréhension du cadre de conception détaillé dans la section suivante.

6.3 Implémentation de transformations 1-1 en Java

6.3.1 Cadre de conception

Les approches par manipulation directe pour les transformations de type *modèle vers modèle* sont basées sur une représentation des modèles au niveau du langage de programmation, typiquement par le biais d'APIs. Nous basons le cadre de conception sur les APIs JMI et utilisons l'implémentation proposée par l'outil *Modfact* [125]. JMI permet de manipuler les modèles soit par le biais d'interfaces réflexives, soit par le biais d'interfaces spécifiques utilisant la terminologie du modèle manipulé. Les premières sont utiles pour le code générique au cœur du cadre de conception, tandis que les secondes sont pratiques pour implémenter les transformations elles-mêmes.

Le cadre de conception présenté ici implémente les mécanismes de base comme l'importation du modèle source ou l'exportation du modèle cible, les routines d'application de la logique des règles à leurs contextes, etc. Les mécanismes de la transformation et ceux des règles sont implémentés dans des classes abstraites distinctes. La conception d'une transformation basée sur ce cadre consiste à *(i)* spécialiser les mécanismes de base de la

transformation en implémentant notamment la stratégie d'application des règles et *(ii)* définir dans cette classe chaque règle comme une classe interne.

6.3.1.1 Mécanismes de base des transformations

Les mécanismes de base des transformations sont implémentés dans deux classes, `Transformation` et `OneToOneTransformation` (Figure 6-2 et Figure 6-3). Si cette séparation ne se justifie pas réellement dans le cadre des transformations 1-1, elle prend son sens dans l'extension ultérieure du cadre pour supporter l'implémentation des transformations 2-1. Le rôle de la classe abstraite `OneToOneTransformation` est de permettre le chargement du modèle source par le biais de la méthode `setSourceModel`, prenant comme paramètre le modèle sous forme sérialisée. La classe `Transformation` spécifie simplement que transformer un modèle consiste à appliquer des règles et à sérialiser le résultat. La méthode abstraite `applyRules` correspond à l'élément *RuleApplicationLogic* de la Figure 6-1. Le concepteur de la transformation implémente dans cette méthode les appels aux règles définies dans des classes internes.

```
package transformations;

public abstract class Transformation {
  protected RefPackage _TargetModel;
  public void dumpTargetModel(String filename) {
      // non détaillé ici : écriture du modèle au format XMI
      // dans le fichier 'filename'
  }
  public void transforms(String filename) {
      this.applyRules();
      this.dumpTargetModel(filename);
  }
  abstract protected void applyRules();
  abstract protected void initModels() throws Exception;
}
```

Figure 6-2 : Classe abstraite Transformation

```
package transformations;

public abstract class OneToOneTransformation extends Transformation {
  protected RefPackage _SourceModel;
  public OneToOneTransformation(){
      // traitement des exceptions non détaillé ici
      initModels();
  }
  public void setSourceModel(String filename) {
      // non détaillé ici : chargement du modèle depuis son format sérialisé
  }
}
```

Figure 6-3 : Classe abstraite OneToOneTransformation

6.3.1.2 Mécanismes de base des règles

La classe abstraite `Rule` implémente l'ossature d'une règle de transformation. Cette classe est spécialisée par des classes internes dans la transformation finale. La classe `RuleContext`, qui définit deux vecteurs et leurs accesseurs pour les côtés gauche et droit

(*LeftHandSide* et *RightHandSide* sur la Figure 6-1) n'est pas décrite ici. L'exécution d'une règle consiste à créer un ensemble de contextes, puis à appliquer la logique de la règle à chacun d'eux.

```
package transformations;

public abstract class Rule {
  protected Vector _RuleContexts = null;
  public Rule() {}
  public Collection applyRule(RefPackage rp, String en) {
      _RuleContexts = new Vector();
      createRuleContexts(rp,en);
      return executeRules();
  }
  public Collection executeRules() {
      Vector result = new Vector();
      Iterator it = _RuleContexts.iterator();
      while (it.hasNext()) {
          RuleContext rc = (RuleContext)it.next();
          result.addElement(applyRuleLogic(rc));
      }
      return result;
  }
  public void createRuleContexts(RefPackage modelsource, String conceptname)
  {
      RefClass rc = modelsource.refClass(conceptname);
      Iterator it = rc.refAllOfClass().iterator();
      while (it.hasNext()) {
          _RuleContexts.addElement(
          createRuleContextForElement((RefObject)it.next()));
      }
  }
  abstract protected RuleContext createRuleContextForElement(RefObject rc);
  abstract protected RuleContext applyRuleLogic(RuleContext rc);
}
```

Figure 6-4 : Classe abstraite Rule

La Figure 6-4 détaille la classe `Rule` :

- *Le rôle de la méthode* `createRuleContexts` est de parcourir le modèle source (`modelsource`) pour rechercher toutes les instances du concept pour laquelle la règle est définie (`conceptname`). Elle libère le développeur de la transformation de cette tâche, mais lui impose d'implémenter la construction du contexte associé à chacune des instances par le biais de la méthode abstraite `createRuleContextForElement`. Le résultat de cette méthode est un vecteur de contextes (`_RuleContexts`).
- Le rôle de la méthode `executeRules` est de parcourir le vecteur de contextes et d'appliquer à chacun la logique de la règle. Le codage de cette logique reste à la charge du développeur de la transformation qui doit implémenter la méthode abstraite `applyRuleLogic`. Le résultat de la méthode executeRules est une collection de contextes qui peut être utilisée comme paramètre d'une règle imbriquée (classe `SubRule`, non décrite ici).
- Le rôle de la méthode `applyRule` est simplement d'initialiser le vecteur de contextes et d'appeler les deux méthodes précédentes.

6.3.2 Exemple de mise en œuvre

Dans cette section, nous illustrons l'utilisation du cadre de conception pour l'implémentation de la transformation spécifiée dans la section 5.4.1. Cette transformation prend comme source un modèle d'application annoté pour la personnalisation (conforme au méta-modèle `PersoAnnotationsMM`) et créé le modèle des fabriques de personnalisation (conforme à `PersonalizationFactoryMM`). La Figure 6-5 détaille la classe Java implémentant cette transformation qui met en œuvre quatre règles. L'ordre d'application des règles déterminé par la méthode `applyRules` est important. Par exemple, la transformation des éléments `M2AttributePersonalizer` du modèle source requerrant la création d'associations entre les éléments `AttributeSetter` et `ClassFactory` dans le modèle cible, il est nécessaire que ces derniers aient déjà été créés.

```
package transformations;

public class AAM2PersonalizationFactory extends OneToOneTransformation {
  public void initModels() throws Exception {
      _SourceModel = (PersoAnnotationsMMPackageImpl.
          create(SystemPaths.PROJECTPATH + "/metamodels/metamodels.xml"));
      _TargetModel = PersonalizationFactoryMMPackageImpl.
          create(SystemPaths.PROJECTPATH + "/metamodels/metamodels.xml");
  }
  public PersoAnnotationsMMPackage getSourceModel() {
      return (PersoAnnotationsMMPackage)_SourceModel;
  }
  public PersonalizationFactoryMMPackage getTargetModel() {
      return (PersonalizationFactoryMMPackage)_TargetModel;
  }
  protected void applyRules() {
      M2ClassPersonalizerRule cpr = new M2ClassPersonalizerRule();
      cpr.applyRule(getSourceModel(),"M2ClassPersonalizer");
      M2AttributePersonalizerRule apr = new M2AttributePersonalizerRule();
      apr.applyRule(getSourceModel(),"M2AttributePersonalizer");
      M2InstantiationHandlerRule ihr = new M2InstantiationHandlerRule();
      ihr.applyRule(getSourceModel(),"M2InstantiationHandler");
      M2AssociationPersonalizerRule apr = new M2AssociationPersonalizerRule();
      apr.applyRule(getSourceModel(),"M2AssociationPersonalizer");
  }
  class M2ClassPersonalizerRule extends Rule {
      protected RuleContext createRuleContextForElement(RefObject element) {
          // implémentation non détaillée ici
      }
      protected RuleContext applyRuleLogic(RuleContext rc) {
          // implémentation non détaillée ici
      }
  }   // autres classes internes de règles non détaillées ici
}
```

Figure 6-5 : Exemple de transformation 1-1

Dans la classe de transformation, deux méthodes `getSourceModel()` et `getTargetModel()` ont été créées. Elles ont pour but de faciliter l'écriture du code en évitant la multiplication des typages forcés[35] lors de l'utilisation des interfaces spécifiques de JMI plutôt que des interfaces réflexives.

[35] *Cast* dans le vocabulaire anglo-saxon.

```
class M2AttributePersonalizerRule extends Rule {
    protected RuleContext createRuleContextForElement(RefObject element) {
        M2AttributePersonalizer ap = (M2AttributePersonalizer) element;
        RuleContext rc = new RuleContext();
        // Côté gauche : ajout de l'élément et du type de l'attribut
        rc.addLHSElement(ap);
        M2Attribute a = ap.getM2Attribute();
        String typename = a.getM2Type().getTypeName();
        // Côté gauche : ajout du type de l'attribut
        rc.addLHSElement(typename);
        // Recherche dans modèle cible de l'élément ClassFactory correspondant
        M2Class c = a.getM2Class();
        M2ClassPersonalizer cp = c.getClassPersonalizer();
        Iterator it = getTargetModel().
            getM2ClassFactory().refAllOfClass().iterator();
        while (it.hasNext())    {
            M2ClassFactory cf = (M2ClassFactory)it.next();
            if ((cf.getName()).equals(cp.getName())) {
                // Côté droit : ajout de l'élément ClassFactory correspondant
                rc.addRHSElement(cf);
                break;
            }
        }
        return rc;
    }
    protected RuleContext applyRuleLogic(RuleContext rc) {
        M2AttributePersonalizer ap = (M2AttributePersonalizer)
            rc.getLeftHandSide().get(0);
        String typename = (String) rc.getLeftHandSide().get(1);
        String name = ap.getName();
        ClassFactory cf = (ClassFactory) rc.getRightHandSide().get(0);
        // création de l'élément AttributeSetter dans le modèle cible
        AttributeSetter as = getTargetModel().
        getAttributeSetter().createAttributeSetter(name, typename);
        // création d'un lien avec l'élément ClassFactory correspondant
        this.getTargetModel().getHasAttributesSetters().add(as,cf);
        return null;
    }
}
```

Figure 6-6 : Implémentation d'une règle de transformation

La Figure 6-6 détaille l'implémentation de la règle qui permet de créer les configurateurs d'attributs (`AttributeSetter`) dans le modèle de la fabrique de personnalisation (voir la spécification de la transformation dans la section 5.4.1). Cette règle est intéressante dans la mesure où elle illustre l'utilisation du côté droit d'un contexte (*LeftHandSide*) et la création de liens entre éléments dans le modèle cible. Le rôle de la méthode `createRuleContextForElement` est de préparer les paramètres d'application de la règle pour un élément donné (paramètre `element`) :

- *Stockage* de l'élément source dans le LHS,
- *Recherche* de l'attribut annoté et ajout dans le LHS,
- *Recherche* de l'élément `ClassFactory` dans le modèle cible et ajout dans le RHS.

La logique de la règle consiste, d'une part, à créer un élément `AttributeSetter` dans le modèle cible avec les attributs `Name` et `Type` récupérés dans le contexte d'application, et d'autre part, à établir une liaison entre cet élément et l'élément `ClassFactory`

correspondant. Cette règle ne contenant pas de règle imbriquée, il n'est pas nécessaire de formater un nouveau contexte d'application dans le résultat de `applyRuleLogic`.

La transformation décrite ici n'est pas spécifique à l'application *LoyaltyManager* que nous avons utilisée comme cas d'étude, elle est donc réutilisable telle quelle pour la personnalisation d'autres applications. Elle n'est en fait dépendante que des méta-modèles `PersoAnnotationsMM` et `PersonalizationFactoryMM`.

Le cadre structurant implémente des mécanismes de base et permet, dans une certaine mesure, de réduire la charge de développement incombant au concepteur d'une transformation. Néanmoins, l'implémentation des règles reste à la charge de ce dernier. La difficulté de cette tâche croît avec la complexité des méta-modèles source et cible.

6.4 Proposition pour le tissage de modèles

Le cadre de transformation présenté dans la section précédente permet l'implémentation de transformation 1-1 mais ne permet pas celle des transformations 2-1, plus complexes à mettre en œuvre. En se basant sur le contexte de la personnalisation des applications embarquées, cette section pose le problème du tissage de modèles et détaille une proposition de solution.

6.4.1 Problématique

Dans le contexte de la personnalisation des applications embarquées, l'obtention des modèles d'activateurs est le résultat de combinaisons entre le modèle de l'application annotée (conforme au méta-modèle `PersoAnnotationsMM`) et les profils des utilisateurs (conformes au méta-modèle `UserProfileMM`).

La Figure 6-7 illustre des extraits des modèles impliqués dans la personnalisation des activateurs pour l'application *LoyaltyManager*. Par exemple, la création des porte-monnaie pour un utilisateur donné consiste, à partir de l'élément `M2ClassPersonalizer` dont l'attribut `Name` vaut *'Purse'* dans le modèle annoté, à effectuer les opérations suivantes :

- Parcourir dans le profil utilisateur l'ensemble des `Item` (appartenant au `DataSet` dont l'attribut `Name` est *'Purses'*) dont l'attribut `Tag` est *'Purse'*. Pour chacun de ces items, créer un élément `Instance` dans le modèle cible, avec *'Purse'* comme valeur de l'attribut `BaseClass`.
- Selon les valeurs des attributs de l'élément `InstantiationHandler` correspondant dans le modèle annoté, définir les valeurs des attributs `ExplicitKey` et `InstantiationKind` de l'élément `Instance` dans le modèle cible (ici, *'WebPurse'* et *'EXPLICIT'*).

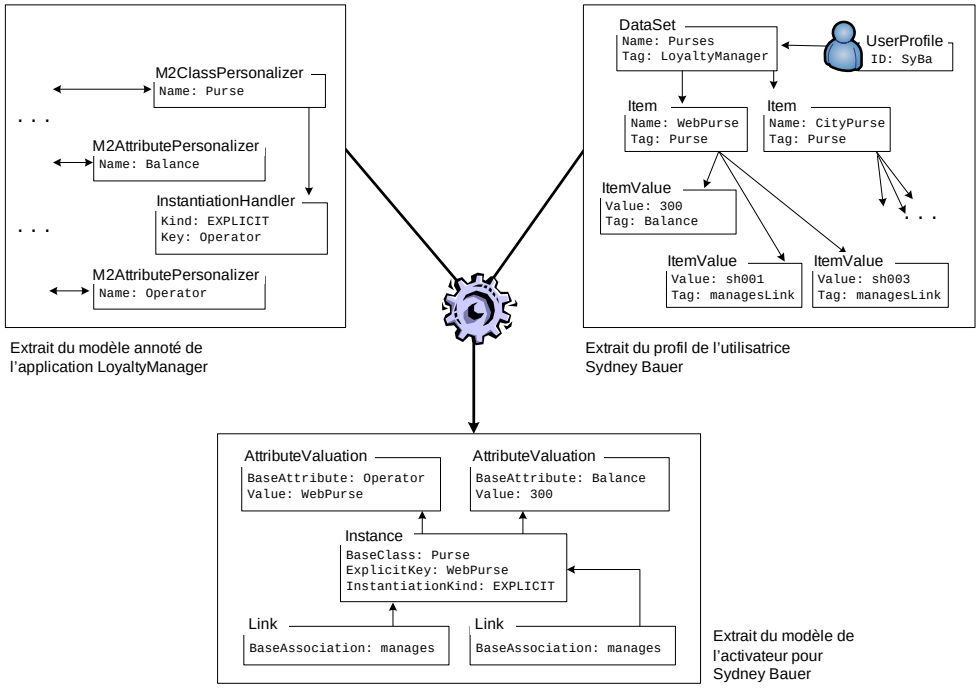


Figure 6-7 : Obtention d'une modèle d'activateur personnalisé

- Les éléments `M2AttributePersonalizer` dans le modèle annoté nécessitent la création d'éléments `AttributeValuation` dans le modèle cible. L'attribut `BaseAttribute` des seconds prend la valeur de l'attribut `Name` des premiers.
- Dans le cas de l'élément `M2AttributePersonalizer` dont l'attribut `Name` est *'Operator'*, la valeur de l'attribut `Value` de l'élément `AttributeValuation` créé est définie par la valeur de l'attribut `Name` de l'élément `Item` correspondant, c'est-à-dire *'WebPurse'*.
- Dans le cas de l'élément `M2AttributePersonalizer` dont l'attribut `Name` est *'Balance'*, la valeur de l'attribut `Value` de l'élément `AttributeValuation` créé est définie par la valeur `Value` de l'élément `ItemValue` dont le `Tag` est *'Balance'*, c'est-à-dire *'300'*.

Des règles de combinaisons similaires sont nécessaires pour la création des instances de comptes de fidélité ainsi que pour l'établissement des liens entre ces dernières et les instances de porte-monnaie. La particularité de cette transformation est de ne pas être une transformation 1-1, mais une transformation 2-1, c'est-à-dire une transformation ayant deux modèles sources. La Figure 6-7 illustre une configuration en Y : tandis que les deux branches supérieures du Y correspondent aux deux modèles sources de la transformation, le pied représente le modèle cible.

Contrairement à la transformation permettant d'obtenir les modèles de fabriques de personnalisation qui est générique, la transformation permettant d'obtenir les activateurs est spécifique à chaque application. De surcroît, elle doit être exécutée pour chaque utilisateur. Le cadre structurant détaillé dans la section précédente étant destiné à

l'implémentation de transformation 1-1, il n'est pas conçu pour supporter un deuxième modèle source. Il est donc nécessaire de le modifier à cet effet.

6.4.2 Vers une transformation 1-1 paramétrée

Malgré des caractéristiques et des implémentations différentes, les transformations de type *modèle vers modèle* présentées dans la section 4.3.2 reposent pour la plupart sur une stratégie commune qui consiste à parcourir le modèle source, et appliquer à chacun de ses éléments des traitements spécifiés par des règles de projections. Cette stratégie, naturelle pour les transformations 1-1, n'est pas applicable telle quelle dans le cas de transformations 2-1 où la présence de deux modèles sources rend plus complexes les relations entre les modèles sources et cibles. Une transformation 2-1 est en fait un tissage de modèles.

La configuration en Y est un schéma récurrent dans l'ingénierie logicielle [149]. La programmation orientée aspects [51] [86] présentée dans la section 3.2 est notamment une illustration très pertinente de la nécessité de combiner plusieurs sources d'informations – le code original et les aspects. Dans le contexte du MDD, ce schéma en Y est notamment au cœur de la définition de la MDA : tandis que les deux branches supérieures du Y représentent la logique du système (PIM) et la description de la plate-forme cible (PDM, *Platform Description Model*), le pied du Y représente le modèle spécifique à cette plate-forme (PSM) [102]. La mise en pratique concrète d'approches logicielles dirigées par les modèles met en avant le besoin de répondre à cette problématique du schéma en Y.

Puisque l'on ne dispose pas de la même connaissance sur les tissages de modèles que sur les transformations 1-1 (section 4.3), nous nous proposons ici de raffiner la configuration en Y pour la faire ressembler à une problématique plus connue. Deux aspects paraissent essentiels :

- *Le cœur d'une opération de tissage réside dans les corrélations entre les deux modèles sources*. Puisque la création d'un élément dans le modèle cible est le résultat de combinaisons d'informations exprimées dans deux modèles sources, il est nécessaire de formaliser les liens entre les éléments de chacun des deux modèles.
- *Il n'est pas possible de baser le mécanisme de la transformation sur un parcours de deux modèles simultanément*. Deux stratégies sont envisageables pour mettre en œuvre un tissage de modèles. La première consiste à baser le tissage sur la création successive de chacun des éléments du modèle cible. Les règles de transformation ne sont donc plus définies pour des concepts des méta-modèles sources, mais pour ceux du méta-modèle cible. La deuxième stratégie consiste à ne pas considérer les deux modèles sources comme réellement équivalents. Le tissage est alors basé sur le parcours d'un des deux modèles, tandis que l'autre est utilisé comme un modèle de paramétrage.

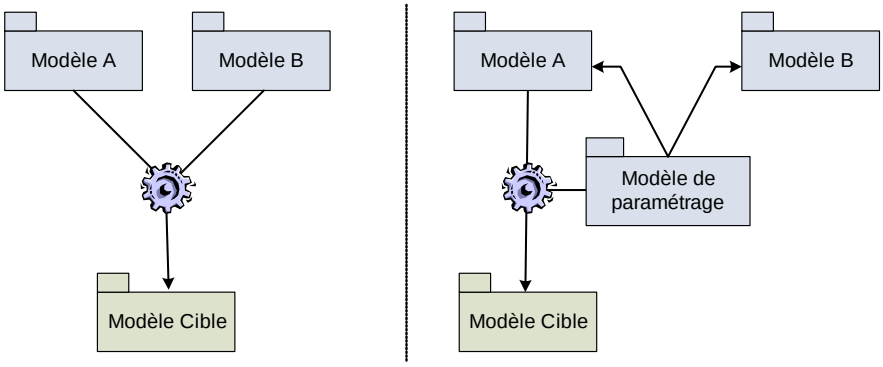


Figure 6-8 : Raffinement du schéma en Y

La Figure 6-8 présente une proposition de raffinement de la configuration en Y. La transformation 2-1 devient une transformation 1-1 paramétrée. Les modèles A et B étant les deux modèles sources, notre proposition consiste à utiliser le modèle A comme base du parcours de la transformation, et à utiliser un modèle de paramétrage contenant les corrélations entre A et B.

Selon les contextes, le tissage entre deux modèles sources est susceptible de requérir un grain de relations plus fin que la mise en correspondance des concepts définis par leurs deux méta-modèles. Dans le cas de l'obtention du modèle des activateurs personnalisés notamment, le tissage entre le modèle de l'application annoté et les profils d'utilisateurs est basé sur une mise en correspondance au niveau des éléments eux-mêmes (section 6.4.1).

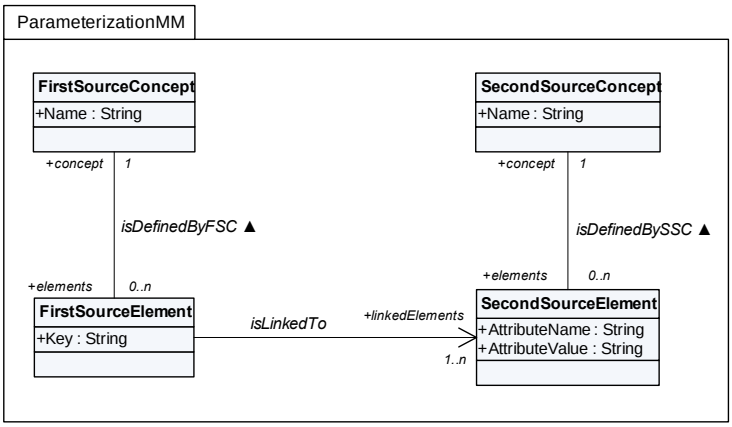


Figure 6-9 : Méta-modèle des modèles de paramétrage

La Figure 6-9 présente le méta-modèle `ParameterizationMM`, qui définit des concepts pour l'expression des relations entre éléments de deux modèles. Chaque élément du premier modèle source (`FirstSourceElement`) peut être relié à un ou plusieurs éléments du second modèle source (`SecondSourceElement`). Chaque élément est une instance d'un concept défini dans un méta-modèle. Ces concepts sont représentés par `FirstSourceConcept` et `SecondSourceConcept`. Chaque élément du premier modèle source est identifié par une

clef (`Key`), qui est choisie de manière à ce qu'un élément donné du premier modèle source ne soit représenté que par un seul et unique élément `FirstSourceElement` dans le modèle de paramétrage. Les éléments du second modèle source sont identifiés par un couple attribut - valeur (`AttributeName` et `AttributeValue`). La Figure 6-14 illustre un modèle de paramétrage conforme à ce méta-modèle.

La création de liens entre des éléments de deux modèles est complexe, elle repose essentiellement sur des décisions de conception. Par exemple, la structure des informations stockées dans le profil utilisateur est totalement indépendante du méta-modèle d'annotations pour la personnalisation : certains éléments `ItemValue` du profil utilisateur sont utilisés pour la spécification d'attributs (donc étroitement liés au concept `M2AttributePersonalizer`) tandis que d'autres éléments `ItemValue` sont utilisés pour l'établissement de liaisons entre porte-monnaie et comptes de fidélité (donc étroitement liés au concept `M2AssociationPersonalizer`).

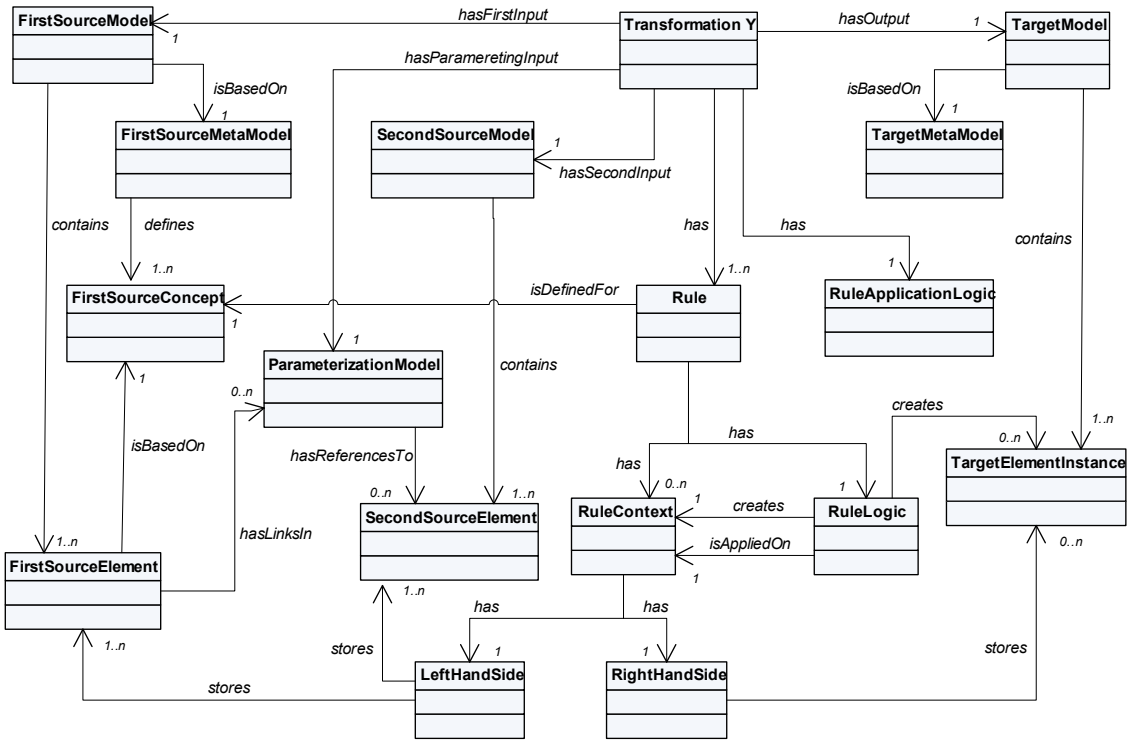


Figure 6-10 : Structure de la transformation 1-1 paramétrée

La Figure 6-10 présente la structure d'une transformation 1-1 paramétrée, sur les bases de du raffinement de la configuration en Y et du méta-modèle `ParameterizationMM`. Par rapport à la structure d'une transformation 1-1 simple, on note que la transformation prend comme paramètres entrants trois modèles : les deux modèles sources (`FirstSourceModel` et `SecondSourceModel`) et le modèle de paramétrage (`ParamaterizationModel`). Une règle est définie pour un concept donné du premier modèle source (`FirstSourceConcept`). Le mécanisme de paramétrage entre en œuvre lors de la création des contextes d'exécution

des règles : si l'élément du modèle source est référencé dans le modèle de paramétrage, alors son ou ses élément(s) correspondant(s) dans le second modèle source (`SecondSourceElement`) est (sont) ajoutés dans le côté gauche du contexte (`LeftHandSide`). Le reste de la transformation est similaire à celle décrite précédemment. Pour améliorer la lisibilité de la figure, certains concepts dont le rôle n'est pas essentiel à la compréhension ne sont pas représentés (`SecondSourceMetaModel`, `ParameterizationMM`, `SecondSourceConcept`, etc.).

Comme la structure présentée par la Figure 6-1 a été la base du cadre de conception Java pour l'implémentation des transformations 1-1, la structure présentée ici constitue la base d'une modification du cadre de conception pour supporter l'implémentation de transformation 1-1 paramétrées.

6.5 Implémentation du tissage de modèles en Java

6.5.1 Adaptation du cadre de conception Java

L'implémentation des transformations 1-1 paramétrées nécessite un enrichissement du cadre de conception décrit précédemment. L'objectif ici est *(i)* de fournir les méthodes pour le chargement des trois modèles entrants, *(ii)* de fournir une API permettant l'exécution de requêtes sur le modèle de paramétrage et enfin *(iii)* d'adapter le mécanisme d'exécution des règles.

6.5.1.1 Chargement des modèles entrants

La Figure 6-11 décrit la classe abstraite `YTransformation` qui spécialise la classe `Transformation` pour permettre le chargement des deux modèles sources et du modèle de paramétrage. Le mécanisme global de la transformation reste similaire à celui d'une transformation 1-1 simple.

```
package transformations;

public abstract class YTransformation extends Transformation {
  protected RefPackage _FirstSourceModel;
  protected RefPackage _SecondSourceModel;
  protected RefPackage _ParamModel;
  public YTransformation() {
      initModels();  // traitement des exceptions non détaillé
  }
  public void setFirstSourceModel(String filename) {
      // non détaillé ici : chargement du modèle depuis son format sérialisé
  }
  public void setSecondSourceModel(String filename) {
      // non détaillé ici : chargement du modèle depuis son format sérialisé
  }
  public void setParameterizationModel(String filename) {
      // non détaillé ici : chargement du modèle depuis son format sérialisé
  }
}
```

Figure 6-11 : Classe abstraite YTransformation

6.5.1.2 Requêtes sur le modèle de paramétrage

Le paramétrage de la transformation est réalisé lors de l'exécution des règles. Les opérations de requêtes effectuées sur le modèle de paramétrage étant similaires d'une règle à l'autre, il est intéressant de factoriser dans une classe l'ensemble des routines de manipulation de ce modèle. Ceci est fait dans la classe `ParameterizationModelHandler`, dont l'implémentation est partiellement décrite par la Figure 6-12.

```
package transformations;

public class ParameterizationModelHandler {
  private ParameterizationMMPackage _ParameterizationModel;
  private Collection _ParameterizedElements;
  public ParameterizationModelHandler(ParameterizationMMPackage pm) {
      _ParameterizationModel = pm;
  }
  public void getParameterizedElementsForConcept(String concept) {
      // Récupère tous les éléments d'un concept du modèle source impliqués
      // dans des relations de paramétrage
  }
  public Collection getParametersForFirstSourceElement(String k) {
      // Retourne la collection d'éléments SecondSourceElement associés à
      // l'élément FirstSourceElement dont l'attribut Key vaut 'k'
  }
  public SecondSourceElement getParameterForFirstSourceElement(String k) {
      // Utilisée dans le cas d'une relation 1-1
      Iterator it = _ParameterizedElements.iterator();
      while (it.hasNext()) {
          FirstSourceElement fse = (FirstSourceElement)it.next();
          if (fse.getKey().equals(k)) {
              return ((SecondSourceElement)fse.getLinkedElements().
                          iterator().next());
          }
      }
      return null;
  }
  public RefObject getSecondSourceElement(RefPackage ssm,
                                          SecondSourceElement sse) {
      SecondSourceConcept ssc = sse.getConcept();
      RefClass rc = ssm.refClass(ssc.getName());
      Iterator it = refc.refAllOfType().iterator();
      while (it.hasNext()) {
           RefObject ro = (RefObject)it.next();
           if (ro.refGetValue(sse.getAttributeName()).
                 equals(sse.getAttributeValue()))
              return ro;
      }
      return null;
  }

  public Collection getSecondSourceElements(RefPackage ssm,
                                            SecondSourceElement sse)
      // Retourne un ensemble de références vers les éléments du deuxième modèle
      // source (ssm) représentés par l'élément SecondSourceElement sse
  }
}
```

Figure 6-12 : Classe abstraite ParameterizationModelHandler

Cette classe, qui constitue une sorte d'interface pour le modèle de paramétrage, fournit principalement :

- *Deux méthodes permettant l'interrogation du modèle de paramétrage* : `getParametersForFirstSourceElement` et `getParameterForFirstSourceElement` retournent le ou les éléments `SecondSourceElement` liés à l'élément `FirstSourceModel` dont l'attribut `Key` est passé en paramètre de la méthode.
- *Deux méthodes permettant de récupérer des références vers les éléments du second modèle source* : `getSecondSourceElement` et `getSecondSourceElements` parcourent le second modèle source pour retourner les éléments pointés par leur paramètre `SecondSourceElement`.

6.5.1.3 Règles paramétrées

La classe abstraite `ParameterizedRule` décrite par la Figure 6-13 spécialise la classe abstraite `Rule` présentée dans la section 6.3.1. Le rôle de cette classe est principalement d'initialiser l'interface de manipulation du modèle de paramétrage par le biais des méthodes `setParameterizationModel` et `retrieveParameterElements`. Le reste du mécanisme d'application d'une règle paramétrée est similaire à celui d'une règle simple. Les requêtes sur le modèle de paramétrage sont effectuées lors de l'implémentation de la règle, notamment dans la méthode `createRuleContexts`.

```
package transformations;

public abstract class ParameterizedRule extends Rule {
  ParameterizationModelHandler _ParameterizationModelHandler;
  public ParameterizedRule() { }
  public void setParameterizationModel(ParameterizationMMPackage pmp) {
      _ParameterizationModelHandler = new ParameterizationModelHandler(pmp);
  }
  public void retrieveParameterElements(String concept) {
      _ParameterizationModelHandler.getParameterizedElementsForConcept(concept);
  }
  public ParameterizationModelHandler getParameterizationModelHandler() {
      return _ParameterizationModelHandler;
  }
  public void createRuleContexts(RefPackage rp,
                       ParameterizationMMPackage pmmp, String elt) {
      setParameterizationModel(pmmp);
      retrieveParameterElements(elt);
      createRuleContexts(rp,emt);
  }
  public Collection applyRule(RefPackage rp,
                       ParameterizationMMPackage pmmp, String elt) {
      _RuleContexts = new Vector();
      createRuleContexts(rp,pmmp,elt);
      return callRules();
  }
}
```

Figure 6-13 : Classe abstraite ParameterizedRule

6.5.2 Exemple de mise en œuvre

La mise en œuvre du cadre de conception est illustrée ici dans le contexte de la personnalisation de l'application *LoyaltyManager*, avec l'implémentation d'une règle de la

transformation 2-1 `AAM2PersonalizedActivator` permettant de produire le modèle de l'activateur personnalisé d'un utilisateur spécifique.

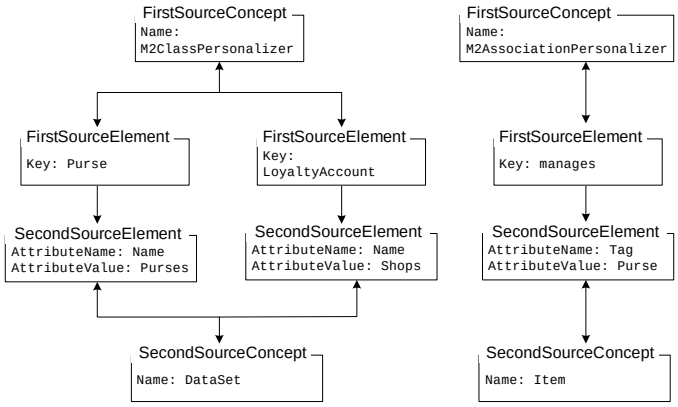


Figure 6-14 : Modèle de paramétrage pour l'application LoyaltyManager

La Figure 6-14 illustre le modèle de paramétrage définit pour l'application *LoyaltyManager*. Ce modèle, conforme au méta-modèle `ParameterizationMM`, formalise les corrélations entre le modèle de l'application annoté pour la personnalisation et le profil utilisateur. Ce modèle est utilisé pour la mise en œuvre de la transformation 2-1 dont la règle qui produit les éléments `Instance` dans le modèle cible est décrite ci-dessous.

```
class InstanceCreationRule extends ParameterizedRule   {
  protected RuleContext createRuleContextForElement(RefObject ro) {
      M2ClassPersonalizer cp = (M1ClassPersonalizer) ro;
      RuleContext rc = new RuleContext();
      // Côté gauche du contexte
      rc.addLHSElement(cp);
      // Parcours du modèle de paramétrage pour récupérer
      //une référence vers un éventuel élément lié
      SecondSourceElement sse = getParameterizationModelHandler().
          getParameterForFirstSourceElement(cp.getName());
      if (sse != null) {
          // on récupère ici l'élément du second modèle source
          DataSet linkeddataset = (DataSet)(getParameterizationModelHandler().
              getSecondSourceElement(getSecondSourceModel(),sse));
          // Chaque élément Item du DataSet récupéré correspond à une instance
          // de porte-monnaie
          // Ajout de la collection des éléments Item au côté gauche du contexte
          Collection items = null;
          items = linkeddataset.getItems();
          rc.addLHSElement(items);
      }
      return rc;
  }
  protected RuleContext applyRuleLogic(RuleContext rc) {
      M2ClassPersonalizer cp = (M2ClassPersonalizer)rc.getLeftHandSide().get(0);
      Collection parameters = (Collection) rc.getLeftHandSide().get(1);
      InstantiationHandler ik = cp.getInstanciationHandler();
      if (ik.getKind() == M1InstanciationKindEnum.EXPLICIT) {
          Vector instances = new Vector();
          Iterator it = parameters.iterator();
          while (it.hasNext()) {
          // Pour chaque élément Item, une instance est créé
              Item ite = (Item)it.next();
```

```
                Instance i = getTargetModel().getInstance().createInstance();
                i.setBaseClass(cp.getName());
                i.setInstantiationKind(InstanciationKind.EXPLICIT);
                inst.setExplicitKey(item.getName());
                instances.addElement(inst);
            }
            // Création d'un contexte pour l'application d'une règle imbriquée
            rc.addRHSElement(instances);
        }
        //  Autres cas qu'explicites non détaillés ici
        return rc;
    }
}
```

Figure 6-15 : Code d'une règle paramétrée

La Figure 6-15 décrit la règle paramétrée `InstanceCreationRule`, implémentée comme une classe interne de la classe `AAM2PersonalizedActivator`. Cette règle est définie pour le concept `M2ClassPersonalizer` du méta-modèle de la première source (`PersoAnnotationsMM`). Comme dans le cas d'une règle simple, la classe `InstanceCreationRule` est composée d'une méthode pour la préparation du contexte d'application et d'une autre pour implémenter la logique de la règle. Le paramétrage de la règle est effectué dans la première de ces deux méthodes, c'est-à-dire dans `createRuleContextForElement` :

- La règle prend comme paramètre entrant '`ro`', une référence sur un élément `M2ClassPersonalizer`. Cet élément est ajouté au côté gauche du contexte.
- Par le biais de l'appel à la méthode `getParameterForFirstSourceElement` de l'interface du modèle de paramétrage, on récupère l'élément `SecondSourceElement` 'sse' associé à l'élément `FirstSourceElement` représentant '`ro`'. Selon le modèle de paramétrage décrit par la Figure 6-14, cet élément associé est une instance du concept `DataSet`.
- Par le biais de la méthode `getSecondSourceElement`, on récupère dans le second modèle source une référence sur l'élément `DataSet` pointé par '`sse`'.
- Dans le second modèle source, l'élément `DataSet` contient plusieurs éléments `Item` décrivant chacun une instance à créer. Ces derniers sont ajoutés sous la forme d'une collection au côté gauche du contexte.

La logique de la règle, implémentée dans la méthode `applyRuleLogic`, consiste à créer des éléments `Instance` dans le modèle cible. Dans la mesure où elles ne constituent pas une aide significative à la compréhension de la mise en œuvre présentée ici, nous ne décrivons ni les autres règles de la transformation ni l'implémentation de la classe `AAM2PersonalizedActivator`. Pour un profil utilisateur donné, l'exécution globale de la transformation `AAM2PersonalizedActivator` produit un modèle d'activateur personnalisé comme celui décrit par la Figure 5-16.

6.6 Synthèse

Dans le Chapitre 5, nous avons décrit un processus génératif de personnalisation d'applications embarquées. Nous avons d'abord détaillé de manière précise les abstractions des espaces problème et solution avant de spécifier de manière informelle les transformations à mettre en œuvre pour réaliser la projection des premières sur les secondes. L'objectif de ce Chapitre 6 était de présenter l'implémentation de ces transformations de modèles.

En raison de l'intérêt grandissant porté au développement logiciel génératif et aux techniques de modélisation, les travaux de recherche sur les transformations de modèles sont nombreux. Ceci a pour conséquence une multiplication des approches – présentées dans la section 4.3 – et le développement de nombreux outils. Certains de ces outils sont propriétaires, tandis que d'autres sont encore en cours de développement ou l'étaient au moment de l'expérimentation du processus génératif de personnalisation. Ne souhaitant pas lier notre approche à un outil donné, nous avons choisi d'implémenter nos transformations en Java et mettre en œuvre un cadre de conception pour factoriser les mécanismes de base des transformations.

En nous basant sur l'analyse réalisée dans la section 4.3.2, nous avons détaillé notre vision de la structure d'une transformation 1-1 : une transformation est composée d'un ensemble de règles de projections définies par rapport à des concepts du méta-modèle source, et appliquées de manière ordonnée. Un contexte d'application contenant divers paramètres nécessaires à l'exécution de la règle est construit pour chaque instance du concept à transformer. Appliquer une règle consiste à exécuter la logique de la règle sur chacun des contextes. Cette vision a guidé la création d'un cadre structurant sous la forme d'un ensemble de classes abstraites Java implémentant les mécanismes de base des transformations : chargement des modèles, appels des règles, parcours des contextes d'application des règles. La logique même de chaque règle reste à la charge du concepteur qui doit l'implémenter lors de la spécialisation du cadre structurant. L'utilisation de ce dernier a été illustrée par l'implémentation de la transformation permettant de produire les modèles des fabriques de personnalisation, spécifiée dans la section 5.4.1.

Dans le processus génératif de personnalisation, la production des modèles d'activateurs personnalisés repose sur la mise en œuvre d'une transformation 2-1, c'est-à-dire d'un tissage de modèles. Ce schéma en Y, bien que récurrent dans l'ingénierie logicielle, n'est pas aujourd'hui aussi bien compris et maîtrisé que les transformations 1-1. Postulant que le cœur d'un tissage de modèles réside dans les corrélations entre les deux modèles sources, nous avons présenté une solution consistant à raffiner la transformation 2-1 en une transformation 1-1 paramétrée. Les relations entre les éléments des deux modèles sources sont spécifiées par le biais d'un modèle de paramétrage dont la complexité croît en fonction de la taille de ces derniers. Nous avons décrit une adaptation du cadre structurant pour supporter ce type de transformation et illustré son utilisation avec l'implémentation de la transformation permettant de produire des modèles d'activateurs personnalisés de l'application *LoyaltyManager*.

L'approche par manipulation directe pour la mise en œuvre des transformations offre une souplesse appréciable dans un contexte d'expérimentation tel que le nôtre. Si le cadre structurant présenté dans ce chapitre libère le concepteur du codage des routines de base de la transformation, la charge de travail liée à l'implémentation de la logique des règles reste très importante. La prochaine disponibilité du langage MOF 2.0 QVT permettra de considérer et d'exprimer les transformations comme des modèles, ce que notre approche ne fait pas. L'apparition des premiers outils implémentant cette spécification est donc susceptible de constituer une alternative intéressante dans la perspective d'un déploiement effectif de l'approche générative pour la personnalisation.

Quatrième partie

Conclusion

Chapitre 7

Conclusion et perspectives

Le développement important et continu des systèmes informatisés conduit à une multiplication des services et des façons de les délivrer. En raison d'une très forte compétition au sein de chaque corps de fournisseurs – commerçants en ligne, portails Internet, éditeurs de logiciels, opérateurs de téléphonie mobile, etc. –, la personnalisation prend aujourd'hui une importance considérable en tant que facteur de différentiation. La personnalisation a pour objectif d'adapter l'offre de service à la spécificité de chaque utilisateur.

Grâce à ses propriétés de sécurité et de mobilité, la carte à puce est depuis des années une représentante des individus au sein de certaines infrastructures informatiques comme les systèmes bancaires ou la téléphonie mobile. Les prochaines évolutions matérielles et logicielles des cartes vont leur faire jouer un rôle privilégié dans le déploiement de services personnalisés, en tant que support des informations personnelles mais également en tant que support d'exécution des services personnalisés. Le principal objectif de nos travaux de recherche était de proposer et expérimenter des solutions pour mettre en œuvre la personnalisation des applications embarquées.

Ce mémoire de thèse a présenté les travaux réalisés dans cette perspective. Sur la base d'une solution dictée par des considérations méthodologiques et techniques liées respectivement au respect de la séparation des préoccupations et au contexte industriel de

fabrication des cartes, nous avons détaillé la mise en œuvre d'une approche générative dirigée par les modèles. Cette mise en œuvre a notamment nécessité la réalisation d'un état de l'art sur les techniques disponibles pour l'automatisation de la construction de logiciels.

Les contributions résultant de ce travail de recherche sont relatives au domaine de la personnalisation et des cartes à puces ainsi qu'à celui du développement logiciel dirigé par les modèles. Ce chapitre de conclusion discute ces différentes contributions et introduit de nouvelles perspectives de recherche.

La conclusion est organisée en deux parties :

- La section 7.1 présente une synthèse des travaux réalisés. Elle résume et discute les principales contributions.
- La section 7.2 est une évaluation de l'approche de personnalisation proposée. Les principales contributions sont discutées et leurs limites établies.
- La section 7.3 présente un ensemble de perspectives ouvertes par ce travail de recherche, dont celle d'une ligne de produits logiciels embarqués.

7.1 Résumé des travaux

Une analogie entre l'industrie du logiciel et celle du textile rapproche la notion de personnalisation logicielle à celle de la confection sur mesure, dans laquelle les vêtements sont taillés selon les mensurations de leur porteur final. Développer un logiciel sur mesure implique que le processus de conception du logiciel intègre des mécanismes de prise en compte des spécificités et des attentes de l'utilisateur final (section 2.1). Dans le contexte des applications embarquées pour cartes à puce, la personnalisation consiste à automatiser sa configuration : pour un langage de programmation orienté objets, il s'agit de contrôler les créations et configurations d'objets ainsi que l'établissement de liaisons entre eux.

Les multiples niveaux de configuration d'une carte à puce rendent son processus de fabrication complexe (section 1.3). L'étape de personnalisation est l'étape qui consiste à charger sur la carte des informations – données ou applications – spécifiques au porteur final de la carte. Au contraire des informations génériques qui sont répliquées par le biais de protocoles de copie rapide sur un lot de carte, le chargement des données personnelles se fait par le biais de protocoles de communications relativement lents. Il est donc nécessaire d'optimiser l'étape de personnalisation en minimisant la quantité d'informations personnelles à charger. Dans cette perspective, et pour respecter le principe de séparation des préoccupations, nous distinguons trois types de code (section 2.5) :

- Le *code fonctionnel original* définit la logique de l'application et implémente ses fonctionnalités sans prendre en compte les préoccupations de personnalisation.
- Le *code personnalisable* est le résultat de l'ajout dans le code fonctionnel original de fabriques de personnalisation, c'est-à-dire d'artefacts permettant d'intervenir sur les créations d'instances et l'établissement de liens entre elles. Ce code étant commun à un lot de cartes, il peut être chargé par le biais de protocoles de copies rapides.

- Le *code personnalisé* correspond à un activateur personnalisé utilisant les interfaces fournies par le code personnalisable. L'activateur est appelé lors de l'amorce de l'application. Son code étant spécifique à chaque utilisateur et donc à chaque carte, il doit être chargé séparément sur chaque carte.

Nous avons illustré la mise en œuvre de ces concepts dans le cas d'un exemple d'application embarquée Java typique, le gestionnaire de porte-monnaie et de comptes de fidélité *LoyaltyManager* (section 2.5.2). Notre objectif étant de fournir aux développeurs d'applications pour cartes à puce les moyens de réaliser une personnalisation à grande échelle de leurs produits, il était nécessaire d'automatiser au maximum le processus de personnalisation, et donc la production du code personnalisable et du code personnalisé.

Dans cette perspective, nous avons étudié et évalué la pertinence par rapport à notre problématique de différentes options technologiques destinées à automatiser la construction de logiciel (Chapitre 3). Si les lignes de produits et la programmation orientée aspects ne répondaient pas directement à notre besoin, les concepts sur lesquels elles sont basées – notions de produits configurables et de respect de la séparation des préoccupations – ont inspiré nos choix d'implémentation. En revanche, nous avons trouvé dans la programmation générative (GP et GSD) et dans le développement logiciel dirigé par les modèles (MDD) des briques technologiques concrètes, sur lesquelles nous avons basé une approche générative pour la personnalisation des applications embarquées.

Les techniques de modélisation et méta-modélisation inhérentes au MDD nous ont permis d'élever le niveau d'abstraction pour généraliser notre solution de personnalisation. Ainsi, nous avons formalisé les concepts de personnalisation au travers d'un ensemble de méta-modèles conformes au MOF (sections 5.2 et 5.3). Les principes à la base du GSD ont fourni un cadre structurant pour la mise en œuvre des mécanismes de transformations de modèles et de génération de code permettant de produire automatiquement, à partir d'une description de l'application et de profils utilisateurs, le code personnalisable et le code personnalisé (sections 5.3 et 5.4).

L'élévation du niveau d'abstraction global nous a conduit à nous intéresser à l'aspect méthodologique du MDD, et notamment à la structuration des méta-modèles utilisés pour la représentation d'un système (section 4.4.1). Afin de respecter le principe de séparation des préoccupations, nous nous sommes appuyés sur le concept des plans d'annotations : un vocabulaire de base, propre à un domaine, est enrichi par des vocabulaires spécifiques associés à des plans d'annotations. Nous avons donc défini un plan d'annotation dédié à la personnalisation.

L'implémentation du processus génératif de personnalisation a nécessité la définition de plusieurs transformations de modèles. Sur la base d'une taxonomie des transformations et d'une étude des différentes approches et outils disponibles (sections 4.3 et 6.1), nous avons adopté une approche par manipulation directe et construit un cadre structurant pour l'implémentation en Java (section 6.3). La prise en charge des profils utilisateurs dans le processus génératif posant une problématique de tissage de modèles, nous avons finalement proposé une solution pour ce type de transformation et adapté le cadre structurant Java (sections 6.4 et 6.5).

7.2 Évaluation de l'approche

Évaluer une démarche ou une méthodologie d'ingénierie logicielle est un exercice difficile, dans le sens où il n'est pas toujours possible d'en mesurer quantitativement les gains ou les coûts. Une possibilité d'évaluation est de comparer les avantages et inconvénients avec ceux d'autres démarches permettant d'obtenir les mêmes résultats similaires.

Dans le cas de la personnalisation des applications embarquées pour les cartes à puce, l'étude approfondie du domaine n'a pas permis d'identifier d'approche similaire ayant donné lieu à des publications. La personnalisation est en fait principalement traitée aujourd'hui dans le cadre de travaux sur les systèmes et algorithmes de recommandation (section 1.1). Dans une certaine mesure, les travaux portant sur la configuration de logiciels, notamment dans le cadre des lignes de produits, peuvent également être considérés comme connexes, mais pas suffisamment pour constituer une alternative (section 3.1.3.2).

La démarche que nous proposons se caractérise principalement par la richesse accrue de la personnalisation qu'elle permet par rapport aux procédés en place aujourd'hui. Le manque d'éléments de comparaison ou d'étalonnage nous conduit à évaluer notre approche d'un point de vue essentiellement qualitatif, en analysant concrètement ce qu'elle permet de réaliser et dans quelles limites. Nos travaux constituent un ensemble de propositions et contributions relatives à la fois aux domaines des cartes à puce, de la personnalisation et des techniques de modélisation :

- Nouvelles perspectives pour le déploiement de services personnalisés
- Introduction des concepts de code personnalisable et de code personnalisé
- Définition d'un processus génératif de personnalisation
- Proposition pour le tissage de modèles

Dans la suite de cette section d'évaluation, nous discutons chacune de ces contributions.

7.2.1 Nouvelles perspectives pour le déploiement de services personnalisés

Alors que les offres des fournisseurs de service se multiplient et se diversifient, la personnalisation apparaît comme un aspect essentiel du paysage numérique émergent. Le premier objectif de nos travaux de recherche était d'étudier le rôle potentiel de la carte à puce dans le déploiement des solutions de personnalisation. Nous avons donc recensé les différentes solutions existantes ou possibles.

Aujourd'hui, la carte est souvent cantonnée à être le support d'une personnalisation basique, consistant au simple stockage de données personnelles : les cartes SIM actuelles, par exemple, contiennent diverses données privées ainsi qu'une sélection d'applications *SIM Toolkit*. Des solutions plus expérimentales telles qu'un système de recommandation de films dans la télévision interactive [25], ou que Césure [26] pour le déploiement

personnalisé d'applications à base de composants, ont donné à la carte un rôle plus important. Dans ces approches, la carte n'est plus seulement le support du profil utilisateur, mais également celui de l'ensemble ou d'une partie du mécanisme de personnalisation lui-même. Les services personnalisés sont délivrés hors carte.

Nos travaux de recherche s'inscrivent dans cette démarche visant à exploiter au mieux le potentiel de la carte à puce. La méthodologie que nous proposons permet de profiter des évolutions matérielles et logicielles des cartes à puce pour réaliser une personnalisation fonctionnelle plus riche des applications embarquées elles-mêmes. Dans le contexte de nomadisme d'un paysage numérique en pleine évolution, cette approche ouvre de nouvelles perspectives pour le déploiement de services mobiles personnalisés.

7.2.2 Introduction des concepts de code personnalisable et personnalisé

La personnalisation est une forme particulière d'adaptation ou de configuration. Une approche simple pour contrôler l'instanciation des applications embarquées aurait pu consister à *(i)* considérer les profils utilisateurs comme de simples fichiers de configuration et *(ii)* fournir une interface de programmation pour les manipuler. La personnalisation se ferait alors au moment de l'exécution de l'application. Cette approche souffre de plusieurs inconvénients majeurs : elle impose un format précis de structuration des données utilisateurs, elle ne guide pas vraiment le concepteur dans l'implémentation des mécanismes de personnalisation, elle est spécifique à un langage de programmation donné et surtout, elle ne respecte pas le principe de séparation des préoccupations.

Réaliser la personnalisation au niveau du code permet d'aborder la problématique de la personnalisation de manière méthodologique. En introduisant les concepts de code personnalisable et de code personnalisé, nous avons atteint deux de nos objectifs initiaux :

- *Le respect de la séparation des préoccupations*. Le concept de code personnalisable, ou plus précisément celui des fabriques de personnalisation ajoutées ultérieurement au code fonctionnel, permet de minimiser l'impact des mécanismes de personnalisation sur le code original. Celui-ci est réduit à quelques conventions de dénominations de champs et méthodes.
- *L'optimisation de l'industrialisation de la personnalisation*. En réduisant aux activateurs des applications embarquées les parties spécifiques aux utilisateurs, nous avons minimisé la quantité de données devant être chargée dans la carte par le biais de protocoles de communication lents. Le code personnalisable peut, quant à lui, être répliqué sur toutes les cartes d'un lot par le biais de protocoles de copie rapide.

L'approche que nous proposons combine une structuration rigoureuse et optimisée des mécanismes de personnalisation tout en laissant aux fournisseurs une grande liberté dans la définition de leurs profils utilisateurs. Le processus génératif que nous avons construit sur la base de ces concepts permet d'automatiser une grande partie de la personnalisation.

7.2.3 Définition d'un processus génératif de personnalisation

7.2.3.1 Proposition

L'objectif initial de nos travaux de recherche était de proposer une solution donnant aux fournisseurs de services les moyens méthodologiques et techniques de réaliser une personnalisation efficace à grande échelle de leurs applications embarquées. La définition d'un processus génératif, applicable aussi bien sur site industriel lors de la fabrication qu'en *post-issuance* après le déploiement des cartes, constitue par conséquent notre contribution majeure.

Mettre en œuvre la personnalisation d'une application embarquée par le biais de notre approche consiste *(i)* à spécifier sur le modèle de l'application par le biais du langage d'annotation fourni ce qui doit être personnalisé, *(ii)* à stocker dans des profils les données utilisateurs nécessaires à la personnalisation et *(iii)* à implémenter le tissage entre le modèle annoté et les profils utilisateurs. Cette dernière phase étant la plus complexe, nous avons fourni une méthodologie et un cadre structurant pour son implémentation en Java. Le résultat de ce tissage est un modèle utilisé pour la génération du code personnalisé. L'obtention du code personnalisable à partir du modèle de l'application annotée est automatique.

7.2.3.2 Limites

Si les travaux d'expérimentation réalisés valident les concepts que nous avons introduits et confirment la pertinence globale de l'approche, proposer réellement cette dernière aux fournisseurs nécessite une phase de consolidation :

- *Les différents méta-modèles introduits ne sont pas figés*. Selon les besoins spécifiques des fournisseurs, un enrichissement des méta-modèles peut s'avérer nécessaire.
- *La disponibilité de nouveaux standards et outils de transformation*, dont celle de QVT et de ses implémentations, devrait constituer une alternative à la mise en œuvre de notre approche.
- *Le développement d'outils supportant l'approche est nécessaire.* Un outil graphique pour l'annotation du modèle original rendrait cette étape plus pratique. Un tel outil pourrait, par exemple, prendre la forme d'une extension à un environnement de modélisation existant.

7.2.4 Une proposition pour le tissage de modèles

7.2.4.1 Proposition

Bien qu'étant un schéma récurrent de l'ingénierie logicielle, la configuration en Y n'a pas encore suscité dans le contexte du MDD autant de travaux de recherche que les transformations 1-1. Pourtant, ce schéma est par exemple au cœur de la définition de la MDA, dont l'objectif est de tisser les modèles représentant la logique d'un système (PIMs)

avec ceux décrivant les plates-formes cibles (PDMs) pour produire des modèles spécifiques à ces dernières (PSMs).

Lors de la conception du processus génératif de personnalisation, nous avons été confrontés à la problématique du tissage de modèle pour la production des modèles d'activateurs personnalisés : cette dernière dépend à la fois du modèle de l'application annoté pour la personnalisation et des profils d'utilisateurs. Arguant que le cœur d'un tissage de modèle réside dans les corrélations entre les éléments et concepts des deux modèles sources, nous avons proposé un raffinement de la configuration en Y pour en faire une problématique de transformation 1-1 paramétrée. Un seul des deux modèles sources est utilisé comme base du parcours pour l'application des règles. Un modèle exprimant les liaisons entre les éléments des deux modèles sources, est utilisé pour paramétrer les règles.

7.2.4.2 Limites

Une des limites principales de la solution proposée réside dans la production du modèle de paramétrage. L'établissement de ces relations reposant sur des choix humains de conception, la création d'outils génériques pour supporter la création de ce modèle est difficile. Le niveau de paramétrage que nous avons choisi est également discutable : établir des corrélations au niveau des concepts plutôt que des éléments eux-mêmes peut s'avérer suffisant dans certains cas. Il est prévisible que la mise en pratique concrète des approches dirigées par les modèles établisse à court terme la problématique du tissage de modèles comme un des thèmes de recherche essentiels liés au MDD.

7.3 Perspectives

La discussion des différentes contributions dans la section précédente a permis d'identifier quelques perspectives à nos travaux. Alors que la poursuite des recherches sur le tissage de modèles semble être une priorité, les autres pistes évoquées constituent davantage des travaux d'ingénierie logicielle du ressort de départements de recherche et développement industriels que des problématiques de recherche fondamentale.

Certains concepts que nous avons introduits ou expérimentés ont d'ores et déjà été réutilisés : par exemple, la définition et le traitement automatisé de vocabulaires d'annotations au niveau des modèles ont inspiré l'architecture de l'outil de configuration d'une machine virtuelle Java de prochaine génération. Cette application partielle de nos résultats confirme le bien fondé de l'utilisation des approches génératives dans le contexte de la carte à puce.

Une carte à puce offre typiquement un environnement logiciel composé d'un système d'exploitation, de couches logicielles intermédiaires et d'applications. Chacun de ces niveaux requiert une certaine forme de configuration. La personnalisation des applications n'est finalement que l'étape ultime de cette chaîne de configuration. Une ligne de produits logiciels pour cartes à puce permettrait de gérer les différents niveaux de configuration d'une façon homogène. Sur des bases communes propres à chaque domaine – téléphonie mobile, bancaire –, différentes configurations pourraient être déclinées selon les requêtes

des clients. Une telle structuration de l'offre logicielle pour cartes à puce présenterait plusieurs avantages, notamment l'optimisation du temps de mise sur le marché des produits et la possibilité de développer des outils de simulation utilisables pour l'estimation des coûts et délais avec les clients.

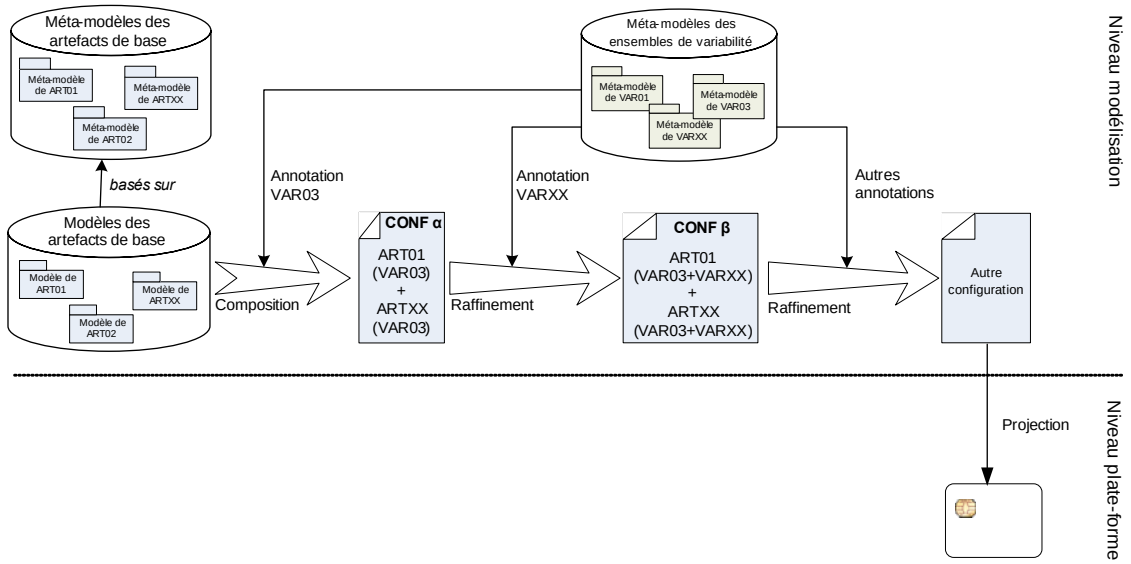


Figure 7-1: Architecture d'une ligne de produits logiciels pour cartes à puce

Notre travail de recherche sur la personnalisation ouvre la voie à la définition d'une architecture possible pour cette ligne de produits. Les mises en œuvre d'une approche de modélisation basée sur des mécanismes d'annotations et d'un traitement automatisé de ces dernières sont à l'origine de l'ébauche d'architecture présentée par la Figure 7-1 :

- *Les artefacts de base sont décrits par le biais de modèles* basés sur des méta-modèles correspondant aux domaines d'applications des cartes.
- *Les points de variabilité de la ligne de produits sont regroupés par préoccupations*, qui peuvent être fonctionnelles ou techniques. La personnalisation, par exemple, constitue un ensemble de points de variabilité.
- *Chaque ensemble de points de variabilité est associé à un ou plusieurs méta-modèles* définissant les concepts pour l'expression de la préoccupation qu'il représente.
- *Un mécanisme à base de plans d'annotations* permet le marquage des modèles des artefacts de base.

La dérivation de produits logiciels sur la base de cette architecture est réalisée par le biais d'une succession d'annotations et de transformations de modèles. Chaque modèle représente une configuration intermédiaire ou finale d'un produit donné. Les configurations finales peuvent être projetées sur les cartes cibles.

La mise en œuvre du processus génératif de personnalisation a illustré la manière dont nous avons utilisé le GSD comme support méthodologique du MDD. La mise en œuvre de

cette ligne de produits logiciels serait une autre illustration de la complémentarité entre ces deux approches. La convergence du développement logiciel génératif et du développement dirigé par les modèles semble constituer une prochaine étape de l'ingénierie logicielle.

Acronymes

AOP	Aspect-Oriented Programming
APDU	Application Protocol Data Unit
API	Application Program Interface
CLI	Common Language Infrastructure
CMOF	Complete MOF
CORBA	Common Request Broker and Architecture
CWM	Common Warehouse Metamodel
DES	Data Encryption Standard
DSL	Domain Specific Language
DTD	Document Type Definition
EEPROM	Electrical Erasable Programmable Read Only Memory
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
ETSI	European Telecommunications Standards Institute
GMT	Generative Model Transformer
GP	Generative Programming
GSD	Generative Software Development
GSM	Global System for Mobile Communications
IDL	Interface Definition Language
ISO	International Standard Organization
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JMI	Java Metadata Interface

JVM	Java Virtual Machine
LHS	Left Hand Side
MDA	Model Driven Architecture
MDD	Model-Driven Development
MDSD	Model Driven Software Development
MOF	Meta Object Facility
OMG	Object Management Group
OTA	Over The Air
PDA	Personal Digital Assistant
PIM	Platform Independent Model
PIN	Personal Identification Number
PSM	Platform Specific Model
QVT	MOF 2.0 Query / Views / Transformations
RAM	Random Access Memory
RF	Radio Frequency
RFP	Request For Proposals
RHS	Right Hand Side
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
ROM	Read Only Memory
RPC	Remote Procedure Call
RSA	Rivest Shamir Adleman
SIM	Subscriber Identity Module
SMS	Short Message Service
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition
XSLT	XSL Transformation

Annexe A :
Publications associées à cette thèse

Conférences internationales avec comité de sélection et actes.

A Model-Driven Approach for Smart Card Configuration
S.Bonnet, O. Potonniée, R. Marvie, J.-M. Geib, dans *actes de la troisièmeconférence internationale sur Generative Programming and Component Engineering (GPCE'04)*, Lecture Notes in Computer Science, Volume 3286, pages 416-435, Springer-Verlag, Octobre 2004, Vancouver, Canada.

Towards a Model-Based Software Product Line for Smart Cards
S. Bonnet, O. Potonniée, dans *actes de la première conférence internationale sur Economic, Technical, and Organisational aspects of Product Configuration Systems*, Technical University of Denmark, Juin 2004, Copenhague, Danemark.

Model-Driven Software Personalization
S. Bonnet, dans *actes du second symposium international sur Communicating Objects (sOc'2003)*, pages 114-117, Mai 2003, Grenoble, France.

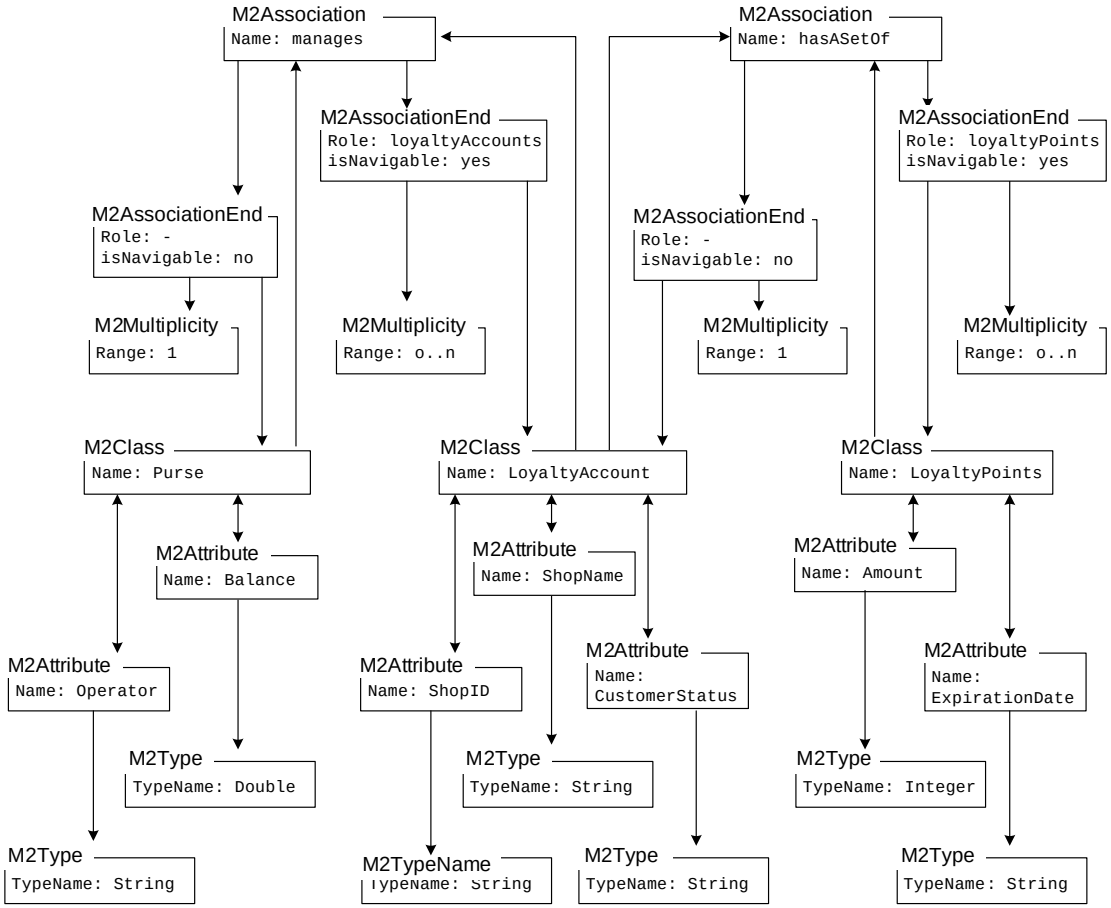
Atelier de travail international avec comité de sélection et actes.

Putting Concern-Oriented Modeling into Practice
S. Bonnet, R. Marvie, J.-M. Geib, dans *actes du second Nordic Workshop on the Unified Modeling Language (NWUML'2004)*, TUCS General Publication, n°35, pages 99-113, Août 2004, Turku, Finlande.

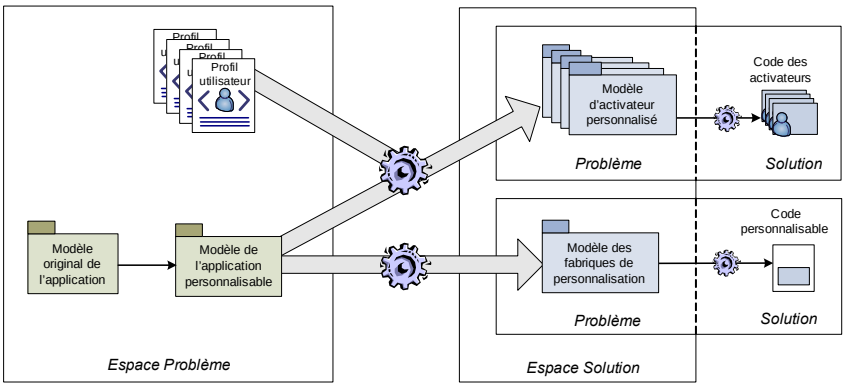
Annexe B : Cas d'étude

L'objectif de cette annexe est de rassembler les modèles utilisés pour l'illustration de la mise en œuvre du processus génératif de personnalisation des applications embarquées décrite dans le Chapitre 5. Les modèles présentés dans cette annexe sont ceux de l'application *LoyaltyManager*, introduite dans la section 2.4.

Modèle original de l'application LoyaltyManager



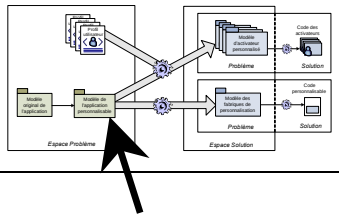
Processus génératif de personnalisation dirigé par les modèles (section 5.1) :

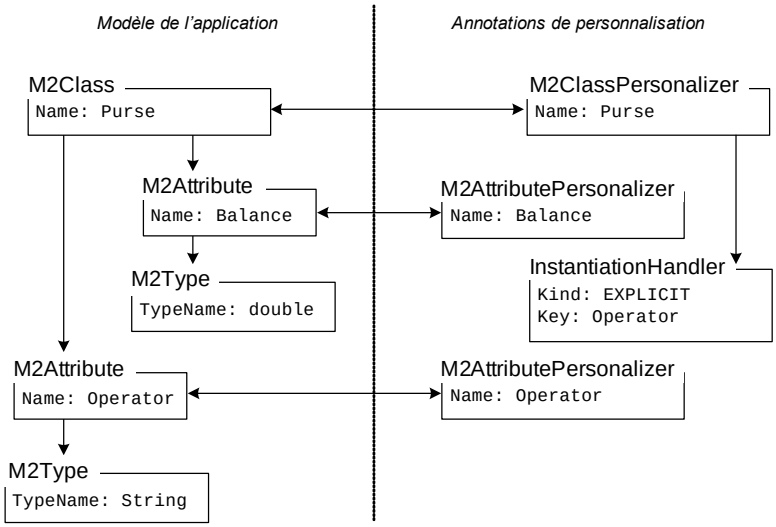


A partir du modèle original de l'application, la mise en œuvre du processus génératif consiste à :

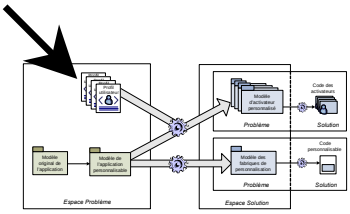
- Annoter selon les préoccupations de personnalisation
- Structurer les informations utilisateur au sein d'un profil
- Obtenir le modèle des fabriques de personnalisation
- Obtenir le code des fabriques de personnalisation
- Obtenir les modèles d'activateurs personnalisés
- Obtenir le code des activateurs personnalisés

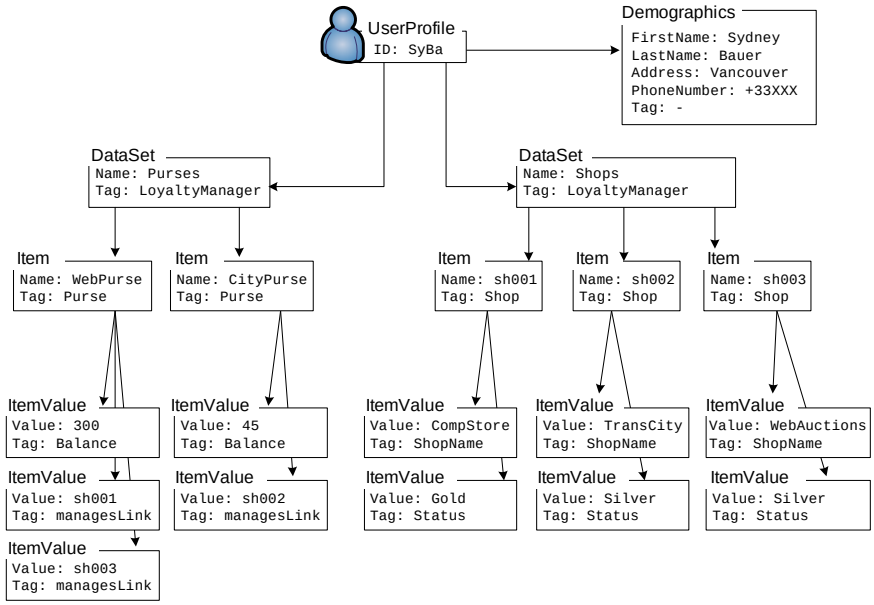
Extrait du modèle annoté (pour la M2Class 'Purse')



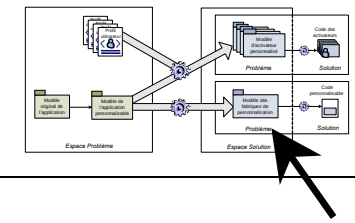


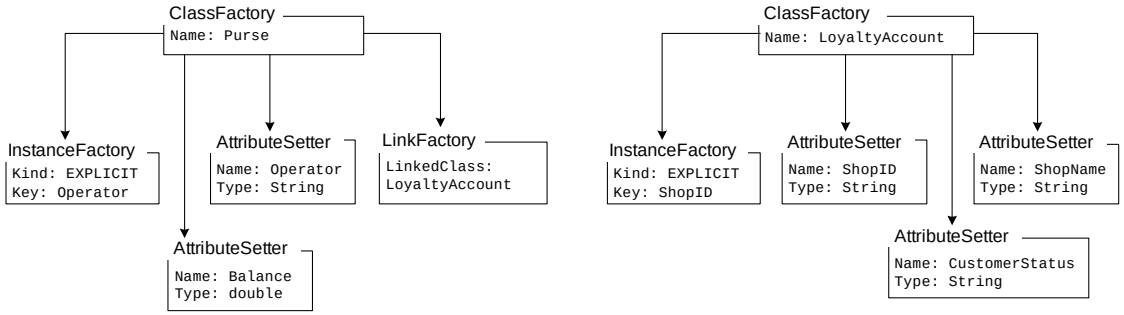
Profil utilisateur de ‘Sydney Bauer’



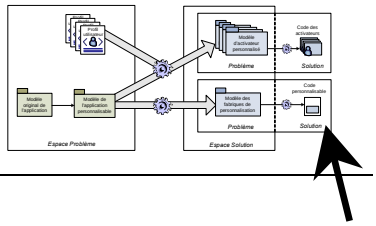


Modèle des fabriques de personnalisation



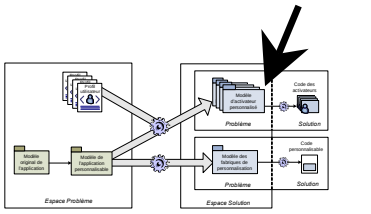


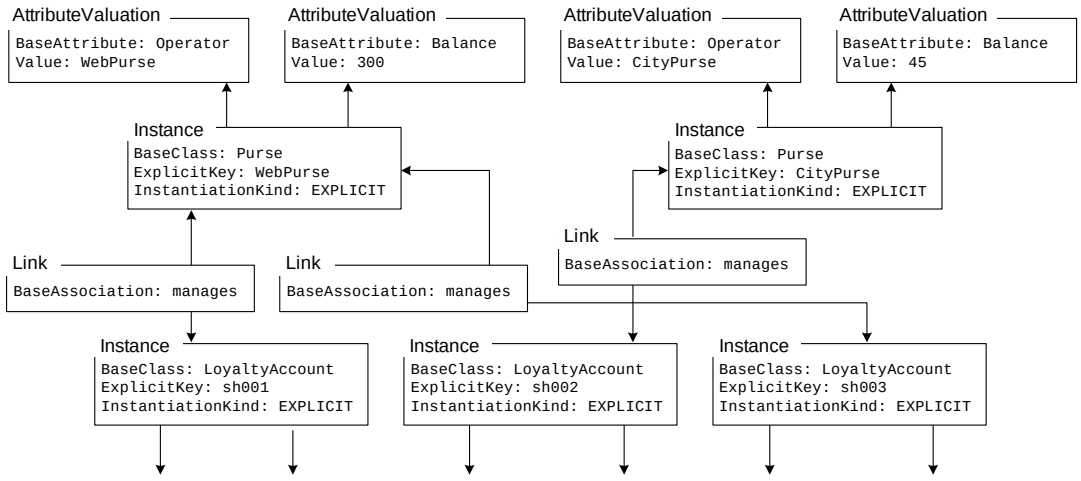
Code Java d'une fabrique de personnalisation



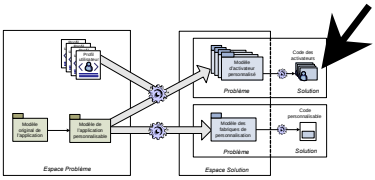
```
public static class Personalizer {
    static Hashtable instances = new Hashtable();
    public static Purse create(Object key) {
        Purse instance = new PurseImpl();
            instances.put(key, instance);
            return instance;
    }
    public static void setBalance (Purse inst, double value) {
        ((PurseImpl)inst). Balance = value;
    }
    public static void setOperator (Purse inst, String value){
        ((PurseImpl)inst).Operator = value;
    }
    public static void addLoyaltyAccount (
        Purse inst, LoyaltyAccount linkedinst) {
        ((PurseImpl)inst).LoyaltyAccounts.add(linkedinst);
    }
}
```

Extrait du modèle de l'activateur personnalisé de Sydney Bauer





Extrait du code Java de l'activateur pour Sydney Bauer



```
public static void personalize ( String[] argv ) {

  // activateur pour Sydney Bauer
  // instanciations et configurations des porte-monnaie
  Purse inst_WebPurse_Purse = PurseImpl.Personalizer.create("WebPurse");
  PurseImpl.Personalizer.setOperator(WebPurse_Purse,"WebPurse");
  PurseImpl.Personalizer.setBalance(WebPurse_Purse,300);
  Purse City_Purse = PurseImpl.Personalizer.create("CityPurse");
  PurseImpl.Personalizer.setOperator(CityPurse_Purse,"CityPurse");
  PurseImpl.Personalizer.setBalance(CityPurse_Purse,45);

  // instanciation et configuration des comptes de fidélité non détaillés ici

  // gestion des liaisons pour les instance de porte-monnaie
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_WebPurse_Purse,inst_sh001_LoyaltyAccount);
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_WebPurse_Purse, inst_sh003_LoyaltyAccount);
  PurseImpl.Personalizer.addLoyaltyAccount
      (inst_CityPurse_Purse, inst_sh002_LoyaltyAccount);
  // . . . autres liaisons non détaillées ici . . .
}
```

Le code des fabriques de personnalisation et des activateurs obtenu à la suite de ce processus génératif est utilisé pour le déploiement des applications personnalisées selon les schémas détaillés dans la section 2.5.3.

Bibliographie

1 D. Riecken, Personalized Views of Personalization, dans *Communications of the ACM*, volume 43, n°8, pages 27-28, ACM Press, Août 2000.

2 The Personalization Consortium, site Internet. Hhttp://www.personalization.orgH

3 J.S. Breese, D. Heckerman et C.Kadie, Empirical Analysis of Predictive Algorithme for Collaborative Filtering, *Microsoft Technical Report,* MSR-TR-98-12, Octobre 1998.

4 Amazon France, site Internet. Hhttp://www.amazon.frH

5 B. Smyth et P. Cotter, A Personalized Television Listings Service, dans Communications of the ACM, volume 43, n°8, pages 107-111, ACM Press, Août 2000.

6 MyYahoo!, site Internet. Hhttp://my.yahoo.com/?myHomeH.

7 U. Manber, A. Patel et J. Robison, Experience with personalization of yahoo!, dans Communications of the ACM, volume 43, n°8, pages 35-39, ACM Press, Août 2000.

8 International Standard Organization (ISO), Information Technology – Identification Cards – Integrated circuit(s) cards with contacts – Numéros de référence ISO/IEC 7816 (parts 1 to 10).

9 International Standard Organization (ISO), Information Technology – Identification Cards – Integrated circuit(s) cards with contacts – Part 4: interindustry commands for interchange, Numéro de référence ISO/IEC 7816-4, 1995.

10 P. Paradinas et J.-J. Vandewalle, A Personal and Portable Database Server: the CQL Card, dans *Application of Databases (ADB)*, Lecture Notes in Computer Science, volume 819, pages 444-457, Springer, June 1994.

11 International Standard Organization (ISO), Information Technology – Identification Cards – Integrated circuit(s) cards with contacts – Part 7: interindustry commands for Structured Card Query Language (SCQL), Numéro de référence ISO/IEC 7816-7, 1999.

12 Europay international, MasterCard International, Visa International (EMV), Integrated circuit card specification for payment systems.
Hhttp://www.emvco.comH

13 BMS Développement, Porte-monnaie électronique Moneo.
Hhttp://www.moneo.netH

14 Proton World, Porte-monnaie électronique Proton.
Hhttp://protonworl.st.comH

15 3rd Generation Partnership Project (3GPP), Specification of the Subscriber Identity Module – Mobile Equipment (SIM-ME) Interface, numéro de référence: 3GPP TS 11.11 version 8.12.0, Avril 2004.
Hhttp://www.3gpp.orgH

16 Calypso Networks Association, Calypso technology.
Hhttp://www.calypsonet-asso.orgH

17 Groupe d'Intérêt Economique (GIE) Sesam-Vitale, Carte Vitale.
Hhttp://www.sesam-vitale.frH

18 European Commission. Trans-european access to health services for mobile citizens.
Hhttp://www.netcards-project.comH

19 International Civil Aviation Organization (ICAO). Machine Readable Travel Documents (MRTD).
Hhttp://www.icao.int/mrtd/H

20 3rd Generation Partnership Project (3GPP), Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface. Numéro de référence: 3GPP TS 11.14 version 8.17.0, Septembre 2004.
Hhttp://www.3gpp.orgH

21 O. Potonniée, Advanced Card Platform: Configuration and Personalization, Rapport technique interne, Gemplus, 2004.

22 E. Volokh, Personalization and Privacy, dans Communications of the ACM, volume 43, n°8, pages 84-88, ACM Press, Août 2000.

23 L. Faith Cranor, 'I didn't buy it for myself' privacy and e-commerce personalization, dans Proceedings of the ACM workshop on Privacy in the electronic society (*WPES'03*), pages 111-117, ACM Press, Washington DC, October 2003.

24 O. Potonniée, Ubiquitous Personalization: a Smart Card Based Approach, dans 4th Gemplus Developer Conference, Singapore, Novembre 2002.

25 O. Potonniée, A Decentralized privacy-enabling TV personalization framework, dans 2nd European Conference on Interactive Television: Enhancing the Experience (euroITV 2004), Brighton, Royaume-Uni, Avril 2004.

26 M.-C. Pellegrini, O. Potonniée, R. Marvie, S. Jean et M. Riveil, CESURE : une plateforme d'applications adaptables et sécurisées pour usagers mobiles, dans Evolution des plateformes orientées objets répartis, édition spéciale de Calculateurs Parallèles, éditions Hermès, Paris, France, Septembre 2000.

27 National Institute of Standards and Technology (NIST), FIPS PUB 46-2 : Data Encryption Standard (DES), Décembre 1993.

28 R. L. Rivest, A. Shamir et L. M. Adleman, A Method for obtaining digital signatures and public-key cryptosystems, dans *Communications of the ACM*, volume 21, n°2, pages 120-126, ACM Press, 1978.

29 D. Deville, A. Galland, G. Grimaud et S. Jean, Smart Card Operating Systems: Past, Present and Future, dans fifth USENIX / NordU Conference, Västerås, Suède, Février 2003.

30 Sun Microsystems, Java Card 2.2.1 Platform specification, 2003.
Hhttp://java.sun.com/products/javacard/H

31 Maosco Consortium Ltd, Multos™: The multi-application operating system for smart cards.
Hhttp://www.multos.comH

32 J. Gosling, B. Joy, G. Stelle et G. Bracha, The Java Language Specification, *The Java Series*, Addison-Wesley, Second Edition, 2000.
Hhttp://java.sun.com/docs/books/jls/H

33 Java Card Forum, site Internet.
Hhttp://www.javacardforum.orgH

34 Hive Minded Inc. Nectar VM Platform – Smartcard.Net 1.1, 2004.
Hhttp://www.hiveminded.comH

35 Microsoft Corp. .Net Framework.
Hhttp://microsoft.com/net/H

36 Common Language Infrastructure (CLI), Standard ECMA-335, second edition, Décembre 2002.

37 Sun Microsystems, Java Technology.
Hhttp://java.sun.comH

38 T. Lindholm et F. Yellin, The Java™ Virtual Machine Specification, *The Java Series*, Addison-Wesley, Second Edition, 1999,
Hhttp://java.sun.com/docs/books/vmspec/H

39 G. Grimaud et J.-J. Vandewalle, Introducing Research Issues for Next Generation Java-based Smart Card Platforms, dans Smart Objects Conference (sOc'03), pages 138-141, Grenoble, France, Mai 2003.

40 Sun Microsystems, Java 2 Enterprise Edition (J2EE).
Hhttp://java.sun.com/j2ee/H

41 Sun Microsystems, Java 2 Micro Edition (J2ME).
Hhttp://java.sun.com/j2me/H

42 L. Lagosanto, Next-Generation Embedded Java Operating System for Smart Cards, dans 4th Gemplus Developer Conference, Singapore, Novembre 2002.

43 C. Muller et E. Deschamps, Smart Cards as First-Class Network Citizens, dans 4th Gemplus Developer Conference, Singapore, Novembre 2002.

44 G. Bussard, Next-Generation Java Card Framework, dans 4th Gemplus Developer Conference, Singapore, Novembre 2002.

45 U.Varshney et R. Vetter, Mobile commerce: framework, applications and networking support, dans Mobile Networks and Applications, volume 7, n°3, pages 185-198, Kluwer Academic Publishers, Juin 2002.

46 S. F. Mjølsnes et C. Rong, On-line e-wallet system with decentralized credential keepers, dans Mobile Networks and Applications, volume 8, n°1, pages 87-99, Kluwer Academic Publishers, Février 2003.

47 Octopus Cards Limited, Electronis Payment System using Octopus Card™. Hhttp://www.octopuscards.com/eng/H

48 NTT DoCoMo, i-Mode Felica: Mobile Wallet, site Internet. Hhttp://www.nttdocomo.com/corebiz/imode/services/felica.htmlH

49 C. Lopes et W. Hursh, Separation of Concerns, rapport technique, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, Février 1995.

50 E. W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, NJ, 1976.

51 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin, Aspect-Oriented Programming, dans Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finlande, Lecture Notes in Computer Science, volume 1241, pages 220-242, Springer, Juin 1997.

52 M. Aksit, K. Wakita, J. Bosh et L. Bergmans, Abstracting Object Interactions Using Composition Filters, volume 791, pages 152-184, 1994.

53 H.Ossher, K. Kaplan, W. Harrison, A. Matz et V. Kruskal, Subject-Oriented Composition Rules, dans Proceedings of OOPSLA'95, Sigplan Notice, volume 30, pages 235-250, ACM Press, 1995.

54 P. Tarr, H. Ossher, W. Harrison et S. Sutton, N Degrees of Separation: Multi-Dimensional Separation of Concerns, dans Proceedings of the International Conference on Software Engineering (ICSE'99), pages 107-119, Los Angeles, Etats-Unis, Mai 1999.

55 H. Ossher et P. Tarr, Multi-dimensional separation of concerns and the hyperspace approach, chapitre de Software Architectures and Component Technology: The State of the Art in Research and Practice, éditeurs L. Bergmans et M. Aksit, Kluwer academic publishers edition, 2001.

56 E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Addison-Wesley Professional, 1995.

57 F. van der Linden, Development and Evolution of Software Architectures for Product Families, dans *proceedings of the Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Espagne, Lectures Notes in Computer Science, volume 1429, Springer, Février 1998.

58 J.Bosch, Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley, 2000.

59 R. Johnson et B. Foote, Designing Reusable Classes, dans *Journal of Object Oriented Programming*, pages 22-30, Juin-Juillet 1988

60 C. Szyperski, Component Software - Beyond Object Oriented Programming, Addison-Wesley, 1997.

61 J. van Gurp, J. Bosch, M. Svahnberg, Managing Variability in Software Product Lines, Landelijk Architectuur Congres, Amsterdam, Hollande, 2000.

62 P. Clements et C. W. Krueger, How Can *You* Get Started with *Software Product Lines* ?, site Internet, Hhttp://www.softwareproductlines.comH

63 P. Clements et L. Northrop, Software Product Lines: Practices and Patterns, *SEI series in Software Engineering*, Addison-Wesley, 2002.

64 Software Engineering Institute (SEI), Framework for Product Line Practice, site Internet.
Hhttp://sei.cmu.edu/plp/H

65 P. America, H. Obbink, J. Muller et R. van Ommering, COPA: A Component-Oriented Platform Architecting Method of Families of Software Intensive Electronic Products, dans *the First Conference on Software Product Line Engineering*, Denver, Colorado, 2000.

66 D. Weiss, C. Lai et R. Tau, Software product-line engineering: a family-based software development process, Addison-Wesley, Reading, MA, 1999.

67 K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin et M. Huh, FORM: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures, dans *Annals of Software Engineering*, volume 5, pages 143-168, 1998.

68 C. Atkinson et al., Component-based product line engineering with UML, Addison-Wesley, 2002.

69 M. Matinlassi, E.Niemelä et L. Dobrica, "Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture", VTT Technical Research Centre of Finland, Espoo, Finlande, 2002.

70 M. Matinlassi, Comparison of Software Product Line Architectures Design Methods: COPA, FAST, FORM, KobrA, and QADA, dans *proceedings of the 26th International Conference on Software Engineering (ICSE'04),* Edinburgh, Écosse, pages 127-136, IEEE Computer Society, mai 2004.

71 J. van Gurp, J. Bosch et Mikael Svahnberg, On the Notion of Variability in Software Product Lines, dans *proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, pages 45-55, août 2001.

72 K. Kang, S. Cohen, J. Hess, W. Nowak et S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, rapport technique, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pa, États-Unis, Novembre 1990.

73 M. L. Griss, J. Favaro et M. d'Alessandro, Integrating feature modelling with the RSEB", dans *proceedings if the Fifth International Conference on Software Reuse (Cat.No.98TB100203),* pages 76-85, IEEE Computer Society, Los Alamitos, CA, USA, 1998.

74 Object Management Group (OMG), The Unified Modeling Language™ (UML), spécification version 1.5, Mars 2003.

75 Object Management Group (OMG), The Unified Modeling Language™ (UML), spécification version 2.0 en cours de finalisation, 2004.

76 T. Ziadi, L. Helouët et J.-M. Jézécquel, Modélisation de Lignes de Produits en UML, dans *Langages et modèles à objets (LMO 2003)*, Vannes, Février 2003.

77 T. Ziadi, L. Helouët et J.-M. Jézécquel, Product Line Derivation with UML, dans *Proceedings Software Variability Management Workshop*, University of Groningen, Février 2003.

78 T. Ziadi, L. Helouët et J.-M. Jézécquel, Towards a UML Profile for Software Product Lines, dans *5th International Workshop on Software Product Family Engineering (PFE 2003)*, Siena, Italie, 2003.

79 M. Clauß, Generic Modeling using UML Extensions for Variability, dans *Workshop on Domain Specific Visual Languages (OOPSLA 2001)*, Tampa bay, Etats-Unis, 2001

80 Object Management Group (OMG), Model Driven Architecture™ (MDA), site Internet, 2004.
Hhttp://www.omg.org/mda/H

81 S. Deelstra, M. Sinnema, J. van Gurp et J. Bosch, Model Driven Architecture as Approach to Manage Variability in Software Product Families, dans *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003)*, University of Twente, Enschede, Hollande, 2003.

82 Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik et Arnor Solberg, An MDA-based framework for model-driven product derivation, in *the eighth IASTED International Conference on Software Engineering and Applications*, pages 709-714, Cambridge, États-Unis, ACTA press, Novembre 2004.

83 I. Jacobson, M. Griss et P. Jonsson, Software Reuse: Architecture, Process, and Organization for Business Success, Addison-Wesley, 1997.

84 M. Svahnberg et J. Bosch, Issues Concerning Variability in Software Product Lines, dans *proceedings of the Third International Workshop on Software Architectures for Product Families*, Las Palmas de Gran Canaria, Espagne, Lectures Notes in Computer Science, volume 1951, pages 146-157, Springer, Mars 2000.

85 M. Anastasopoulos et C. Gacek, Implementing Product Line Variabilities, IESE-Report No. 089.00/E, Version 1.0), Kaiserslautern, Allemagne, Fraunhofer Institut Experimentelles Software Engineering, novembre 2000.

86 Aspect-Oriented Programming Project, Xerox PARC, site Internet.
Hhttp://www.parc.xerox.com/aop/H

87 Xerox PARC, AspectJ project, 2004.
Hhttp://eclipse.org/aspectj/H

88 Eclipse Foundation, Eclipse Project, 2004.
Hhttp://www.eclipse.orgH

89 Xerox PARC, The AspectJ™ Programming Guide, 2004.

90 IBM Research, Hyper/J™ : Multi-Dimensional Separation of Concerns for Java™.
Hhttp://www.research.ibm.com/hyperspace/index.htmH

91 ObjectWeb consortium, The JAC (Java Aspects Component) project, version 0.12.1.
Hhttp://jac.objectweb.org/H

92 D. Shukla, S. Fell, C. Sells, Aspect Oriented Programming Enables Better Code Encapsulation and Reuse, dans MSDN Magazine, Mars 2002.

93 C. Czarnecki et U. W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

94 J. Greenfield et K. Short, Software Factories, Wiley Publishing, 2004.

95 J. Bézivin, From Object Composition to Model transformation with the MDA, dans *proceedings of TOOLS'USA 2001*, Santa Barbara, États-Unis, volume IEEE Publications TOOLS'39, 2001.

96 J. Bézivin, MDA™: From hype to Hope, and Reality, invited talk, dans *<<UML>>2003*, San Franscisco, États-Unis, 2003.

97 J. Bézivin et O. Gerbé, Towards a Precise Definition of the OMG MDA Framework, dans p*roceedings of the Conference on Autonomous Software Engineering (ASE'01)*, San Diego, CA, Etats-Unis, Novembre 2001.

98 J.Rothenberg, The nature of Modeling, dans *Artificial Intelligence, Simulation and Modeling*, pages 75-92, John Wiley and Sons, Inc, 1989.

99 Metamodel.com, site Internet, 2004.
Hhttp://www.metamodel.comH

100 Object Management Group (OMG), The Meta-Object Facility™ (MOF), spécification version 1.4, Avril 2002.

101 Object Management Group (OMG), The Meta-Object Facility™ (MOF), spécification version 2.0 en cours de finalisation, 2004.

102 Object Management Group (OMG), MDA Guide version 1.0.1, Juin 2003.

103 OMG Architecture Board MDA Drafting Team, Model-Driven Architecture: A Technical Perspective, document ab/2001-02-04, Février 2001.

104 J. D. Poole, Model-Driven Architecture : Vision, Standards and Emerging Technologies, dans *workshop on Metamodeling and Adaptive Object Models (ECOOP 2001)*, Budapest, Hongrie, Avril 2001.

105 Object Management Group (OMG), XML Metadata Interchange™ (XMI), specification version 1.2, Janvier 2002.

106 Object Management Group (OMG), XML Metadata Interchange™ (XMI), specification version 2.0, Mai 2003.

107 Object Management Group (OMG), Common Warehouse Metamodel™ (CWM) Resource Page, 2004.
Hhttp://www.omg.org/cwm/H

108 QVT-Merge Group, Revised submission for MOF 2.0 Query / Views / Transformations (ad/2002-04-10), version 1.8, Octobre 2004.

109 J. Bézivin, Les évolutions récentes en ingénierie des modèles, dans *it-expert*, numéro 51, Septembre/Octobre 2001.

110 Projet ModelWare, site Internet, 2004.
Hhttp://www.modelware-ist.org/H

111 Model Driven Software Development (MDSD), site Internet, 2004,
Hhttp://www.mdsd.infoH

112 J. Bettin, Model-Driven Software Development, dans *The MDA Journal (editeur D. S. Frankel)*, Business Process Trends, Avril 2004.

113 Manifesto for Agile Software Development, site Internet, 2004.
Hhttp://agilemanifesto.org/H

114 Generative Model Transformer (GMT) project, site Internet, 2004.
Hhttp://www.eclipse.org/gmt/H

115 G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, B. Selic, The IBM MDA Manifesto, dans *The MDA Journal (editeur D. S. Frankel)*, Business Process Trends, Mai 2004.

116 C. Czarnecki, Generative Software Development, Invited talk, <<UML>> 2004, Lisbon, Portugal, Octobre 2004.

117 Microsoft Corporation, Microsoft Visual Studio .NET, outil de développement, Hhttp://msdn.microsoft.com/vstudio/H

118 Eclipse Foundation, Eclipse Modeling Framework (EMF), 2004. Hhttp://www.eclipse.org/emf/H

119 A. Gerber et K. Raymond, MOF to EMF: There and Back Again, dans *proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, Anaheim, États-Unis, pages 60-64, ACM Press, 2003.

120 World Wide Web Consortium (W3C), Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommandation, Février 2004.

121 World Wide Web Consortium (W3C), WML Schema Part 0-2 (Primer, Structure, Datatype), W3C Recommandation, Octobre 2004.

122 Java Community Process, Java™ Metadata Interface (JMI) Specification, version 1.0, Juin 2002.

123 Sun Microsystems, Java Metadata Interface (JMI), site Internet, 2004. Hhttp://java.sun.com/products/jmi/H

124 NetBeans IDE, Metadata Repository (MDR) project, 2004. Hhttp://mdr.netbeans.org/H

125 ObjectWeb, Modfact JMI Repository Maker, 2004. Hhttp://forge.objectweb.org/projects/modfact/H

126 The Jamda Project, site Internet, 2004. Hhttp://jamda.sourceforge.netH

127 AndroMDA, site Internet, 2004. Hhttp://www.andromda.orgH

128 Velocity 1.4, The Apache Jakarta Project, site Internet, 2004. Hhttp://jakarta.apache.org/velocity/H

129 K. Czarnecki et S. Helsen, Classification of Model Transformation Approaches, dans *proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim*, États-Unis, Octobre 2003.

130 A. Gerber, M. Lawley, K. Raymond, J. Steel et A. Wood, Transformation: The Missing Link of MDA, dans *proceedings of the First international conference on Graph Transformation (ICGT 2002)*, Barcelona, Espagne, Octobre 2002.

131 M. Andries, G. Engels, A. Habel, B. Hoffmann, h.-J. Kreowski, s. Kuske, D. Pump, A. Schürr et G. Taentzer, Graph transformation for specification and programming, dans *Science of Computer programming*, volume 34, n°1, pages 1-54, Avril 1999.

132 Object Management Group (OMG), MOF 2.0 Query / Views / Transformations (QVT), Request For Proposal, Avril 2002.

133 QVT-Merge Group, Revised submission for MOF 2.0 Query / Views / Transformations RFP, version 1.8, Octobre 2004.

134 Raphaël Marvie, Vers des patrons de méta-modélisation, dans les *Actes de l'atelier de travail du groupe OCM du GDR ALP*, Vannes, France, Février 2003.

135 R. France, I. Ray, G. Georg et S. Ghosh, An aspect-oriented approach to early design modelling, dans *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, volume 151, n°4, page 173, Août 2004.

136 J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale et B. Natarajan, An approach for supporting aspect-oriented domain modelling, dans *Proceedings of the second international conference on Generative programming and component engineering (GPCE'03)*, Erfurt, Germany, Lecture Notes in Computer Science, volume 2830, pages 151-168, 2003.

137 J. Suzuki et Y. Yamamoto, Extending UML with Aspects: Aspect Support in the Design Phase, dans *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, Juin 1999.

138 W.-M. Ho, J.-M. Jézéquel, F. Pennaneac'h et N. Plouzeau, A Toolkit for Weaving Aspect Oriented UML Designs, dans *proceedings of the 1st international conference on Aspect-Oriented Software Development (AOSD'O2)*, pages 99-105, ACM Press, Enschede, Hollande, Avril 2002.

139 D. Stein, S. Hanenberg et R. Unland, Designing aspect-oriented crosscutting in UML, dans *AOSD-UML Workshop (AOSD '02)*, Enschede, Hollande, Avril 2002.

140 O. Aldawud, T. Elrad et A. Bader, UML Profile for Aspect-Oriented Software Development, dans *Proceedings of Third International Workshop on Aspect-Oriented Modeling*, Boston, États-Unis, Mars 2003.

141 R. Silaghi et A. Strohmeier, Generic Concern-Oriented Model Transformations meet AOP, dans *proceedings of the first workshop on Model-driven Approaches to Middleware Applications Development (MAMAD 2003)*, pages 307-311, Rio de Janeiro, Brésil, Juin 2003.

142 R. Marvie, Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants, *thèse de Doctorat*, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'Ascq, France, décembre 2002.

143 Codagen Technologies Corp, Codagen Architect, outil MDA, Hhttp://www.codagen.com/products/architect/default.htmH

144 Interactive Objects Software GmbH, Arc Styler, outil MDA, Hhttp://www.io-software.com/products/arcstyler_overview.jspH

145 Mia Software - Sodifrance, MIA-Transformation, outil MDA, Hhttp://www.mia-software.comH

146 ATLAS Group, ATLAS Transformation Language, Hhttp://www.sciences.univ-nantes.fr/lina/atl/atlProject/H

147 Triskell Team, MTL Engine et UMLAUT NG framework, outils pour la transformation de modèles Hhttp://modelware.inria.fr/rubrique.php3?id_rubrique=8H

148 S. Bonnet, R. Marvie et J.-M. Geib, dans *proceedings of the second Nordic Workshop on the Unified Modeling Language (NWUML'2004)*, TUCS General Publication, n°35, pages 99-113, Turku, Finlande, Août 2004.

149 J. Bézivin, F. Jouault et P. Valduriez, First Experiments with a ModelWeaver, dans *Workshop on Best Practices For Model Driven Software Development (OOPSLA'04)*, Vancouver, Octobre 2004.

Résumé

Face à la multiplication et la diversification des services informatisés proposés aux individus, la personnalisation devient un élément majeur de différentiation pour les fournisseurs. Grâce à ses propriétés de sécurité et de mobilité, la carte à puce dispose du potentiel nécessaire pour jouer un rôle clef dans le déploiement de services personnalisés nomades.

Les plates-formes logicielles des cartes à puce de prochaine génération offrent des possibilités accrues en terme de programmation et d'intégration au sein des infrastructures informatiques. Cette évolution des applications encartées nécessite à la fois un enrichissement et une modernisation du processus industriel de personnalisation. Les travaux décrits dans cette thèse constituent une proposition d'approche novatrice, fournissant aux développeurs les moyens méthodologiques et techniques de réaliser une personnalisation fonctionnelle à grande échelle de leurs applications.

Dans la perspective d'optimiser le processus industriel de personnalisation, les concepts de code personnalisable et de code personnalisé sont introduits : le second résulte du tissage entre le premier et un ensemble de données utilisateurs structurées dans des profils. Le code des applications encartées personnalisées est produit de manière semi automatisée sur la base d'une démarche exploitant la complémentarité entre les techniques de modélisation et le développement logiciel génératif. Un ensemble de méta-modèles structurés de manière à respecter la séparation des préoccupations permet d'exprimer les besoins de personnalisation, tandis qu'un cadre de conception permet d'implémenter les transformations et les tissages de modèles en Java.

Mots clefs : personnalisation, cartes à puce, développement logiciel génératif, méta-modélisation, transformations de modèles.

Abstract

Considering the multiplication and the diversification of ubiquitous computing services, personalization is becoming a major stake enabling providers to differentiate each other. Thanks to their reduced size and their security features, smart cards have the potential for playing a key role in the wide deployment of personalized services.

Software platforms coming along with next generation smart cards bring considerable improvements, such as more advanced programming models or better integration in computing infrastructures. As the forthcoming evolutions of embedded applications require an enrichment and a modernization of the personalisation process, both methodological and technological means to implement a large-scale personalization are proposed.

In order to optimize the industrial personalization process, a clear distinction is made between personalizable code and personalized code: the latter results from a weaving between the former and user-specific data gathered in profiles. The code of personalized embedded applications is produced in a semi-automated way, using a combination of model-driven techniques and generative software development. While a set of dedicated meta-models organized according to the separation of concerns principle allows to express the personalization requirements, a framework allows the implementation of transformation and weaving in Java.

Keywords: personalization, smart cards, generative software development, meta-modeling, model transformations.