THESE DE DOCTORAT DE L'UNIVERSITE PARIS 6

Spécialité :
Informatique

Option :
Intelligence Artificielle

présenté par

M. Seyed Reza RAZAVI EBRAHIMI

pour obtenir le grade de
DOCTEUR de l'UNIVERSITE PIERRE et MARIE CURIE (PARIS 6)

Sujet de la thèse :

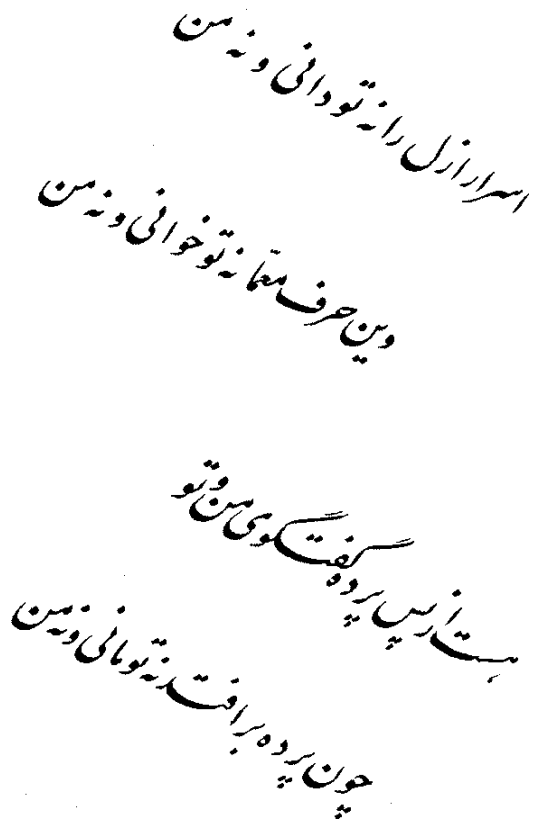
Outils pour les Langages d'Experts

Adaptation, R*efactoring* et Réflexivité

soutenu le 30 novembre 2001 devant le jury composé de MM.

Isabelle BORNE, Université de Bretagne Sud	**Rapporteur**
Christophe DONY, Université Montpellier - II	**Rapporteur**
Max FONTET, Université Pierre et Marie Curie (Paris 6)	**Président**
Vincent GINOT, Institut National de la Recherche Agronomique	**Examinateur**
Philippe KRIEF, *Object Technology International Inc.* (OTI)	**Examinateur**
Gil BLAIN, Université Pierre et Marie Curie (Paris 6)	**Invité**
Jean-François PERROT, Université Pierre et Marie Curie (Paris 6)	**Directeur de thèse**

Avant-propos



Voici la Porte à laquelle je ne trouverai point Clef,
Voici le Voile au travers duquel je ne pus voir : Pourquoi
Quelque temps parla-t-on un peu de Moi et de Toi,
Et plus tard, ne parlera-t-on plus jamais de Toi ni de Moi ?

Omar Khayyâm, Robaiyyat, n° 2
[Ed. et trad. M. Ramasani, Padideh, Téhéran, 1981]

Les langages d'experts présentés et outillés dans ce mémoire traitent la question de l'adaptation de logiciels aux changements de la réalité modélisée par ces derniers. Il convient donc de se demander ce qu'est la *réalité* de façon générale.

Khayyâm veut évoquer ici l'idée selon laquelle *la réalité n'est qu'un effet individuel et changeant, produit par un mécanisme d'interprétation.* Pour ce faire, il s'appuie sur la métaphore du "rideau" (mot traduisant mieux que "Voile" le terme persan "pardeh") qui lui offre trois propriétés intéressantes :

1) En premier lieu un rideau peut être le support d'un reflet ;
2) C'est l'état du rideau qui détermine la forme du reflet : le rideau peut changer, tout comme un interprète méta-circulaire, et conduire au changement de la "forme", la réalité ;
3) En fin, le rideau peut tomber et faire disparaître avec lui le reflet.

Khayyâm observe donc qu'entre nous, (en tant qu'interprète,) et la réalité il existe un rapport comparable avec celui du rideau et du reflet : *il faut un support pour que cette réalité prenne forme ; et la réalité change et disparaît en même temps que ce support*. Il est important de noter ici le lien de dépendance entre la forme que prend la "réalité" et la réalité du support qui a porté cette forme.

Un exemple remarquable de telles mises en relations est fourni par les ordinateurs et par la programmation. L'informatique peut s'expliquer comme étant l'art de la création d'interprètes et d'effets par programmation. C'est là qu'à nos yeux l'Intelligence Artificielle, c'est-à-dire la *réalité programmée*, prend sens.

Pour revenir aux langages d'experts, le choix de ce quatrain s'explique par la résonance que nous trouvons entre la pensée de Khayyâm, il y a près de 1000 ans, et notre principale observation lors de ce travail de recherche décrite très succinctement à la section 4.2, page 174 du chapitre IV.

Cette interprétation est sûrement influencée par des travaux de recherche présentés dans ce mémoire, lesquels ont sans nul doute été à leur tour influencés par notre culture persane de façon générale et la lecture des Robaiyyat de façon plus particulière. Le "théorème" de Khayyâm est donc bien vérifié ici, la culture jouant évidement le rôle du support.

A nos yeux, le but de la recherche scientifique peut-être décrit dans ce cadre comme étant un effort d'explicitation de la nature du nombre infini d'interprètes qui nous entourent ainsi que leurs *interdépendances*.

Les langages d'experts constituent une contribution à cet effort dans le sens où ils expliquent la nécessité de conserver le "lien de programmation" entre un logiciel et l'outil de programmation qui a servi à sa création. De façon plus précise, on apprend qu'il devient possible de faire évoluer un logiciel objet lorsque la réalité qu'il modélise évolue *si lui-même est conçu comme un interprète*, un "rideau". On explique également une technique qui montre comment accomplir une telle tâche.

Remerciements

A tous ceux qui ont su aimer et respecter l'Homme

Je tiens à remercier toutes les personnes qui, par leur sympathie pour mon projet, leurs compétences intellectuelles et leur professionnalisme, retrouvent un peu d'elles-mêmes dans ce travail, même si ses inévitables imperfections ne sont imputables qu'à moi.

Cette thèse est le résultat d'un long cheminement, malaisé, qui n'aurait jamais pu aboutir sans la générosité et l'humanisme de nombreuses personnes que je souhaiterais également remercier à cette occasion.

Je remercie sincèrement et avec simplicité :

- Jean-François Perrot pour avoir accepté de diriger ces travaux et m'avoir accordé une grande confiance dans le développement de mes idées ; pour son rôle primordial lors des choix cruciaux et ses conseils toujours très pertinents. Mais ma dette excède le cadre de cette thèse car c'est d'abord en tant que son élève de Compilation que j'ai appris à mêler réflexions esthétiques sur les langages de programmation et préoccupations techniques et scientifiques. J'aimerais profiter de l'occasion pour lui témoigner mon admiration pour son œuvre intellectuelle et sa grande probité.

- Isabelle Borne pour avoir accepté de rapporter ce travail et pour sa lecture minutieuse de ce mémoire ; pour ces travaux sur les schémas de conception et les frameworks qui étaient pour moi une source d'inspiration.

- Christophe Dony pour avoir accepté de rapporter ce travail ; pour ses nombreuses questions et remarques fines et pertinentes ; pour ses travaux sur les langages à prototypes qui m'ont servi de cadre lors du rapprochement entre les "classes autonomes" et les prototypes.

- Max Fontet pour l'honneur qu'il me fait en présidant le jury de cette thèse ainsi que pour son grand intérêt pour nos travaux et ses encouragements.

- Vincent Ginot pour avoir accepté d'examiner ce travail ; pour sa confiance en les résultats de cette recherche, son soutien financier durant les trois derniers mois de la rédaction de ce mémoire (à travers notre projet de collaboration et le contrat avec l'INRA) ; pour son amitié.

- Philippe Krief pour avoir accepté d'examiner ce travail ; pour m'avoir toujours conseillé avec intelligence et pertinence. Il a été pour moi un guide dans mon itinéraire professionnel et dans mon ascension dans l'univers des objets. Il est, à mes yeux, un modèle en incarnant l'art de la programmation par objets.

- Gil Blain pour être présent dans mon jury de thèse ; pour m'avoir initié à la méta-modélisation et sensibilisé sur le rôle potentiel des experts dans un processus de développement ; pour m'avoir accueilli au sein de l'équipe Métafor du Lip6 et pour m'avoir assisté lors de nombreuses démarches administratives du début de cette thèse.

- Jean-Pierre Briot pour son œuvre scientifique qui a été source d'inspiration pour moi ; pour son soutien, en tant que responsable du thème OASIS et du groupe Framework, lors de mes nombreuses initiatives et voyages.

- Tous les membres du groupe Framework de l'équipe OASIS pour m'avoir écouté avec intérêt et conseillé tout au long de l'avancement de cette thèse.

- Nicolas Revault, Houari Sahraoui et Mikal Ziane, membres de l'équipe Métafor pour avoir partagé avec moi leur expérience du déroulement d'un projet de thèse et pour m'avoir initié à la construction d'outils de méta-modélisation.

- Noury Bouraqadi pour ses remarquables travaux sur la réflexion dans les langages à objets et le système MetaClasstalk qui été l'un des piliers de cette recherche ; pour sa générosité à finaliser et mettre à ma disposition son système.

- Ralph Johnson et tous les membres du groupe Software Architecture Group à UIUC pour l'ensemble de leur œuvre scientifique qui est sans conteste la base de nos travaux ; pour les nombreux échanges fructueux que j'ai pu avoir avec eux. Je n'aurais jamais pu développer le lien avec les travaux sur le *End user programming*, sans le conseil pertinent de Ralph Johnson à propos des travaux de Nardi. J'aimerais lui réaffirmer ici mon amitié et mon admiration pour ses valeurs humaines et son intelligence.

- Joseph W. Yoder, Ali Arsanjani et tous les autres collègues "adaptatives" avec qui j'ai partagé des grands moments d'émotion lors de nos workshops à ECOOP et OOPSLA.

- Stéphane Ducasse, Hafedh Mili, Francois Pachet, Dirk Riehle, Michel Tilman, Francis Wolinski et tout ceux qui ont bien voulu relire et commenter mes écrits, pour leur patience et intérêt.

- Tous les doctorants et membres du Laboratoire d'Informatique de Paris 6 avec qui j'ai eu le plaisir de partager trois des meilleures années de ma vie professionnelle.

- Ghislaine Mary et Jacqueline Lebaquer et leurs collègues pour leur gestion sans faille des différents aspects administratifs de cette thèse. Christophe Bouder qui cache sous l'appellation d'Ingénieur Système un ami serviable et impressionnant de compétences.

- Jeannine Sivel et tous les membres du service de la Formation Permanente de l'Université Paris 6 pour avoir géré avec une très grande efficacité ces longues années de formation jusqu'en thèse.

- Jacques Nefussy, président de la Société Cablaccès à Villejuif, pour avoir appuyé ma demande d'inscription à l'Université Paris 6, alors qu'en 1989 j'ai été en stage dans son entreprise.

- Aristide Schlienger et tout mes anciens collègues de la société Julia SA dont les exigences en matière d'outils pour l'écriture des gammes de mesure m'ont permis de développer les premières versions des modèles complexes présentées dans ce mémoire. C'est également A. Schlienger qui a fait pour la première fois, lors de l'une de nos conversations vieux déjà de près de dix ans, la correspondance entre le modèle de programmation des gammes dans Marlene et celui des tableurs. Le hasard a voulu que je me retrouve encore une fois sur cette trajectoire suite au conseil de R. Johnson d'étudier les travaux de Nardi, laquelle évoque la pertinence du modèle des tableurs pour la programmation par des experts. Le chemin fut donc long, mais l'exemple de réutilisation efficace de Dyctalk dans Mobidyc le montre, bien fructueux.

- Jean Gouy et tout mes anciens collègues du laboratoire de métrologie qui porte son nom, pour donner un exemple vivant et efficace du rôle que les experts non-informaticiens peuvent jouer dans un processus de développement ; pour leur intérêt à mener à bien le projet Calibres malgré l'originalité de notre démarche. C'est lui qui est le point de départ des travaux formalisé lors de cette thèse.

- Rachid Senoussi et tout mes collègues de l'unité Biométrie de l'INRA-Avigon, ainsi que François Rodolph et mes collègues de l'unité Mathématiques, Informatique et Génome de l'INRA-Versailles pour la chaleur de leur accueil et les moyens mis à ma disposition pour tester les résultats de cette recherche.

Bien qu'il y ait de multiples occasions de dire à ses proches combien leur compétences et leur compréhension furent précieuses, je souhaite conclure ces remerciements en leur adressant mes plus affectueuses pensées. Mes parents ont jouté un rôle clé dans mon intérêt pour l'enseignement et la recherche ; mes amours, Shafagh, Cinna et Goya, m'ont permis de réaliser ce rêve vieux de quarante ans. Merci …

Paris le 8 mars 2002,
Reza Razavi

Table des matières

Conventions typographiques

Le texte général est en Garamond 11.

`Le code source en SMALLTALK est encadré et en Courier 9.`

`Le nom des éléments du code apparaissant dans un texte est en Courier 11.`

Le nom des schéma de conception est en Garamond 11 et en italic.

`Le nom des attributs et de procédures définis par des experts est en Courier 9.`

LE NOM DES LOGICIELS, SYSTEMES DE CLASSES ET LES FRAMEWORKS EST EN GARAMOND 11 ET EN PETITES MAJUSCULES.

Introduction

1 Vue d'ensemble

1.1 Cadre

1.1.1 Motivation : logiciels dynamiquement modifiables par des experts

L'utilisation de certaines applications nécessite deux niveaux d'intervention : la mise en œuvre courante et l'adaptation à de nouveaux besoins. L'idée est que la spécification du service rendu par le logiciel peut varier au cours du temps. Il y a donc d'une part les utilisateurs habituels, qui utilisent le logiciel pour obtenir le service en question. Mais il y a également des utilisateurs privilégiés qui peuvent apporter au système, alors même qu'il est en fonctionnement, des adaptations qui viendront modifier le service obtenu par l'utilisateur final. Nous appellerons "experts" ces utilisateurs privilégiés.

Par exemple, un établissement bancaire propose à ses clients divers produits financiers. Ces produits donnent lieu à différentes procédures qui interviennent dans la manipulation des *comptes* des clients dans le système d'information de la banque. De notre point de vue, les utilisateurs finaux sont les guichetiers et tous les employés qui ont à manipuler des comptes individuels.

Dans le contexte fort concurrentiel d'aujourd'hui il est fréquent qu'un tel établissement conçoive et propose à ses clients de nouveaux produits financiers [Rie98]. La conception de ces produits est le fait d'organismes spécialisés. Il faut ensuite les intégrer dans le système d'information de la banque. Les spécialistes financiers qui effectuent cette intégration sont nos "experts". Leur intervention ne doit pas nécessiter l'appel aux informaticiens pour mettre à jour le code du système, ce qui nécessiterait le transfert des connaissances des experts sur le sujet à ces derniers et entraînerait également l'arrêt de l'exploitation du système actuel, l'installation du nouveau système, son démarrage, paramétrage, etc.

Un autre exemple tiré d'un domaine tout à fait différent est celui de systèmes de facturation de services télécoms [AJ98]. La facturation consiste à traiter les transactions et calculer les différents montants à payer par chaque client. La facturation des services télécoms est une tâche difficile car le type des transactions et les règles de leur facturation sont en évolution constante et diffèrent pour chaque compagnie. Pour faire face à cette situation il est nécessaire de créer des logiciels de facturation qui automatisent la définition de nouveaux services et leur gestion informatique par des experts - sans transfert de connaissances aux informaticiens et sans interruption du service, bien entendu.

Le même besoin se retrouve dans le cas des systèmes CAO. Frédéric Duclos, Jack Estublier et Rémy Sanlaville décrivent ainsi dans leur récente publication [DES00] les préoccupations de la société Dassault Systèmes, numéro un mondial en édition de systèmes CAO, à propos de l'adaptabilité de leur logiciel CATIA V5 aux besoins locaux du métier de leurs clients (aéronautique, automobile, biens de consommation, ...) :

> "Les logiciels de grande taille ont de plus en plus besoin d'être adaptés par le client. Chaque client a en effet ses besoins propres en termes de fonctionnalité. Pour des raisons de temps de développement et de coûts, le fournisseur ne peut pas adapter le logiciel à chaque client." [DES00]

Selon nous, là aussi, l'intervention directe de l'expert CAO devait être possible et cela sans nécessiter l'arrêt du système. Autrement dit, le logiciel doit être conçu pour ce type d'adaptation.

Il est important de noter ici la différence entre le phénomène décrit ci-dessus et le paramétrage. En effet, le paramétrage est utilisé pour faire varier le fonctionnement d'un système selon la valeur courante des options disponibles. Ces options sont définies en fonction des différentes variations possibles dans le déroulement du code que les programmeurs ont pu mettre en évidence lors de l'analyse. C'est donc un moyen utile dans le cas ou *les variations sont connues d'avance*. Or, ici nous nous intéressons à l'ajout dynamique de nouvelles structures et de nouveaux procédés non prévus lors de la livraison. Le but de cette intervention est de modifier les structures et comportements (classes) déjà définis par les programmeurs qui ont écrit le système.

1.1.2 Notions d'adaptation et de langage d'experts

La création de ces logiciels à deux niveaux d'usage pose des problèmes particuliers. Il faut en effet, que les experts puissent introduire dynamiquement à la fois la définition de structures de données, et celle de procédés de calcul qui produisent ces données et opèrent sur elles. Il faut aussi que les utilisateurs finaux puissent instancier ces nouvelles structures de données et exécuter ces nouveaux procédés.

Nous faisons l'hypothèse que ces logiciels sont créés à l'aide des langages à objets [Per92]. Nous raisonnons donc en termes de classes, d'instances, de méthodes et d'héritage. Nous définissons alors *l'adaptation* comme la spécialisation dynamique de classes par l'ajout lors de l'exécution de nouveaux types d'objets ainsi que la définition de leur structure et procédures et cela par des experts du domaine et non par des programmeurs. Nous définissons également un *langage d'experts* comme une spécialisation d'un langage à objets (réflexif) qui assure et systématise l'adaptation *pour un domaine* d'application donné.

A titre d'exemple, un langage d'experts dédié à la gestion de comptes bancaires sert dans un premier temps aux experts pour créer facilement et dynamiquement de nouveaux types de comptes bancaires. Il sert ensuite aux utilisateurs, e.g. guichetiers, leur permettant de proposer ces nouveaux comptes aux clients.

1.1.3 Notre expérience industrielle en création de langages d'experts

Nous avons déjà rencontré une situation similaire dans le cadre de nos projets antérieurs[1]. Notre expérience industrielle s'est déroulée de 1993 à 1998 au sein d'une *start up* française, la société Julia SA. Elle a consisté essentiellement à mettre au point une plate-forme pour la création de certaines applications de métrologie. Cette plate-forme a été développée à partir du logiciel MARLENE, destiné au contrôle 3D, dont le noyau en SMALLTALK-80 a été réalisé entre 1992-93 par la Sté ACKIA, et tout particulièrement par Philippe KRIEF [Kri90, Kri96]. De cette plate-forme nous avons dérivé deux logiciels dédiés à la programmation par des experts. L'un sert à créer des logiciels de contrôle 3D (PRELUDE INSPECTION [ASM99]) et l'autre des logiciels d'étalonnage de calibres (CALIBRES [Raz99, Raz00a, Raz00c]).

[1] Cette thèse s'est déroulée dans le cadre d'une formation professionnelle organisée par la Formation Permanente de l'Université Paris 6 et financée par les Assedic.

PRELUDE INSPECTION est sur le plan fonctionnel dédié à la programmation des gammes de mesure de la géométrie des pièces usinées. Toutefois, il n'est pas un langage d'experts au sens défini ici, car les procédures (de mesure) définies dynamiquement n'opèrent pas sur des données qui soient également définies dynamiquement. En effet, dans ce cas toutes les structures de données sur lesquelles opèrent les procédures, essentiellement les objets qui modélisent des formes géométriques 3D, sont connues des programmeurs lors du développement initial.

Au contraire, le système CALIBRES peut être considéré comme un langage d'experts. Il permet aux experts en métrologie de développer relativement facilement et rapidement des logiciels pour l'étalonnage des calibres en métrologie dimensionnelle. Un calibre est un appareil de mesure de grande précision. L'utilisateur final du système exécute sur un jeu de calibres une procédure d'étalonnage qui détermine les valeurs d'écarts par rapport aux références nationales [NFX07]. Vu la multiplicité et la diversité des types de calibres, les demandes spécifiques des clients[2] et aussi en raison de l'évolution constante des normes, il est régulièrement nécessaire de faire mettre à jour le logiciel par un expert métrologue. Celui-ci qui va introduire de nouveaux types de calibres et de nouvelles procédures d'étalonnage. Cela consiste en la spécification des attributs caractéristiques des calibres ainsi que de leurs procédures et certificats d'étalonnage. On trouve donc dans CALIBRES à la fois les deux aspects "structure" et "comportement" envisagés ci-dessus.

Nous fournissons en annexe VIII, page 283, plus d'informations sur nos activités industrielles.

1.2 Propriétés des langages d'experts

Nous classifions en deux catégories les propriétés que nous estimons souhaitables des langages d'experts.

La première catégorie comporte les aspects plutôt techniques, c'est-à-dire ceux qui rendent l'adaptation possible, opérationnelle et adaptée à son objet. Cela comprend quatre volets :

1. permettre la *spécialisation dynamique* ;
2. permettre la gestion des workflows (désormais, en abrégé, la *dimension workflow*) ;
3. permettre le travail collaboratif entre les programmeurs et les experts (désormais, en abrégé, le *travail collaboratif*) ;
4. permettre le choix local, par des programmeurs et experts, du type d'adaptation (désormais, en abrégé, le *choix local du type d'adaptation*).

La seconde catégorie regroupe les aspects plutôt d'ordre cognitifs. Ce sont ceux qui rendent l'adaptation facile à réaliser et bien intégrée dans un processus de développement de logiciels. Cela comprend deux volets :

1. être facile à apprendre et à utiliser par des experts (désormais, en abrégé, *apprentissage par les experts*) ;
2. assurer un lien causal entre la définition des structures et leurs instanciations ainsi que la définition des procédures et leurs interprétations (désormais, en abrégé, le *lien causal*).

Notre travail vise à "simultanément" obtenir l'ensemble de ces propriétés.

[2] Il s'agit en règle générale de très grands industriels des domaines comme l'aéronautique, l'automobile, l'armement, etc.

1.2.1 Aspects techniques

Spécialisation dynamique

La spécialisation dynamique constitue la propriété principale des langages d'experts. En effet, adapter et spécialiser visent le même objectif, à savoir structurer ou organiser, suivant une certaine logique, un ensemble d'individus, *d'êtres* [Lie86]. Autrement dit, vouloir adapter reflète, du point de vue de langages à objets, le besoin de *prolonger* la spécialisation de classes lors de l'exécution de logiciels objets.

C'est pourquoi nous concevons l'intervention des experts comme une forme particulière de la spécialisation. Celle-ci se déroule lors de l'exécution et consiste à raffiner des concepts existants par l'ajout de nouveaux concepts et la définition de leurs attributs (structures et procédures).

De ce fait, les langages d'experts peuvent être considérés dans la catégorie des environnements de programmation basés sur la création et l'interprétation de modèles (model-based programming systems) [RFBO01][3]. Dans ces cas, l'idée consiste à décrire le programme dans un langage indépendant d'une implantation spécifique et puis interpréter cette description (modèle de programme) et/ou la transformer en code exécutable.

De façon semblable au cas des langages d'experts, un tel outil doit fournir un langage qui permet de décrire des structures et des procédures, alors qu'il est lui-même en cours d'exécution[4]. Nous reviendrons dans le §1.4.1, page 22 sur les problèmes posés actuellement pour la création systématique de ce type de logiciels.

Il est important de noter ici que l'adaptation se différencie principalement de la spécialisation par le fait qu'elle est mise en œuvre par des experts non-informaticiens. Le système (langage d'experts) doit donc être conçu pour assurer cette fonctionnalité[5]. Toutefois, pour atteindre effectivement cet objectif et cela de façon appropriée, il est nécessaire de prendre dans cette conception en considération d'autres aspects dont l'exposé va suivre.

Dimension workflow

Le *Workflow* se trouve aujourd'hui au cœur des systèmes d'informations des entreprises [LR00] et la modélisation des procédés métier constitue l'une des activités majeures des experts non-informaticiens. Dans la mesure où les langages d'experts sont conçus pour ces derniers, il convient alors qu'ils assurent également cette fonction.

Mais notre objectif, tout comme le suggère Dragos Manolescu [Man00], consiste à aller plus loin que les systèmes de gestion de workflow traditionnels car une description fine et *adaptative* des workflows nécessite non seulement pouvoir décrire le déroulement d'une suite d'applications (niveau "macro"), mais aussi et surtout pouvoir aisément éditer et suivre l'exécution des collaborations entre les objets métiers (niveau "micro").

Nous avons déjà rencontré cette nécessité lors de nos projets industriels. A titre d'exemple, une procédure d'étalonnage est un ensemble d'actions de mesure et de calcul sur des objets du domaine tels que cercles, plans, etc. Une telle procédure est écrite par des métrologues comme des appels à des méthodes primitives de ces objets, lesquelles sont codées par des programmeurs. Elle est donc comparable

[3] Ce qui par ailleurs, différencie ces outils des langages d'experts, c'est la recherche systématique d'une accessibilité aux experts par la conception elle même d'un langage d'experts.

[4] Un des principaux éléments qui caractérisent cette catégorie de logiciels du point de vue de la programmation par objet, c'est qu'ils comportent des classes dont chaque instance représente elle-même une classe. Il s'agit d'une situation relativement récurrente, en particulier dans le cas de logiciels objets dédiés à la méta-modélisation [MM01] ou encore les machines virtuelles UML [RFBO01]. C'est, à titre d'exemple, le cas du système METAGEN [RSBP95, LSGBP99, RBP00] où chaque instance de la dasse `MetaIndividu` décrit une classe d'objets et est effectivement compilée sous forme d'une sous-classe de la classe `Individu`. Un autre exemple remarquable, c'est celui des langages à objets réflexifs où chaque classe est elle-même instance de sa méta-classe.

[5] Rappelons, en effet, que notre objectif est ici de proposer et valider une conception qui permet d'outiller la création systématique de ce type de logiciels.

à un workflow classique dans la mesure où elle décrit des tâches à exécuter, synchroniser, superviser, etc. Mais, elle est en même temps différente car elle remplace de fait une procédure qui est dans la pratique courante des langages à objets est écrite sous forme d'une méthode. Le problème est que les langages de programmation ne sont pas adaptés aux experts (métrologues) qui préfèrent un langage plus facile à apprendre et à mettre en œuvre.

Notre choix répond donc à une nécessité réelle.

Travail collaboratif (Refactoring et édition des adaptations par les programmeurs)

Assurer le bon fonctionnement, la maintenabilité et l'évolutivité d'un logiciel relève de la responsabilité des informaticiens. Les langages d'experts suivent cette règle générale, même s'il permettent la modification du système par des experts. Comme le précise Bonnie A. Nardi[6] [Nar93, pages 103-121], ce type d'environnements doit alors être conçu pour favoriser le *travail collaboratif* entre les programmeurs de différents niveaux, y compris des experts.

Cette collaboration dans le contexte des langages d'experts prend une forme particulière. Il s'agit, en effet, de permettre aux programmeurs d'éditer, et plus particulièrement procéder au *refactoring* [Opd92, Rob99, FBBOR99] des spécialisations dynamiques des modèles objets (opérées par des experts) à travers leurs outils habituels.

Nous estimons que cette dimension doit aussi être intégrée dans la conception elle-même des langages d'experts.

Choix local du type d'adaptation

Lors de cette étude nous avons observé des "types" divers d'adaptation, que nous avons, par ailleurs, répertorié (cf. une liste, très probablement non-exhaustive, au §1.4, page 150 du chapitre IV). A tire d'exemple, l'*adaptation prototypique* (cf. le §2.1, page 207 sur les perspectives) fait partie de cet ensemble. Son rôle est de permettre de prolonger l'adaptabilité au niveau de *chaque instance terminale* et de ce fait assurer une activité de modélisation du type prototypage.

De plus, par rapport à ces types "standard" d'adaptation, il existe également des types d'adaptation qui sont propres au domaine d'application visé. Ces derniers sont, en règle générale, créés par la spécialisation des types standard pré-définis. Un exemple d'une telle spécialisation est fourni dans le paragraphe 2.4, page 188 du chapitre V (cas des comptes bancaires).

Sur le plan technique, nous estimons que cette situation est comparable à celle qui a conduit à l'étude du choix explicite des méta-classes par Pierre Cointe & al. [LC96]. C'est d'ailleurs, la même technique que nous proposons d'utiliser ici afin d'apporter une solution satisfaisante au problème posé par cette variation sur le type d'adaptation.

La différence des deux cas est que les *méta-classes se retrouvent ici au centre des pré-occupations des programmeurs, lors de la création de "l'aspect de base"* (cf. le §4.2.1, page 196 du chapitre V) des applications.

Dans ce contexte, il se pose alors le problème du choix local du type d'adaptation (de façon similaire à la question posée par Pierre Cointe & al. sur le choix des propriétés de classes [LC96]). Celui-ci est indispensable pour assurer un traitement adapté à chaque cas d'usage de l'adaptation.

Par exemple, si une adaptation est du type prototypique, il ne faut pas nécessairement que ses sous-classes/adaptations soient également de ce type. Cela conduirait à une généralisation de l'usage de

[6] *Bonnie A. Nardi* se présente comme une anthropologue "high-tech". Ses travaux de recherche actuels portent sur l'étudie du comportement au travail des spécialistes en biologie moléculaire. Cette activité se déroule dans le cadre du projet *Experiment Management Project* au département *BioScience Information Solutions* au sein de laboratoire *Agilent Laboratories* chez *Agilent Technologies*. Avant de rejoindre Agilent, elle a mené des études sur le *social networks in the workplace au sein du département Human Computer Interaction Department* du *laboratoire Information Systems and Services Research Lab* chez *AT&T*. Auparavant, elle s'est également intéressée à la théorie de l'activité (*Activity Theory* de Leont'ev 1974) et à la création des *ateliers d'experts* (end user programming systems) [Nar93].

l'adaptabilité *prototypique*[7] et à la disparition de la notion d'abstraction au profit des prototypes. Les conséquences négatives d'une telle situation sont déjà étudiées par ailleurs [Mal97, DMB98][8].

Nous estimons donc indispensable qu'un outillage dédié à la création de langages d'experts assure le choix local du type d'adaptation.

1.2.2 Aspects cognitifs

Apprentissage par les experts

Un langage d'experts doit être facile à apprendre et à maîtriser par des experts. C'est un objectif tout à fait réaliste. En effet, comme le précise également Nardi, un expert est un non-informaticien qui a la capacité, le savoir métier et l'intérêt pour programmer[9]. Il peut créer, personnaliser, spécialiser et faire évoluer ses propres applications [Nar93, page 5][10].

Le vaste usage de tableurs par des personnels administratifs pour développer des applications de gestion offre une bonne illustration de la faisabilité et de l'intérêt de cette approche. Notre propre expérience de cinq années de collaboration avec des métrologues dont la programmation, l'écriture des gammes de contrôle 3D, fait partie des activités quotidiennes confirme également ces observations.

En ce qui nous concerne ici, l'enjeu consiste alors à prévoir cette dimension dans la conception de l'outillage des langages d'experts. Il faut en particulier que les experts puissent mettre facilement en œuvre la spécialisation dynamique. Mais, il faut aussi que la prise en charge de cette propriété par notre outillage se fasse en harmonie avec celles déjà énumérées ci-dessus.

Lien causal (Causal connection)

Dirk Riehle & al. [RFBO01], insistent sur l'importance du *lien causal* (*causal connection*) à savoir la propriété d'un système où entre les différents niveaux de modélisation, e.g. les méta-modèles et les modèles ou les modèles et leurs instanciations, il existe une relation telle qu'un niveau N se conforme toujours aux niveau N+1[11]. Cela implique que toute modification du niveau N+1 se répercute (immédiatement) sur le niveau N. Cette définition est empruntée des travaux sur la réflexion et les systèmes à méta-niveaux [Foote92, Zim96].

Par exemple, si le niveau N+1 comporte la définition d'un type d'objet tel qu'un compte en banque (son nom, sa structure et ses comportements), alors tout changement de cette définition se répercute sur la structure et le comportement des instances d'un tel compte. Si une procédure de calcul des intérêts (qui est écrit au niveau N+1) est modifiée, alors toutes les exécutions en cours d'une telle procédure (au niveau N) doivent (optionnellement) se conformer (immédiatement) à la nouvelle définition.

Cette caractéristique offre une importante réactivité dans l'exploration de modèles de programmes, tout comme le cas des programmeurs SMALLTALK-80 [GR83, Ing81] qui peuvent écrire et immédiatement exécuter leur code dans un espace de travail (*workspace*)[12].

---

[7] Dans le cas où le choix local du type d'adaptation ne serait n'est pas assuré.

[8] Un exemple plus familier est celui de la programmation simultanée en SMALLTALK et JAVA (par exemple, dans le cas des systèmes SMALLTALK/X [CG01] et FROST [Frost97]), où il devient de fait nécessaire de pouvoir choisir la nature SMALLTALK ou JAVA de chaque classe, indépendamment de la nature de sa super-classe ou ses sous-classes.

[9] *End users have the detailed task knowledge necessary for creating the knowledge-rich applications they want and the motivation to get their work done quickly, to a high standard of accuracy and completeness* [Nar93].

[10] *Highly specialized end user programming environments that leverage users' existing task-related interests and skills* [Nar93].

[11] *A modeling level is causally connected with the next higher modeling level, if the lower level conforms to the higher level and if changes in the higher level lead to according changes in the lower level* [RFBO01, page 3].

[12] Même s'il n'y a pas de relation causale, au sens de D. Riehle & al., entre un programme SMALLTALK et ses exécutions.

L'absence d'une telle fonctionnalité rend l'activité de modélisation frustrante et impraticable pour des experts. En effet, ces derniers ne peuvent pas observer dans des délais acceptables les effets produits par un changement au sein du modèle au niveau de ses instances [RFBO01].

C'est pourquoi nous estimons important q'un langage d'experts soit également muni de cette propriété. Il est aussi important de noter que celle-ci concerne plus particulièrement le processus de mise en œuvre de la spécialisation dynamique.

1.3 Objectif : faciliter et systématiser la création de langages d'experts

1.3.1 Notion d'outillage

Nous entendons ici par *système de classes* la spécification d'un framework orienté-objets [Joh92, Joh97a, Joh97b, FSJ99, HJ98, Rie99, AB01] sous forme d'un ensemble de classes et de leurs relations[13]. Celle-ci utilise de préférence les schémas de conception [GHJV95, ABW98] afin de non seulement documenter une solution abstraite réutilsable (le "quoi"), mais aussi de justifier les décisions prises lors de sa conception (le "pourquoi"). Elle comporte également du texte et des diagrammes UML [FS97]. Dans ce mémoire nous utilisons indifféremment *système de classes* ou *solution de conception*.

Nous utilisons le terme *framework* pour se référer à une implantation (code exécutable) d'un système de classes. Pour désigner l'ensemble, c'est-à-dire un système de classes et son implantation, nous utilisons le terme *outillage*. Les frameworks sont déjà par ailleurs connus pour leur intérêt fondamental pour outiller la création de logiciels objets [Foote88][14].

Cette distinction entre un framework et sa documentation sous forme d'un système de classes est faite dans le but de mettre l'accent sur l'importance de la documentation des frameworks, qui, de part leur nature, apparaissent lors des évolutions successives d'un système [RJ97, RJ98] sans pour autant être documentés [Foote88][15].

1.3.2 Outiller l'adaptation dynamique de programmes par des experts

A ce jour les environnements de programmation ne supportent pas ce type de développement de façon standard, et il n'existe pas de système de classes standard pour outiller ce type de création. En somme, les outils actuellement mis à la disposition des programmeurs objets ne correspondent pas à leurs besoins quand il s'agit de créer des systèmes dont il est ici question. Les programmeurs sont alors obligés de créer et de maintenir d'outils *ad-hoc*. Ce fut, à titre d'exemple, notre cas lors de la création du système CALIBRES.

Cette situation constitue un frein important au développement de ce type de logiciels, pourtant essentiels notamment pour mieux confronter les évolutions constantes actuelles des métiers [RDREM00].

C'est pourquoi notre *objectif* est ici la recherche d'un système de classes qui décrit la nature de l'adaptation. Par la création d'un framework orienté-objets sur la base de ces spécifications, nous

[13] *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes [JF88]. Recall that one way to look at a framework is as an abstract design. Such a design is extended and made concrete via the definition of new subclasses. Each method that a subclass adds to such a framework must abide by the internal conventions of its super-classes.* [Foote88]

[14] *A framework for a given application domain can often serve as the basis for the construction of tools and environments for constructing and managing applications.* [Foote88]

[15] *It is possible, albeit difficult, to design good class libraries and frameworks in a top-down fashion. More frequently, good class libraries and frameworks emerge from attempts to solve individual problems as the need to solve related problems arises. It is through such experience that the designer is able to discern the common factors present in solutions to specific problems and construct class hierarchies that reflect these commonalities. It is the ability of inheritance hierarchies to capture these relationships as they emerge that makes them such powerful tools in environments that must confront volatile requirements.* [Foote88]

montrons expérimentalement dans quelle mesure un tel système permet d'outiller effectivement la création systématique et contrôlée de langages d'experts.

Comme le précisent Ralph Johnson & Brian Foote [JF88] ou encore Brian Foote [Foote88] [16] et Nardi [Nar93, page 6], la définition de systèmes de classes est une tâche extrêmement difficile à mener à bien :

> *We believe that there are generalizable lessons that can be learned from successful systems and applied to the design of new software systems --- as well as to new social systems to support effective software use. Of course design still is, and almost certainly always will be, a* ***black art*** *whose most crucial elements remain an incalculable mix of imagination, intuition, and intellectual interaction with one's fellow. [Nar93, page 6]*

Par ailleurs, selon Johnson & al. [JF88, MJ98a], des expériences concrètes sont nécessaires avant de pouvoir atteindre cet objectif [17]. Notre activité industrielle décrite brièvement dans le paragraphe 1.1.3, page 16 ainsi que l'annexe VIII, est une des bases du travail présenté ici.

1.4 Quel système de classes pour outiller la création de langages d'experts ?

1.4.1 Nature du problème central

Nous considérons ici que le problème central de l'outillage de la création de langages d'experts est celui de l'outillage de la co-évolution dynamique de structures et procédures. Cette co-évolution se concrétise sous forme de la spécialisation dynamique de classes (cf. 1.2.1).

Les dernières avancées dans ce domaine (en particulier les travaux de Ralph Johnson et de son équipe, qui constituent le point de départ de cette thèse) ont abouti à créer des systèmes qui traitent seulement en partie le cahier des charges d'outillage de langages d'experts (cf. §1.2, page 17). En somme, ces systèmes proposent des conceptions pour créer des logiciels qui permettent une description dynamique de structures de données et de leurs relations. Autrement dit, ils permettent de définir de nouvelles classes d'objets et de préciser leurs structures et associations.

[16] *Design in practice is neither truly top-down or bottom up, but it is a hierarchical process. At the top of the hierarchy are the broad outlines of the product being designed. At the bottom are those things we already know how to do that we recognize as likely to play a role in the solution to the problem at hand. The designer fixes decisions at a given level by considering the impact they will have on their ancestors, neighbors, and descendents in the design hierarchy. At each level, decision making is influenced by both top-down and bottom-up feedback. The designer attempts to come to an optimal solution to the issues present at each level of the hierarchy before turning his or her attention to the next level. As decisions at each level are fixed, they become fixed constraints around which other constraints are relaxed. Attempts to violate the structure of a system so constructed can be very disruptive. To give an architectural analogy, it is much more appropriate to make changes to the wiring plan of a building after the floor plan is determined, and before the wallboard is up. … It is far more difficult to design a framework that attempts to accommodate future expansion and extension requirements than it is to merely meet the requirements at hand. How does one trade off the simplicity of solving only the current problem with the potential benefits of designing more general components? There is, (to paraphrase Randall Smith), a tension between specificity and generality.* [Foote88]

[17] *Developers need a few concrete examples to factor out the abstractions and metaphors that are suitable for a particular domain.*[MJ98a] *Useful abstractions are usually designed from the bottom up, i.e. they are discovered, not invented. We create new general components by solving specific problems, and then recognizing that our solutions have potentially broader applicability.* [FJ88]

Désormais, comme le précisent Dirk Riehle & al. dans leur publication très récentes (octobre 2001) dans les actes du congrès OOPSLA'2001 [RFBO01], la difficulté majeure est la modélisation dynamique du comportement et son exécution :

> *Our biggest remaining problem is modeling of behavior and execution of the modeled behavior. At the time of writing, we still have to implement a significant amount of code (policies) to add behavior to models. UML's behavior modeling features are not sufficient to completely describe desired behavior and our behavior modeling extensions and implementations have not fully overcome this problem [RFBO01, page 13].*

Ces auteurs précisent que des compagnies comme Project Technology [PT00] ou Kennedy Carter [KC00] ont montré la faisabilité de la modélisation dynamique du comportement sur la base des extensions d'UML[18].

1.4.2 Précisions sur le problème

Cette situation pose plusieurs problèmes par rapport à notre préoccupation d'outiller la création systématique de logiciels "à deux niveaux" et plus précisément les langages d'experts.

Tout d'abord, comme le précise D. Riehle et al., la modélisation dynamique de comportement reste un problème ouvert. La question d'une solution de conception canonique et documentée à ce problème en vue de la création d'outils est encore plus difficile à résoudre. Les travaux actuels, e.g. ceux de l'OMG sur les ASL (*Action Specification Language*) [OMG98], sont encore en phase de spécification et ne s'intéressent ni à la facilité d'apprentissage par des experts, ni à la dimension workflow, ni le lien causal, ni le travail collaboratif, ni à la question centrale de l'outillage de la création systématique de ce type de logiciels.

Par ailleurs, les travaux publiés sur ces ateliers de modélisation décrivent seulement les principes généraux de leur création, mais les formalisent très peu sur le plan conceptuel et architectural. Or, une documentation de l'architecture de ces systèmes, par exemple, à l'aide de schémas de conception [GHJV95, ABW98, BR99][19], pourrait permettre aux autres programmeurs de les réutiliser tout en les adaptant à leurs besoins spécifiques. Un pas en avant pour faciliter l'outillage de création de ce type de logiciels serait la création de frameworks orienté-objets sur la base de telles documentations. Les programmeurs seraient alors munis d'un outil opérationnel, documenté et aussi adaptable[20] à leurs besoins spécifiques.

Troisièmement, à notre connaissance ces travaux ne considèrent pas de façon systématique la facilité d'apprentissage par des experts. De notre point de vue, ce problème rejoint le précédent, c'est-à-dire une manque de documentation formelle des architectures de ces systèmes. En effet, les experts ont leurs propres modèles de programmation dont la mise en œuvre doit être de toute évidence intégrée à la conception elle-même de ces systèmes.

Enfin, les travaux actuels ne considèrent non plus pas l'outillage du travail collaboratif entre les experts et les programmeurs. En effet, les langages de programmation offrent une puissance d'expression importante dont les langages de modélisation, e.g.UML [FS97], sont à ce jour dépourvus. Aussi, les utilisateurs de ces systèmes sont-ils amenés à compléter leurs modèles par l'ajout de code "à la main". Dans le contexte actuel où la gestion de ces deux types d'intervention n'est pas outillée, cela conduit à des problèmes qui sont liés au manque de concordance entre les modèles et les programmes.

---

[18] *However, other companies, for example Project Technology* [PT00] *or Kennedy Carter* [KC00] *have shown that precise behavior modeling is possible with (an extended form of)* UML. *Their system allow the execution of models based purely on modeled rather than implemented behavior. Key to their approach as well as our approach is knowledge about the target runtime architecture.* [RFBO01]

[19] Ce qui n'est, certes, pas facile à réaliser.

[20] Par des techniques classiques de spécialisation de frameworks.

La prise en considération de la "dimension workflow" et du lien causal dans la conception d'un tel outillage ajoute à la complexité de cette entreprise. Il n'y a pas à l'heure actuelle une conception standard dédiée à l'outillage de la création de systèmes intégrant à la fois tous ces aspects.

1.5 Notre démarche

1.5.1 Analyse du problème

Nous estimons que d'une manière générale la clé d'une solution aux problèmes mentionnés dans le paragraphe précédent (§1.4.2) réside dans la recherche d'une structure de représentation de programmes telle qu'elle satisfasse aussi bien les contraintes liées à la mise en œuvre de langages de programmation, que celle des langages d'experts telle que la spécialisation à l'exécution, l'apprentissage par des experts et toutes les autres propriétés énumérées dans le §1.2, page 17.

A titre d'exemple, la représentation d'un programme écrit dans le langage SMALLTALK-80 par des instances de classes comme `Metaclass`, `Class`[21] et `CompiledMethod`, etc. n'est pas suffisante pour l'outillage de la programmation par des experts et la création d'environnement de modélisation[22]. Cette représentation ne fournit pas certaines informations qui sont nécessaires au bon fonctionnement de tels outils, comme par exemple toutes celles qui servent à guider les experts lors de l'élaboration de leur modèles ainsi que lors de leurs exécutions. Il en est de même en ce qui concerne l'automatisation de certaines tâches comme la gestion des liens de dépendances entre les calculs, e.g., dans le cas du langage de feuilles de calcul utilisé dans les tableurs.

La représentation de programmes des langages à objets est, du point de vue de notre étude, encore plus incomplète en ce qui concerne leur exécution. A titre d'exemple, à notre connaissance aucun langage à objets n'offre la possibilité d'adapter localement et facilement (pour une exécution donnée) un programme à un contexte d'exécution particulier. En règle générale, on peut dire qu'aucun langage de programmation n'aborde la question d'exécution de modèles (programmes) dans le cadre d'une relation du type classe/instance. Autrement dit, les exécutions d'un programme ne sont jamais considérées comme des instanciations de celui-ci. De ce fait, aucun langage de programmation n'est en mesure de fournir, à titre d'exemple, une information sur le procédé dont l'exécution a permis de produire une certaine donnée. Or, dans le cas des modèles élaborés par des experts, e.g. le modèle d'un produit d'assurance, il est nécessaire (pour une bonne gestion informatisée des dossiers) de savoir avec quel modèle chaque police d'assurance a été créée[23]. Chacun de ces manques constitue un obstacle à l'outillage de la création des systèmes qui nous intéressent ici.

Dans ce cadre, la question principale posée peut être formulée de la façon suivante : dans quelle mesure un environnement de programmation par objets peut-il permettre une prolongation de la spécialisation à l'exécution, et cela par l'intervention des experts ? Autrement dit, qu'elle est la *structure de représentation* de la définition et de l'exécution de programmes qui assure l'adaptation dynamique de *parties* de programmes ? C'est la question centrale à laquelle notre contribution essaie de répondre[24].

---

[21] Ou plus précisément des instances des sous-classes de la classe `Class`.

[22] Dans leur publication sur quelques simplifications du langage SMALLTALK-80, A. Borning & T. O'Shea mentionnent déjà les insuffisances de la représentation de ce langage des programmes, du point de vue d'outils qui manipulent les programmes comme données: "*Finally, we recommend some changes to the parse tree classes to make it easier to manipulate programs as data. Historically, these classes were developed for use by the compiler. Later, they were used as well by the decompiler, and then by other applications that manipulate programs as data. They should be cleaned up to make them more suitable as general vehicles.*" [BO87, page 9].

[23] A noter que le modèle des produits ainsi défini peut évoluer et doit donc être considéré comme une nouvelle classe de produits.

[24] Cette question se présente sous une forme similaire dans le cas des environnements de programmation par objets qui assurent l'intervention des programmeurs qui utilisent des langages différents. C'est, à titre d'exemple, le cas des systèmes SMALLTALK/X [CG01] et FROST [Frost97] qui proposent la programmation simultanée en SMALLTALK et JAVA.

Nous supposons ici que la prise en considération de la dimension workflow, du travail collaboratif, de l'apprentissage par des experts et du lien causal, fait également partie du cahier des charges de cette représentation recherchée. De manière générale, l'outillage (tel que défini dans le paragraphe 1.3.1, page 21) dédié à l'adaptation que nous proposons dans ce mémoire prend en considération toutes les propriétés des langages d'experts telles qu'elles sont définies dans le paragraphe §1.2.

1.5.2 Thèse

Au vu de l'exposé qui précède, nous énonçons notre thèse de la façon suivante :

> Il est possible d'outiller[25] la création de langages d'experts[26] munis de toutes les propriétés énoncées dans le paragraphe 1.2, page 17.

Pour des raisons pratiques, lors de l'exposé qui suit nous parlerons souvent de la première et la seconde partie de notre thèse. Il convient de préciser ici que nous faisons ainsi allusion aux deux parties que nous distinguons quant à la procédure de validation de cette thèse :

1. Une première validation, qui s'avère partielle, à l'aide des techniques standards (cf. le chapitre III : le framework DYCTALK).
2. Une validation complète, par l'usage simultané des techniques standards et réflexives (cf. chapitres IV et V : les framework MIDYCTALK et MXDYCTALK).

1.5.3 Point de départ

Pour valider notre thèse, nous partons d'une part des recherches de Ralph Johnson et de son école (sur la définition de solutions de conception canonique pour notamment créer des logiciels dynamiquement adaptables) et d'autre part de celles de Bonnie Nardi sur la programmation par des experts. Cela nous permet de nous appuyer sur des solutions éprouvées que nous complétons et que nous assemblons de façon harmonieuse afin d'aboutir à une solution appropriée à notre problème.

Plus précisément, nous empruntons tout d'abord le schéma de conception *Dynamic Object Model* (désormais, en abrégé DOM) [RTJ00] aux travaux de Ralph Johnson et al. sur les modèles objets adaptatifs. Celui-ci constitue un premier effort de formalisation des structures mises en jeu lors de la création de langages d'experts. Les auteurs de ce schéma reconnaissent son manque de mécanisme standard pour associer dynamiquement du comportement aux compléments de classes (forme particulière d'adaptation : cf. la chapitre II, paragraphe 2.2, page 91) :

> Dynamic behavior. *The core of the Dynamic Object Model pattern provides only a structure to which dynamic behavior needs to be hooked up to. However, there is no standardized way to do so (like a programming language for a traditional system). Hence you have to add a whole bunch of further patterns (like Strategy, Chain of Responsibility, Interpreter, or Observer) [RTJ00].*

D'autres chercheurs se sont intéressés à des problèmes semblables mais toutefois assez différents par rapport à nos préoccupation ici. Par exemple, Roel Wuyts a étudié l'intégration de la programmation logique à un langage à objets (SMALLTALK) [Wuy98].

[25] Dans le sens de la notion d'outillage, décrit dans le § 1.3.1, page 21.

[26] Dans le sens de la notion de langage d'experts, décrit dans le § 1.1.2, page 16.

Aussi, nous prenons en considération deux nouveaux éléments :

1. d'une part les principaux schémas de conception du système *Micro-workflow* [MJ98a, MJ99a, MJ99b, MJ99c, Man00, MJ00], également issu de l'école de Ralph Johnson (thèse de Dragos Manolescu, UIUC, octobre 2000). Il s'agit plus particulièrement des schémas *Type Object* [JW97] et *Composite* [GHJV95],
2. d'autre part les analyses de Bonnie A. Nardi sur la programmation par les experts et le langage de feuilles de calcul [Nar93].

et définissons un nouveau système de classes qui remplit pleinement notre cahier des charges. Celui-ci est validé expérimentalement de la façon suivante.

1.5.4 Validation selon trois axes

Sur le plan technique notre démarche est centrée sur la recherche d'un nouveau système de classes pour représenter et interpréter les programmes définis par des experts tout en inscrivant leur intervention dans la continuité des efforts des programmeurs objets et en assurant en contrepartie l'accessibilité des résultats de leurs interventions aux programmeurs. Plus précisément il s'agit :

1. de modéliser l'adaptation par la recherche d'une organisation de concepts dédiée à l'outillage de la programmation lors de l'exécution par des experts ;
2. de déterminer la nature de relations que le système de classes défini en (1) doit entretenir avec la représentation de programmes du langage à objets hôte lui-même, dans le but
   a. d'assurer l'intervention harmonieuse des experts et des programmeurs (ou experts-programmeurs) sur la mise en œuvre d'un même logiciel ;
   b. d'assurer l'application locale de l'adaptation au niveau de chaque classe.

Le modèle obtenu en (1) satisfait tous les éléments caractéristiques des langages d'experts (cf. paragraphe §1.2) à l'exception du travail collaboratif et du choix local du type d'adaptation. L'objet de la seconde partie de nos travaux est d'outiller la mise en œuvre de ces deux derniers aspects.

Pour ce faire, en partant des bases esquissées plus haut, nous construisons trois nouveaux frameworks qui sont mise en œuvre dans les cas suivants :

1. l'intégration en cours au sein de l'atelier multi-agents MOBIDYC de l'INRA (cf. ci-dessous paragraphe 1.5.5.7, page 30)
2. l'exemple classique de comptes en banques, inspiré et adapté à notre étude du papier sur DOM [RTJ00]. Cet exemple est détaillé au fil de notre exposé.
3. démonstration de l'usage possible de nos frameworks dans le but différent de génération (semi-)automatique de nouveaux types d'objets, leur structure et procédures à travers l'exemple des *lignes brisées* qui figure en annexe VIII, page 264.

L'outillage le plus performant que nous proposons est composé d'un système de classes appelé DYCRA-II et d'un framework orienté-objets appelé MXDYCTALK. C'est seulement lui qui est en mesure de satisfaire l'ensemble des propriétés des langages d'experts[27].

Toutefois, avant de construire cette solution complète, nous procédons par deux étapes intermédiaires qui nous conduisent chacune à une solution partielle : le système de classes DYCRA et le frameworks DYCTALK et ensuite le système de classes DYCRA-II et le frameworks MIDYCTALK. Nous nous référons désormais à ces solutions par le nom des frameworks correspondants.

[27] Un cas d'usage intéressant d'un tel outillage peut être illustré par l'exemple de son usage réflexif dans le but de permettre de modifier dynamiquement la définition de ses propres abstractions. Nous avons proposé dans [Raz99] d'appeler ce type d'usage que l'on rencontre par ailleurs, e.g. les éditeurs de menu et d'interface graphique du système VISUALWORKS [Cin01], la *réflexion applicative*. Cela signifie d'appliquer l'usage d'une fonction rendue par un système au développement du système lui-même.

DYCTALK et MIYCTALK sont, par ailleurs, réalisé en SMALLTALK-80 [GR93], à l'aide du système VISUALWOKS NC 5I.3 de la compagnie CINCOM [Cin01]. En ce qui concerne MXDYCTALK, il est implanté à l'aide du du système METACLASSTALK. [Bou99], une extension de la version 2.7 du système SQUEAK [IKMWK97].

Ces trois solutions se partagent les techniques que nous avons mis ici au point en ce qui concerne l'outillage de la définition de procédures par des experts, l'activation des procédures, ainsi que les dimensions workflow et le lien causal.

Elles se distinguent, toutefois, de la façon suivante :

1. DYCTALK : outille toutes les propriétés des langages d'experts, à l'exception du travail collaboratif ni le choix local du type d'adaptation.
2. MIDYCTALK : conserve toute les propriétés de DYCTALK, et y ajoute l'outillage du travail collaboratif. Il abandonne la technique standard de l'ajout dynamique de nouveau types de classes, celle de DOM, au profil d'une technique basée sur l'usage de la réflexion et plus particulièrement des méta-classes (cf. le chapitre IV, page 145).
3. MXDYCTALK : conserve toutes les propriétés de MIDYCTALK (et donc DYCTALK) et y ajoute le choix local du type d'adaptation (cf. le chapitre V, page 179).

Sur le plan technique, ce qui nous a permis de faire évoluer la première solution vers la seconde et ensuite la troisième concerne uniquement les choix relatifs à l'outillage de la définition de nouveaux type d'objets. C'est simplement en jouant sur ce paramètre que nous levons les contraintes posées successivement par la première, mais aussi la seconde solution.

Nous fournissons ci-dessous, dans la section intitulée Organisation de la thèse, page 48, plus d'information au sujet de ces frameworks.

1.5.5 Publications et communications

Nous résumons ici le déroulement temporel de notre travail ainsi que des publications auxquelles il a donné lieu.

1.5.5.1 Caractérisation des phénomènes observés lors de projets industriels

La première étape de la réalisation des travaux présentés dans ce mémoire a été la caractérisation des phénomènes observés lors de nos projets industriels. Pour ce faire nous avons rapproché les techniques mises en œuvre (et les effets produits) aux celles de la méta-modélisation, la méta-programmation, ainsi que les modèles objets adaptatifs.

Nous en avons conclu que les AOMs sont les mieux en mesure de décrire la nature de ces systèmes. Nous avons concrétisé ce premier résultat par la participation à un atelier de travail sur ce thème (*workshop*) qui à eu lieu lors du congrès OOPSLA'99, avec la soumission suivante :

> *Building an End-user-oriented Application Framework by Meta-programming -- A Case Study. Position Paper to OOPSLA'99 Metadata and Dynamic Object-Model Pattern Mining Workshop. Nov. 1999, Denver, USA. [Raz99]*

Cette participation a été suivie d'un séjour à UIUC qui, en accord avec la société MATRA DATAVISION[28], nous a permis de présenter nos réalisations industrielles aux membres du *Software Architecture Group* à UIUC, notamment à Ralph Johnson, Joseph W. Yoder, Don Roberts et Dragos Manolescu (l'inventeur du M*icro-workflow*). Ces rencontres se sont avérées toutes très déterminantes dans la suite de cette thèse. Nous y reviendrons dans les paragraphes suivants.

[28] Nous profitons de cette opportunité pour remercier nos anciens collègues du département contrôle 3D de la société MATRA DATAVISION pour avoir notamment mis à notre disposition lors de ces rencontres une version de démonstration du logiciel PRELUDE INSPECTION.

1.5.5.2 Rapprochement du système CALIBRES et les AOMs

La seconde étape de cet effort de caractérisation a été le rapprochement de l'architecture du système CALIBRES avec les résultats des travaux sur les AOMs. La conclusion de ces travaux a été une publication à la conférence OCM'2000 (Nantes) :

> Active Object-Models et Lignes de Produits -- Application à la création des logiciels de Métrologie. La conférence OCM'2000 : Objets, Composants et Modèles. 18 - May 2000, Nantes, France. Actes de l'OCM, pages 130-144. [Raz00a]

Cet article montre l'utilité des schémas de conception décrivant les AOMs pour documenter de telles architectures. Mais, l'apport majeur de notre communication se situe dans l'identification de ces schémas au sein de l'architecture du système CALIBRES, ce qui en raison de la nature même des schémas de conception (solutions à des problèmes récurrents de conception) ouvre la voie à une généralisation de la démarche de conception de CALIBRES et à son application à d'autres domaines.

1.5.5.3 Constat de la nécessité d'un couplage

Comme nous avions conclu dans notre publication OCM, les mécanismes utilisés dans CALIBRES pour la modélisation dynamique des procédures d'étalonnage appliquées sur des types de calibres qui sont définis dynamiquement étaient différents de ceux documentés notamment par le schéma de conception DOM, basés sur le schéma de conception *Strategy*.

Ce constat, et la rencontre mentionnée avec Manolescu à UIUC nous ont conduit à une étude plus précise du *Micro-workflow* et à son rapprochement avec les mécanismes utilisés dans MARLENE, CALIBRES et PRELUDE INSPECTION (initialement conçus et réalisés par Philippe Krief lors du projet MARLENE). Cette étude a conduit au choix du *Micro-worklow* comme support pour l'outillage de la définition dynamique de procédures.

Toutefois, cette étude a mis en évidence qu'une simple juxtaposition des deux modèles DOM et *Micro-workflow* ne suffisait pas à résoudre le problème de l'outillage de la co-évolution dynamique de structures et de procédures. Aussi, nous avons proposé la mise en œuvre d'un couplage entre ces deux systèmes. Cette proposition est formulée pour la première fois dans les deux soumissions suivantes :

> *Coupling The Core of Active Object-Models and Micro Workflow -- Foundations of a Framework for Developing End User Programming Environments. Position paper to ECOOP '2000 workshop on Metadata and Active Object-Model Pattern Mining, June 2000, Cannes, France. [Raz00b]*

> *Type Cube: Foundation for an Architectural Style aimed at Building Adaptive and Flow-Independent Software. OOPSLA'2000 First Workshop on Best-practices in Business Rule Design and Implementation. October 2000 at Minneapolis, MN, USA. [Raz00c]*

1.5.5.4 Co-organisation d'un workshop ECOOP et d'un poster à OOPSLA

Notre collaboration sur ce sujet avec l'équipe de UIUC à conduit à la co-organisation avec Joseph W. Yoder d'un *workshop* à ECOOP'2001, suivi par une session de *poster* tenue à OOPSLA'2000 sur nos travaux communs. Cet effort s'est concrétisé dans les deux publications suivantes :

> *Metadata and Adaptive Object-Models (co-author with Joseph W. Yoder). Published in the ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. no. 1964; pages 104-113, Springer Verlag 2000. [YR00a]*

Adaptive Object-Models (co-author with Joseph W. Yoder). OOPSLA'2000 Poster Session. OOPSLA'2000 Companion, pages 81-82. October 2000 at Minneapolis, MN, USA. [YR00b]

Cette collaboration va se poursuivre notamment par l'organisation d'une nouvelle session de workshop *Best-practices in Business Rule Design and Implementation* à OOPSLA'2001 en octobre à Tempa, avec la soumission :

Concepts and Tools to Support Building Expert-Programmable Software. Submission to the third Workshop on Best-practices in Business Rule Design and Implementation. OOPSLA'2001, October 2001, Tampa, FL, USA. 14-18 October 2001 at Tampa, Florida, USA. [Raz01c]

1.5.5.5 La "dimension expert"

L'étape suivante du travail de thèse a été la recherche d'une solution au problème d'outillage de la mise en œuvre par des experts de l'adaptation. Suite à une proposition de R. Johnson, nous avons entamé une étude sur la base des travaux de Bonnie A. Nardi. Celle-ci a conduit à des résultats qui seront présentés lors d'une session poster à OOPSLA'2001.

Reusable Designs for Building Dynamically Programmable and Workflow-enabled Object-Oriented Software. In Companion Papers of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). ACM Press, 2001, pages unknown at this moment. 14-18 October 2001 at Tampa, Florida, USA. [Raz01a]

L'ensemble des travaux réalisés jusqu'alors constitue la première partie de notre thèse, c'est à dire l'outillage de l'adaptation et sa mise en œuvre par les experts. La deuxième partie est consacrée à l'usage de la réflexion pour améliorer cet outillage.

1.5.5.6 Usage des méta-classes pour améliorer l'outillage de langages d'experts

L'étape suivante a été l'étude de l'usage des méta-classes pour améliorer l'outillage des langages d'experts. Elle s'est déroulée en deux temps.

La première étape a consisté à utiliser les méta-classes standard (selon le système SMALLTALK-80) afin de créer un outillage assurant également le travail collaboratif.

L'étape la plus importante a toutefois été l'usage du système METACLASSTALK, grâce auquel nous avons pu concrétiser nos idées sur l'intérêt des méta-classes explicites pour le choix du type d'adaptabilité comme une propriété locale à chaque adaptation.

La réalisation effective de cette étude doit beaucoup à la collaboration de N. Bouraqadi qui a bien voulu finaliser, lors de ses vacances de nouvel an 2001, le portage de ce système, dont il est inventeur, en SQUEAK version 2.7.

Ce travail a donné lieu au framework MXDYCTALK. Il a été l'objet de la communication suivante :

Why Object-Oriented Languages Should Support Building Tools for Adaptive Object-Models? Submission to ESUG'2001 1st Doctoral Symposium. Tuesday, August 28th - Friday, August 31st, Essen, Germany. [Raz01b]

1.5.5.7 Suite des travaux

Hormis les collaborations avec l'équipe de Ralph Johnson et des chercheurs directement concernés par les travaux sur les modèles objets adaptatifs, nous avons une collaboration en cours avec des chercheurs à l'INRA-Avignon (Vincent Ginot [Gin90]) dans le but d'appliquer nos techniques à la création de simulateurs d'écosystèmes. Il s'agit de l'atelier de simulation multi-agents [Fer95] MOBIDYC dédié à la création et la simulation de MOdèles Basés sur les Individus pour la DYnamique des Communautés [GLP98]. Les experts utilisateurs de *MOBIDYC* sont des chercheurs (biométriciens) en dynamique des populations.

L'annexe VII, page 278, ainsi que les pages Web se trouvant à l'URL *http://www-poleia.lip6.fr/~razavi/Myctalk/* , fournissent plus de précisons sur l'état d'avancement de ce projet.

2 Illustration : exemple d'adaptation de comptes bancaires

Nous développons ici un exemple particulièrement simple de langage d'experts. Dans la mesure où l'exposé qui suit semble inhabituellement (mais en raisons de la complexité du sujet, inévitablement) long, nous esquissons d'abord notre démarche avant d'énoncer l'exemple et enfin, sur cette base, illustrer le fonctionnement de notre outillage.

2.1 Démarche

Dans un premier temps (à partir du §2.3, page 32), nous exposons le fonctionnement "externe" du système. Nous énonçons à cette occasion d'abord le scénario applicatif (§2.3.1, page 32) qui constitue, en effet, une ébauche du cahier des charges de ce langage.

Cet exposé prend en considération toutes les propriété des langages d'experts :

1. la définition de nouveau type d'objets (§2.3.2, page 33), leur structure (§2.3.3, page 33), procédures (§2.3.4, page 33) ainsi que leur instanciation (§2.3.6, page 38).
2. le *refactoring* des adaptations (§2.3.7, page 39) ainsi que le choix local du type d'adaptation (§2.3.8, page 40).

Ensuite (à partir du §2.4, page 41), nous expliquons sommairement comment ce fonctionnement est obtenu à partir de nos trois frameworks, en comparant, le cas échéant, leurs modes d'action. Il s'agit plus particulièrement d'illustrer :

1. les trois techniques dédiées à l'ajout de nouveaux types d'objets (sans, à ce niveau, se préoccuper de la définition de leur structure et procédures) (cf. le paragraphe 2.4.1, page 41) ;

ainsi que la technique dédiée :

2. à la définition de structures (cf. le paragraphe 2.4.2, page 44) ;
3. à la préparation de la composition (cf. le paragraphe 2.4.3, page 44) ;
4. à la composition de procédures par les experts (y compris la dimension workflow) (cf. le paragraphe 2.4.4, page 45) ;
5. à la composition dynamique de services dynamiquement définis (cf. le paragraphe 2.4.5, page 46) ;
6. à l'activation de procédures (cf. le paragraphe 2.4.6, page 46) ;
7. au travail collaboratif, notamment le *refactoring* des adaptations (cf. le paragraphe 2.4.7, page 47) ;
8. au choix local du type d'adaptation (cf. le paragraphe 2.4.8, page 47).

On trouvera plus de détails sur chacun de ces sujets dans le corps de la thèse.

2.2 Précisons sur nos contributions

Nous tenons à préciser à cette occasion que nos contributions majeures se situent au niveau des alinéa (1), (3), (4), (5) et (6) ci-dessus. Les travaux présentés en (1) constituent, par ailleurs, les bases des résultats annoncés en (7) et (8).

Quant à la dimension workflow, nous nous appuyons entièrement sur les travaux de Dragos Manolescu sur le Micro-workflow.

Il importe de mentionner ici, que notre effort de conserver la compatibilité de notre outillage avec celui de Micro-workflow conduit, en contre partie, à améliorer ce framework en le munissant des mécanismes nécessaires à la définition de procédés par des experts et cela, de plus, dans le contexte des modèles objets adaptatifs (AOMs).

Il s'agit là d'une contribution importante en soi dans la mesure où le Micro-workflow constitue l'état de l'art en matière de *l'outillage* de la définition explicite des collaborations entre les objets sous forme de procédés workflow (cf. le paragraphe 3.11, page 171, du chapitre IV).

En ce qui concerne le *refactoring* des adaptations, outre le fait d'avoir évoqué sa nécessité, à notre connaissance pour la première fois, nous en étudions ici uniquement une condition nécessaire, à savoir la possibilité pour des programmeurs d'éditer les adaptations par les mêmes outils dont ils se servent pour éditer les classes (le flâneur).

Il importe de préciser que cet effort va tout à fait dans le sens des travaux réalisés depuis plusieurs années par John Brandt et Don Roberts [FBBOR99]. Ces derniers ont mis au point le *Refactoring Browser* qui se présente aujourd'hui des perspectives intéressantes notamment en ce qui concerne l'outillage de la mise en ouvre de l'*eXtreme Programming* [Bec99, BF00].

Quant à la gestion du lien causal, nous appliquons à notre outillage une technique de conception inspirée de celle du Micro-workflow, laquelle est basée sur le schéma *Type Object* [JW97]. Cela nous conduit à un outillage qui a, par sa conception, le potentiel d'assurer le lien causal. Toutefois, nous n'étudions pas cet aspect ici de façon systématique sur toutes ses dimensions.

2.3 L'exemple d'adaptation par des experts : Compte-Service et PEP

Pour mieux fixer les idées nous nous appuyons ici sur l'exemple classique de comptes bancaires.

Cet exemple est initialement inspiré des travaux de Dirk Riehle sur les systèmes bancaires (cf. §1.1.1, page 15) qui se reflètent également dans l'article DOM [RTJ00]. Nous l'avons, toutefois, adapté à notre étude qui s'intéresse aux dimensions nouvelles comme la co-évolution dynamique de procédures et structures, le travail collaboratif et le choix local du type d'adaptation.

2.3.1 Le scénario applicatif

Supposons qu'un langage d'experts dédié à la gestion de comptes bancaires soit livré aux utilisateurs finaux et qu'il soit en cours de fonctionnement. Il gère actuellement des comptes chèques et des comptes d'épargne. Les utilisateurs finaux (e.g. guichetiers) peuvent donc proposer des comptes de ce type à leurs clients.

En matière de l'adaptation par des experts, nous considérons, à titre d'exemple, l'ajout dynamique de deux nouveaux types de compte : le *Plan d'Epargne Populaire* (désormais, en abrégé PEP) et le *Compte-Service*[29].

Le PEP est un type particulier du compte d'épargne, bien connu de tous.

Quant au Compte-Service, il est attaché à un compte-chèque et sert à contractualiser les relations entre le client titulaire du compte-chèque et sa banque en matière de quelques règles de fonctionnement de ce compte. Globalement, un client s'engage à payer une certaine somme à sa banque pour en contre partie bénéficier des quelques services comme une carte de paiement, une protection des moyens de paiement (par un contrat d'assurance), une autorisation de découvert à taux préférentiel, etc.

Par ailleurs, afin de couvrir les besoins des différentes catégories de clients, plusieurs types de Compte-Service sont prévus : les *Comptes-Service Equilibre*, *Confort* et *Privilège*.

Une description plus exhaustive du Compte-Service est fournie en annexe II, page 256.

Nous nous servons ici des Comptes-Service dans un premier temps pour illustrer notre outillage de base de l'adaptation, c'est à dire, la possibilité d'ajouter dynamiquement un nouveau concept, définir sa structure et ses procédures. Ensuite il sera le support de notre exposé sur le *refactoring* des adaptations par des programmeurs.

Nous faisons ensuite appel au compte PEP afin d'illustrer l'intérêt du choix du type d'adaptation à travers une application de l'adaptation prototypique.

[29] Exemple tiré du site Web du Crédit Agricole de Vendée http://www.ca-vendee.fr/libre.asp?id=10495.

2.3.2 Ajouter de nouvelles adaptations (par des experts)

La première action de l'expert consiste à ajouter au système de nouvelles adaptations, ici les trois types de Compte-Service. Pour ce faire, le langage d'experts lui propose simplement une action comme par exemple *Introduire un Nouveau Concept*. Le déclenchement de cette action fait en principe apparaître une boîte de dialogue qui demande la saisie du nom du concept concerné, par exemple le Compte-Service Equilibre.

2.3.3 Définir la structure (par des experts)

L'étape suivante consiste à définir dynamiquement la structure de nouvelles adaptations, ici toujours les Comptes-Service. Pour l'expert il s'agit de définir les attributs caractéristiques des concepts qu'il vient d'ajouter. Chaque attribut est caractérisé au moins par son nom et son type.

Le nom d'un attribut correspond à une chaîne de caractères qui désigne clairement la caractéristique qu'il modélise, e.g., `Type de carte bancaire associé`[30]. Une appellation informatique du genre `typeDeCarteBancaire` ne convient pas ici[31].

En ce qui concerne le type de l'attribut, le but est de permettre à l'expert d'associer un type à chaque attribut qu'il définit. Cette information joue un rôle fonctionnel important au niveau de l'usage par des utilisateurs du système ainsi créé. Par exemple, lors du choix de la valeur du type de carte bancaire le guichetier se voit proposer la liste des trois valeurs possibles (cette liste de trois valeurs est aussi définie par l'expert). Il est ainsi non seulement guidé mais aussi contrôlé dans le choix de la valeur des attributs.

2.3.4 Descriptifs de service

L'étape suivante de la spécialisation dynamique est la définition de procédures. Toutefois, dans la mesure où cette fonctionnalité s'appuie pleinement sur la notion de *descriptif de service*, il convient de la définir en premier lieu.

2.3.4.1 Rôles

Les descriptifs de service constituent les granules de base de la composition dynamique de procédures. Le rôle premier des descriptifs de service est "d'habiller" les différentes formes d'expression d'un service ou calcul (cf. ci-dessous le paragraphe 2.3.4.2, page 34).

Par exemple, au lieu de proposer à l'expert la fonction primitive `affecterSolde:` qui prend en entrée un argument et l'affecte à la variable d'instance solde du compte concerné, l'expert se voit proposer le descriptif de service *Affecter Solde* avec des explications sur le rôle de la fonction, la nature de ses arguments, son résultat, etc.

Ensuite, les descriptifs de service sont utilisés lors de la composition de procédures. En effet, une procédure est définie comme un ensemble d'instances de descriptifs de service. Une instance d'un descriptif de service correspond à un cas d'usage de celui-ci au sein de la définition d'une procédure. Cela consiste à définir les arguments en entrée ainsi que l'emplacement de l'instance lui-même dans la feuille de composition.

Nous proposons d'appeler ce procédé la *composition d'instances de descriptifs de service* , ou en abrégé la *composition de procédures*. En effet, par cet acte un lien de dépendance est établi entre les différentes instances de descriptifs, ce qui matérialise la composition. Des exemples sont fournis dans la sous-section suivante (§2.3.5, page 36).

---

[30] Tout au long de ce document le nom des attributs et les traitements sera en police `courrier`. Pour permettre de distinguer les attributs des traitements, le nom de ces derniers est suivi par ().

[31] Un programmeur est, malheureusement encore, obligé par son compilateur de choisir des noms de ce genre !

Ce sont également les descriptifs de service qui prennent en charge la responsabilité de porter les informations nécessaires au guidage des experts lors de la composition. Cela comprend des informations sur le nombre d'arguments nécessaires et leur type, ainsi que le type du résultat.

Le descriptif de service intervient également lors de l'activation des procédures. En effet, la stratégie d'activation de chaque instance de descriptif de service est décidée par le descriptif lui-même (cf. le paragraphe 2.2, page 59, ainsi que le paragraphe 3.3, page 67).

2.3.4.2 Différents types

Notre étude a permis de mettre en évidence la nécessité de distinguer plusieurs types de descriptifs de service. Le Tableau 1 ci-dessous récapitule les différents types que nous avons rencontrés. La section 3, page 158 du chapitre IV en fournit des exemples.

Type de descriptif de service	Sémantique opérationnelle
Méthode	Activation d'une méthode.
Méthode statique	Activation d'une méthode de classe (création d'instances).
Primitive externe	Activation d'une primitive externe.
Getter	Accès en lecture à la valeur courante d'un attribut.
Setter	Accès en écriture à un attribut pour affecter sa valeur courante.
Procédure	Activation de procédures définies à l'exécution (micro-compositions ou micro-procédés).
Argument	Recherche de la valeur dans le contexte courant d'exécution.
Component Factory	Instanciation d'adaptations.

Tableau 1 : Les différents types de descriptifs de services.

Voici une description succincte de chacun de ces types de descriptif de service :

Type *méthode*

Le rôle des descriptifs de service du type *méthode* est de rendre possible l'expression par des experts de l'activation des méthodes (des objets métier). De tels descriptifs servent donc à "habiller" les méthodes du langage d'experts qui sont jugées utiles lors de la composition dynamique de procédures.

Type *méthode statique*

Le rôle des descriptifs de service du type *méthode statique* est de permettre l'expression par des experts de la création d'objets métier. De tels descriptifs servent donc à "habiller" les constructeurs du langage d'experts qui sont jugées utiles lors de la composition afin de décrire la création d'instances.

Type *primitive externe*

Le rôle des descriptifs de service du type *primitive externe* est de permettre l'expression par des experts de l'activation des méthodes qui se trouvent en dehors de l'espace d'adressage du langage d'expert. C'est le cas, à titre d'exemple, des méthodes invoquées par des mécanismes d'interopérabilité comme CORBA [OHE97] ou encore des procédures stockées dans des bibliothèques comme les DLL sous Windows.

Type *getter* et *setter*

Le rôle des descriptifs de service du type *getter* et *setter* est de permettre l'expression par des experts des accès en lecture et en écriture aux attributs dynamiques des instances d'adaptations. Ces descriptifs sont auto-générés par le langage d'experts (cf. le paragraphe 3.4, page 163 du chapitre IV).

Type *procédure*

Le rôle des descriptifs de service du type *procédure* est de permettre l'expression par des experts de l'activation d'autres procédures. Nous proposons d'appeler *macro-procédure*, une procédure utilisée lors de la définition d'une autre (cf. le paragraphe 2.4, page 60 du chapitre I).

C'est grâce à cette possibilité que nos langages d'experts offrent la possibilité de composer dynamiquement des services définis eux-mêmes dynamiquement.

Type *argument*

Le rôle des descriptifs de service du type *argument* est de permettre d'homogénéiser la définition des procédures avec arguments par rapport à celles qui sont sans arguments. En effet, un argument reçu par une procédure est considéré comme une instance de descriptif de service. Il occupe donc une place dans la matrice de composition et peut être utilisé comme argument lors de la définition d'autres instances de descriptifs de service (cf. le paragraphe 2.4.5, page 46 ci-dessous, ainsi que le paragraphe 2.4, page 60 du chapitre I).

Type *component factory*

Le rôle des descriptifs de service du type *component factory* est de permettre l'expression par des experts de la création d'instances des adaptations. Ces descriptifs habillent donc les constructeurs dédiées qui savent notamment tenir compte de la technique particulière utilisée dans la définition de la structure des adaptations (les descriptifs d'attribut).

2.3.4.3 Référentiel de descriptifs de service et son évolution dynamique

Un langage d'experts gère, par ailleurs, un *référentiel* de descriptifs de service. Le rôle de celui-ci consiste à contenir l'ensemble des services accessibles aux experts lors de la définition de procédures.

Lors de sa livraison ce référentiel contient un ensemble de descriptifs qui sont du type *méthode, méthode statique* et *primitive externe* (cf. le paragraphe 2.3, page 60, chapitre I). Il contient également un descriptif du type *argument* pour permettre la définition d'arguments.

Ce référentiel peut également être complété lors de l'exécution et cela au moins de trois façons :

1. par la génération automatique de nouveaux descriptifs qui sont du type *getter* ou *setter* (cf. le paragraphe 0, page 131 du chapitre III).
2. par "l'habillage" de macro-procédures (cf. page 46 ainsi que le paragraphe 2.4, page 60 du chapitre I) qui produit des descriptifs du type *procédure* ; ou encore
3. par la génération de nouveaux descriptifs qui "habillent" des bibliothèques externes ou des classes/méthodes chargées lors de l'exécution (quand le langage le permet), qui produit des descriptifs de service du type méthode, méthode statique et primitive externe. Cette possibilité est brièvement documentée dans le paragraphe 3.4.2, page 72 du chapitre I.

Ces possibilités d'évolution dynamique constituent une caractéristique fondamentale de notre outillage. En effet, ces descriptifs qui sont définis à l'exécution peuvent également être utilisés lors de la définition de nouvelles procédures par des experts : c'est *la composition dynamique de services définis dynamiquement* (cf. ci-dessous, le paragraphe 2.4.5, page 46).

Il est aussi important de noter qu'en présence d'algorithmes adéquats, il est possible de faire évoluer ce référentiel automatiquement, comme c'est le cas déjà pour les descriptifs du type *getter* et *setter*.

2.3.5 Définir des procédures (par des experts)

A présent l'expert doit définir dynamiquement les procédures attachées aux nouvelles adaptations. Pour ce faire, il procède simplement par l'instanciation de *descriptifs de services*.

2.3.5.1 Procédure Traiter les agios du jour()

A titre d'exemple, considérons de plus près la définition de la procédure `Traiter les agios du jour()`. Le rôle de cette procédure est de calculer les agios journaliers et de les cumuler sur la valeur courante de l'attribut `Cumul d'agios` du compte-service concerné. Pour ce faire, il utilise la valeur courante des attributs solde du `compte chèque associé`, `montant de découvert forfaitaire` et `taux préférentiel de calcul d'agios`. Il appelle également deux autres procédures, `Calculer les agios journaliers()` et `Cumuler les agios du jour()`.

L'algorithme de calcul est le suivant : si le solde du `compte chèque associé` dépasse le `montant de découvert forfaitaire` alors `Calculer les agios journaliers()`à l'aide du `taux préférentiel de calcul d'agios` et ajouter le montant résultant au cumul des agios de la période à l'aide de la procédure `Cumuler les agios du jour()`. Cet algorithme est mis en œuvre par une méthode statique qui est habillée par le descriptif de service appelé `Calculer les agios journaliers`.

Aussi, la définition dynamique de cette procédure nécessite l'existence des descriptifs de service récapitulés par le Tableau 2 ci-dessous.

Nom du descriptif de service	Type du descriptif
`Obtenir Compte-chèque associé`	*Getter*
`Obtenir Solde`	Méthode
`Obtenir Taux préférentiel de calcul d'agios`	*Getter*
`Obtenir Montant de découvert forfaitaire`	*Getter*
`Calculer les agios journaliers`	Méthode statique
`Cumuler les agios du jour`	Micro- composition

Tableau 2 : Descriptifs utilisés pour définir la procédure `Traiter les agios du jour()`.

Les trois descriptifs de service `Obtenir Compte-chèque associé`, `Obtenir Taux préférentiel de calcul d'agios` et `Obtenir Montant de découvert forfaitaire` sont auto-généré lors de la définition des attributs concernés. Ce sont donc des descriptifs de service du type *Getter*. Ils permettent d'accéder aux valeurs courantes de chacun des attributs concernés. Le descriptif de service `Obtenir Solde` est livré avec le langage d'experts et permet d'accéder au solde des comptes, ici le compte-chèque associé. Le descriptif de service `Calculer les agios journaliers` habille une méthode statique. Le descriptif `Cumuler les agios du jour` habille la procédure dynamiquement définie du même nom (cf. ci-dessous, le paragraphe 2.3.5.2).

Le Tableau 3 ci-dessous présente enfin la définition de la procédure `Traiter les agios du jour()` sous forme d'un tableau. Une interface graphique semblable à celle des tableurs[32] peut ici parfaitement convenir à l'expert pour réaliser cette composition[33]. La première colonne de ce tableau comporte le nom par défaut affecté par notre outillage à chacune des instanciations de descriptifs de service utilisés. La seconde colonne comporte le nom du descriptif de service instancié. La troisième colonne énumère la liste des arguments utilisés.

[32] Les tableurs utilisent le même modèle de langage, c'est à dire celui des feuilles de calcul.

[33] Nous tenons à préciser ici que nos framework n'offrent pas une telle interface dont le développement coûteux ne présente pas d'intérêt particulier pour les travaux présentés ici.

Il est important de noter que le descriptif du type *argument* (`Le compte`) est d'un genre particulier et est, par défaut, proposé aux experts. Il en est de même en ce qui concerne le descriptif `Le montant` qui figure dans le Tableau 5 ainsi que la Figure 2.

Nom	Nom du descriptif de service ou d'argument	Utilise	Est utilisé par
C11	`Le Compte` (*argument*)	-	C12, C24, C25, C32, C33
C12	`Obtenir Compte-chèque associé`	C11	C23
C23	`Obtenir Solde`	C12	C32
C24	`Obtenir Taux préférentiel de calcul d'agios`	C11	C32
C25	`Obtenir Montant de découvert forfaitaire`	C11	C32
C32	`Calculer les agios journaliers`	C11, C23, C24, C25	C33
C33	`Cumuler les agios du jour`	C11, C32	-

Tableau 3 : Définition de la procédure `Traiter les agios du jour()` (tableau).

La Figure 1 ci-dessous offre une visualisation différente de la même définition. Chaque rectangle représente une instance de descriptif de service (la définition d'un appel de service), avec le cas échéant des arguments en entrée et un résultat en sortie. Pour simplifier les diagrammes, nous n'avons pas visualisé les flèches qui devaient relier l'argument `me` (qui représente l'instance courante du type d'objet lors de l'exécution de cette procédure) aux descriptifs de service qui l'utilise comme argument en entrée. Cette flèche est remplacée par l'apparition de la chaîne de caractères `me` sur une des entrées de cette instance.

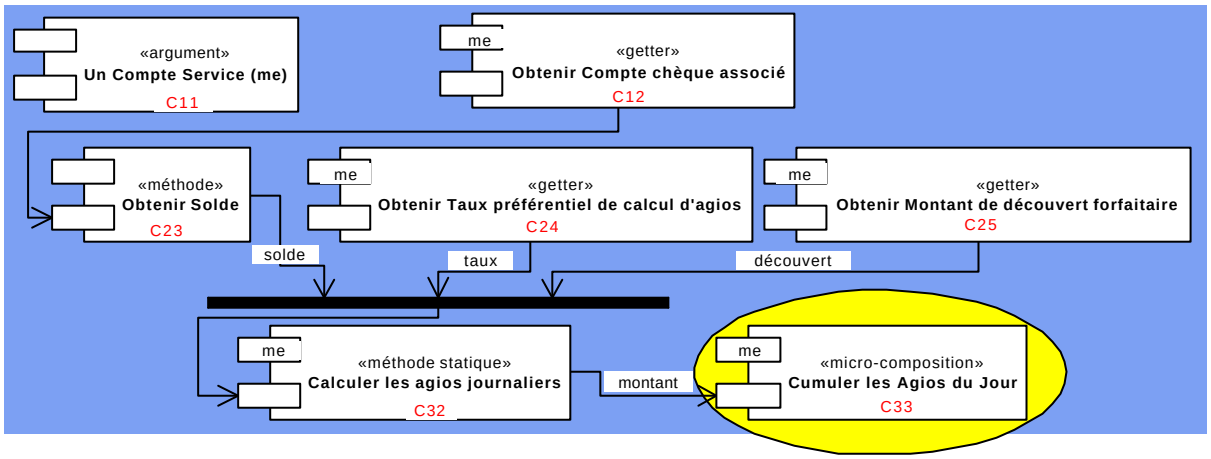


Figure 1 : Définition de la procédure `Traiter les agios du jour()` (dessin).

2.3.5.2 Procédure Cumuler les agios du jour()

La procédure `Traiter les agios du jour ()` décrite ci-dessus fait appel à la (sous-) procédure `Cumuler les agios du jour()` qui elle-même est définie à l'exécution. Le rôle de cette procédure est d'ajouter le montant des intérêts journaliers à la valeur courante de l'attribut `Cumul d'agios`. Ce dernier conserve le cumul du montant des intérêts journaliers sur une période de référence (*a priori* d'un mois) avant son imputation à la fin de la période.

Nom du descriptif de service	Type du descriptif
`Obtenir Cumul d'agios`	*Getter*
`Affecter Cumul d'agios`	*Setter*
`Additionner deux nombres`	Primitive Externe

Tableau 4 : Descriptifs utilisés pour définir la procédure `Cumuler les agios du jour()`.

Le Tableau 5 ci-dessus récapitule les descriptifs de service utilisés pour la définition de cette procédure. Le descriptif de service `Obtenir Cumul d'agios` est du type *getter* et sert à accéder en lecture à l'attribut `Cumul d'agios`. Le descriptif de service `Affecter cumul d'agios` est du type *setter* et sert à accéder en écriture à l'attribut `Cumul d'agios`. Le descriptif de service `Additionner deux nombres` est du type primitive externe est sert à réaliser la somme des deux valeurs qui lui sont passées en argument.

Nom [34]	**Nom du descriptif de service ou d'argument**	**Utilise**	**Est utilisé par**
C11	`Le Compte` (*argument*)	-	C21
C12	`Le montant` (*argument*)	-	C31
C21	`Obtenir Cumul d'agios`	C11	C31
C31	`Additionner deux nombres`	C12, C21	C41
C41	`Affecter Cumul d'agios`	C31	-

Tableau 5 : Définition de la procédure `Cumuler les agios du jour ()` (tableau).

Le Tableau 5 ci-dessous présente la définition de cette seconde procédure (une macro-procédure). Celle-ci définit deux arguments `Le compte` et `Le montant`. Ces deux arguments sont utilisés dans les calculs qui suivent. D'abord, il y a un accès en lecture à la valeur courante de l'attribut `Cumul d'agios` (C21). Ensuite, la valeur résultante est additionnée à la valeur du second argument à l'aide de la primitive externe `Additionner deux nombres` (C31). La valeur résultante est ensuite affectée à l'attribut `Cumul d'agios` à l'aide de la fonction d'accès en écriture qui porte le même nom (C41). La Figure 2 ci-dessous offre une visualisation différente de cette définition.

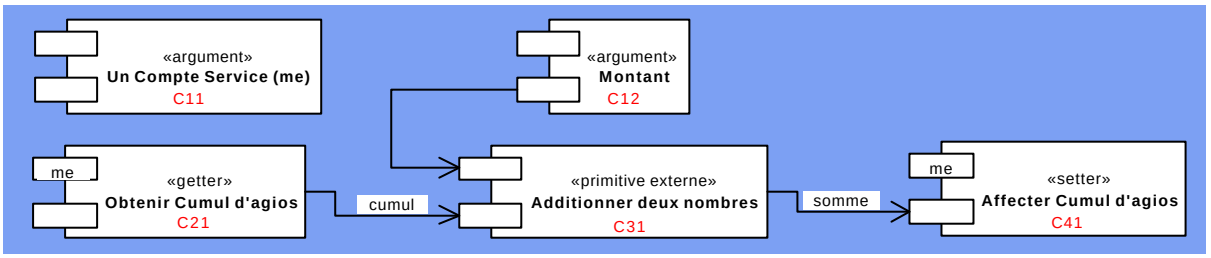


Figure 2 : Définition de la procédure `Cumuler les agios du jour ()` (dessin).

A noter que le modèle présenté ci-dessus met en œuvre une gestion implicite et dynamique des relations entre les objets. Cette relation s'établit implicitement lorsque l'expert utilise le résultat d'un calcul (représenté par une cellule de la matrice) dans la définition d'un autre calcul. Dans l'exemple de la Figure 1, une relation s'est établi entre `Montant` et `Taux`, car le `Taux` est utilisé dans le calcul du `Montant`.

2.3.6 Instancier les adaptations et activer les procédures

Les interventions décrites ci-dessus donnent lieu à l'ajout de nouvelles adaptations, ici des Comptes-Service. Les utilisateurs finaux peuvent alors les instancier et leur appliquer les procédures associées.

Par exemple ici, les utilisateurs peuvent à présent proposer à leurs clients trois nouveaux types de comptes prévus par le cahier des charges : *Compte-Service Equilibre*, *Confort* et *Privilège*. Créer un compte-service consiste à renseigner les informations qui le caractérisent, comme par exemple le `compte-chèque associé`, le `type de carte bancaire associé` et le `taux préférentiel de calcul d'agios`.

Une procédure de gestion applicable à chaque instance de ces comptes est la procédure `Traiter les agios du jour()` définie dynamiquement par l'expert au paragraphe 2.3.5.1 ci-dessus.

[34] Nom par défaut.

2.3.7 Refactoring et édition des adaptations

Une autre fonction considérée par nos langages d'experts est le *refactoring* des adaptations. En effet, les experts ne sont pas des programmeurs objets et ne raisonnent pas en termes de hiérarchies de classes. Aussi, il appartient aux programmeurs de veiller à l'intégrité et la solidité du langage d'experts, malgré les interventions des non-spécialistes en la matière de la programmation par objets.

Nous estimons que le principe de *refactoring* énoncé par Ralph Johnson et al. dans le cas des programmeurs, peut s'appliquer ici parfaitement au cas du travail collaboratif et des langages d'experts. L'usage des algorithmes de restructuration automatique de hiérarchies de classes comme celui de *Ares* [DDHL96, HLD99] semble également être ici pertinent.

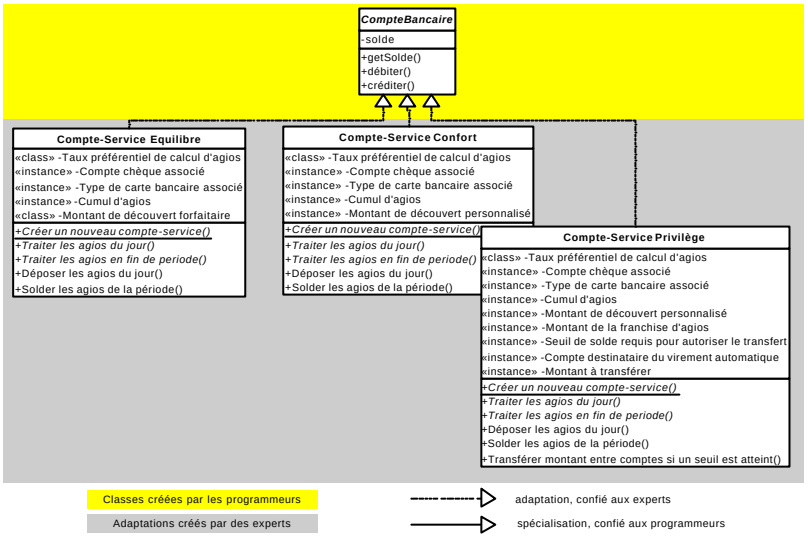


Figure 3 : Exemple d'évolution dynamique d'un modèle objet.

Notre objectif, à plus long terme, est d'assurer qu'un modèle objet adapté dynamiquement par des experts, par exemple celui de la Figure 3 ci-dessus, qui correspond au cas de notre exemple en cours d'adaptation de comptes, puissent être transformé à l'aide d'outils appropriés, en modèle de la Figure 4 ci-dessous. Pour l'heure nous assurons ici la possibilité pour les programmeurs d'éditer les adaptations au même titre que les classes et cela à l'aide de leurs outils habituels (e.g. le flâneur).

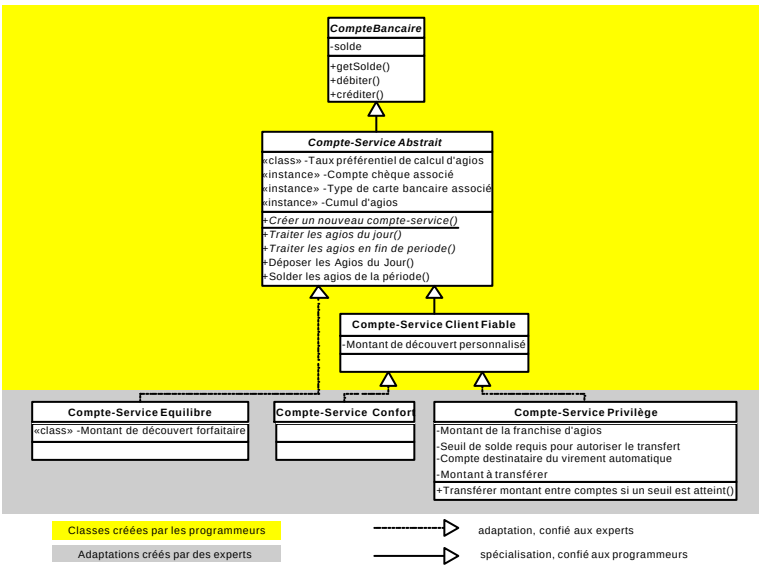


Figure 4 : Modèle objet de la Figure 3 après le *refactoring* des adaptations.

Le modèle de la Figure 3 comporte des redondances (e.g. l'attribut `Cumul d'agios` ou la procédure `Traiter les agios du jours ()`). Or, ce n'est plus le cas du modèle de la Figure 4.

Nous fournissons en annexe II, page 258 une analyse du cahier des charges des Comptes-Service qui explique mieux le pourquoi du modèle de la Figure 4.

2.3.8 Choix local du type d'adaptation

La dernière propriété des langages d'experts que nous devons outiller correspond à celle du choix local du type d'adaptation.

Comme l'illustre la Figure 5 ci-dessous, le but premier du choix local du type d'adaptation est de rendre potentiellement chaque classe d'un système adaptable, tout en lui assurant, de plus, le type d'adaptation adéquat (cf. la page 150, §1.4, chapitre IV pour une description des différents types d'adaptation). Par ailleurs, ce choix doit idéalement pouvoir être fait lors de l'exécution.

L'exemple de la Figure 5 veut montrer que les classes `ObjectBancaire, ObjectCompte` et `CompteBancaire` sont créées sans aucune contrainte particulière d'héritage. De plus, chacune des sous-classes (adaptations) directes et indirectes de la classe `ObjectCompte` sont munies du type d'adaptation qui convient à leur cas.

Il s'agi ici, à titre d'exemple, de pouvoir choisir librement le type d'adaptation pour chacun des deux types de compte PEP et Compte d'Epargne. Autrement dit, le fait que le compte PEP soit une sorte de Compte d'Epargne ne doit pas avoir d'impact sur le libre choix du type d'adaptation du compte PEP (ou vice versa).

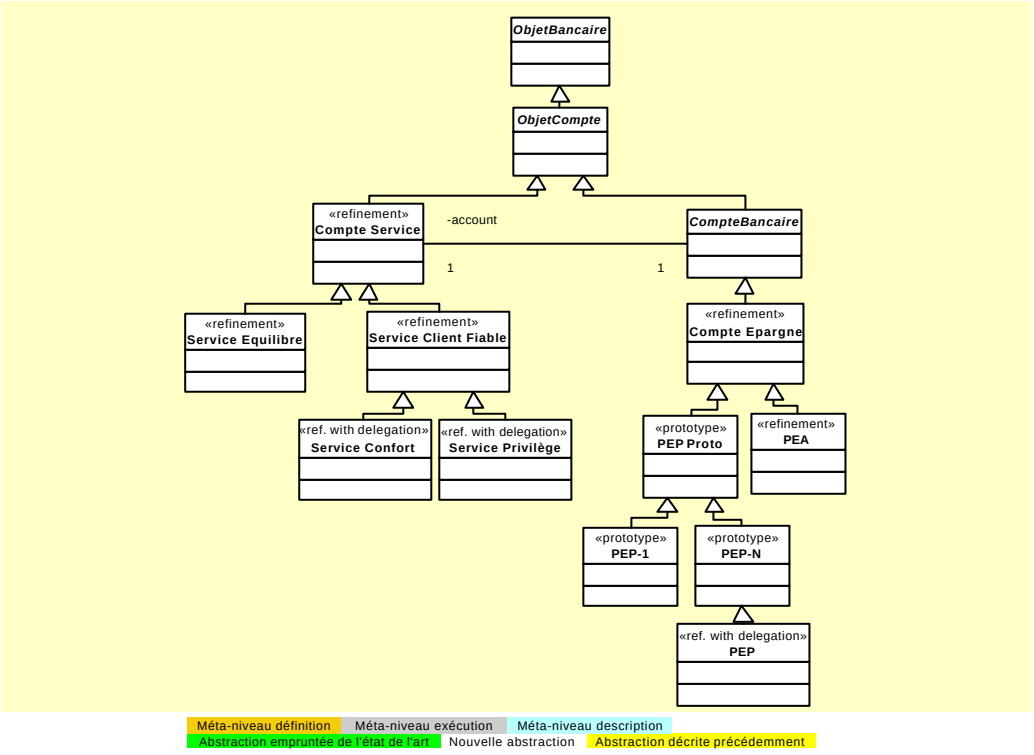


Figure 5 : Exemple pour illustrer la nécessité du choix local de différents types d'adaptabilité.

Ceci conclut la présentation "externe" de l'exemple. Nous passons à présent au fonctionnement "interne" de notre outillage.

2.4 Mise en œuvre comparée des trois frameworks

Le but de cette section est de montrer dans quelle mesure les techniques mises au point dans le cadre de notre étude outillent la création de langages d'experts qui satisfont le cahier des charges décrit ci-dessus.

Il s'agit, à titre d'exemple, de montrer comment un expert en conception de produits bancaires peut faire évoluer le langage d'experts dédié à la gestion de comptes vers un nouveau système qui offre aux utilisateurs (ici guichetiers) les fonctions nécessaires à la mise sur le marché des deux nouveaux produits de l'établissement possédant ce langage d'experts.

Il est évident qu'informatiser la gestion d'un produit est une tâche complexe qui requiert un effort considérable. Nous n'étudions pas ici tous les aspects relatifs à un tel travail, comprenant notamment la gestion des bases de données, la distribution, la synchronisation, etc.

Notre contribution se situe ici uniquement au niveau des *adaptations*. Nous esquisserons, toutefois, une solution à ce genre de problèmes (aspects techniques) dans le chapitre IV, paragraphe 4.2.1, page 196, lorsque nous évoquons les liens entre les langages d'experts et la *programmation par aspects* (AOP) [HL95, KLMMLI97].

Dans l'exposé qui suit, nous décrivons, sur l'exemple des comptes énoncé dans le paragraphe §2.3, d'abord la nature de nos trois solutions DYCTALK, MIDYCTALK et MXDYCTALK, en ce qui concerne l'ajout dynamique de nouveaux types d'objets. Nous poursuivons ensuite notre exposé par la description de l'outillage relatif aux autres aspects de notre cahier des charges.

2.4.1 Outillage de l'ajout dynamique de nouveaux types d'objets

Le point de départ des experts pour élaborer le modèle objet de la Figure 3, page 39, est la classe `CompteBancaire` (Figure 6). En effet, l'implantation, sous forme de classes, d'une modélisation par objets plus ou moins abstraite du domaine, fait partie de la contribution des programmeurs lors de la phase initiale du développement d'un langage d'experts.

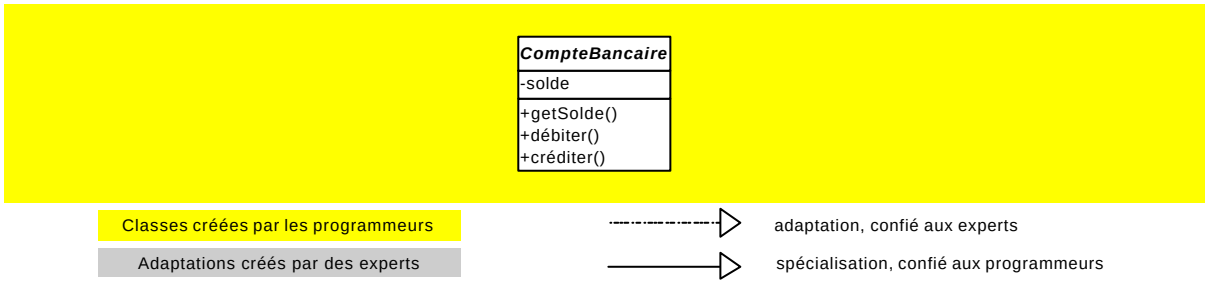


Figure 6 : Point de départ des experts pour la définition dynamique de Comptes-Service.

Comme permet de l'illustrer la Figure 7 ci-dessous, la première action de l'expert va raisonnablement être l'ajout d'un premier Compte-Service, supposons Compte-Service Equilibre.

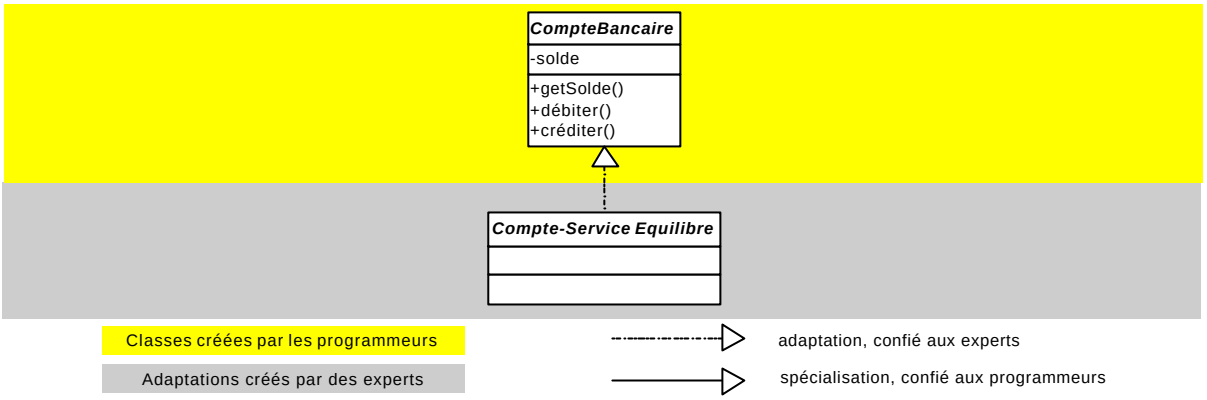


Figure 7 : Evolution du modèle objet par l'ajout du type `Compte-Service Equilibre`.

Cette action, qui fait évoluer le modèle de la Figure 6 vers celui de la Figure 7, fait intervenir, dans une certaine mesure indifféremment, l'un des trois mécanismes suivants.

Cas de DYCTALK

Dans le cas de DYCTALK, c'est le sous-système FDOM qui prend cette fonction en charge. Celui-ci correspond à notre implantation suivant le schéma DOM d'un framewotk orienté-objets qui outille la définition de nouveaux types d'objets.

Le schéma de conception DOM est rappelé dans le paragraphe 2.2, page 91 du chapitre II. Cet exposé est précédé de la description des systèmes existants qui utilisent déjà cette conception (cf. le paragraphes 2.1 du chapitre II). Nous détaillons également notre conception et implantation du framework FDOM dans le paragraphe 2.1, page 110 du chapitre III.

Le fonctionnement de FDOM est basé sur le principe d'interprétation d'instances terminales comme des *compléments de classe*. L'idée consiste à associer à une classe adaptable `C`, une seconde classe `CType` dont chaque instance comporte la définition d'une spécialisation de la classe `C`.

Plus concrètement, il s'agit ici d'associer à la classe `CompteBancaire` une seconde classe `CompteBancaireType`. Le type de compte Compe-Service est alors décrit par la création d'une instance de la classe `CompteBancaireType`. C'est elle qui comporte en l'occurrence le nom de ce nouveau type de comptes.

Pour que cette mécanique fonctionne, la classe `CompteBancaire` est programmée de façon à rechercher certaines informations au sein de son complément de classe. Pour ce faire, cette classe contient une variable d'instance `type` qui lui permet de se référer à son complément de classe. Cette possibilité est ici, toujours à titre d'exemple, utilisée pour que les instances de la classe `CompteBancaire` cherchent leur nom au sein du complément de classe.

A noter que cela suppose ici également que la classe `CompteBancaireType` comporte une variable d'instance `nom`, afin de stocker le nom des nouveaux types de compte, mais également la définition des attributs (la dimension comportement n'est pas abordée de façon systématique par DOM).

Une autre "astuce" qui permet à ce modèle de fonctionner consiste à instancier systématiquement la classe `CompteBancaire`. En effet, dans la mesure où ici les nouveaux types de comptes sont représentés par des *instances terminales* (les compléments de classe) qui complètent la définition initiale de la classe (ici `CompteBancaire`), il n'est pas possible de les instancier au sens des langages à objets.

Dans ce contexte, il n'est pas en réalité possible d'atteindre la situation illustrée par la Figure 7. En effet, une abstraction instanciable du type classe (au sens de la programmation par objets) ne peut pas être créée avec cette technique, qui produit des instances terminales jouant le rôle des classes.

Cela veut également dire qu'il n'existera jamais véritablement dans le système une instance de la classe `Compte-Service Equilibre`. En effet, cette dernière n'existe pas sous cette forme de classe est donc ne peut pas être instanciée.

Nous reviendrons sur les inconvénients multiples de cette conception dans la sous-section suivante. L'implantation détaillée de cet exemple est fournie dans le paragraphe 3.2, page 137 du chapitre III.

Cas de MIDYCTALK

Dans le cas de MIDYCTALK, c'est notre nouveau modèle de la spécialisation dynamique DARC-II et son implantation à l'aide des méta-classes standard de SMALLTALK-80 qui prennent en charge la définition de nouveaux type d'objets.

En effet, la conception standard de la spécialisation dynamique, décrite par DOM, est largement critiquée par l'article qui l'expose [RTJ00]. Nous reprenons ces critiques et les complétons par des éléments qui sont propres à notre contexte (§1.1, page 146 du chapitre IV). Après un diagnostic des causes de ces problèmes (§1.2, page 148 du chapitre IV), selon nous liés à l'interprétation d'instances terminales comme des classes, telle que nous venons de l'écrire ci-dessus, nous proposons une solution basée sur l'usage de la réflexion et des méta-classes afin de rapprocher la représentation des adaptations à celle des classes (§1.3, page 149, toujours du chapitre IV).

Cette analyse nous conduit à présenter un nouveau modèle, DARC-II, d'outillage de la spécialisation dynamique (§1.3.2, page 149). Ce modèle est en grande partie identique à son prédécesseur DARC-I (§1.1, page 108), qui est celui du DOM. Nous observons simplement que les langages à objets sont déjà confrontés à ce problème de modélisation de la spécialisation/adaptation. Ils doivent, en effet, assurer la définition de nouvelles classes, par la spécialisation des classes existantes.

Nous proposons donc de ne pas outiller la spécialisation dynamique en mettant en place des mécanismes parallèles (ce qui résulte actuellement de la conception décrite par DOM) à ceux de leur homologues dans le cas des langages à objets qui sont, eux, dédiés à la spécialisation. Autrement dit, nous proposons d'intégrer l'outillage de l'adaptation de façon la plus harmonieuse, mais aussi pratique (cf. le choix local du type d'adaptation) que possible au sein de du langage à objet d'implantation (sujet de notre extension).

Pour la mise en œuvre pratique et la validation expérimentale de cette idée, nous proposons de faire appel à la propriété des langages à objets *réflexifs* qui consiste à réifier leur modèle de la spécialisation à travers les méta-classes.

Aussi, après une présentation (cf. la paragraphe 1.5 page 151) de ce modèle réifié de spécialisation dans le cas du langage SMALLTALK-80, le langage à objets réflexif le plus largement utilisé, nous mettons en œuvre notre solution à partir du paragraphe 2, page 153 du chapitre IV.

Les adaptations créées suivant ce modèle sont de même nature que les classes, puisqu'elles sont créées par l'instanciation des méta-classes. Aussi, tous les inconvénients de la solution standard DOM des AOMs, liés à l'interprétation d'instances terminales comme des classes, disparaissent. Par ailleurs, cette solution permet bien de mettre en œuvre le schéma illustré par la Figure 7. L'adaptation `Compte-Service Equilibre` sera bien créée comme une sous-classe de la classe `CompteBancaire`. Une description détaillée de cette création, accompagnée d'une comparaison avec l'approche de DYCTALK, est fournie dans le paragraphe 3.2, page 160 du chapitre IV.

L'un des résultats majeurs de cette nouvelle solution est qu'il rend possible d'éditer les adaptations au même titre que les classes. Cela permet alors notamment d'outiller à terme le *refactoring* des adaptations.

Ces résultats sont décrits dans les paragraphes 3.9 et 3.10, page 170du chapitre IV. La description de ces résultats s'appuie toujours sur notre exemple en cours d'adaptation de comptes bancaires. Nous reviendrons sur cette question également dans le §2.4.7, page 47, ci-dessous.

Cas de MXDYCTALK

Dans le cas de MXDYCTALK, la définition de nouveaux types d'objets s'appuie sur les mêmes principes que ceux décrits pour MIDYCTALK. Seulement par l'usage de méta-classes "explicites", MXDYCTALK apporte une solution plus performante.

En effet, nous montrons dans le paragraphe 1.1, page 180 du chapitre V quelques inconvénients importants de la solution mise en œuvre par MIDYCTALK. Cela comprend notamment la propagation du choix du type d'adaptation au niveau de toute une hiérarchie de classes/adaptations. Rappelons à cette occasion que le répertoire des différents types d'adaptation que nous avons observés lors de cette étude est exposé dans le paragraphe 1.4, page 150 du chapitre IV.

Aussi, après une analyse de la cause de ce problème (§1.2, page 181 du chapitre V) que nous estimons lié au choix implicite de la méta-classe mise en œuvre par le langage SMALLTALK-80, nous proposons une solution basée sur l'idée de ramener le problème du choix local du type d'adaptation au problème déjà connu et résolu du choix des propriétés de classes [LC96] par le choix explicite de leur méta-classe (§1.3, page 182).

Enfin, nous mettons en œuvre cette solution grâce au système METACLASSTALK de Noury Bouraqadi [Bou99b] (§2, page 183). Nous discutons ensuite les avantages procurés par ce nouvel outillage et notamment le choix local du type d'adaptation (§3, page 189 du chapitre V). Les paragraphes §3.1.1, page 190, §3.1.2, page §191 et §3.1.3, page 192 du chapitre V fournissent des exemples de création de nouveaux types de comptes dans le cas de MXDYCTALK.

Toutes les autres propriétés des langages d'experts qui ont été outillées par MIDYCTALK, restent ici toujours acquises.

A travers ce cas d'ajout de nouveaux types d'objets nous venons d'illustrer les travaux des deux derniers chapitres IV et V, ainsi qu'une partie de ceux du chapitre III.

A présent nous nous intéressons à l'outillage d'autres propriétés des langages d'experts, à savoir, la définition dynamique de structures et de procédures et leur co-évolution, ainsi que la mise en œuvre de l'adaptation par des experts. Les dimensions workflow et lien causal seront traités brièvement en parallèle.

2.4.2 Outillage de la définition dynamique de structures et leur instanciation

L'épate suivante consiste à définir dynamiquement la structure de nouveaux types d'objets, et en l'occurrence celle du `Compte-Service Equilibre`.

L'outillage de la définition dynamique des structures est un sujet relativement facile et bien documenté notamment par l'article sur DOM. L'idée consiste à utiliser deux classes, `PropertyType` et `Property`. Les instances de la première permettent de décrire la structure d'un nouveau type d'objets. Chaque instance est appelée un descriptif d'attribut. Celui-ci comporte, en règle générale, un nom et un type. Les instances de la seconde servent à contenir les valeurs des "instances" des descriptifs d'attributs.

Cette fonction est outillée ici par le framework FDOM, présenté dans le paragraphe 2.1, page 110 du chapitre III. En outre, le paragraphe 3.3, page 161 du chapitre IV fournit une description détaillée et comparative de la mise en œuvre de cet exemple à l'aide de notre outillage.

Nous exposons également brièvement notre modèle d'outillage de la gestion des "types métier" dans l'annexe VI.

2.4.3 Outillage de la préparation pour la composition

La composition de procédures, telle que nous la concevons ici, s'appuie sur les descriptifs de service. Nous avons lors de cette étude observé plusieurs types de descriptifs que nous avons décrits dans le §2.3.4, page 33 ci-dessus. Comme nous l'avons annoncé à cette même occasion, chaque langage d'experts gère un référentiel de descriptifs. Celui-ci est enrichi de deux façons : automatiquement et manuellement.

Pour l'heure nous avons automatisé deux cas :

1. génération des *getters* et *setters* : la composition de procédures capables d'opérer sur la structure de nouveaux types requiert une étape intermédiaire. Celle-ci consiste à générer, pour chaque descriptif d'attribut ajouté, deux descriptifs de services. Le rôle de ces derniers et de permettre l'expression lors de la composition des accès en lecture et en écriture à ces attributs (cf. la sous-section suivante). Le paragraphe 3.4, page 163 du chapitre IV détaille, en s'appuyant sur notre exemple en cours, la mise en œuvre de la génération automatique de ces descriptifs.
2. génération du descriptif des macro-procédures (semi-automatique) : pour permettre l'appel de sous-procédures nous proposons d'habiller les procédures existantes par des descriptifs de service. Notre analyse sur ce sujet figure dans le §2.4.2, page 61 du chapitre I. Elle est suivie par un modèle de conception, présenté dans le §3.5.2, page 74 du même chapitre I. Cette génération est semi-automatique dans la mesure où il ne semble pas opportun d'"habiller" systématiquement toutes les procédures. La partie manuelle consiste donc simplement à annoncer le souhait d'associer à une procédure un descriptif de service. Le reste du travail est réalisé automatiquement.

La partie manuelle consiste à choisir simplement les méthodes, primitives externes, constructeurs, etc. (cf. les différentes possibilités au §2.3.4, page 33) qui sont requises pour la description des procédures envisagées par l'expert. Ce choix est suivi par la création effective du descriptif. Cela nécessite, en règle générale, de fournir des informations comme le nom du descriptif, la liste de ses arguments et un commentaire. Il convient également de fournir un nom pour chaque argument ainsi qu'un type. Ces informations sont utiles lors de la création des dialogues qui sont affichés lors de la composition et qui permettent d'instancier les descriptifs de service.

Nous outillons cette préparation par un modèle exposé dans le §3.4, page 70 du chapitre I. Nous illustrons ensuite, dans le paragraphe 3.5, page 164 du chapitre IV, le fonctionnement de ce mécanisme à l'aide de notre exemple en cours.

2.4.4 Outillage de la définition dynamique de procédures par les experts

L'étape suivante consiste à composer effectivement des procédures. Nous outillons cette fonction sur la base d'un modèle emprunté initialement des travaux sur la programmation par des experts et plus particulièrement le langage de feuilles de calcul (e.g. tableurs) [Nar93]. Nous avons, toutefois, adapté ce modèle à notre cas notamment par l'intégration de la composition dynamique de procédures qui sont elles même définies dynamiquement (cf. sous-section suivante).

A titre d'exemple, la procédure `Cumuler les agios du jour()` est définie en créant cinq instances de descriptif de service. Deux entre elles représentent des arguments en entrée : `Le Compte` et `Le montant`. Ces deux instances n'ont pas d'argument. La troisième est l'instance du descriptif `Obtenir Cumul d'agios`. Elle prend en entrée `Le Compte` et retourne le cumul des agios. La quatrième est l'instance du descriptif `Additionner deux nombres` qui prend en entrée `Le montant` et le cumul des agios et qui retourne leur somme. La dernière est l'instance du descriptif `Affecter Cumul d'agios` qui prend en entrée `Le Compte` et la somme et met ainsi à jour le cumul des agios.

Les deux *seules* activités demandées aux experts sont ici :

1. le choix des descriptifs à instancier ; ainsi que
2. le choix des instances de descriptifs déjà définies comme argument de nouvelles instances.

Par ces deux actions simples, que l'on peut retrouver de façon semblable dans les tableurs, les experts peuvent décrire des modèles complexes. Notre contribution ici est de définir un système de classes

et de créer un framework qui permettent d'intégrer de façon systématique ce modèle de programmation dans des logiciels objets.

Nous permettons ainsi *l'expression par des experts des modèles qui manipulent des* ***objets du domaine****, lesquels sont aussi définis par des experts.*

Nous décrivons dans le paragraphe 1, page 53 du chapitre I notre analyse de la composition par des experts. En s'appuyant sur cette base, nous concevons un modèle qui outille la composition de procédures par des experts (§2, page 57 du chapitre I).

Le paragraphe 3.6, page 166 du chapitre IV illustre le fonctionnement de l'outillage mentionné ci-dessus en s'appuyant sur l'exemple en cours.

2.4.5 Outillage de la composition dynamique de procédures définies à l'exécution

A présent nous disposons du matériel nécessaire à l'illustration d'une autre fonction importante de notre outillage, c'est à dire, la composition dynamique de services dynamiquement définis.

Il s'agit des macro-procédures dont le modèle d'analyse est présenté dans le paragraphe 2.4, page 60 du chapitre I. Le modèle de conception des macro-procédures est détaillé le paragraphe 3.5, page 74 également du chapitre I.

Les macro-procédures servent à définir des appels de *sous-procédures*. Par exemple, la procédure `Traiter les agios du jour()` appelle la sous-procédure `Cumuler les agios du jour()`. Pour mettre en œuvre cet appel, l'expert procède toujours de la même façon : choisir le descriptif de service, ici celui associé à la procédure `Cumuler les agios du jour()`, et de préciser les instances de descriptifs de service existantes qui produiront, lors de l'activation, les arguments dont la sous-procédure a besoin pour s'exécuter (ici `Le Compte` et `Le montant`).

Nous tenons à préciser qu'avec ce simple procédé il est possible de mettre en œuvre des programmes assez complexes. Un *programme* se présente toujours sous forme d'une feuille de calcul (ou structure similaire). Elle contient un ensemble d'instances de descriptifs de service. En zoomant sur les instances du type *procédure* on se retrouve dans une autre feuille de calcul qui contient à son tour la définition de la procédure appelée. Les deux feuilles "appelante" et "appelée" sont reliées entre elles à travers les cellules de la feuille appelante qui sont utilisées comme argument de la feuille appelée.

Ce procédé est particulièrement simple à mettre en œuvre. De plus, les séquences de feuilles peuvent, théoriquement, se poursuivre indéfiniment, permettant ainsi la description de modèles de complexité considérable (cf. §1.2.2, page 55 du chapitre I).

Le paragraphe 3.7, page 167 du chapitre IV illustre le fonctionnement de cet outillage à l'aide de l'exemple en cours.

Note importante au sujet des interfaces graphiques

Il est important de préciser ici que nos frameworks ne disposent pas d'interfaces graphiques. Il s'agit certes d'un manquement important dont l'étude systématique sort du cadre des travaux présentés ici. En effet, les techniques et outils classiques de création d'interfaces ne conviennent pas aux systèmes du type AOM où la structure et le comportement du logiciel évolue lors de l'exécution.

2.4.6 Outillage de l'activation de procédures et du lien causal

Les experts achèvent à présent la définition des procédures de leurs nouveaux types d'objets et sont donc en mesure de s'intéresser à leur activation. L'idée de la solution que nous mettons ici en œuvre consiste à créer des modèles à deux "étages" où on fait correspondre, suivant le schéma de conception *Type Object*, de façon systématique à chaque couche d'objets qui modélise la définition des comportements ou des structures, une couche qui modélise leur activation ou instanciation.

Cela fournit tout particulièrement les structures nécessaires à la représentation explicite des activations.

Un autre élément qui entre en jeu est la notion de *stratégie d'activation*, qui, suivant le schéma de conception *Strategy*, permet d'appliquer un algorithme adéquat à chaque type de service. Par exemple, un appel d'une méthode (descriptif de service du type *méthode*) n'est pas exécuté de la même façon q'un appel de primitive externe (descriptif de service du type *primitive externe*).

Pour que ce modèle fonctionne, il devient nécessaire de faire intervenir le descriptif de service dans le processus d'activation en lui déléguant le choix de la stratégie d'activation adéquate.

Le modèle d'analyse de l'activation des procédures est décrit dans le paragraphe 2.2, page 59 du chapitre I. Le modèle de conception correspondant, basé sur la notion de stratégie d'activation, est décrit dans le paragraphe 3.3, page 67 du même chapitre.

La même question se pose au sujet des macro-procédures.

Le modèle d'analyse de l'activation des macro-procédures (appels de sous-procédures) est exposé dans les paragraphes §2.4.3, page 61 et §2.4.4, page 61 du chapitre I. Le modèle de conception correspondant figure au §3.5.3, page 75, ainsi que le paragraphe 3.5.4 page 75 toujours du chapitre I.

Enfin, le paragraphe 3.7, page 167 du chapitre IV illustre le fonctionnement de cet outillage à travers notre exemple en cours.

C'est cet ensemble de modélisation dédié à l'activation des procédures qui constitue notre outillage du lien causal. Il s'agit d'un travail, certes indispensable, mais encore insuffisant par rapport à l'importance du sujet.

2.4.7 Outillage du travail collaboratif (*refactoring*)

A présent les experts ont fait évoluer le modèle objet initial (cf. la Figure 6, page 41) du langage d'experts de notre exemple vers le modèle illustré par la Figure 3, page 39. Nous avons déjà esquissé la nature de ce travail dans le §2.4.1, page 41 ci-dessus (cas de MIDYCTALK).

Les paragraphes 3.9, page 170 et 3.10, page 170 du chapitre IV expliquent en quoi notre solution outille l'édition et le *refactoring* des adaptations par les programmeurs.

Le paragraphe 3.11, page 171 du même chapitre expose notre contribution à l'outillage proposé par Dragos Manolescu en le rendant capable de servir à la composition de procédures par des experts et cela dans le contexte des AOMs.

2.4.8 Outillage du choix local du type d'adaptation

Enfin, le chapitre V est entièrement dédié à l'exposé de notre outillage du choix local du type d'adaptation. Nous avons également déjà exposé la nature de ce travail dans le §2.4.1, page 41 ci-dessus (cas de MXDYCTALK).

Le paragraphe 3, page 189 illustre son fonctionnement sur la base de l'exemple en cours. Il montre la création par des experts du nouveau type de compte PEP suivant une démarche basée sur le prototypage.

3 Organisation de la thèse

Nous exposons la validation expérimentale de notre thèse par une succession de cinq chapitres qui vont se compléter harmonieusement et nous conduire progressivement à un outillage capable de permettre la création de langages d'experts munis de toutes les propriétés énumérées dans le paragraphe 1.2, page 17.

Chapitre I : Mise en œuvre par des experts : le composant DART

Nous définissons dans un premier temps un nouveau système de classes, appelé DART[35], qui permet la création de langages qui offrent la définition dynamique de procédures par la composition d'instances de descriptifs de service. Dans la conception de DART nous avons tenu compte des analyses de Nardi [Nar934] sur les techniques de conception de langages dédiés aux experts. Aussi, ce système garantie la facilité d'apprentissage de la programmation dynamique des langages d'experts.

Ce système de classes se concrétise sous la forme d'un framework appelé FDART qui entrera dans les trois frameworks DYCTALK, MIDYCTALK et MXDYCTALK. C'est pourquoi nous commençons notre exposé ainsi.

Chapitre II : AOM et Micro-Workflow

Le chapitre II expose les technologies standard dont nous nous servons dans l'outillage de la création de langages d'experts. Nous y distinguons deux sous-parties :

La première, cas de DOM, concerne l'outillage de l'ajout dynamique de nouveaux types d'objets ainsi que la définition de leur structures.

La seconde partie, cas du Micro-workflow, concerne l'outillage de la création d'applications à objets qui intègrent un mécanisme dédié à la définition explicite des collaborations entre les objets. Le but de cette démarche consiste à améliorer l'adaptabilité des procédés mise en œuvre par des logiciels en les réifiant et les rendant ainsi plus facilement l'objet de future modifications.

Nous terminons ce chapitre par l'exposé du problème de couplage. Il s'agit de montrer que le problème de co-évolution dynamique de procédures et de structures ne peut pas être résolu par une simple juxtaposition de DOM et de Micro-workflow.

Chapitre III : Premier outillage de l'adaptation (DYCTALK)

Nous nous appuyons ensuite sur les technologies "Johnsonniennes" DOM et *Micro-workflow* pour définir un second système de classes, dédié à l'outillage de la spécialisation dynamique. Nous définissons alors un modèle de couplage de ces deux structures et nous montrons comment le mettre en œuvre.

Ces travaux se concrétisent au sein du second composant du framework DYCTALK, appelé FDARC.

Chacun des deux systèmes définis en chapitre I (FDART) et ci-dessus (FDARC) offre une partie de notre première solution dédiée à l'outillage de l'adaptation. Nous couplons subtilement ces deux solutions partielles. Cela permet essentiellement d'apporter au mécanisme de spécialisation dynamique obtenu grâce à FDARC la *dimension d'expert* obtenu grâce à FDART. Dans d'autres termes, nous obtenons un système de classes qui permet de créer d'outils qui assurent l'ajout dynamique de nouveaux types d'objets, et la définition de leur structure et procédures et cela par des experts. Ce système satisfait également le "lien causal" et la "dimension workflow".

Ces travaux finalisent la création du framework orienté-objet DYCTALK.

[35] **D**ynamic **ART**ifact-driven class specialization.

Nous validons ainsi la première partie de notre thèse. Cette validation s'appuie sur des technologies standards. Celles-ci atteignent ici leurs limites de performances. Elles ne permettent en particulier pas d'assurer le travail collaboratif ni le choix local du type d'adaptation.

Chapitre IV : Second outillage de l'adaptation (MIDYCTALK)

De nombreux problèmes liés à la mise en œuvre standard des modèles objets adaptatifs, donc notre premier outillage de l'adaptation, ont mis au jour dans des systèmes concernés [RTJ00, page 6], [YBJ01a, YBJ01b]. Cela comprend notamment l'impossibilité pour les programmeurs d'éditer les adaptations à l'aide de leurs outils habituels (e.g. cas de *refactoring*).

Nous montrons que le problème de fond est l'usage des structures de représentation de l'adaptation qui ne sont pas engagées dans des *relations* appropriées avec celles des spécialisations. Nous montrons ensuite comment résoudre ces problèmes par l'usage de la réflexion et plus particulièrement ici les méta-classes [Coi97a, Coi97b, KdRB91, Per98, FP99] de SMALLTALK-80 [FJ89, Riv96a]. Nous nous servons des méta-classes comme un moyen pour rapprocher la représentation des spécialisations et des adaptations.

Ces travaux conduisent à la création d'une nouvelle version du framework orienté-objets DYCTALK, appelé MIDYCTALK. Celui-ci constitue notre second outillage de l'adaptation. Il offre la nouvelle possibilité d'assurer le travail collaboratif, tout en conservant toutes les propriétés déjà acquises des langages d'experts.

Ce travail sert également d'une étape intermédiaire vers la validation de la seconde partie de notre thèse.

Chapitre V : Troisième outillage de l'adaptation (MXDYCTALK)

La solution obtenue lors de l'étape précédente manque d'une dernière propriété des langages d'experts, c'est le choix local du type d'adaptation. En effet, elle conduit au choix statique du type d'adaptabilité au niveau de toute une hiérarchie de classes.

Après un diagnostic des causes de ce problème, lié au choix implicite par SMALLTALK-80 de la méta-classe de chaque classe, nous proposons une solution basé sur le choix explicite de la méta-classe, que nous mettons en œuvre à l'aide du système METACLASSTALK de Noury Bouraqadi [Coi90, BSLRC96, BLR98, Bou99].

Ces travaux conduisent à la création d'une nouvelle version du framework orienté-objets DYCTALK, appelé MXDYCTALK. Celui-ci assure toutes les propriétés des langages d'experts décrite dans le paragraphe §1.2, page 17.

Nous validons ainsi la second et la dernière partie de notre thèse.

Conclusions et Perspectives

Nous terminons ce mémoire par l'exposé de nos conclusions qui portent sur la faisabilité de l'outillage de la création systématique des langages d'experts, munies de toutes les propriétés décrites dans le paragraphe 1.2, page 17.

Nous décrivons ensuite les trois axes de recherches que nous estimons important de mener à bien. Il s'agit d'étudier les rapports entre les langages d'experts et d'une part les langages à prototypes, et d'autres part la réflexion. Une autre dimension importante est celle de la méthodologie de création des langages d'experts.

Chapitre I :
Mise en œuvre par des experts

Chapitre I : Mise en œuvre par des experts

1 Introduction

1.1 Rôle des systèmes dédiés aux experts

Le processus actuel de développement, de la maintenance et de l'évolution de logiciels est, en règle générale, fondé sur le principe de transfert de connaissances qui relève de l'expertise du domaine aux informaticiens. Le rôle potentiel des experts[36] est sous-estimé au profit d'une démarche qui n'a jamais véritablement pu conduire à des résultats très satisfaisants.

Une alternative à cette approche consiste à envisager un partage plus équilibré de responsabilités entre les informaticiens et les experts [LSGBP99]. Le but est d'éliminer deux obstacles majeurs dans le processus de développement de logiciels : 1) la dépendance des informaticiens aux experts en matière de connaissances métier ; 2) la dépendance des utilisateurs aux informaticiens en matière de la description des procédés métier à informatiser aussi bien durant la phase de développement initial que les phases de maintenance et d'évolution.

Dans cette approche, l'expert est considéré comme le *partenaire* indissociable de programmeurs dans le cycle de développement de logiciels. L'enjeu consiste alors à rendre effective la participation des experts dans ce cycle par la création de nouvelles techniques. *L'objectif des langages d'experts est d'aider à la mise en œuvre de ce partage plus équilibré de rôles par la spécialisation de langages à objets (réflexifs)*[37].

[36] Dans le cadre des travaux que nous présentons ici, l'expert est considéré tout d'abord comme un individu non informaticien. Celui-ci est motivé pour participer au processus de développement de logiciels dont le but est d'automatiser ses procédés de travail. De plus il dispose des connaissances requises pour une telle participation [Nar93].

[37] Nous empruntons les idées générale sur les systèmes dédiés aux experts des travaux réalisés par Bonnie A. Nardi et al [Nar93], mais aussi ceux du projet METAGEN [RSBP95, RBP00] que nous avons étudié de très près et auquel nous avons participé indirectement [Rev96]. Nous avons également eu l'occasion d'évaluer ces idées dans le cadre de projets industriels Calibres et Prelude Inspection [Raz00a, ASM99].

L'idée de systèmes dédiés aux experts[38] n'est pas nouvelle. Déjà en 1967 James Martin [Mar67] insistait sur leur nécessité :

> *"Much of the development in the years to come will probably be in the area of languages, especially languages for on-line use. Now [that] we have this immensely powerful tool available to us, it is important to extend its use to the maximum number of people. We must develop languages that the scientist, the architect, the teacher, and the layman can use without being computer experts. The language for each user must be as* natural *as possible to him. The statistician must talk to his terminal in the language of statistics. The civil engineer must use the language of civil engineering. When a man learns his profession he must learn the* problem-oriented languages *to go with that profession." [Mar67]*

Nous partageons cette idée avec Bonnie A. Nardi qui estime qu'une part importante de la vision de Martin n'a pas encore été appréciée à sa juste valeur[39]. Les travaux présentés dans ce mémoire veulent contribuer à la mise en oeuvre de cette idée par la proposition d'une conception qui permet de systématiser la création de ce type de systèmes dans le contexte de la programmation par objets. La valeur de telles contributions est mieux appréciée si l'on considère les propos de Nardi qui précise :

> *"The task-specific language approach is not without its problems. First, it is expensive to build that many different task-specific languages that are needed for the myriad uses to which computers are put, or could be put. Further progress in reusable software components will be needed to alleviate this problem, to bring down the cost of developing task-specific systems. … A second possible problem with a plethora of task-specific programs is that users will be forced to switch between many different systems, leaning a new user interface every time. The third, and the most serious, problem is that it is difficult to know just how specific a task-specific system should be." [Nard93, page 50]*

En matière des solutions possibles, elle ajoute que :

> *"The notion of families of shared, specialized, task-specific applications holds the promise for providing both consistency and flexibility." [Nard93, page 51]*

La solution que nous proposons ici s'inscrit dans le cadre de l'outillage dédié à la création systématique de ce type de familles de logiciels. Nous considérons dans cette conception non seulement la facilité d'apprentissage par des expert, mais aussi d'autres aspects techniques importants comme la "dimension wokrflow", le lien causal mais aussi l'intégration harmonieuse dans les langages à objets pour permettre le travail collaboratif ainsi que le choix local du type d'adaptation.

[38] De l'anglais, "*domain-oriented computer languages*".

[39] "*the easy availability of end-user programming systems for smaller niches of users whose computing needs are more specialized that, but just as important as those of systems that command large markets. Task-specific programming languages and environments that would allow users who do not belong to vast market segments to create their own applications have not yet been developed, except as expensive and relatively inflexible custom solutions to specific problems for specific users*." [Nar93, page 40]

1.2 Assurer la facilité d'apprentissage

La facilité d'apprentissage est l'une des qualités importantes de systèmes dédiés aux experts [Nar93]. Selon Nardi, celle-ci peut être acquise par la technique suivante :

1.2.1 Primitives de "haut niveau"

La démarche mise en avant par les systèmes dédiés aux experts consiste à capitaliser sur l'expertise des utilisateurs pour créer des langages qui leur permettent de programmer avec des primitives qui relèvent de leur domaine de compétences [Nar93, page 39]. La connaissance effective de la sémantique de ces actions par les experts du domaine facilite grandement l'apprentissage de ces systèmes[40].

A titre d'exemple, un système dédié aux responsables de comptes dans un établissement bancaire doit proposer des primitives comme : ouvrir un compte, lire le solde et calculer les intérêts journaliers. Dans le cas d'un logiciel de contrôle 3D comme PRELUDE INSPECTION, parmi ces primitives on retrouve les fonctions de la géométrie analytique.

1.2.2 Structures de contrôles simples mais efficaces

Un système dédié aux experts doit proposer des structures de contrôle simples mais efficaces. La définition d'une itération ou d'une conditionnelle ne doit pas nécessiter de compétences en informatique.

Un exemple célèbre est le langage de feuilles de calcul. En effet, dans ce cas la définition d'une itération sur un ensemble de cellules, e.g. pour calculer la somme de leur valeur, consiste simplement à *sélectionner* les cellules concernées par l'itération (ici l'addition).

La situation est similaire en ce qui concerne les conditionnelles. En effet, la condition est associée à une cellule et ne transfère pas le contrôle d'une partie de programme à une autre. Son effet reste local à la cellule où elle est définie.

Ces propriétés sont obtenues grâce à la prise en considération dans la conception de ce langage lui-même de l'usage d'un formalise visuel, le tableau [Nar93, pages 45-48]. La programmation dans ce cadre requiert uniquement des experts la maîtrise de deux concepts: *cellule* (qui jouent le rôle de variable) et *fonction* décrite sous la forme de relations entre cellules.

Nardi et al. observent que malgré la simplicité des moyens de programmation offerts par ce langage, il permet aux experts d'écrire des programmes très complexes. Cette complexité est due à la richesse de relations établies entre les éléments d'une feuille de calcul. Ces relations correspondent, bien sûr, à celles qui existent déjà entre les entités du domaine modélisé.

La description de ces relations, bien connues des experts, n'implique pas toujours l'usage des techniques de programmation sophistiquées et des langages de programmation puissants. Même avec des primitives de haut niveau et des structures de contrôle très simples mais efficaces, comme c'est le cas des tableurs, il est possible d'énoncer des relations complexes. Cette même étude montre, en effet, que la plupart des utilisateurs des tableurs utilisent en règle générale moins de dix primitives dans leurs modèles. Ce sont des fonctions arithmétiques élémentaires et les fonctions d'arrondi.

---

[40] *The spreadsheet formula language is accessible because its task-specific operations are already familiar to users with applications requiring numeric manipulation. In addition, users can invoke the operations without having to do the tedious non-task/domain-related activities required in general programming languages such as declaring data types, including files, naming variables, "making" the system, and compiling. Being free of the tedium of low-level programming minutiae allows users to concentrate on the problem at hand, to be engaged with the problem-solving semantics of the application itself. … Spreadsheets, then, offer a set of very useful control structures—conditionals in formulas, iteration over cell ranges, and the modeling of cell dependencies through one way constraints. But these control constructs are characterized by conceptual clarity and simplicity in terms of required programming effort.* [Nardi, pages 45-48]

Aussi, Nardi propose d'appliquer systématiquement cette approche à la conception de systèmes dédiés aux experts. Celle-ci lève un obstacle important pour les experts en leur permettant de réaliser des programmes et résoudre des problèmes, sans pour autant maîtriser des concepts qui ne relèvent pas de leur domaine de compétences.

La facilité d'apprentissage s'accentue également par le fait que l'utilisateur est dispensé des tâches comme la compilation, la déclaration de types de données, la "make", le déploiement, etc., lesquelles ne relèvent *a priori* non plus pas de son domaine de compétences.

Le système DART que nous proposons dans ce chapitre et montrons comment le mettre en œuvre sous forme d'un framework orienté-objets FDART correspond à un système de classes qui modélise la définition et l'exécution de programmes suivant le modèle du langage de feuilles de calcul. En outre, nous prévoyons dans cette modélisation des facilités d'extensions qui vont nous permettre de le coupler ultérieurement avec le système DARC (cf. le chapitre III). C'est ce couplage qui nous conduit à notre première solution d'outillage dédié à la création systématique de langages d'experts.

1.3 Note technique : trois types d'objets de méta-niveau

Avant d'entamer la description de nos systèmes de classes, il convient de préciser que les abstractions que comportent ces derniers servent à décrire des programmes et leurs exécutions. Aussi, par analogie avec les langages à objets réflexifs, elles peuvent être considérées comme des objets du méta-niveau au sens de N. Bouraqadi [Bou99][41]. Nous y distinguons trois types d'objets de méta-niveau :

1. Objets du *méta-niveau définition* : ce sont les objets qui servent à définir des adaptations. Les abstractions de ce type sont identifiables dans les illustrations par leur position dans un cadre de couleur *orange*[42]. Des exemples de ce type d'objets existent dans les composants du système DART, mais aussi au sein du micro-workflow qui est utilisé pour la définition dynamique de procédures.
2. Objets du *méta-niveau exécution* : ce sont les objets qui implantent les stratégies d'interprétation des procédures définies en (1). Des exemples de ce type d'objets existent à nouveau au sein des composants du système DART (stratégies d'exécution), mais aussi le micro-workflow, la partie concernant la modélisation de l'exécution des procédures. Les abstractions de ce type sont identifiables dans les illustrations par leur position dans un cadre de couleur *gris clair*.
3. Objets du *méta-niveau description* : ce sont les objets qui guident, dans le sens de fournir un complément d'information, la réalisation des étapes (1) et (2). Dans le cas de la définition, ils fournissent ici des informations comme le nombre d'argument nécessaire ou le type de chacun de façon à mieux orienter le procédé de définition de procédures. Dans le cas de l'exécution ils permettent ici le choix de la stratégie d'exécution adéquate suivant la nature du service. En effet, à titre d'exemple, un service du type *Getter* (cf. Tableau 1, page 34) n'est pas exécuté de la même façon qu'un service du type *Méthode*. L'unique exemple de ce type d'objets dans notre système sont les descriptifs de service. Les abstractions de ce type sont identifiables dans les illustrations par leur position dans un cadre de couleur *bleu clair*.

---

[41] "Un système réflexif peut donc être décomposé en différents niveaux d'abstraction. Le premier niveau, appelé *niveau de base*, décrit les traitements à réaliser, i.e. les tâches que le système doit réaliser. Le second niveau appelé *méta-niveau* (ou niveau méta) interprète le niveau de base. … Cette séparation a pour conséquence de faciliter la maintenance et l'évolution des systèmes réflexifs. En effet, les développeurs disposent de deux niveaux d'intervention clairement séparés : le niveau de base et le méta-niveau. Les services d'un système (i.e. le "Quoi") sont décrits dans le niveau de base et les mécanismes d'exécution (i.e. le "Comment") sont décrits dans le méta-niveau [WY88, McA95b, GC96, SW96]. … Nous appelons *méta-objet* un objet qui joue le rôle d'interprète pour un ou plusieurs autres objets. Les méta-objets permettent de contrôler la structure et le comportement d'autres objets. Par exemple, un méta-objet définit la manière d'allouer la mémoire qui correspond à la structure des objets qu'il contrôle. Nous appelons *réification* le résultat de l'opération du même nom qui consiste à représenter des éléments de programmes sous forme d'objets qui existent à l'exécution. A titre d'exemple, les classes et les méthodes constituent des réifications dans un langage réflexif. Un méta-niveau est constitué de méta-objets et de réifications. Nous désignons donc par objet du méta-niveau, un objet faisant partie du méta-niveau sans préciser si cet objet est un méta-objet ou une réification." [Bou99, pages 20-25]

[42] Cette couleur peut exceptionnellement être portée par le rectangle qui représente une abstraction, c'est à titre d'exemple, le cas pour `ServiceDescription` dans la Figure 8.

2 DART : le modèle d'analyse

Conformément aux principes et recommandations exposés ci-dessus, le système DART modélise la mise en œuvre par des experts de la définition dynamique de procédures, ainsi que leurs activations. La Figure 8 ci-dessous offre une vue d'ensemble de ce modèle que nous allons détailler dans la suite de cette section.

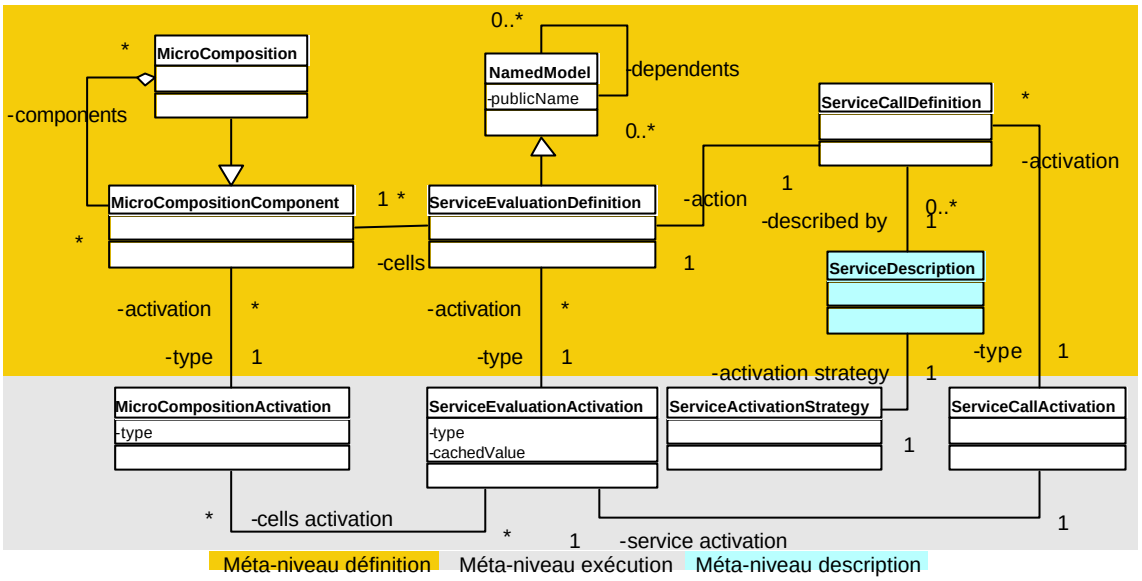


Figure 8 : Modèle d'analyse du système de classes DART.

2.1 Composition d'instances de descriptifs de service

Dans le paragraphe 2.3.4, page 33 de l'introduction, nous avons exposé les descriptifs de service ainsi que leurs différents rôles au sujet de la composition et de l'activation des procédures. Nous revenons ici sur ces aspects.

2.1.1 Structuration

La partie haute de la Figure 8 comporte les abstractions (figurant sur une couleur de fond orange) qui modélisent la définition de procédures.

Ici, suivant le schéma de conception *Composite* [GHJV95], une procédure (`MicroComposition`) est définie comme une collection de composants (`MicroCompositionComponent`). Chaque composant d'une micro-composition est à son tour composé d'un ensemble d'*instances de descriptifs de service* (`ServiceEvaluationDefinition`). Ces instances sont organisées à l'aide d'une matrice (le lien appelé `cells`). Autrement dit chaque composant d'une micro-composition joue le rôle d'une matrice. Chaque instance de descriptif de service est alors affectée à une cellule de cette matrice[43]. Cette affectation permet d'identifier de façon discriminante chaque instance de descriptif de service.

Ce choix de modélisation suivant le schéma *Composite* permet d'imbriquer au sein d'une même définition de procédure (une micro-composition), d'autres définitions de procédures (des micro-compositions ou composants).

[43] Le choix de la structure matricielle est ici arbitraire, d'autres structures comme liste ou arbre peuvent être utilisées pour organiser les cellules.

A chaque instance de descriptif de service est à son tour associée une *définition d'appel de service* (le lien `action` vers `ServiceCallDefinition`). Cette abstraction représente la définition d'un calcul. Toutefois, les instances de descriptifs de service ne font pas d'hypothèse particulière sur la nature de la représentation de cette définition de calcul. La seule contrainte est que toute représentation utilisée doit lors de l'interprétation répondre au message `evaluate`.

Cette modélisation constitue l'une des caractéristiques importantes de DART. C'est elle qui permet de définir les calculs aussi bien sous forme de micro-procédés à la Micro-worklfow que des micro-compositions imbriquées à la DART (cf. ci-dessous les macro-procédures) ou encore des arbres de syntaxe abstraite crées suivant le schéma de conception *Interpreter* [GHJV95]. Les constantes sont modélisées comme un cas particulier de telles représentations.

A chaque *définition d'appel de service* est, par ailleurs, associé un descriptif de service. La responsabilité de la création d'instances de descriptifs de services appartient au descriptif lui-même (envoi du message `asExpressionEvaluation`). Cette modélisation a l'avantage de permettre à chaque descriptif de créer le type d'instance de descriptif qui lui convient.

A titre d'exemple, la définition par l'expert de la procédure `Traiter les agios du jour()` (cf. l'introduction, le §2, à partir de la page 31) donne lieu à la création d'une instance de la classe `MicroComposition`. Celle-ci pointe sur une collection qui contient une seule instance de la classe `MicroCompositionComponent`. Cette dernière comporte une matrice de trois lignes et cinq colonnes. Des instances de la classe `ServiceEvaluationDefinition` sont affectées aux cellules C12, C23, C24, C25, C32, C33 de cette matrice. Elles représentent les instanciations de descriptifs de services qui sont récapitulés par le Tableau 2 exposé dans l'introduction.

Une autre dimension de cette modélisation concerne la gestion des dépendances entre les différents calculs. Toute représentation de calcul est rendue dépendant des arguments requis pour sa réalisation. Cela implique que la valeur de l'expression représentée doit être recalculée en cas de changement de la valeur de chacun de ses arguments.

Par exemple, puisque la procédure `Calculer les agios journaliers()` (cf. Figure 1), affectée à la cellule C32, utilise comme argument les valeurs courantes de cellules C23, C24 et C25, alors C32 est *automatiquement* ajoutée à la liste des dépendants de C23, C24 et C25 de façon à être averti de tout changement de valeur de ses arguments.

Il nous semble également nécessaire que les langages d'experts permettent une gestion manuelle des relations de dépendances (ajout ou suppression manuelle).

2.1.2 Mode d'emploi

A priori, l'éditeur de composition de procédures d'un langage d'experts organise les descriptifs de service, qui se trouvent dans le référentiel concerné, suivant une logique qui convient aux experts. A titre d'exemple, le système CALIBRES propose une palette qui comporte des icônes. Chaque icône regroupe les descriptifs de service suivant le type du résultat retourné. En effet, dans le cas de ce système, tous les descriptifs de service sont du type *méthode statique* (constructeur), c'est à dire que leur exécution donne lieu à la création d'un nouvel objet qui est retourné comme le résultat de l'activation du service concerné. Par exemple, tous les descriptifs de service dont le calcul retourne une distance se trouvent associés à un même icône.

Comme nous l'avons évoqué brièvement plus haut dans ce chapitre, nos frameworks *ne fournissent pas actuellement un tel éditeur*. En effet, la création de celui-ci n'entre pas directement dans le cadre de notre travail. Aussi, dans les exemples que nous exposons ici, nous utilisons les scripts textuels pour montrer le mode d'emploi des systèmes de classes que nous mettons en œuvre.

Un expert commence la composition de procédures par le simple choix dans cette palette du descriptif de service qu'il souhaite instancier (DS). En l'absence d'un protocole par défaut[44], il doit également choisir la cellule d'affectation de cette instance de descriptif de service (IDS)[45]. Cette dernière référence une définition d'appel de service (`DDS`), laquelle référence à son tour le descriptif de service choisi par l'expert (DS).

Cette configuration permet, lors de l'exécution, à DDS de se renseigner auprès du DS correspondant afin d'obtenir la stratégie d'exécution adéquate (cf. le paragraphe suivant) ou le type de chacun des paramètres (pour la vérification facultatif de types).

Une autre partie de cette activité consiste à choisir les arguments nécessaires à la réalisation du service demandé (DS)[46]. Sur le plan ergonomique, ce choix s'effectue toujours par un simple clic, e.g. Excel, sur les cellules concernées. Suite à cette action, l'instance du descriptif de service associée à la cellule choisie, IDS, est ajoutée dans la liste des arguments requis à l'exécution du service et cela, *a priori*, dans l'ordre de la sélection de l'expert. De plus, l'instance du descriptif en cours de définition (DS) est ajoutée dans la liste de dépendants du descriptif déjà instancié (IDS). Cette conception suit le schéma de conception *Observer* [GHJV95, ABW98].

2.2 Exécuter les procédures composées

La partie basse de la Figure 8 comportent les abstractions (figurant sur une couleur de font gris clair) qui modélisent l'exécution de procédures. Ce modèle, suivant le schéma de conception *Type Object* [JW97], associe à chaque abstraction du méta-niveau définition, une abstraction du méta-niveau exécution. Le rôle de ce dernier est la prise en charge de l'interprétation d'instances de son homologue du méta-niveau définition.

Ici, chaque composant de micro-composition est associée à une *activation de micro-composition* (`MicroCompositionActivation`)[47]. Celle-ci comporte à son tour une collection d'activations d'instances de descriptifs de service (`ServiceEvaluationActivation`). Chacune de ces instances contient la valeur issue de l'exécution effective du service associé (envoie du message `evaluate`). Les activations d'instances de descriptifs de service sont alors ici des *Value Holders* [Rie97a, FY98b].

A titre d'exemple, à chaque activation de la procédure `Traiter les agios du jour()`, une instance de `MicroCompositionEvaluation`[48] est créée. Celle-ci référence six autres instances de `ServiceEvaluationActivation`. Chaque instance de cette dernière porte la valeur qui résulte de l'exécution effective du service concerné. Chaque service correspond à l'un des calculs récapitulés par le Tableau 2, page 36 de l'introduction.

L'exécution d'un service s'appuie sur un autre objet dont le rôle est de mettre en œuvre la stratégie d'exécution approprié (`ServiceActivationStrategy`). Celui-ci est conçu suivant le schéma de conception *Strategy* [GHJV95]. Le choix du type de cet objet appartient au descriptif de service (cf. le Tableau 1, exposé dans l'introduction). Le descriptif de service a également la responsabilité de fournir des informations nécessaires à la vérification du type des arguments et le résultat de calcul.

Par ailleurs, avant de lancer un calcul, le système évalue d'abord chacun de ses arguments (message `value`). Aussi, à titre d'exemple, avant d'activer la procédure `Calculer les agios journaliers()` (cf. Figure 1, page 36), notre système procède à l'activation du calcul de ses arguments,

[44] Par exemple le système CALIBRES structure le programme dans une matrice à une colonne (comme une liste) et affecte une nouvelle instance de descriptif de service à la dernière cellule libre de cette colonne.

[45] Le choix de la cellule d'affectation correspond au choix du canal de sortie du composant, dont nous avons brièvement parlé dans la sous-section 2.3.4.1 de l'introduction.

[46] Le choix des arguments correspond au choix des canaux en entrée du composant, dont nous avons brièvement parlé dans la sous-section 2.3.4.1 de l'introduction.

[47] Le même comportement est hérité par une micro-composition qui est une sous-classe de composant de micro-composition, suivant le schéma de conception Composite (cf. Figure 8).

[48] Pour simplifier, nous faisons ici l'abstraction de l'instance de `MicroCompositionEvaluation` qui correspond à l'objet du méta-niveau définition micro-composition (`MicroComposition`).

c'est à dire `Obtenir Solde` (C23), `Obtenir Taux préférentiel de calcul d'agios` (C24) et `Obtenir Montant de découvert forfaitaire` (C25).

L'activation du calcul `Obtenir Solde` à son tour requiert l'activation au préalable du calcul `Obtenir Compte chèque associé`.

Cette modélisation explicite de l'exécution des procédures constitue la fondation requise à la mise en œuvre du lien causal (cf. §1.2, page 17 de l'introduction). Cette approche constitue une condition nécessaire à l'outillage satisfaisant du lien causal.

2.3 Préparer la composition

Dans ce cadre, le point de départ de la programmation par l'expert est l'initialisation du référentiel de services (cf. §2.3.4.3, page 35). Cette tâche est réalisée par des programmeurs lors de la création du langage d'experts et cela en collaboration avec des experts. Le rôle des experts consiste à déterminer et à spécifier le jeu initial de descriptifs de service qui sont nécessaires à la composition. C'est "l'habillage" de fonctions primitives dont nous avons parlé dans le paragraphe 2.3.4.1 de l'introduction.

Le jeu initial de descriptifs de service peut, toutefois, évoluer lors de l'exécution. Nous reviendrons sur ce sujet lors de la construction de notre solution finale par le couplage des deux systèmes DARC & DART dans la section 2.5, page 128 du chapitre III.

2.4 Macro-procédures et appels de sous-procédures

Une macro-procédure est une procédure appelée lors de la définition d'autres procédures. C'est donc avant tout une procédure. Nous utilisons une appellation différente surtout pour insister sur le fait que du point de vue des experts il n'y a pas d'intérêt à ce que n'importe quelle procédure puissent être réutilisée et appelée comme une sous-procédure.

Une macro-procédure marque, en effet, la volonté des experts de définir des procédures qui seront appelées ultérieurement.

Cette possibilité est une caractéristique importante de notre outillage. C'est grâce à elle que les langages d'experts assurent la *composition dynamique de services définis eux-mêmes dynamiquement*.

Un exemple d'une telle composition est énoncé dans l'introduction, le paragraphe 2.3.5.2, page 37. Sa mise en œuvre technique est esquissé dans le paragraphe 2.4.5, page 46. L'exposé de la mise en œuvre effective de cet exemple se trouve dans le paragraphe 3.7, page 167 du chapitre IV.

2.4.1 Arguments comme un type particulier d'instances de descriptifs de service

Sur le plan fonctionnel, le rôle d'un argument ne change pas ici par rapport à son rôle classique (exprimer un transfert de données d'une procédure à l'autre). Seulement, notre analyse est que du point de vue des experts il est important que l'introduction des arguments n'engendre pas de complexité au niveau de la composition.

Nous proposons donc de concevoir les arguments comme des instances de descriptifs de service. Notre objectif est d'homogénéiser la composition : l'expert annonce la nécessité d'un argument de la même façon qu'il instancie d'autres descriptifs de service. De plus, il les utilise à leur tour comme argument lors de création d'autres instances de descriptifs de service.

A titre d'exemple, la Figure 2, page 38 de l'instruction montre la définition de la procédure `Cumuler les agios du jour()` qui nécessite deux arguments en entrée : `Le Compte` et `Le Montant`.

Aussi, sur le plan technique, une macro-procédure se distingue d'une procédure par le fait qu'elle nécessite la gestion des arguments en entrée[49].

2.4.2 Habiller des procédures par des descriptifs de service

Toujours dans le soucis d'homogénéiser la composition, nous proposons d'habiller les macro-procédures par des descriptifs de service. Cela permet de ramener la définition d'un appel de procédure à une simple instanciation de descriptif de service.

Le descriptif de service aura par défaut le même nom que la procédure. La liste des arguments du descriptif peut être calculée automatiquement à partir des arguments utilisés lors de la définition de la procédure. En cas d'absence d'argument le descriptif n'aura pas d'argument non plus.

Par exemple, le descriptif de service associé à la procédure `Cumuler les agios du jour()`, s'appelle `Cumuler les agios du jour` et aura deux arguments.

Pour obtenir une assistance au niveau du choix des arguments effectifs lors d'une future instanciation d'un tel descriptif, il faut pour chaque argument préciser son type.

Il est donc possible d'automatiser la génération de ce descriptif, dès lors que l'expert exprime sa volonté de considérer une procédure comme une macro-procédure. En effet, toutes les informations nécessaires sont déjà connues d'avance.

2.4.3 Appeler des sous- procédures

Lorsqu'un descriptif de service est associé à une (macro-)procédure, il devient alors possible de définir, de façon intuitive, des appels de procédures lors de la définition de nouvelles procédures.

Il suffit, en effet, de procéder de la même façon que nous l'avons décrit brièvement dans le paragraphe 2.3.4.1, page 33 de l'introduction, à propos de l'instanciation de descriptifs de service.

La section 3.7, page 167 du chapitre IV expose en détail l'appel de la procédure `Cumuler les agios du jour()` par la procédure `Traiter les agios du jour()`.

2.4.4 Exécuter les appels de sous-procédures

Lors de l'activation d'une procédure et lorsqu'il existe des appels d'autres procédures, il faut calculer la valeur des arguments et les passer à la procédure appelée.

Nous proposons de s'appuyer ici à nouveau sur le descriptif de service qui connaît le nombre d'arguments requis et leur nom (voire leur ordre). Une solution possible est alors de calculer la valeur effective de chaque argument (connue de la procédure appelante) et de la stocker dans le contexte initial d'exécution de la procédure appelée. Pour ce stockage nous proposons d'utiliser le nom de l'argument, tel qu'il est connu du descriptif associé.

La procédure appelée commence son exécution, lorsque des arguments existent, par l'évaluation de ses arguments en entrée. L'évaluation d'un argument correspond à chercher la valeur se trouvant dans le contexte initial d'exécution. Pour se faire chaque argument accède au contexte en utilisant son propre nom comme clé.

On suppose donc ici, naturellement, qu'il y a une concordance entre la sous-procédure appelée et le descriptif de service qui l'habille : le nombre d'arguments (au sens de descriptif d'argument d'un descriptif de service) définis au sein du descriptif, leur nom et type correspondent bien au nombre

[49] Il est important de préciser qu'en pratique la plupart des procédures se trouvent dans cette situation. En effet, nous considérons l'instance courante d'une adaptation (ce qui correspond au receveur, `self` ou `this`, dans les langages à objets) comme le premier argument d'une procédure (quand cela est explicitement demandé par l'expert).

d'arguments (cette fois au sens du descriptif de service de service du type *argument*), leur nom et type qui existent au sein même de la définition de la sous-procédure appelée (et habillée par ce descriptif).

Il importe de préciser ici que le Micro-workflow ne dispose pas des mécanismes requis pour la mise en œuvre des appels de procédures, telle que nous venons de le décrire. Il s'appuie sur le langage d'implantation.

3 DART : le modèle de conception

Le modèle d'analyse du système DART que nous venons d'exposer se décline sous forme d'un modèle de conception dont les composants sont décrits ci-dessous. Rappelons que ce modèle est conçu pour une implantation à l'aide du système VISUALWORKS [Cin01].

3.1 Micro-compositions

Nous concevons une micro-composition comme une instance de la classe `MicroComposition`. Celle-ci hérite de la classe `MicroCompositionComponent` qui hérite de la classe `Object`.

La classe `MicroCompositionComponent`

Le rôle principal de chaque composant consiste à gérer une matrice. Pour ce faire, la classe `MicroCompositionComponent` utilise sa variable d'instance `compositionSheet`. Celle-ci référence sur une instance de la classe `TwoDList`. La méthode de classe `getSheet:` permet de créer la matrice. Le nombre de lignes et de colonnes de cette matrice est passé en argument, sous forme d'un point (instance de la classe `Point`).

Cette classe offre deux protocoles. Le premier regroupe les méthodes utilisées lors de la composition et le second les méthodes utilisées lors de l'activation.

Le rôle de chaque méthode du protocole de composition est de permettre de créer une instance de descriptif de service d'un type particulier. Le Tableau 1, page 34 récapitule les différents types de descriptif que nous avons rencontrés jusqu'alors.

Le Tableau 6 ci-dessous décrit la correspondance entre les différents types de descriptifs de service répertoriés par le Tableau 1, page 34 et les méthodes du protocole de composition mentionné ci-dessus.

Type de descriptif de service	Méthode
Méthode	addMethod:of:with:at:
Méthode statique	addStaticMethod:of:with:at:
Primitive externe	addExternalPrimitive:of:with:at:
Getter	addGetter:of:with:at:.
Setter	addSetter:of:with:at:
Procédure	addProcedure:of:with:at:
Argument	addInPin:at:
Component Factory	addComponentFactory:of:with:at:

Tableau 6 : Protocole d'instanciation des différents types de descriptifs de services.

Toutes les méthodes reçoivent les mêmes arguments :

1. le nom du descriptif de service à instancier
2. l'adaptation qui contient le descriptif de service à instancier. Ces deux premiers arguments permettent de retrouver le descriptif.
3. la liste des arguments requis pour la création d'une instance. Ces arguments sont eux-mêmes des instances de descriptifs de service.
4. les coordonnées en x et y (sous forme d'un point) de la cellule à laquelle l'instance créée doit être affectée.

Des exemples d'usage de ce protocole sont fournis dans la section 3.6, page 166 du chapitre IV.

Le protocole d'activation de cette classe comporte les méthodes suivantes :

1. La méthode `execute` active le calcul de toutes les cellules de la matrice, colonne par colonne. Il est important de noter que le lancement de ce calcul peut déclencher le calcul de la valeur de ses arguments, lesquels peuvent déclencher d'autres calculs se trouvant après par rapport à la position de l'instance de descriptif en cours de calcul.
2. La méthode `execute:` active le calcul associé à la cellule dont les coordonnées sont passées en argument.
3. La méthode `executeFromHere:` applique la règle décrite dans le cas de la méthode `execute`, mais en partant de la position reçue en argument.
4. La méthode `executeSelection` active le calcul de la sélection courante de cellules.

La class `MicroComposition`

La classe `MicroComposition` permet d'enchaîner un ensemble de composants. Pour ce faire il comporte une variable d'instance `microCompositions`. Celle-ci référence une collection ordonnée (`OrderedCollection`) qui contient l'ensemble des composants dans l'ordre de leur ajout.

Cette classe réimplante le protocole de la classe `MicroCompositionComponent` exposé ci-dessus. La nouvelle implantation consiste à itérer sur la collection des composants et à envoyer le message reçu à chacun des composants.

3.2 Instances de descriptifs de service

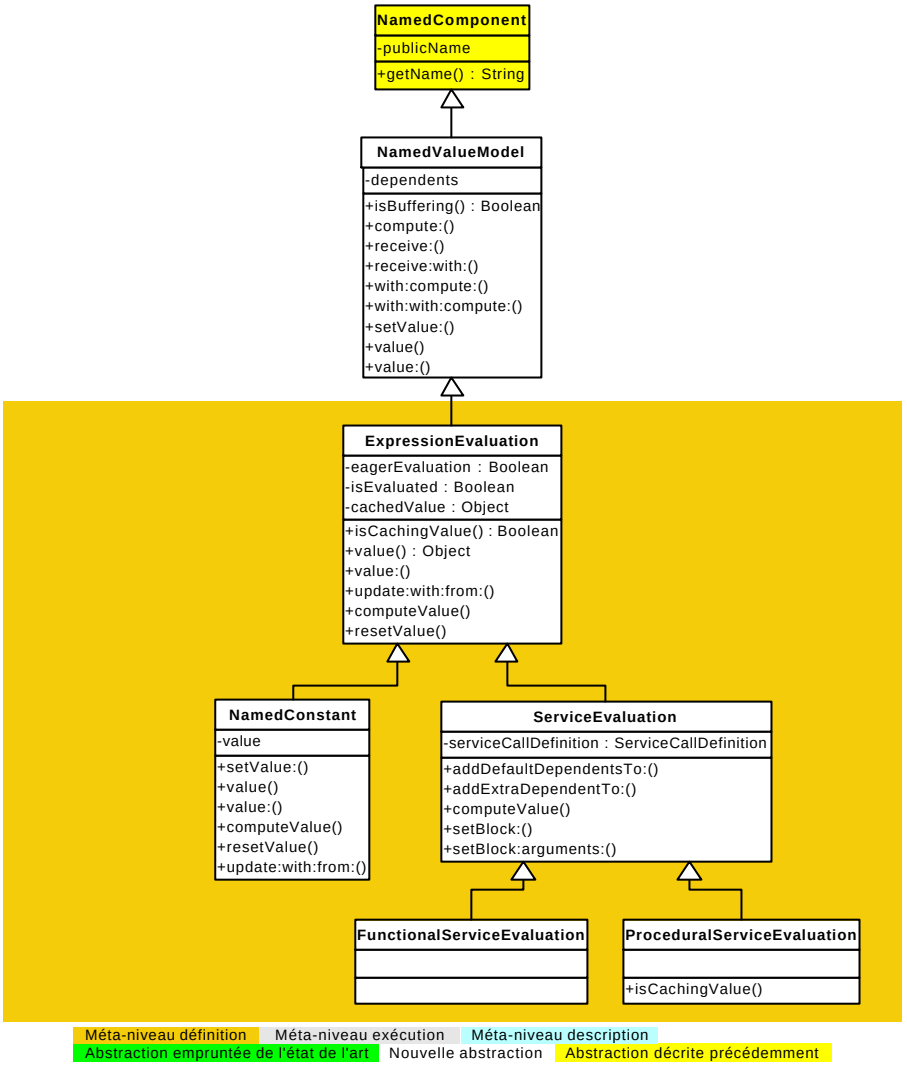


Figure 9 : Modèle de conception des instances de descriptifs de service.

La seconde partie de notre conception concerne les instances de descriptifs de service, autrement dit, les différentes représentations de calcul qui peuvent être affectées à une cellule. Nous poursuivons notre exposé ainsi car ce sont ces objets qui sont associés aux cellules de chaque composant, tel que nous venons de les présenter.

La Figure 9 ci-dessus illustre notre modèle de conception. Ce qui suit expose la structure et le comportement des différentes classes de ce modèle.

La classe `NamedValueModel`

La classe abstraite `NamedValueModel` hérite de la classe `NamedComponent`. Son rôle est de mettre en œuvre la gestion des dépendances entre la définition d'un calcul et ses arguments. Pour ce faire elle comporte la variable d'instance `dependents`. Celle-ci est une instance de la classe `DependentsCollection` (empruntée du système VISUALWORKS).

Le protocole de cette classe est également inspiré des classes `Model` et `ValueModel` du système VISUALWORKS. Cela comporte les méthodes d'ajout et de suppression de dépendances. La méthode principale de cette classe est la méthode abstraite `value`. C'est elle qui déclenche la réalisation du calcul qu'il représente. Son implantation concrète dépend des sous-classes (cf. ci-bas).

La méthode `value:` joue également un rôle important. C'est elle qui stocke le résultat d'un calcul et qui avertit les dépendent d'une instance de descriptif de service du changement de valeur.

La classe `ExpressionEvaluation`

La classe `ExpressionEvaluation` est également une classe abstraite, sous-classe de la classe `NamedValueModel`. Elle est également une adaptation de la classe `ComputedValue` du système VISUALWORKS.

`ExpressionEvaluation` comporte trois variables d'instances :

1. `eagerEvaluation` est une valeur booléenne qui indique si la valeur de l'expression doit être immédiatement recalculée après la réception d'un message de mise à jour (cf. ci-dessous, la méthode `update:with:from:`).
2. `isEvaluated` est également une valeur booléenne qui indique si la valeur de l'expression a été calculée.
3. `cachedValue` comporte le résultat de l'évaluation de l'expression.

Comme permet de le constater la Figure 10 ci-dessous, cette classe donne une implantation concrète de la méthode `value`. Celle-ci déclenche l'évaluation de l'expression que le receveur représente, si cette expression n'a pas été évaluée (le test `isEvaluated`). Après le calcul procède au stockage du résultat de calcul (affectation à la variable d'instance `cachedValue`), s'il y a lieu (le test `isCachingValue`).

A la fin de ce calcul il prévient ses dépendants du changement de sa valeur. Les dépendants sont eux-mêmes des instances des sous-classes de `ExpressionEvaluation`. Lorsqu'un tel objet reçoit le message de mise à jour (`update:with:from:`), il libère sa valeur courante et déclenche (si la mise à jour immédiate est demandée via `eagerEvaluation`) un nouveau calcul de sa valeur (par la méthode `resetValue`).

```
value
	"Answer the cached value for the receiver. If the value is unknown,
	then compute the value."

	| aProcedureActivation |
	self isEvaluated ifFalse:
		[[aProcedureActivation := self computeValue] ensure:
			[isEvaluated := true].
		self isCachingValue ifTrue:
			[cachedValue := aProcedureActivation localState: self getName].
		self changed: #value].

	^cachedValue
```

Figure 10 : Déclenchement d'un calcul et le stockage du résultat.

Cette classe est abstraite car elle délègue à ses sous-classes l'implantation de la méthode de calcul `computeValue`.

La classe `NamedConstant`

La sous-classe concrète la plus simple, mais toutefois importante, de la classe `ExpressionEvaluation` est la classe `NamedConstant`. Celle-ci modélise les constantes. Elle comporte une variable d'instance `value` qui référence la valeur constante utilisée dans la définition d'une procédure (du type micro-composition)[50].

Dans la mesure où la valeur est fournie, cette classe ré-implante la méthode `value` qui retourne simplement la valeur stockée dans la variable d'instance du même nom.

Par ailleurs, les constantes n'ont pas de dépendants. Elles ne réagissent donc pas à la méthode de mise à jour `update:with:from:`.

La classe `ServiceEvaluation`

La sous-classe la plus largement utilisée à l'heure actuelle est la classe `ServiceEvaluation`. C'est elle qui assure la transition entre le stockage de la définition, le déclenchement du calcul et le stockage du résultat d'une expression (rôle des instances de descriptifs de service) et la définition elle même d'une expression.

En effet, `ServiceEvaluation` ajoute à sa super-classe `ExpressionEvaluation` une variable d'instance `serviceCallDefinition`. Celle-ci, conçue suivant le schéma *Bridge* [GHJV95], permet de dissocier l'abstraction d'un calcul de ses implantations.

Pour ce faire, comme permet de le constater la Figure 11 ci-dessous, la classe `ServiceEvaluation` donne une implantation concrète de la méthode `computeValue` . Celle-ci consiste à envoyer systématiquement le message `execute` à l'objet qui représente le calcul désiré et à retourner le résultat de cette exécution. C'est cette modélisation qui permet de stocker dans une cellule une expression aussi bien décrite comme un micro-procédé qu'une micro-composition ou encore un arbre de syntaxe abstraite.

```
ServiceEvaluation >>computeValue
      ^serviceCallDefinition execute
```

Figure 11 : Rendre la réalisation d'un calcul indépendant de sa représentation.

La classe `ServiceEvaluation` ajoute également une méthode au protocole d'initialisation des instances de descriptifs de service. En effet, comme l'illustre la Figure 12 ci-dessous, ses instances sont initialisée par la méthode `setBlock:arguments:` (nom inspiré de la classe `BlockValue` du système VISUALWORKS).

```
ServiceEvaluation >> setBlock: aServiceCallDefinition arguments: aCollection
      "Set the receiver's block to be aBlock and the arguments to be aCollection."

      self setBlock: aServiceCallDefinition.
      self addDefaultDependentsTo: aCollection
```

Figure 12 : Initialisation particulière des instances de la classe `ServiceEvaluation`.

Cette méthode a la particularité de faire appel à la méthode `addDefaultDependentsTo:`. Comme permet de l'illustrer la Figure 13 ci-dessous, celle-ci itère sur la liste des arguments et rend le receveur dépendant de chacun de ses arguments. De plus, il réalise une vérification supplémentaire afin

[50] A noter que le micro-workflow ne modélise pas les constantes.

d'éviter les relations circulaires. Il s'agit d'éviter que le calcul de l'expression A dépende du calcul de l'expression B et vise versa.

```
ServiceEvaluation >> addDefaultDependentsTo: aCollection
      | anotherServiceEvaluation newColl |
      1 to: newColl size do: [:i |
            anotherServiceEvaluation := newColl at: i.
            self myDependents notNil ifTrue:
                  [(self   myDependents   detect:   [:anObject   |   anObject   ==
anotherServiceEvaluation] ifNone: []) notNil ifTrue:
                        [self notifyError: #'Cicularity']].
            anotherServiceEvaluation
                  removeDependent: self;
                  addDependent: self]
```

Figure 13 : Gestion de dépendances par la classe `ServiceEvaluation`.

La classe `ServiceEvaluation` dispose, par ailleurs, de deux autres sous-classes : `FunctionalServiceEvaluation` et `ProceduralServiceEvaluation`. L'idée consiste à différencier les instances de descriptifs de service dont l'activation retourne une valeur (tu type fonctionnel) ou pas (du type procédural).

Pour ce faire, la classe `ProceduralServiceEvaluation` ré-implante la méthode `isCachingValue` pour renvoyer systématiquement faux (`false`). Rappelons que cette méthode, initialement implantée dans la classe `ExpressionEvaluation` retourne par défaut systématiquement vrai (true).

3.3 Stratégies d'activation

L'activation de services s'appuie sur un modèle qui distingue différents types d'exécution. Celui-ci est conçu suivant le schéma de conception *Strategy*. La Figure 14 illustre la hiérarchie de classes qui met en œuvre ce modèle.

La classe `AbstractServiceActivationStrategy` fédère les différents types de stratégies qui comporte ce modèle. Ce dernier distingue deux "grandes" catégories de stratégie d'activation : celle des services globaux et celle des services du type envoi de message.

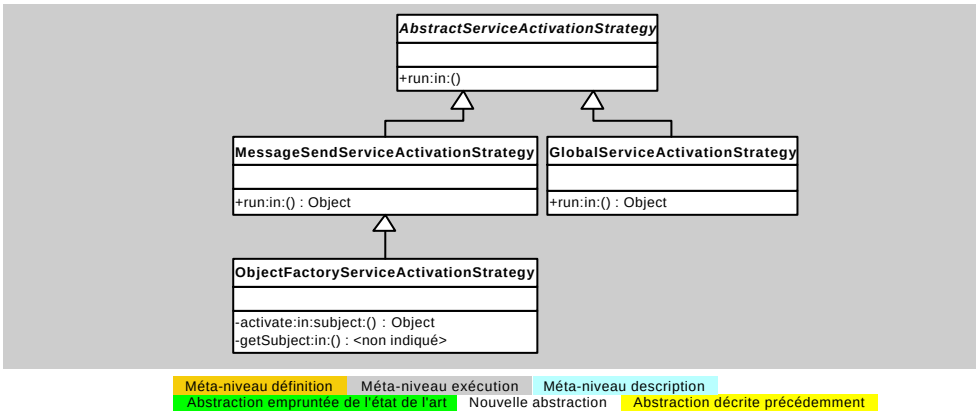


Figure 14 : Modèle de conception des stratégies d'exécution de DART.

Ce qui suit fournit une information plus détaillée sur la conception et l'implantation de ce composant de DART.

La classe `AbstractServiceActivationStrategy`

La classe abstraite `AbstractServiceActivationStrategy` implante simplement la méthode abstraite `run:in:`. C'est cette méthode qui déclenche la réalisation effective d'un service. Elle est implantée par différentes stratégies que nous allons exposer ci-dessous.

Cette méthode est déclenchée via la méthode `computeValue` de la classe `ServiceEvaluationDefinition` décrite ci-dessus.

La classe `MessageSendServiceActivationStrategy`

La première grande catégorie de stratégies d'exécution de services est modélisée par la classe `MessageSendServiceActivationStrategy`. Celle-ci offre un cadre pour une panoplie de services du type envoi de message. Un service de ce type se caractérise par l'existence d'un sujet, le destinataire du message, l'existence optionnelle d'une liste d'arguments ainsi que le nom du message à envoyer.

Toutefois, tous les services proposés ne sont pas de ce type. Notamment nous exposerons plus loin la classe `GlobalServiceActivationStrategy` qui modélise les services dont l'exécution s'appuie sur une approche non objet/envoi de message.

Sur le plan ergonomique il est important d'homogénéiser, du point de vue des experts, la définition et l'instanciation de ces différents types de service. Cela nous a conduit à inclure le nom du sujet dans la liste des arguments. En effet, le premier argument dans cette liste est interprété comme étant le sujet. Le reste des éléments, s'ils existent, constituent les arguments de l'appel. Le message à envoyer est connu du descriptif de service.

L'exécution d'un envoi de message est divisé en quatre étapes majeures :

1. la recherche du destinataire du message (le sujet)
2. l'évaluation des arguments
3. l'activation effective du message
4. le traitement du résultat, si celui-ci existe

La recherche du destinataire du message (le sujet) s'appuie sur l'hypothèse que celui-ci correspond à la valeur du premier argument. Cette stratégie est mise en œuvre par la méthode `getSubject:in:`.

Il est important de noter que cette méthode procède à une évaluation systématique du sujet ainsi que les arguments d'appel (message `value`), avant de procéder à la réalisation effective du calcul.

Cette implantation effectue également un contrôle de type sur le sujet. Un exposé détaillé de cet aspect sort du cadre de notre thèse. Toutefois, un cadre pour l'implantation du module dédié à la gestion des types métier est fourni en annexe VI.

L'étape suivante consiste en la recherche de la valeur effective des arguments. Cette recherche s'effectue dans le contexte courant d'exécution et cela grâce au nom de chaque argument, lequel est détenu par l'instance de descriptif de service. Chaque argument est évalué suivant le principe décrit ci-dessus pour le cas du sujet.

L'activation effective du message est réalisée à l'aide des primitives du langages (e.g. `perform:` en SMALLTALK et `invoke:` en JAVA).

Le traitement du résultat consiste à le stocker dans le contexte courant d'activation.

Cette stratégie d'exécution de services peut être spécialisée pour modéliser différents types d'envoi de messages. L'accès en lecture et en écriture aux attributs (et variables d'instances) et l'usage récursif des procédures dans la définition d'autres procédures en sont des exemples. Ces cas seront étudiés dans le chapitre III, paragraphe 2.5.2.1, page 133.

La classe `ObjectFactoryServiceActivationStrategy`

Les experts ont besoin d'exprimer dans leur procédure la création d'instances d'objets métier. Les langages d'experts rendent cette expression possible grâce à une stratégie d'activation bien particulière. Celle-ci est mise en œuvre par la classe `ObjectFactoryServiceActivationStrategy`.

Par exemple, la composition de services dans le cas du système CALIBRES se repose *uniquement* sur ce type d'expression. En effet, une procédure d'étalonnage est décrite comme une séquence d'expression de création d'objets qui servent ensuite d'argument pour réaliser la création d'autres objet. Chaque opération de création est précédée par des calculs ou des mesures effectuées par des robots spécialisés.

La solution que nous proposons ici s'appuie sur une particularité du langage cible SMALLTALK, c'est-à-dire les méta-classes. L'idée consiste à stocker le sujet, c'est à dire la classe à instancier, au sein d'un descriptif de service. En effet, dans ce cas le sujet est un objet connu lors de la définition. A cela peut évidemment, s'ajouter le nom des arguments.

Au niveau du protocole, cette classe réimplante principalement la méthode `getSubject:in:`. En effet, le sujet n'est pas ici recherché dans le contexte courant d'exécution mais au sein du descriptif de service associé (`^aServiceCall getServiceDescription getObjectFactory`).

La classe `GlobalServiceActivationStrategy`

Une autre stratégie d'exécution qui doit être envisagée est celle de services globaux, c'est-à-dire tout service qui n'est pas implanté par une classe. Cela permet notamment aux experts d'utiliser, à titre d'exemple, des fonctions d'une bibliothèque externe. Cette stratégie est mise en œuvre dans notre framework par la classe `GobalServiceActivationStrategy`.

L'exécution d'un service suit une procédure composée de deux grandes étapes :

1. appel de la fonction qui met en œuvre le service
2. le traitement du résultat de l'appel, si celui-ci existe.

L'implantation effective de l'appel de la fonction dépend du type de service. La méthode `primCall:in:` est donc une méthode abstraite. Une réalisation plus poussée de cette stratégie d'exécution n'a pas d'intérêt direct pour les travaux présentés dans ce mémoire et n'a pas été mise œuvre.

En ce qui concerne le traitement du résultat, ce dernier existe seulement si le service est du type fonctionnel. Dans ce cas, la définition de l'appel de service concerné dispose de la clé qui doit servir à stocker le résultat dans le contexte courant d'activation

3.4 Descriptifs de service

3.4.1 Première partie : classes abstraites

Nous arrivons enfin à l'exposé du concept central de la composition des procédures par des expert, c'est la notion de descriptif de service. Celui-ci a déjà été introduit dans la section 2.3.4, page 33 de l'introduction. Nous y avons annoncé trois foncions attachées aux descriptifs de service :

1. "habiller" les fonctions primitives créées par des programmeurs afin de cacher à l'expert des aspects purement informatique d'une fonction. L'objectif est de présenter les primitives aux experts avec une apparence métier.
2. fournir des informations utiles à la création d'une instance de descriptif de service, comme par exemple le nombre et le type des arguments.
3. fournir des informations utiles à l'exécution des services, comme la stratégie d'activation appropriée.

La Figure 15 ci-dessous illustre une première modélisation de cette notion. Deux nouvelles classes sont introduites. Il s'agit des deux classes abstraites `EnityDescription` et `AbstractServiceDescription`. Les méta-classes de celles-ci joue également un rôle dans notre conception.

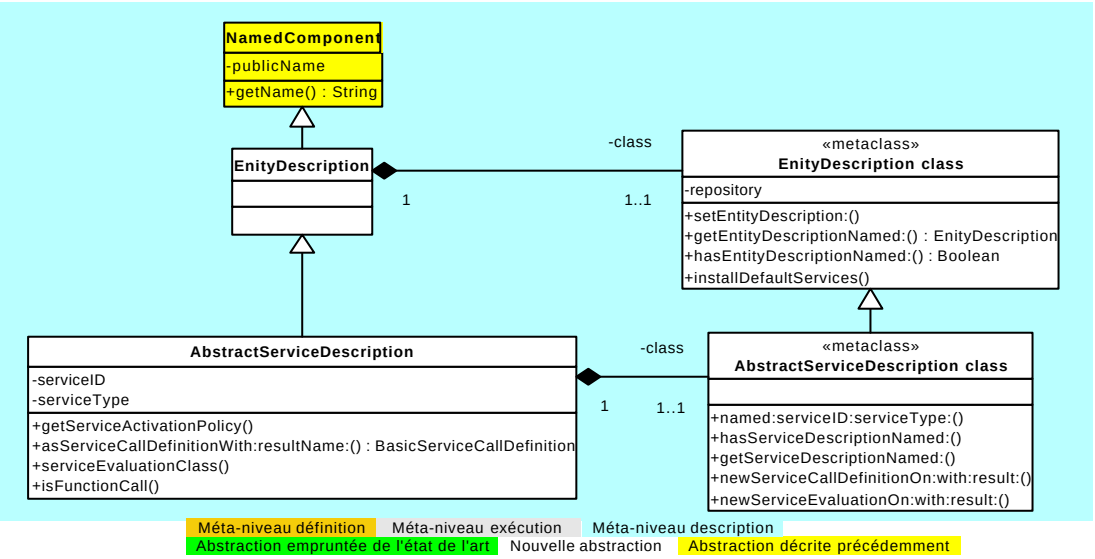


Figure 15 : Modèle abstrait des descriptifs de service.

La classe EnityDescription et sa méta-classe EnityDescription class

`EnityDescription` est une classe abstraite qui permet de fédérer les différentes classes qui représentent les descriptifs qui font partie de la conception de notre système. Le trait commun de toutes ces classes est la gestion d'un référentiel de descriptifs courants. C'est le contenu de ces référentiels qui constitue le point de départ de la programmation par les experts.

Dans notre mise ne œuvre en SMALLTALK nous utilisons les variables d'instances de classe pour allouer ce référentiel. La variable d'instance de classe `repository` permet à chaque classe de constituer un référentiel de descriptifs courant du système.

Le protocole de cette méta-classe est composé des méthodes suivantes :

1. La méthode `installDefaultServices` permet d'initialiser le référentiel de chaque sous-classe avec les descriptifs de service par défaut. Il s'agit des descriptifs prévus par les programmeurs pour constituer un point de départ du système. A titre d'exemple, la sous-classe `MethodDescription` comportera les descriptifs des méthodes d'instances (ou, le cas échéant, de classes) auxquels les experts doivent pouvoir avoir accès, comme le message `getBalance` dans le cas d'un compte bancaire.
2. Les méthodes `setEntityDescription:` et `getEntityDescriptionNamed:` permettent respectivement d'ajouter et de rechercher un descriptif au sein d'un référentiel.
3. La méthode `ensureBeingNewEntityDescription:` permet d'assurer qu'il n'existe pas déjà un descriptif portant le nom reçu en argument.
4. La méthode `hasEntityDescriptionNamed:` permet de vérifier l'existence d'un descriptif au sein d'un référentiel.

La classe `AbstractServiceDescription` et sa méta-classe

La classe `AbstractServiceDescription` est une sous-classe de la classe `EnityDescription`. Elle est une classe abstraite, la super-classe de toutes les classes dont les instances sont des descriptifs de service.

La variable d'instance `serviceID` de cette classe comporte le nom du service. La sémantique opérationnelle de cette information dépend du type de service. Pour un service du type envoi de message, par exemple, le `serviceID` correspond au nom de la méthode à invoquer.

La variable d'instance `serviceType` permet de fournir, lors de la création d'un descriptif de service, une information qui va permettre le choix de la bonne stratégie d'exécution.

Le protocole de la méta-classe `AbstractServiceDescription class` est la suivante :

1. Pour plus de clarté, la méthode `setEntityDescription` est redéfinie en `setService:`. Celui-ci permet d'ajouter un descriptif de service au référentiel de descriptifs de service du receveur.
2. De même, la méthode `getEntityDescriptionNamed:` est redéfinie en `getServiceDescriptionNamed:`. Celui-ci permet de rechercher un descriptif de service au sein du référentiel de descriptifs de service du receveur.

Le protocole de la classe `AbstractServiceDescription` comprend la méthode abstraite `getServiceActivationPolicy`. Celle-ci retourne l'objet qui met en œuvre la stratégie d'activation appropriée pour le type de service (cf. ci-dessus pour plus d'information sur les stratégies d'exécution).

3.4.2 Seconde partie : services

Le modèle abstrait mis en œuvre ci-dessus doit à présent être spécialisé pour modéliser les différents types de descriptifs de service que nous avons annoncés dans la section 2.3.4, page 33 de l'introduction. La Figure 16 ci-dessous illustre les abstractions ainsi ajoutées que nous allons détailler à présent.

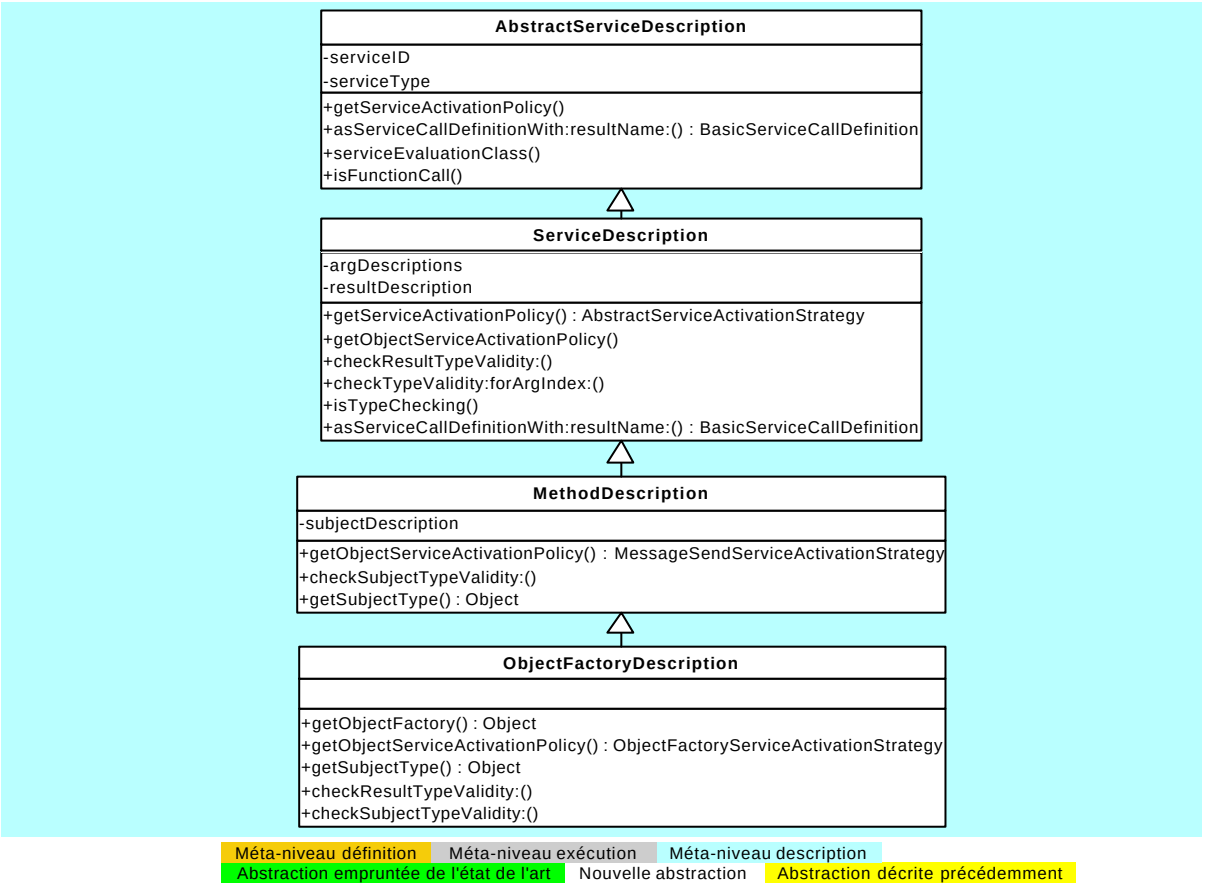


Figure 16 : Modèle des différents types de descriptifs de service utilisés dans DART.

La classe `ServiceDescription`

La classe `ServiceDescription` est la première sous-classe concrète de la classe `AbstractServiceDescription`. Celle-ci ajoute deux variables d'instances :

1. La variable d'instance `argDescriptions` comporte une collection d'instantes de la classe `ArgumentDescription`. Chaque instance de cette classe décrit le nom et le type d'un argument.
2. La variable d'instance `resultDescription` comporte une instance de la classe `ResultDescription`. Celle-ci décrit le nom et le type du résultat.

Cette classe joue un rôle particulièrement important dans la mise en œuvre de la composition. C'est elle qui implante le protocole d'instanciation des descriptifs de service :

1. La méthode `asDefaultExpressionEvaluationWith:` crée une instance de descriptif de service dont le résultat est affecté à la clé `#me`. C'est ainsi que les experts sont dispensés de nommer le résultat d'un calcul qui doit être utilisé comme receveur du message lors d'un calcul ultérieur. La clé `#me` est, en effet, recherchée par défaut lors du calcul du sujet dans les envois de message.
2. La méthode `asExpressionEvaluationWith:result:` crée effectivement une instance du receveur. Le nom du résultat ainsi que les arguments lui sont également transmis.

Les autres méthodes du protocole de cette classe sont les suivantes :

1. La méthode `getServiceActivationPolicy` retourne une instance de la stratégie d'activation qui est appropriée pour l'exécution de ce service. La stratégie d'activation par défaut est `GlobalServiceActivationStrategy`.
2. La méthode `isFunctionCall` retourne vrai (`true`) si le descriptif comporte des informations sur le résultat du calcul et faux (`false`) sinon.

A titre d'exemple, le descriptif de service appelé `Get Balance` de la Figure 17 décrit un service du type d'envoi de message (`getBalance`). L'utilisateur doit, lors de la définition d'un appel de ce service fournir le nom du destinataire de ce message (ici `balance`). Ce nom sera transformé en un objet par la recherche dans le contexte courant d'exécution. Le message sera alors effectivement adressé à cet objet par des mécanismes d'invocation de message calculés comme `perform:` dans le cas du langage SMALLTALK ou `invoke:` dans le cas du langage Java.

```
^self
      named: 'Get Balance'
      serviceID: #getBalance
      serviceType: #messageSend
```

Figure 17 : Exemple de création d'un descriptif de service.

Le rôle principal des autres sous-classes de cette classe consiste à redéfinir la méthode `getServiceActivationPolicy` afin de retourner la stratégie d'activation adéquate. Le Tableau 7 ci-dessous décrit la correspondance entre les types de descriptif de service et la stratégie d'activation correspondante.

Le type du descriptif de service	La stratégie d'activation
`MethodDescription`	`MessageSendServiceActivationStrategy`
`ObjectFactoryDescription`	`ObjectFactoryServiceActivationStrategy`
`ComponentFactoryDescription`	`ComponentFactoryServiceActivationStrategy`
`ComponentAccessorDescription`	`SetterServiceActivationStrategy` ou `GetterServiceActivationStrategy`
`MicroProcessDescription`	`MicroProcessServiceActivationStrategy`

Tableau 7 : Table de correspondance entre les descriptifs de service et les stratégies d'activation.

Certains de ces types de descriptifs de service ne sont pas encore décrits et seront exposés lors de la description des extensions de leur modèle dans le chapitre III. Il s'agit des classes `ComponentFactoryDescription`, `ComponentAccessorDescription`, et `MicroProcessDescription`.

3.5 Macro-procédures et appels de sous-procédures

Afin de faciliter la compréhension de la mise en œuvre des différents aspects de l'outillage des macro-procédures, nous regroupons ici l'exposé de tous les éléments qui relèvent de ce travail.

Notre exposé suit le même plan que notre analyse présentée au paragraphe 2.4, page 60 du même chapitre.

3.5.1 Arguments

Pour mettre en œuvre l'idée d'homogénéiser les arguments avec les instances de descriptifs de service (cf. le paragraphe 2.4.1, page 60 ci-dessus), nous concevons la classe `InPinEvaluationDefinition` comme une spécialisation de la classe `ExpressionEvaluationDefinition`.

Celle-ci ajoute une nouvelle variable d'instance `type` afin de stocker, optionnellement, pour chaque argument le type de la valeur attendue.

De plus elle implante le protocole suivant :

1. La méthode `computeValueIn:` reçoit le contexte initial d'activation en argument et cherche sa valeur dans ce contexte à l'aide de son nom (`^aProcedureActivationOrNil localState: self getName`).
2. La méthode `isInPin` qui retourne systématiquement vrai (`true`) permet de distinguer ce type d'instance de descriptif des autres. Pour ce faire, nous avons également ajouté cette méthode dans la classe `ExpressionEvaluationDefinition`. Dans ce dernier cas elle retourne systématiquement faux (`false`).
3. La méthode `asArgDescription` remplit la fonction importante de conversion des instances de cette classe en instance de la classe `ArgumentDescription`. Cette conversion intervient lors de l'habillage de procédures par des descriptifs de service (cf. paragraphe suivant).

3.5.2 Habillage

Pour habiller une procédure avec un descriptif de service, nous proposons la méthode `asServiceDescriptionNamed:`[51]. Comme l'illustre le script SMALLTALK de la Figure 18 ci-dessous, cette éthode est implantée par la classe `MicroCompositionComponent`.

L'algorithme consiste à créer une instance de la classe `MicroProcessDescription`. Celle-i modélise les descriptifs de services de type procédure (cf. le paragraphe 3.4.2, page 72 ci-dessus). Le nom de ce descriptif est reçu en argument. C'est le même nom qui est aussi utilisé comme le nom interne qui sert au stockage de ce descriptif dans le référentiel correspondant.

```
MicroCompositionComponent >> asServiceDescriptionNamed: aString
      | aMicroProcessDescription |
      aMicroProcessDescription := MicroProcessDescription
            named: aString
            serviceID: aString asSymbol
            resultType: nil.
      aMicroProcessDescription setArgDescriptions: self getArgDescriptions.
      ^aMicroProcessDescription
```

Figure 18 : Habillage d'une procédure.

[51] Pour un exemple, veillez voir au niveau du code source, la méta-classe `CompteBancaire class`, la méthode `habillerProcedureCumulerLesAgiosDuJourEnDescriptifDeProcedureDe:`.

Enfin, la méthode `getArgDescriptions` est invoquée. Celle-ci itère sur les instances de descriptif de service de la procédure qui sont du type argument (cf. méthode `isInPin` ci-dessus). Il envoie ensuite à chaque argument le message `asArgDescription`, que nous venons également de décrire ci-dessus (cf.).

```
MicroCompositionComponent >> getArgDescriptions
	| inPinCollection args |
	self hasAnyInPin ifFalse: [^nil].
	inPinCollection := self inPinCollection.
	inPinCollection size = 1 ifTrue:
		[^inPinCollection first asArgDescription].
	args := inPinCollection collect: [:aPin | aPin asArgDescription].
	^ArgumentDescriptionCollection withArgCollection: args
```

Figure 19 : Transformation des arguments.

3.5.3 Appel de sous-procédures

L'ajout d'un appel de procédure dans la définition d'une autre procédure se fait par l'invocation de la méthode `addInPin:at:` exposée dans le paragraphe 3.1, page 62 ci-dessous et récapitulé par le Tableau 6, page 62.

C'est elle qui crée une instance de la clase `InPinEvaluationDefinition` en fournissant son nom. De façon optionnelle il est aussi possible d'y associer un type.

3.5.4 Activation

L'activation des macro-procédures s'appuie principalement sur deux méthodes. La première, cf. la Figure 20 ci-dessous, est la méthode `activate:in:subject:` implantée par la stratégie d'exécution des procédures (classe `MicroProcessServiceActivationStrategy`, cf. ci-dessus, paragraphe 3.3, page 67).

```
MicroProcessServiceActivationStrategy >> activate: aServiceCall in:
aProcedureActivation subject: theSubject
	| theLastProcedureActivation |

	theLastProcedureActivation := theSubject getType
		run: aServiceCall getServiceID
		in: nil
		initialContext: (aServiceCall getServiceDescription
buildActivationInitialContextUsing: aServiceCall).

	^aServiceCall getServiceDescription isFunctionCall
	ifTrue: [theLastProcedureActivation
		localState: aServiceCall getServiceDescription nameOfTheResultSlot]
	ifFalse: []
```

Figure 20 : Activation d'une macro-procédure.

Le fonctionnement de celle-ci est basée sur la préparation du contexte initial d'activation. C'est lui qui est utilisé par les arguments afin de rechercher leur valeur courante (cf. ci-dessus, le paragraphe 3.5.1, page 74, la méthode `computeValueIn:`).

Le calcul de ce contexte est confié à la méthode `buildActivationInitialContextUsing:` de la classe `MicroProcessDescription`. C'est, en effet, le descriptif de service qui prend ici en charge ce calcul. Pour ce faire, il met en parallèle les descriptifs d'arguments, qu'il contient lui-même, avec les arguments effectifs qui sont contenus dans l'instance de descriptif de service qui représente l'appel de la sous-procédure.

```
MicroProcessDescription >> buildActivationInitialContextUsing: aServiceCall
      | contextDic |
      contextDic := Core.IdentityDictionary new.
      self getArgDescriptions getArgCollection with: aServiceCall arguments do:
            [:argDesc :effectiveArg |
                  contextDic add: (argDesc getName -> (effectiveArg value))].
      ^contextDic
```

Figure 21 : Algorithme de calcul du contexte initial d'appel d'une sous-procédure.

4 Exemple : adaptation de comptes bancaires

A son stade actuel de développement, notre outillage reste encore trop abstrait et incomplet pour permettre de mettre effectivement en œuvre notre exemple en cours d'adaptation de comptes bancaires (cf. l'introduction, le paragraphe §2, page 31).

En effet, même si la mécanique de base de la composition de procédures par des experts est mise ici en place, il nous reste toutefois à concrétiser cette infrastructure par l'apport des éléments qui seront notamment en mesure de prendre en charge la définition dynamique de procédures qui opèrent sur des structures définies également dynamiquement.

En effet, DART se charge de la composition par des experts des descriptifs de service. Aussi, pour l'heure, notre outillage n'a aucune notion de cette co-évolution dynamique de structures et de procédures, qui sera, en effet, le contexte dans lequel nous allons nous servir ici de DART. Cela ne signifie nullement qu'il s'agit du seul et unique cas d'usage possible de DART.

Pour toutes ces raisons, nous sommes donc amenés à retarder l'illustration du fonctionnement de notre outillage jusqu'au chapitre III, paragraphe 3, page 136, mais, surtout le chapitre IV paragraphe 3, page 158.

5 Conclusion

Ce chapitre décrit en détail les modèles d'analyse et de conception du framework DART. Après une analyse détaillée des deux parties du modèle de ce système, l'une dédiée à la définition et l'autre à l'activation des procédures, nous avons décrit l'architecture de ce système à travers l'exposé des quatre modèles qui le composent :

1. Le modèle de la composition ;
2. Le modèle de l'instanciation des descriptifs de service ;
3. Le modèle des stratégies d'activation de services ; et
4. Le modèle des descriptifs de service.

Nous avons également présenté ici le modèle de l'outillage de l'appel des sous-procédures, à travers la notion de macro-procédure. Là encore les descriptifs de service ont joué un rôle primordial.

Nous avons, par ailleurs, suggéré de regrouper les abstractions de ces modèles dans trois catégories d'objets de méta-niveau : le méta-niveau de définition, de description et d'exécution.

Ce travail constitue un élément central de notre étude. En effet, suivant les recommandations de B. A. Nardi, le framework présenté ici outille la composition de procédures par des experts ainsi que leurs activations.

Dans les chapitres suivants nous allons montrer comment il est possible de coupler ce système au système DARC, présenté au chapitre III, afin d'outiller la création de langages d'experts.

Chapitre II : Les AOMs : DOM et Micro-Workflow

Chapitre II : Les AOMs : DOM et Micro-Workflow

1 Introduction

La principale fonction des langages d'experts est d'assurer la spécialisation dynamique de classes et cela par des experts. En terme d'outillage[52] ceci se traduit sous forme d'un framework orienté-objets *documenté*, dédié à l'ajout lors de l'exécution par des experts de nouveaux types d'objets (des sous-classes), leur structure et procédures.

Les travaux les plus récents en matière de cet outillage sont réalisés par l'équipe de Ralph Johnson à UIUC sous l'appellation *Adaptive Object-Models* (AOMs) [JW97, Joh98, FY98a, FY98b, MJ99c, YBJ01b]. Ils se poursuivent depuis 1998 suivant deux axes : 1) l'analyse des systèmes existants en vue de l'extraction de leurs schémas de conception ; et 2) la création de frameworks orienté-objets documentés, sur la base notamment des analyses et expériences menées en (1)[53].

Les deux résultats majeurs de cette étude, toujours en terme d'outillage, sont jusqu'alors :

1. le schéma de conception *Dynamic Object Model* (DOM) [RTJ00], plus récemment présenté également sous forme d'un style architectural[54] [YBJ01b], qui décrit la conception du cœur des AOMs ; ainsi que
2. le Micro-workflow [Man00], un framework orienté-objets documenté, dédié à la création d'applications *flow-independent*, comme une approche pour la réalisation de logiciels dont le comportement peut évoluer lors de l'exécution.

Dans la suite de ce chapitre, après une brève description des principales caractéristiques des AOMs et leurs rapports avec les langages d'experts, nous présentons succinctement les deux travaux mentionnés ci-dessus et concluons sur le problème de couplage, c'est-à-dire le problème de l'outillage de la co-évolution dynamique de structures et de procédures par le couplage de DOM et de Micro-workflow.

---

[52] au sens défini dans le paragraphe 1.3.1, page 21 du chapitre I.

[53] Johnson lui même observe la nécessité des expériences concrètes pour créer des frameworks orienté-objets [MJ98a, JF88]. Il se retrouve ici face à un problème extrêmement difficile à traiter car à notre connaissance, à ce jour tous les systèmes du type AOM ont été créés dans les milieux industriels. L'acquisition de l'expérience requise à l'outillage de la création des AOMs et plus particulièrement les frameworks ne semble donc pas être facile dans des milieux académiques et donnera, de fait, lieu à l'avancement lent de ces travaux.

[54] Terme emprunté de David Garlan et al. [GS94], mais utilisé ici plutôt dans le sens de la description des schémas qui interviennent dans la conception de l'architecture d'un système.

1.1 Présentation Générale des AOMs

Dans cette section nous revenons sur la définition des AOMs et décrivons brièvement leurs avantages et inconvénients, ainsi que la difficulté majeure face à (l'outillage de) leur création.

1.1.1 Qu'est-ce qu'un AOM ?

Un système à modèle objet adaptatif est avant tout un interprète : c'est un logiciel objet qui est *conçu pour interpréter une représentation évolutive du domaine modélisé.*

Il est constitué de trois principaux éléments :

1. une représentation abstraite du domaine d'application[55].
2. des abstractions requises à la spécification, lors de l'exécution, de la représentation souhaitée du domaine (les objets du méta-niveau[56]). Ces spécifications peuvent, en théorie, concerner n'importe quel *aspect* d'un logiciel, mais concerne toujours son modèle objet et ses règles métier [Ars00, Ars01]. Dans ces cas, elles viennent spécialiser le modèle abstrait créé en (1). M. Fowler [Fow97] préfère désigner ces deux couches d'objets (1) et (2) respectivement par *Operational Level* et *Knowledge Level*.
3. un mécanisme d'interprétation de ses spécifications dans le but de leur associer une sémantique opérationnelle.

Un système AOM est donc comparable à une ligne de produits d'une famille de logiciels similaires. En effet, chaque exemplaire du logiciel (chaque rémanence du système à travers un certain nombre de fonctionnalités et une certaine apparence à l'écran) correspond à l'effet produit par l'interprétation d'une spécification donnée en termes d'instances de classes du méta-niveau du modèle objet adaptatif en question.

Les AOMs sont mentionnés dans la littérature également sous des noms comme *dynamic object model*, *metadata driven system*, *user defined products* [Joh98] ou encore *active object-models* [YFRT98]. Johnson et al. estiment qu'ils apparaissent dès 1995 sous l'appellation *Type Instance pattern* [GHV95].

Les AOMs ont été utilisés avec succès dans des domaines où les règles métier changent souvent et où il y avait nécessité de créer des lignes de produits de logiciels similaires.

A titre d'exemple, on peut citer le système OBJECTIVA dédié à la création de logiciels de facturation pour les produits Télécoms [AJ98], le système UDP pour les produits d'assurances [OJ98], ainsi que le système ARGOS pour la gestion administrative des écoles en Belgique [DT98]. Un autre exemple issu de notre propre expérience est le système CALIBRES [Raz00a]. Rappelons que celui-ci est dédié à la création par des experts métrologues d'une famille de logiciels d'étalonnage de calibres. D'autres exemples sont mentionnés dans [YBJ01b].

---

[55] Ceci constitue l'une des différences majeures entre l'approche des AOMs et celle basée sur la méta-modélisation à la METAGEN [RSBP95, Sah95, Rev97]. En effet, le système ACKPRO [Kri91] qui se trouve au cœur du système METAGEN [RB93, Rev97], ne fait aucune hypothèse sur le domaine modélisé. Par conséquence, les objets du discours de l'expert, définis en termes de *méta-individus* (instances de la classe Meta`Individu`), spécialisent une classe abstraite (`Individu`) sans aucune sémantique métier. Les AOMs préfèrent substituer cette représentation par une autre, à notre goût plus pragmatique, qui contient la sémantique « invariante » du domaine, c'est-à-dire celle qui couvre les aspects du domaine qui ne sont pas objet à des modifications récurrentes. Les travaux que nous présentons ici montrent que le choix des AOMs n'est pas en contradiction par rapport à l'outillage de leur création car nous montrons qu'il est possible de conjuguer le travail de modélisation des experts à celui de la programmation des informaticiens, chacun utilisant le langage qui lui permet de mieux *assumer ses responsabilités* (au sens de [LSGBP99]). Il est, par ailleurs, important de noter qu'on retrouve ici également le schéma de conception du cœur des AOMs (DOM) à travers le modèle objet de la définition et d'instanciation des méta-modèles composé des classes comme Meta`Individu` et `Individu`, mais aussi `MetaRubrique`, `MetaRelation`, etc. On peut donc proclamer le système ACKPRO, et par conséquence METAGEN, comme des AOMs.

[56] Cf. le paragraphe 1.3, page 56 du chapitre I.

1.1.2 Avantages et inconvénients des AOMs

Les avantages et les inconvénients des AOMs sont largement discutés dans la littérature correspondante [FY98b, YFRT98, Til99, RTJ00, YR00a, YBJ01b]. Notre propre expérience de création de ce type de systèmes nous permet également d'affirmer ces analyses.

Au chapitre des avantages sont mentionnés les plus souvents le fait qu'un AOM :

1. facilite la mise à jour du logiciel dans le but de l'adapter aux changements du domaine ou de créer de nouvelles applications.
2. rend la mise en œuvre de cette adaptation accessible aux experts.
3. évite la nécessité d'arrêter, de modifier, de recompiler, de redéployer et de reconfigurer un système afin d'y incorporer un changement du cahier des charges.
4. accélèrent, de façon considérable, le temps requis pour mettre à jour les logiciels.
5. réduit le nombre de classes que les programmeurs doivent implanter.

En ce qui concerne les inconvénients, on peut mentionner le fait que :

1. il est, en règle générale, difficile de créer un AOM car cela requiert une infrastructure pour créer, stocker et interpréter des méta-données. Cela comprend la nécessite de créer des outils graphiques spécialisés pour l'édition des spécifications du système, réalisées lors de son exécution.
2. les interfaces graphiques des systèmes AOMs sont plus difficiles à développer et la gestion des données est également plus difficile à mettre en œuvre car la définition des objets concernés varie au cours du temps.
3. il est difficile de comprendre l'architecture à deux niveaux de modélisation d'un système AOM, et par conséquence, il est difficile de la documenter, maintenir et faire évoluer.
4. un AOM peut être lent, car basé sur l'interprétation.
5. il n' y pas de méthodologie établie et outillée pour la création des AOMs. La création d'AOMs dépend grandement des experts spécialisés et expérimentés[57].

1.1.3 Approches comparables

Il existe de très nombreuses autres approches de développement visant à créer des applications plus flexibles et évolutives. Parmi celles-ci on peut citer les travaux à l'INRIA/IRISA/LABRI sur le projet Compose (*http://compose.labri.fr/*) basés sur l'évaluation partielle, ainsi que ceux du projet METAGEN au LIP6 basés sur la double méta-modélisation [RSBP95, Sah95, Rev97].

Dans la littérature sur les AOMs on peut également trouver un rapprochement avec d'autres techniques comme *Code Generators*, *Generative Programming*, *Metamodeling*, *Table Driven systems* et *Grammar-oriented Object Design (GOOD)* [RFBO01, YBJ01b, RY01].

Très souvent ces approches s'appuient sur des mécanismes comparables à celui de la spécialisation de classes, utilisé dans le *cœur* des AOMs et documenté par le schéma de conception DOM (cf. ci-dessus, §2.2, page 91).

A titre d'exemple, le fonctionnement du système METAGEN est basé sur la définition, pour chaque projet, de deux méta-modèles. Ces derniers constituent les formalismes de modélisation dédiés l'un aux experts et l'autre aux programmeurs.

[57] Le plus dur semble être l'acquisition d'une première expérience réussie [YBJ01D].

La réalisation de cette fonction s'appuie sur un modèle objet[58] dédié à la définition et à l'instanciation de méta-modèles. Celui-ci est composé des classes comme `MetaIndividu`, `Individu`, `MetaRubrique`, `MetaRelation`, etc. La nature des relations entre ces classes et également très semblable à celles des modèles conçus suivant le schéma DOM.

Par exemple, chaque terme du discours de l'expert (ou programmeur), est défini comme un méta-individu, instance de la classe `MetaIndividu`, qui spécialise, par ailleurs, la classe abstraite `Individu`. Les modèles des experts et des programmeurs sont créés alors sous forme d'instances de ces spécialisations de la classe `Individu`.

Deux différences avec l'approche des AOMs peuvent ici être remarquées :

1. pour que les spécialisations de la classe `Individu`, décrites initialement sous forme d'instances de la classe `MetaIndividu` (donc des instances terminales), puissent être instanciées lors de la modélisation, ACKPRO [Kri91]/ METAGEN procède à la génération dynamique d'une classe, suivant le modèle décrit par l'instance de la classe `MetaIndividu` concernée, et qui hérite effectivement de la classe `Individu` (ou l'une de ses sous-classes dans certaines implantations du système ACKPRO).
2. le système METAGEN préfère ne faire aucune hypothèse sur le domaine méta-modélisé. Plus précisément, les instances de la classe `MetaIndividu` spécialisent directement la classe abstraite `Individu`. Or, dans les systèmes AOMs, les classes comparable à la classe `Individu` implantent déjà une sémantique métier, même si elle peut rester assez abstraite (suivant les cas). L'implantation un peu différente du système ACKPRO utilisée dans le système MARLENE, préfère au contraire introduire l'héritage entre les spécialisations de la classe `Individu`. Ce choix le rapproche davantage sur ce point à la technique utilisée par les AOMs.

En ce qui concerne l'aspect comportemental (l'outillage de la co-évolution dynamique de procédures et structures) les approches de ces deux systèmes divergent complètement.

En effet, une fois que les instances de la classe `MetaIndividu` sont compilées sous forme de sous-classes de la classe `Individu` (ou l'une de ses sous-classes), ACKPRO confie au langage SMALLTALK-80 la prise en charge de la définition du comportement des abstractions ajoutées[59].

Quant au système METAGEN, celui-ci propose l'usage du moteur d'inférence NéOpus [Pac92, Pac94, PP94, Pac95] pour décrire les règles qui engendrent le code qui opérationnalise les modèles réalisés par des experts.

---

[58] Ce modèle est initialement mise en oeuvre au sein de l'atelier ACKPRO [Kri91], réutilisé par le système METAGEN [RB93, Rev97]. Il convient ici de mentionner également que le système ACKPRO se trouvait également au cœur du logiciel MARLENE dont ont hérité les systèmes CALIBRES et PRELUDE INSPECTION [ASM00].

[59] Du point de vue des langages d'experts, cette approche, qui comprend déjà l'idée du travail collaboratif, représente une solution possible si d'outils appropriés sont créés pour assister, e.g., par analyse du code, les experts lors de la mise en œuvre de l'adaptation. Cela comprend notamment une assistance sur les conséquences des décisions/interventions des experts au niveau du système existant qui est, *a priori*, opérationnel. A notre connaissance il n'existe pas à ce jour de tels outils. De plus, aucun langage à objet capable d'assurer pleinement le cahier des charges de langages d'experts, tel que défini dans le §1.2, page 17 de l'introduction, n'est encore conçu.
Par ailleurs, comme nous le montrons dans ce mémoire (chapitres IV et V), l'approche d'ACKPRO, tout comme celle des systèmes AOMs conçu suivant le schéma DOM, présente de nombreux inconvénients et notamment l'évolution obligatoire du modèle objet autour d'une abstraction (ici la classe `Individu`). Or, l'approche basé sur le choix explicite de méta-classes permet d'adapter les classes en leur imposant moins de contraintes, voire aucune (cas d'adaptation prototypique, cf. chapitres IV, V et Perspectives).

1.1.4 Difficulté majeure face à la création des AOMs

Comme l'observe R. Johnson[60], la difficulté majeure consiste ici à trouver un système de classes qui offre suffisamment de pouvoir d'expression, afin de permettre de tenir un discours convenablement précis sur l'aspect concerné, sans rendre cette expression aussi compliquée que l'écriture du code :

> *MetaData[61] is just saying that if something is going to vary in a predictable way, store the description of the variation in a database so that it is easy to change. In other words, if something is going to change a lot, make it easy to change. The problem is that it can be hard to figure out what changes, and even if you know what changes then it can be hard to figure out how to describe the change in your database. Code is powerful, and it can be hard to make your data as powerful as your code without making it as complicated as your code. But when you are able to figure out how to do it right, metadata can be incredibly powerful, and can decrease your maintenance burden by an order of magnitude. Or two.* [62]

De plus, comme nous l'avons brièvement évoqué dans l'introduction, paragraphe 1.5.1, page 24, la représentation du code (programme) d'aucun langage à objets ne convient au type d'applications dont il est ici question. En effet, la *représentation des programmes* des langages à objets ne fait pas de ces derniers un outil convenable pour la création des AOMs et plus particulièrement de langages d'experts.

Cette représentation ne considère, en effet, pas les dimensions comme la spécialisation à l'exécution par des experts (donc programmation multi-langages), facilité d'apprentissage par des experts (qui doit aussi être considérée dans la conception elle-même du langage et sa représentation des programmes) et toutes les autres propriétés énumérées dans le §1.2, page 17.

Par ailleurs, celle-ci ne considère pas de façon systématique le besoin des programmeurs en matière de la spécialisation de cette représentation[63].

1.2 Langages d'experts et les AOMs

L'objet des recherches sur les AOMs est l'outillage de la création systématique de ce type d'interprètes ou de "langages métiers". Les outils présentés dans ce mémoire s'inscrivent dans ce cadre et s'intéressent plus particulièrement à l'outillage de la spécialisation par des experts du modèle objet.

Les langages d'experts sont des logiciels du type AOM car la création d'une première forme de langages d'experts est ici outillée en partant directement des techniques issues des travaux sur les AOMs :

1. le modèle de la définition et d'instanciation de nouveaux types d'objets et leur structure est initialement fourni par le DOM (cf. le chapitre III) ;
2. le modèle de la définition et de l'activation de procédures est inspiré du Micro-workflow (cf. les chapitres I et III) .

Dans un second temps (cf. les chapitres IV et V), nous proposons un outillage pour la création d'une forme plus élaborée de langages d'experts qui est également née de l'observation des liens étroits entre le cœur des AOMs et du modèle de la spécialisation tel qu'il est mis en œuvre par des langages à objets.

---

[60] Source: Joseph W. Yoder, URL: *www.joeyoder.com/Research/metadata/#Definition*.

[61] Johnson et al. préfèrent d'utiliser le terme *metadata* [FY98b] pour désigner les spécifications dont il est ici question.

[62] Cette approche, appliquée ici à la création de "langages métiers", est, par ailleurs, bien connue de la communauté de recherche sur les langages de programmation réflexifs [Coi87, Coi90, KRB91]. En effet, comme le disent Des Rivières et Smith [DS84] "*As described in Smith a reflective computational system is one in which otherwise implicit aspects of the system's structure and behavior are available for explicit inspection and manipulation*".

[63] Par exemple permettre de changer, de façon contrôlée (avec des effets bien déterminés à l'avance), la sémantique du langage.

En effet, nous partons du schéma de conception DOM (cf. ci-dessous, paragraphe 2.2, page 91), et en analysant son mode de fonctionnement, nous observons qu'il met en œuvre, d'une façon particulière, la *programmation par spécialisation de classes*, dont l'outillage est, par ailleurs, déjà pris en charge par le langage à objets d'implantation des AOMs.

Le *problème* étant que l'outillage de ce type de programmation (par spécialisation de classes) tel qu'il est mis en œuvre par les langages à objets n'est pas assez performant[64] quant il s'agit de créer des langages d'experts, nous proposons alors de créer un framework qui réutilise et étend la conception des langages à objets réflexifs (les seuls à permettre de mettre en œuvre cette démarche) afin d'assurer que la spécification de chaque représentation du domaine modélisé se fasse en harmonie avec celle du langage à objet d'implantation qui comporte de fait[65] une partie de cette représentation.

Nous insistons sur la nécessité de cette intégration harmonieuse, car les études menées par Johnson et al. (et toujours confirmées par notre propre expérience) montrent que l'usage du schéma DOM, qui néglige cette intégration, conduit à une situation complexe et très problématique quant à la création, la maintenance et la documentation des systèmes AOMs [RTJ00, YBJ01b].

Une autre caractéristique importante de notre solution est que lorsque un framework qui lui est conforme est intégré à un langage à objets réflexif (approprié), il permet de rendre adaptable, en théorie[66], n'importe quelle classe du système et cela avec la possibilité du choix local du type d'adaptation.

Autrement dit, les classes d'une application AOM du type langage d'experts ne sont pas contraintes d'être conçues pour être adaptables (e.g. en héritant d'une classe particulière, comme le propose le schéma DOM). C'est, en effet, notre outillage qui est en mesure de rendre toute classe d'un langage d'experts adaptable en offrant les fonctions suivantes :

1. de pouvoir sous-classer n'importe quelle classe du système lors de l'exécution ;
2. de pouvoir choisir, lors de l'ajout d'une telle spécialisation, le modèle de programmation souhaité (objets, procédures, voire prototypes) par le choix de la méta-classe qui met en œuvre ce modèle.

En ce qui concerne l'outillage de l'opérationnalisation des évolutions de la représentation du domaine modélisé, nous proposons un framework qui assure, de façon systématique, la définition par des experts et cela lors de l'exécution des procédures qui agissent sur des structures de données elles-mêmes définies dynamiquement. Ce framework assure également l'interprétation de ces procédures.

Pour ce faire, nous adoptons le modèle de langage de feuilles de calcul, l'adaptons au contexte de la programmation par spécialisation en s'inspirant des travaux sur les logiciels *flow-independent* mais aussi en s'appuyant sur notre propre expérience industrielle de création de ce type de logiciels, et définissons un système de classes et montrons comment l'implanter sous forme d'un framework orienté-objets.

Il est important de noter qu'ainsi faisant, nous franchissons l'obstacle majeur quant à (l'outillage de) la création des AOMs, annoncé ci-dessus par Ralph Johnson (cf. §1.1.4, page85), en trouvant un système de classes, pour un *aspect* fondamental qui est celui de la spécialisation de la structure et du comportement de classes, qui offre suffisamment de pouvoir d'expression aux experts sans rendre cette expression aussi compliquée que l'écriture du code.

Le reste de ce chapitre est dédié à l'étude des deux axes de recherche actuelle sur les AOMs et le problème de l'outillage de la co-évolution dynamique de structures et de procédures.

[64] Car ne permet pas de garantir des propriétés telle que la spécialisation par des experts, le lien causal, etc.

[65] Puisque c'est justement lui qui sert à implanter le système AOM dont fait partie une implantation d'un modèle objet abstrait du domaine (les *Entities*). C'est cette implantation abstraite qui est l'objet d'évolution lors de l'exécution.

[66] Puisque en pratique il faut prendre quelques précautions pour que ce type d'adaptation puisse avoir lieu (e.g. afficher, au moins, la classe dans la liste de celles qui sont supposées être adaptées) et aussi pour que les conséquences de l'adaptation soient prises en compte par l'application de façon appropriée (e.g. mise à jour du schéma de la base de données).

2 Analyse d'applications industrielles existantes

Cette section étudie succinctement le premier des deux axes de recherche actuelles sur les modèles objets adaptatifs, mené par Johnson & al.. Il s'agit d'analyser des applications industrielles du type AOM dans le but d'abstraire leur schémas de conception [UIUC98, OOPSLA98, OOPSLA99, ECOOP2000, OOPSLA'2000, ECOOP2001].

Les analyses et conclusions relatives à ces travaux ont été publiées sous forme de schémas de conception et d'autres documents similaires [FY98b, YFRT98, Til99, RTJ00, YR00a, YR00b, YBJ01a, YBJ01b, YBJ01c].

Il est important de noter que l'ensemble des systèmes AOMs créés à ce jour, y compris le framework Micro-worklfow, sont directement ou indirectement contraints par des obligations de confidentialité (très souvent en raison de leur caractère industriel) et ont donc été très partiellement documentés.

Toutefois, en ce qui nous intéresse ici, c'est-à-dire la nature de la conception du noyau de ces systèmes, les études mentionnées ci-dessus montrent l'usage de techniques très semblables. Nous allons donc nous appuyer sur l'un d'entre eux, le logiciel THE HARTFORD[67], pour illustrer les principaux problèmes posés et les solutions *ad-hoc* qui leur ont été apportées dans le contexte industriel de ces projets.

2.1 Exemple: THE HARTFORD (UDP)

Ce paragraphe est consacré à l'étude, à travers l'exemple du système THE HARTFORD, des schémas de conception qui interviennent de façon récurrente dans la création des AOMs.

2.1.1 Généralités

The Hartford insurance framework ou *User Defined Prodcut* (désormais, en abrégé UDP) [OJ98], a été créé par la compagnie d'assurances américaine *The Hartford*, en réponse au besoin de faire définir par des experts en produits d'assurance, de nouveaux produits et de leur permettre également de mettre à jour des produits déjà définis. Il permet d'éditer des objets métier complexes dont la valeur des attributs est calculée en fonction de leurs composants constitutifs et cela à l'aide des règles décrites également dynamiquement.

Un produit d'assurance est créé par la composition de produits de base. Par exemple, un produit peut combiner une police auto avec une police habitation voire diverses options. De plus, le prix de chaque produit est calculé en fonction des caractéristiques choisies - assurance habitation ou auto, valeur de la propriété assurée, localisation géographique, etc.

UDP a été l'objet d'un rapport écrit par Ralph Johnson et Jeff Oakes [JO98]. Ce rapport représente un intérêt important, c'est d'expliquer le « pourquoi » de l'architecture de ce système a travers les schémas utilisés dans sa conception.

Cette étude montre[68], par ailleurs, que le problème fondamental de co-évolution dynamique de structures et de procédures se pose également ici. Il s'agit en effet, de pouvoir spécifier la règle de calcul du prix d'un nouveau produit d'assurance en fonction de ses composants, lesquels ont été également définies dynamiquement. Ce problème a été à nouveau, comme cela a été le cas pour d'autres systèmes comme CALIBRES, solutionné par des techniques ad-hoc.

UDP pose également le problème de la gestion des flux de travail (workflow), mais ne lui apporte aucune solution. En effet, selon Johnson et Oakes, UDP n'est pas en mesure de définir et d'assurer le déroulement du procédé selon lequel un nouveau produit est validé, mis sur le marché et plus tard retiré du marché. Cette gestion est très liée à la définition des règles métier au niveau de ces produits. En effet, à titre d'exemple, un produit ne peut être vendu que s'il est en cours de commercialisation. Dès que son statut change de nouvelles règles devaient s'y appliquer.

---

[67] THE HARTFORD ainsi que les systèmes OBJECTIVA et ARGOS, ont été l'objet d'une étudie lors du premier workshop sur les AOMs à UIUC en mai 1998 [UIUC98].

[68] Il est important de rappeler que ce travail ne met en évidence que *l'essence* de ce système qui est de nature très complexe, tout comme tous les autres systèmes de sa famille.

2.1.2 Conception pour décrire dynamiquement de nouveaux produits d'assurance

Ce qui suit décrit les schémas qui assurent la définition à l'exécution de nouveaux produits d'assurances au sein du système UDP.

Composite

Le schéma de conception *Composite* est utilisé pour permettre de « simuler » la hiérarchie de classes qui devait sinon modéliser les différents types de produits d'assurance de cette compagnie. Selon ses créateurs, ce système devait alors comprendre près de 10000[69] classes du type `PropertyPolicy`, `AutoPolicy`, `FloodRider`, etc. De plus, la définition de certaines combinaisons pouvait nécessiter l'héritage multiple, ce qui n'était pas en l'occurrence proposé par le langage d'implantation SMALLTALK-80 (VISUALAGE d'IBM).

Les classes en question sont alors définies (« simulées ») sous forme de composition d'instances d'autres classes. Par exemple, une police habitation comprenant une assurance dégât des eaux est créée comme une instance de la classe `Policy` (représentant la police) qui comporte une instance de la classe `Home` (représentant le composant habitation de la police), laquelle comporte une instance de la classe `Flood` (le component dégât des eaux).

Le modèle objet correspondant comporte une hiérarchie de `Policy`. Tout nouvel élément d'une police, comme par exemple, une assurance auto ou habitation, sera modélisé par une sous classe de `Policy`. Ces derniers peuvent, de plus, être à leur tour des éléments qui se composent à partir d'autres éléments. Il faut alors utiliser le schéma de conception composite pour permettre ce type de définition, d'où la nécessité de la classe supplémentaire `CompositePolicy`.

`Flood` et `Collision` sont des exemples de component et `Policy`, `Home` et `Auto` sont des exemples de composites. Cette technique facilite l'ajout de nouveaux types de produits et/ou la modification des types de produits existants.

Property List

Chaque component a ses propres attributs. Par exemple, une police d'assurance (la classe `Policy`) se caractérise par son propriétaire (l'attribut `owner`) et son adresse (l'attribut `address`. Une maison (la classe) se caractérise par sa valeur (l'attribut value) et son adresse (l'attribut `address`).

Une autre caractéristique souhaitée de ce système est de permettre d'ajouter de nouveaux composants dont la structure se caractérise différemment par rapport à ceux déjà existants, ou de modifier la structure des components existants.

Le schéma de conception *Property List* est utilisé pour assurer cette fonction. Les attributs d'un component sont alors stockés dans une table de hachage. L'ajout et le retrait d'attributs est facilité et en particulier ne requiert plus de compilation.

Cet aspect prend une certaine importance dans la mesure où certains langages à objets ne permettent pas le changement « à chaud » de la structure des objets. Il est toutefois important de noter que ce schéma ne résout pas tous les problèmes sous-jacents à ce sujet car dans la plupart des cas il faut aussi prévoir des mécanismes de mise à jour du schéma de la base de données, et aussi des interfaces graphiques.

[69] Suivant les estimation du département systèmes d'information de THE HARTFORD.

2.1.3 Conception pour décrire dynamiquement le comportement de nouveaux produits

Strategy

Le schéma de conception *Strategy* est utilisé pour permettre de définir lors de l'exécution le comportement des différents composants des produits d'assurance par le choix d'un algorithme parmi un ensemble de choix possibles.

Par exemple une police habitation (`PropertyPolicy`) n'a non seulement des attributs différents par rapport à une police auto (`AutoPolicy`), mais elle dispose également des algorithmes spécifiques pour calculer la valeur de ses attributs.

Le schéma de conception *Strategy* permet de créer des classes qui représentent chacune une implantation particulière d'un certain algorithme. La hiérarchie des classes ainsi obtenue constitue un ensemble de choix possibles pour l'expert lors de la définition de chaque component.

Cette technique s'est avérée, selon ces auteurs, assez appropriée dans le cas de ce système d'assurance où chaque component doit répondre à un protocole assez limité : calculer sa valeur (`rating`), s'éditer (`edit`), s'imprimer suivant un format lisible pour le client (`print`) et s'imprimer suivant un format approprié pour les contrôleurs de l'administration[70] (`coding`).

2.1.4 Conception pour assurer la co-évolution dynamique de structures et de procédures

Le schéma de conception *Strategy* ne résout pas tous les problèmes posés, en particulier le plus important ici qui est le problème de la co-évolution dynamique de structure et de procédures. En effet, lorsque les experts ajoutent un nouveau type de component et définissent sa structure, avec le schéma *Strategy* il y a nécessité de créer une nouvelle classe pour coder de nouveaux comportements requis.

Or, coder un algorithme dans un langage de programmation n'est pas la solution idéale pour un expert non informaticien. D'autant plus que souvent il est possible de proposer une meilleure solution. Par exemple, ici Johnson et Oakes observent que les règles relatives aux polices d'assurances pouvaient convenablement être écrites à l'aide d'un langage simple du niveau de celui du langages de feuilles de calcul[71].

Interpreter

Le schéma de conception *Interpreter* est utilisé, conjointement avec le schéma *Composite*, pour permettre de composer lors de l'exécution de nouveaux algorithmes, ici de calcul des primes d'assurances, ou de nouvelles règles métier. Des éditeurs graphiques aident les experts lors de cette activité.

La technique consiste à modéliser la grammaire du langage approprié sous forme d'une hiérarchie de classes et d'implanter au niveau de chaque classe un algorithme qui décrit la sémantique opérationnelle des nœuds de l'élément de discours qu'il représente.

Une expression de ce langage est alors représentée par une composition bien formée d'instances de ces classes. Un éditeur graphique assiste, en règle générale, l'expert lors de cette composition.

Lors de l'interprétation, le message approprié est envoyé à la racine de la composition qui représente l'expression et qui donne lieu, en règle générale, a un parcours récursif de l'arbre qui conduit à son interprétation.

Par exemple, dans le cas du système UDP, les règles de calcul des primes d'assurance sont décrites à l'aide des éléments d'un arbre composé de quatre classes : `AttributeRef`, `Constant`, `TableLookup` et `BinaryOp`. Celles-ci héritent de la classe abstraite `Rule`. Cette hiérarchie permet en

[70] *Government regulators.*

[71] *However, except for summing up values of components, the functions never have to deal with iteration or recursion. Thus, they can be described by a fairly simple language, more at the level of* spreadsheet *rules than a real programming language* [JO98, page 6].

théorie, de décrire toute expression nécessaire. Toutefois, en pratique il peut s'avérer nécessaire d'y apporter des modifications pour l'adapter aux besoins d'une nouvelle situation. Par exemple, la recherche dans une table n'est pas toujours nécessaire.

Chacune de ces classes implante la méthode `valueWith:`. Celle-ci reçoit en argument un contexte, implanté souvent à l'aide d'une table de hachage, qui contient ici l'ensemble des attributs du composant courant ainsi que ceux de ses «parents ». Ce choix permet, par exemple, à un component collision de calculer ses primes en fonction du département d'habitation du conducteur de la voiture.

Il est important de noter ici que le cas de UDP montre encore une fois l'importance de l'outillage de la co-évolution dynamique de structures et de procédures. Il montre également l'intérêt réel des modèles de programmation simples comme celui du langage de feuilles de calcul.

La solution mis en œuvre par UDP comprend déjà des éléments de la conception de l'outillage que nous présentons ici. Mais, notre solution couvre des dimensions importantes, comme la gestion du lien causal, la facilité d'apprentissage par des experts, et le travail collaboratif qui ne sont pas pris en considération systématique dans l'approche de UDP (pas intégrées dans la conception).

2.1.5 Conception pour partager les définitions

Type Object

Le schéma de conception *Type Object* [JW97] est utilisé pour permettre de *partager* la définition des attributs et des stratégies communs des composants. Chaque instance de la classe du méta-niveau contient les descriptions communes à un ensemble de composants (relation comparable à celle des classes et leur instances).

Nous ne détaillons pas d'avantage ce cas car il sera l'objet d'une description détaillée et une implantation sous forme d'un framework dans le chapitre III.

History

Le schéma de conception *History* [And98] est, par ailleurs, utilisé pour assurer la gestion de l'historique des modifications des types de composants ainsi que les composants eux-mêmes.

2.1.6 Cas du système ARGOS

De façon comparable au cas du système UDP, le système ARGOS est dédié à la création de logiciels par des non-programmeurs pour la gestion des écoles publics en Belgique. Ces logiciels sont utilisés par une variété d'utilisateurs dans l'administration centrale, les écoles et les mairies.

Les logiciels produits, e.g. gestion du budget, du personnel et des stocks, font appel aux techniques très variées comme les basses de données, la gestion des documents électroniques, le workflow et l'Internet.

L'approche générale du système ARGOS, tout comme les autres systèmes du type AOM, consiste à créer un «langage » pour chaque aspect changeant du système qui permet aux experts de spécifier, à l'aides des interfaces graphiques spécialisés, les caractéristiques souhaitées pour chaque logiciel. De ce fait, le système peut être considéré comme un environnement de prototypage et d'expérimentation.

Ces spécifications sont associées aux types d'objets concernés.

Ici, les aspects qui ont été l'objet de ce traitement sont :

1. Modèle objet : définir des nouveaux types d'objets, des associations et contraintes élémentaires.

2. Interfaces graphiques : définir les différentes interfaces et spécifier leurs propriétés visuelles (dimension, couleur, etc.) et fonctionnelles (activités et informations disponibles).
3. Règles métier : définir des rôles et autorisations et des règles du type événement, condition, action. Les conditions et actions sont définies à l'aide d'un langage de script qui est une extension du langage SMALLTALK-80. Les événements sont pré-définis et sont de deux sortes : liés aux transactions (création, suppression, modification) et autres (liés aux interfaces et au temps).

Une autre caractéristique remarquable du système ARGOS est qu'il met en œuvre une phase de *bootstrapping* afin de remplacer certains outils de développement "codés en dures" par d'autres qui bénéficient des outils développés pour ARGOS lui-même. Cela permet de rendre configurable, certains aspects de ces outils de développement, e.g. les règles de définition des modèles objets ou règles d'import/export de données.

2.2 Conclusion : le schéma de conception DOM

2.2.1 Observations générales

L'analyse comparative des systèmes d'origines indépendantes et convergeant vers des architectures du type AOM qui a été jusqu'alors menée par Ralph Johnson et al. permet de constater des ressemblances mais aussi des différences entre ces différents systèmes.

Tous ces systèmes permettent de définir lors de l'exécution de nouveaux types d'objets et leur structure. On observe également une nécessité pour la définition dynamique de comportements. Ce qui pose systématiquement, mais avec des intensités différentes, le problème de la co-évolution des structures et des procédures.

Les techniques déployées varient, toutefois, d'un système à l'autre.

En matière de la définition de nouveaux types d'objets et leur structure, UDP met en avant la composition, alors que OBJECTIVA privilégie un schéma similaire à celui des langages à objet et propose également l'héritage et le polymorphisme.

En ce qui concerne le comportement, le système ARGOS fait appel à un langage de script, une forme spécialisé du langage SMALLTALK-80. Le système UDP utilise des techniques documentées par les schémas de conception *Strategie* et *Interpreter*. Le système OBJECTIVA, tout comme le système CALIBRES, font appel à la composition de procédures. Les techniques utilisées ne sont pas identiques. OBJECTIVA semble utilisé une approche comparable à celle du Micro-workflow alors que CALIBRES utilise une technique semblable à celle de DART/DYCRA.

Une dimension partagée par l'ensemble de ces systèmes est l'usage par des experts non informaticiens. Toutefois, la mise en œuvre systématique de cet aspect n'a pas jusqu'alors (avant notre étude) été l'objet d'une étude systématique. Elle a, principalement, été assurée par la création d'interfaces graphiques dédiées.

Au sujet des motivations on peut remarquer souvent l'usage des AOMs pour couvrir une famille de logiciels similaires. La variété dont il est ici question est soit issue de la complexité du domaine à modéliser (exemple de UDP ou OBJECTIVA) soit répond à la nécessité de faire face aux changements fréquents du domaine modéliser (cas de ARGOS) ou encore la volonté de créer des lignes de produits (cas de OBJECTIVA). En sommes, tous ces systèmes sont plus ou moins directement concernés par ces dimensions. Par contre, le choix de l'approche AOM a été suivant le cas motivé différemment.

Dans le cas de notre propre expérience, le système CALIBRES [Raz00a], c'est la dimension ligne de produits accessible aux experts qui a plus directement motivé le choix d'une telle architecture.

2.2.2 Résultats en terme d'outillage : le modèle des compléments de classe

La communauté AOM utilise de manière essentielle les schémas de conception [GHJV95, JW97, ABW98] pour décrire et organiser les architectures complexes qu'elle étudie et produit. D'ailleurs, Ralph Johnson qui est le «père » des AOMs, fait aussi partie du groupe des quatre qui ont publié pour la première fois des schémas de conception de logiciels à objets [GHJV95].

Aussi, les résultats des analyses des systèmes AOMs existants ont également été synthétisés par le schéma de conception *Dynamic Object Model* [RTJ00], appelé également le cœur des modèles objets adaptatifs [Joh98, MJ99]. Ce schéma est illustré par la Figure 22 ci-dessous.

Il s'agit d'une combinaison de trois patterns *Type Object*, *Properties* et *Strategy*, qui constituent ensemble un modèle pour la spécialisation dynamique de classes à l'aide des *compléments de classe*. Nous opposons ici la notion de complément de classe à celle d'adaptation (cf. §1.1.2, page 16 de l'introduction) afin de souligner la différence des modèles de leur mise en oeuvre. En effet, les compléments de classe s'appuient sur le modèle de la Figure 22, or les adaptation sont outillées suivant une variante de ce modèle présenté par la Figure 55, page 149.

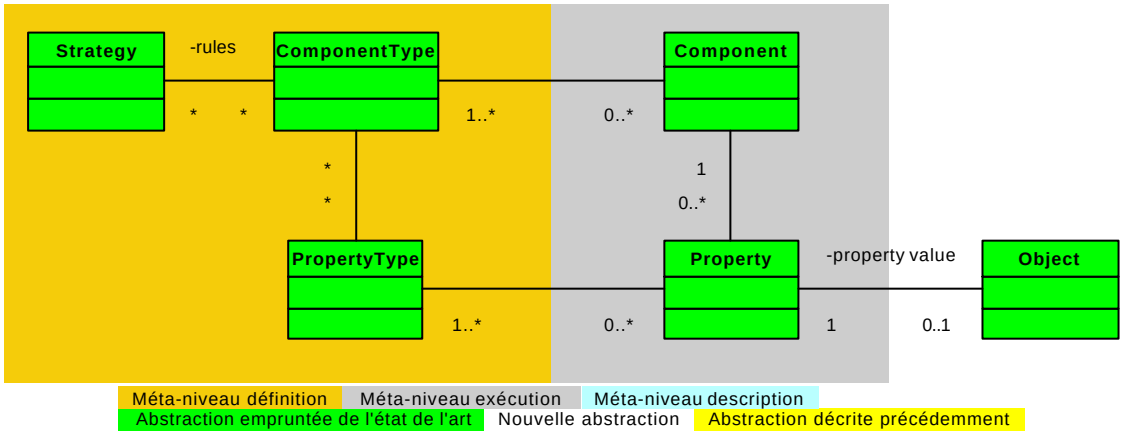


Figure 22 : Le cœur des modèles objets adaptatifs [MJ99b, YBJ01b].

Selon ce modèle, chaque classe (représentée ici par la classe `Component`) qui satisfait un certain nombre de critères peut être spécialisée lors de l'exécution et cela indépendamment du langage à objets d'implantation. Cette solution est en effet, entièrement basée sur l'instanciation des classes prévues à cet effet, lors de la conception initiale du système.

Ces critères sont :

1. disposer d'un lien, appelé souvent `type`, vers un autre objet qui est le complément de classe.
2. disposer d'une variable d'instance, souvent appelée `properties`, qui contient une table de hachage des valeurs des attributs (sous forme d'instances d'une classe comme `Property`) définis au sein du complément de classe concerné.
3. implanter ou hériter l'implantation des protocoles nécessaires à la gestion des valeurs des attributs mais aussi à l'accès au complément de classe.

Les responsabilités du complément de classe sont modélisées par la classe `ComponentType`. Il s'agit tout simplement, de façon comparable au rôle d'une classe, de contenir l'ensemble des définitions d'attributs et de procédures de compléments de classes.

A titre d'exemple, le nouveau type de Compte-Service est ici défini comme une instance d'une classe modélisée suivant `ComponentType`. Chacun de ses attributs est défini comme une instance d'une classe modélisée suivant `PropertyType`.

Lors de l'instanciation, c'est l'abstraction assimilable à la classe `Component` qui est instanciée, e.g. la classe `Account`. Cette instance est, toutefois, interprétée par le système comme une instance du nouveau type Compte-Service.

Nous allons pas nous attarder ici davantage sur ce modèle car il sera l'objet d'une étude plus approfondie lors du chapitre suivant. Pour conclure, nous tenons à préciser qu'en ce qui nous intéresse ici, c'est à dire l'outillage de la spécialisation à l'exécution, ce schéma a deux utilités : 1) il montre une technique standard de conception pour la définition lors de l'exécution de nouveaux types d'objet et leur structure ; et 2) il modélise l'instanciation de ces types d'objets.

Aussi, ce schéma couvre une partie de l'outillage de la spécialisation dynamique et de même une partie de l'outillage de la gestion du *lien causal* des langages d'expert. Nous allons nous intéresser à présent au mécanisme qui va nous permettre de compléter cet outillage et de couvrir la dimension comportemental.

3 Création de nouvelles architectures AOM : le *Micro-workflow*

Un autre axe de travail majeur de l'équipe de Johnson consiste à créer de nouveaux outils dédiés à la création des AOMs. Le Micro-workflow, thèse de Dragos Manolescu à UIUC, octobre 2000, est à notre connaissance le seul système de classes dédié à l'outillage de la création des systèmes de gestion de workflow adaptatifs par des programmeurs objets[72].

Le but de cette section est de présenter succinctement ce système et d'étudier ses intérêts mais également ses limites quant à l'outillage de la co-évolution dynamique de structures et de procédures.

3.1 Introduction

Le Micro-workflow se veut avant tout un framework modulaire pour le développement de systèmes *workflow adaptatifs* [MJ99b, MJ99c, Mal00]. Il est destiné aux programmeurs de logiciels objets, qui peuvent le personnaliser à leur gré. Il a une architecture modulaire constituée d'un noyau pour la description des tâches à réaliser et le flux de contrôle de cette réalisation. D'autres modules pour la persistance, le monitoring, la gestion de l'historique d'exécution, etc. peuvent y être inclus de façon optionnelle.

La réification des flux de contrôle au sein d'une application objet, pris en charge par le noyau du Micro-workflow, permet leur représentation explicite. Cela permet de modifier les flux[73] et donc mettre en œuvre de nouveaux procédés sans changer le code des objets métiers et vice-versa. Une telle application est appelée *flow-independent*. Les recherches récentes ont mis en évidence l'intérêt de cette approche pour améliorer la maintenabilité et l'évolutivité des logiciels.

Le Micro-workflow se propose alors de remplacer les langages de programmation par objets pour ce qui est le codage de procédés qui relient les objets métiers entre eux (le « comment », alias *process logic*). Le langage à objets servira alors uniquement pour le codage au sein des objets métier des tâches primitives (le « quoi » , alias *task logic*). Le Micro-workflow a été implanté en VisualWorks SMALLTALK et a été utilisé dans le cas de plusieurs applications [Mal00].

[72] Il serait important de noter ici que notre thèse entre dans la catégorie des travaux consacrés à la définition des systèmes de classes dédiés à l'outillage de la création des AOMs (ou, de façon plus générale, des systèmes complexes). La thèse de D. Manolescu sur le Micro-workflow en est un autre exemple.

[73] Suivant les choix d'implantation, cette modification peut toutefois nécessiter l'écriture du code relatif aux procédés.

3.2 Conception du Micro-workflow

Comme permet de l'illustrer la Figure 23 ci-dessous, composée de deux parties coloriées en orange et gris, le modèle du Micro-workflow sépare les objets qui servent à la définition des procédés par rapport à ceux qui prennent en charge leurs exécutions. Cette séparation est faite suivant le schéma de conception *Type Object*. C'est pourquoi on estime le Micro-workflow comme lui-même une application du type AOM [YBJ01b].

Ce choix donne lieu à deux couches d'objets de méta-niveau. Celle qui modélise la définition de procédures et permet de définir les règles de déroulement d'un procédé. Une seconde couche modélise leurs activations et plus précisément la gestion des données propres à chaque activation de la procédure.

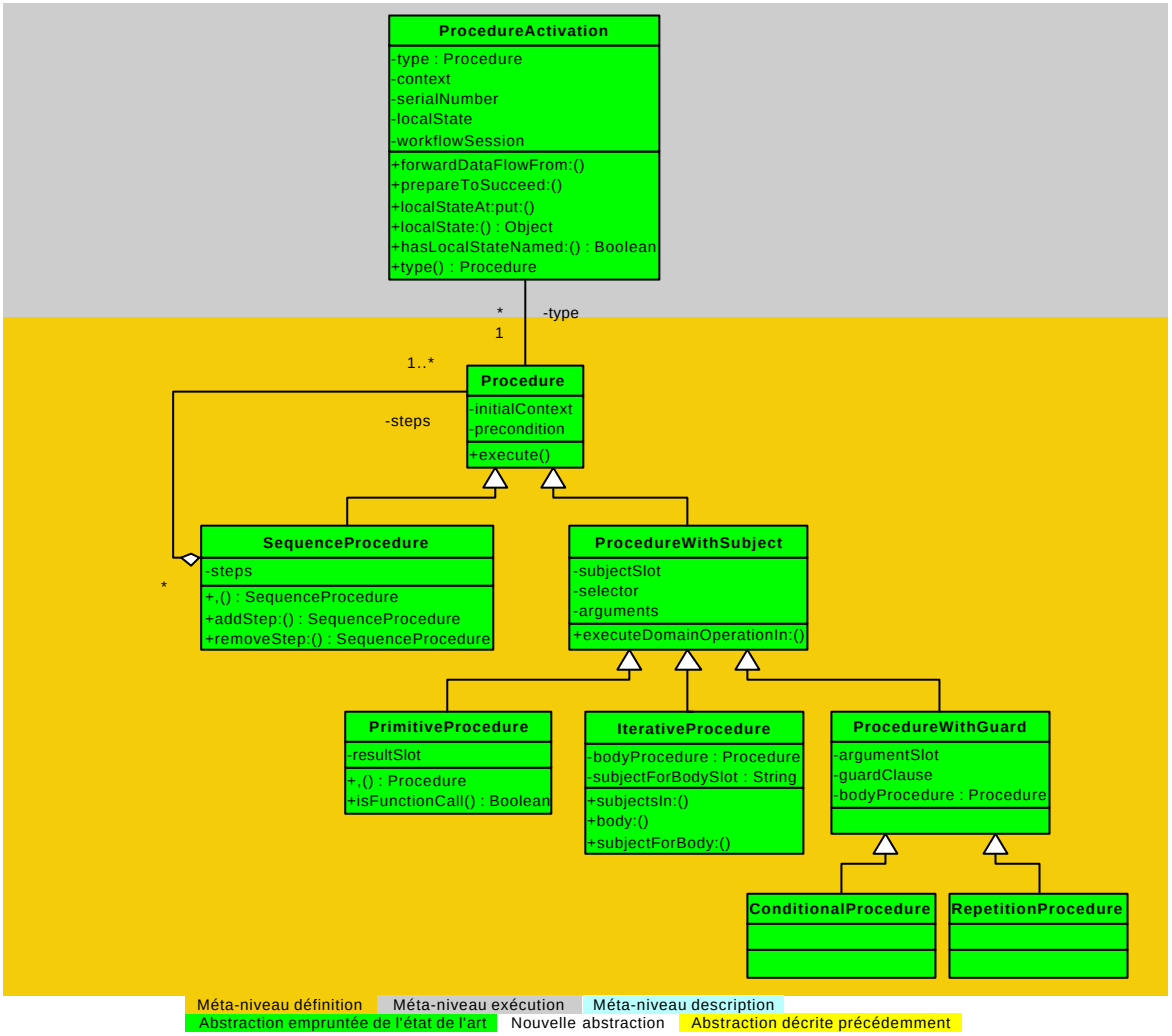


Figure 23 : Le cœur du système de classes micro-workflow [Man00].

Le Micro-workflow est, par ailleurs, conçu sous forme d'une architecture modulaire, dans le but de favoriser le choix local à chaque projet des composants nécessaires. Aussi, il comporte un noyau qui modélise la définition et l'interprétation des procédures, ainsi qu'un ensemble de modules optionnels.

Nous nous intéressons ici plus particulièrement au noyau du Micro-workflow qui nous fournit les abstractions nécessaires à la définition explicite de procédures, mais aussi celles qui assurent leur exécution.

3.2.1 Modèle de définition de procédures

La définition d'un procédé spécifie les activités que les acteurs (ici des objets métiers) doivent réaliser pour atteindre un certain but. Le Micro-workflow réifie les concepts qui permettent de représenter certains type de scripts. Il s'agit fondamentalement des séquences d'instructions avec l'affectation. De plus, ces séquences peuvent comporter des itérations et conditionnelles. Il met en œuvre également un certain type de calcul parallèle à l'aide des abstractions `Fork` et `Joint` [Man00, MJ00].

De façon plus précise, le Micro-workflow prévoit la définition d'un procédé comme une collection de *procédures*. Une `Procedure` est un concept abstrait qui représente un élément quelconque d'un procédé. Des spécialisations de la classe `Procedure` spécifient différents types de procédures. Ce sont, pour l'essentiel, la primitive (`PrimitiveProcedure`), la séquence (`SequenceProcedure`), l'itération (`RepetitionProcedure`) et la conditionnelle (`ConditionalProcedure`).

L'ensemble de ces concepts constitue le méta-niveau description du système de classes définissant le Micro-workflow.

La `PrimitiveProcedure` sert à spécifier la demande de réalisation d'une tâche par un objet du domaine. Elle permet de transférer lors de l'activation, des données vers l'espace des objets métier (par rapport à l'espace workflow où se trouvent les objets qui relèvent de la définition et de l'activation des workflows). Elle permet ensuite de faire opérer les objets métier sur ces données et de retourner le résultat de cette opération qui va enrichir le contexte d'exécution.

La Figure 24 ci-dessous montre un exemple de définition d'un appel de primitive (instance de la classe `PrimitiveProcedure`). Celle-ci formule la demande d'envoi du message `notifyAbout:` à l'objet du contexte d'exécution nommé `reviewer` avec en argument l'objet appelé `proposal` et d'affecter le résultat de cet envoi de message à la variable `review`, toujours dans le même contexte d'exécution.

```
PrimitiveProcedure
        sends: #notifyAbout:
        with: #(#proposal)
        to: #reviewer
        result: #review
```

Figure 24 : Exemple de Syntaxe du Micro-workflow.

Une `SequenceProcedure` est conçue suivant le schéma de conception *Composite*. Cela signifie en particulier que toutes les sous-classes de `Procedure`, y compris `SequenceProcedure` elle-même, peuvent jouer le rôle de composants. L'activation d'une séquence correspond à l'activation successive de chacun de ses composants [Man00, page 50].

Le modèle mis en œuvre jusqu'ici permet la définition et l'exécution des séquences de primitives. Pour obtenir des ruptures de séquence, le Micro-workflow prévoit le concept de `ProcedureWithGuard`, qui associe une condition à une procédure. Ce couple permet la réalisation de diverses constructions comme `ConditionalProcedure` (IF condition THEN body) et `RepetitionProcedure` (DO body UNTIL condition).

3.2.2 Modèle d'activation de procédures

Le concept de `Procedure` sert à la définition des procédés. Le Micro-Workflow met en face de ce concept la notion de `ProcedureActivation` qui sert à l'exécution des procédés.

Une instance de `ProcedureActivation` gère un pointeur sur la définition du procédé auquel il est attaché (instance de la classe `PrimitiveProcedure`). Elle gère également un contexte d'exécution (une table de hachage) qui lui est propre. La mise en œuvre de ce contexte suit le patron de

conception *Property*. Celui-ci permet une expansion du contexte par le rajout de nouvelles entrées à l'exécution afin de définir des couples variable / valeur.

Dans le cas de l'activation d'un appel de `PrimitiveProcedure`, la valeur courante des arguments est recherchée dans ce contexte, et c'est leur valeur immédiate qui est employée. Ceci exclut, par exemple, des arguments qui seraient des expressions à évaluer. Nous revenons sur ce problème plus en détail dans le §3.3.2.1, page 98 de ce même chapitre.

3.2.3 Exemple

La Figure 25 ci-dessous donne l'exemple d'un script particulièrement simple, extrait de l'exemple de la section 4, page 100 ci-dessous, écrit en SMALLTALK-80. Celui-ci calcule le solde d'un compte en lui envoyant le message `getBalance` et affecte le résultat à la variable locale `balance`. Ensuite il affiche ce solde à l'aide de la méthode `show:` en passant en argument la variable `balance`.

Il s'agit donc d'une séquence de deux expressions et d'une affectation.

```
balance := self getBalance.
self show: balance.
```

Figure 25 : Exemple d'un script en SMALLTALK-80.

Ce script veut uniquement permettre de comparer les deux modes d'expression d'un calcul, d'une part d'un langage à objet et de l'autre part le Micro-workflow. La Figure 26 ci-dessous montre, en effet, ce même script écrit en Micro-workflow[74].

Ce dernier représente ce script par une instance de la classe `SequenceProcedure`. Celle-ci est créée par l'envoi du message `,` (virgule) à une instance de la classe `PrimitiveProcedure` qui représente l'envoi du message `getBalance` à un compte et l'affectation de sa valeur à la variable local `balance`. Elle reçoit en argument de ce message une autre instance de la classe `PrimitiveProcedure` laquelle représente l'envoi du message `show:` avec la variable `balance` en argument.

```
| getBalance showBalance thePrograme |
      getBalance := PrimitiveProcedure
                        sends: #getBalance
                        to: #myAccount
                        result: #balance.
      showBalance := PrimitiveProcedure
                        sends: #show:
                        with: #balance
                        to: #myAccount.
      thePrograme := getBalance, showBalance.
      ^thePrograme
```

Figure 26 : Le script de la Figure 25 écrit en Micro-workflow.

Dans le premier cas c'est le message `sends:to:result:` qui est utilisé pour spécifier la primitive. Ce choix permet, en effet, de nommer la variable locale qui reçoit le résultat du calcul à l'exécution.

Dans le second cas c'est le message `sends:with:to:` qui est utilisé, car cette fois il n'y a pas d'affectation, mais en revanche il y a le passage en argument de la variable `balance`.

[74] Ce script est stocké dans la méthode `exampleStandardMFW` de la méta-classe de `Account class`.

Le Micro-workflow permet la description sous forme de micro-procédés des séquences de code plus complexes, qui comportent des itérations, conditionnelles, etc. La thèse de D. Manolescu [Man00] fournit plusieurs exemples détaillés.

3.3 Comparaison entre DART et le Micro-workflow

Le but de cette section est de fournir plus de précisons sur la nature des relations entre les deux systèmes de classes DART et le Micro-workflow, les deux étant dédiés à la définition et l'activation de procédures.

3.3.1 DART hérite les avantages du Micro-workflow

Le Micro-workflow offre de nombreux avantages que DART préserve et cherche à consolider. Parmi ceux-ci on peut citer :

1. le Micro-worklfow réifie les abstractions qui permettent de décrire explicitement une séquence d'activités. Nous montrons que cette mécanique peut servir, moyennant un habillage très particulier, pour la création de langages dédiés aux experts suivants les critères de Nardi.
2. le Micro-workflow modélise explicitement l'activation des procédues. C'est ce choix de modélsiation qui permet d'assurer le lien causal et que nous reprenons dans le cas de DART.
3. le Micro-wokrflow est une architecture dédiée aux programmeurs. Il y a donc une convergence au niveau des objectifs poursuivis et du public visé entre DART et le Micro-workflow.
4. le Micro-workflow veut outiller la création par les programmeurs de logiciels flow-independent et des systèmes de gestion de workflow. En considérant les évolutions récentes de l'usage de l'informatique, de nombreux travaux s'accordent sur l'importance de la prise en charge de ces deux aspects dans les architectures logicielles. Le choix du Micro-workflow nous procure également cet avantage, que nous veillerons à préserver également tout en lui donnant un caractère optionnel au sein de notre architecture.
5. le Micro-workflow s'intéresse également aux aspects pratique de l'outillage et prévoit la définition de procédures à l'aide des outils habituels de programmation.
6. enfin, le Micro-workflow est une architecture bien documentée et facile à mettre en œuvre et à spécialiser par les outilleurs.

3.3.2 DART enrichi le Micro-workflow

Malgré tous ces avantages, le Micro-workflow montre également quelques faiblesses, notamment dans le cas de l'usage que nous envisageons ici d'en faire. DART veille à enrichir le Micro-workflow de façon à ce qu'il puisse effectivement servir à l'outillage de la co-évolution dynamique de procédures et de structures et aussi permettre une composition de procédures par des experts.

Nous évoquons ici deux de ces problèmes. La section suivante en évoquera un troisième, celui du couplage avec DOM.

3.3.2.1 Problème de modélisation de variables

Le Micro-workflow est dédié aux programmeurs. Cela signifie que les programmeurs sont supposés utiliser l'éditeur de programme du langage d'implantation du Micro-workflow et de coder leur procédés dans une syntaxe acceptable par le compilateur de ce langage. Le résultat de compilation d'un tel code est stocké sous forme d'une méthode. Lorsque celle-ci est invoquée, le texte compilé s'exécute et retourne comme résultat un arbre composé d'instances des sous-classes de la classe `Procedure`.

Lors de la définition d'un procédé composé de plusieurs étapes, il est nécessaire de stocker l'objet (l'instance d'une sous-classe de la classe `Procedure`) qui représente l'étape N-1 dans une variable local, afin de pouvoir s'y référer lors de la définition d'autres étape de la procédure et établir des connexions entre la réalisation de ces différentes étapes.

Une connexion remarquable à ce niveau est celle de l'usage du résultat de l'exécution d'une étape comme l'argument lors de l'exécution d'une étape ultérieure.

Au moins deux solutions sont alors envisageables :

1. stocker le résultat de l'exécution de l'étape N-1 dans l'environnement d'exécution et indiquer à la procédure de l'étape N avec quelle clé recherchée cette valeur dans l'environnement concerné.
2. passer l'objet qui représente l'étape N-1 lui-même comme argument à l'objet qui représente l'étape N. Dans ce cas, lorsque l'étape N a besoin de la valeur de cet argument le demande à la procédure qui représente l'étape N-1 et qui lui avait été passé comme argument.

La première solution est celle adoptée par le Micro-workflow. En effet, comme permet de l'illustrer la Figure 26 ci-dessus, le Micro-workflow nomme le résultat de calcul de chaque procédure (lorsqu'il existe) et demande de se référer à ce nom lorsque ce résultat devait être utilisé comme l'argument d'une autre procédure. La Figure 27 ci-dessous permet d'illustrer le même cas de façon graphique. La procédure qui défini l'appel de la primitive `getInterestRate` stocke son résultat dans la clé `interestRate` du contexte d'exécution.

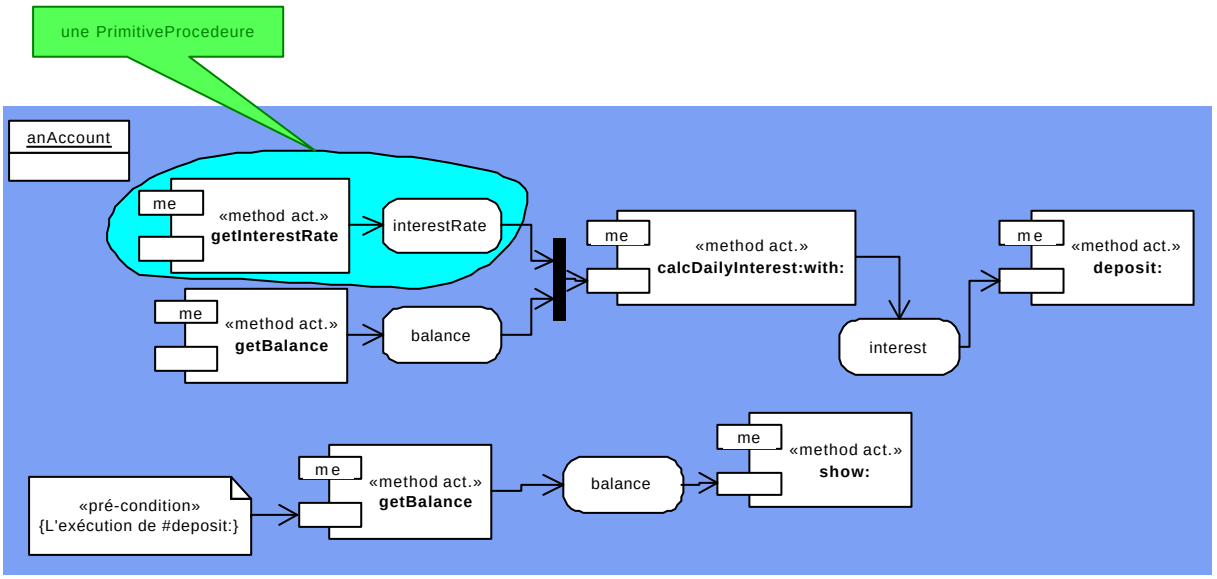


Figure 27 : Exemple de micro-procédé à la Micro-workflow.

La seconde solution est celle de DART qui intègre la notion d'instances de descriptif de service (représentés par l'abstraction `ExpressionEvaluationDefinition`). Comme permet de l'illustrer la Figure 28 ci-dessous le rôle de celle-ci est de regrouper au sein d'un même objet les deux éléments de la définition de une étape dans un procédé, c'est à dire l'objet qui représente l'étape elle-même et le nom de son résultat, ici l'appel de la primitive `getInterestRate` et la clé `interestRate`.

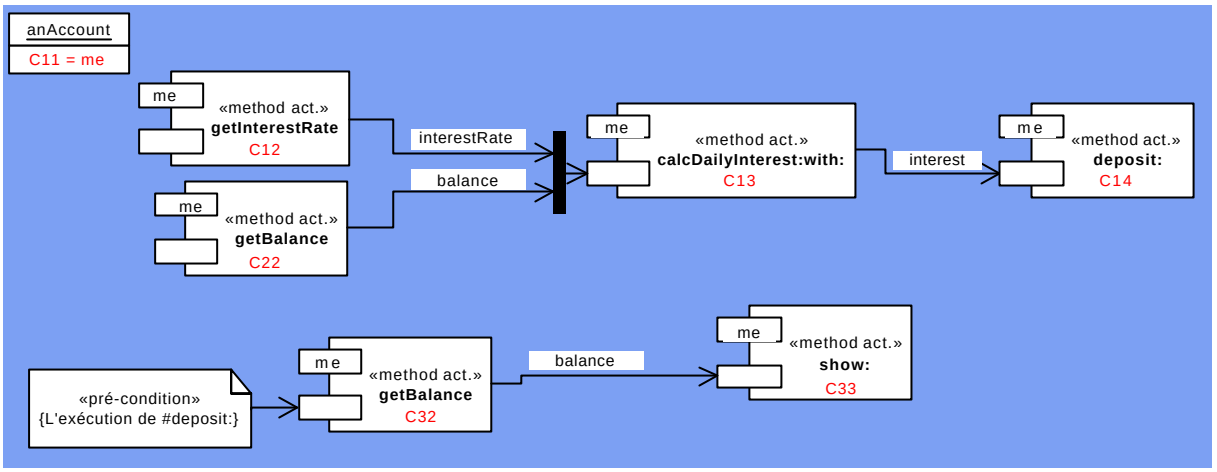


Figure 28 : Exemple de la Figure 27 sous forme de micro-composition, à la DART.

Cette modélisation explicite des « variables locales » a des conséquences importantes. En particulier, dans la mesure ou les arguments ne sont plus des simples clés mais des objets de plein droit, il devient possible, comme le met en œuvre DART, de faire varier la nature de ces objets afin d'obtenir des comportements différents lors de l'exécution, e.g. des arguments qui s'évaluent avant de retourner leur valeur ou qui mettent en œuvre le schéma Observer pour avertir leur dépendants de leur actions, e.g ; les changements de valeur. C'est aussi cette technique qui permet de résoudre le problème de couplage (cf. le paragraphe suivant) en s'appuyant sur les différentes formes de descriptifs de service. Elle est également la clé de la solution à l'appel des sous-procédures, à la modélisation des arguments et à l'usage des expressions écrites suivant les schémas de conception *Interpreter* et *Composite*.

3.3.2.2 Vision trop restreinte de services

Une autre insuffisance du Micro-workflow est qu'il suppose systématiquement que le service élémentaire dont le nom est contenu dans la variable d'instance selector correspond à une méthode implantée dans la classe de l'objet responsable de l'exécution de ce service. Ce dernier est désigné par une clé contenue dans la variable d'instance subject. Lors de l'exécution, cette clé est supposée pointer sur un objet qui est "l'exécuteur" effectif du service.

Or, dans le cas des modèles objets adaptatifs, le subject peut être instance d'un complément de classe et de ce fait ne pas avoir une implantation sous forme de méthode du service demandé. Dart apporte une solution à ce problème à travers l'extension de la notion de service à une série de cas, tel que nous avons décris dans le §2.3.4, page 33 de l'introduction.

4 Exemple : adaptation de comptes bancaires (problème de couplage)

Cette dernière section de ce chapitre est consacrée à l'étude du couplage du DOM et du Micro-workflow. Nous voulons montrer qu'une simple juxtaposition de ces deux modèles ne suffit pas pour assurer l'usage des structures dynamiquement définies dans la définition des procédures également définies dynamiquement.

4.1 Ce qui est possible avec le *Micro-workflow*

Le Micro-workflow est conçu à l'usage des programmeurs. Il intègre explicitement un modèle de définition de procédures dans un langage à objets. Dans ce contexte il y a nécessité d'un "pont" entre les deux "univers" objets et procédures. Le but est de permettre l'usage des objets dans la réalisation des calculs et le transfert des résultats de calculs effectués par des procédures dans l'univers des objets. Il faut aussi une entité qui réalise le calcul à proprement parler.

Le Micro-worklfow met ce pont en œuvre à l'aide des "primitives". Cette notion est représentée dans le système par la classe `PrimitiveProcedure.` Une primitive est conçu suivant une vision très orienté-objet et orienté envoi de message d'un calcul :

1. les données nécessaires à la réalisation d'un calcul sont transférées de l'univers des objets dans le contexte d'exécution de la procédure par l'envoi de message ;
2. le calcul lui même est réalisé par un envoi de message ;
3. le résultat du calcul est transféré depuis le contexte d'exécution des procédures dans le monde des objets, également par un envoi de message.

Cette logique est implantée par la méthode `executeDomainOperationIn:` de la classe `PrimitiveProcedure`. Dans toutes ces situations, l'hypothèse de base est l'existence des méthodes qui implantent les messages.

A titre d'exemple, le script SMALLTALK de la Figure 29 ci-dessous décrit le traitement des intérêts journaliers suivant la logique suivante[75] :

1. récupérer le montant du taux d'intérêt du compte concerné.
2. récupérer le solde du compte.
3. calculer le montant des intérêts journaliers.
4. ajouter ce montant au solde du compte.
5. relire le solde du compte [76]
6. afficher (imprimer) le nouveau solde du compte.

[75] Cet exemple est emprunté de l'article sur le DOM [RTJ00].

[76] Il est important de noter que l'expression d'envoi de message `getBalance` y apparaît deux fois. En effet, la seconde expression est nécessaire car l'exécution de la méthode `deposit:` (n° 4) fait changer la valeur du solde de l'objet compte, sans toutefois changer la valeur du solde qui a déjà été transférée dans le contexte d'exécution de cette procédure (n° 2). Le but de ce second appel est de mettre à jour ce contexte.
Il est évident que cette contrainte qui est liée à la conception de micro-workflow pénalise ses utilisateurs. En l'occurrence un langage d'expert ne peut pas exiger le respect de ce mode d'emploi contraignant aux experts.

```
accrueDailyInterest
        | interest interestRate |
        interestRate := self getInterestRate.
        balance := self getBalance.
        interest := self
                            calcDailyInterest: balance
                            with: interestRate.
        self deposit: interest
        balance := self getBalance.
        self show: balance
```

Figure 29 : Script SMALLTALK du traitement des intérêts journaliers (logique micro-workflow).

En l'occurrence ce calcul, écrit en Micro-workflow dans la Figure 30 ci-dessous, est basé sur l'hypothèse que les méthodes `getInterestRate`, `getBalance`, `calcDailyInterest:with:`, `deposit:` et `show:` existent dans la classe `SavingsAccount`[77].

```
        | getBalance showBalance thePrograme getInterestRate calcDailyInterest
depositInterest newBalance |

        getInterestRate := PrimitiveProcedure
                            sends: #getInterestRate
                            to: #myAccount
                            result: #interestRate.
        getBalance := PrimitiveProcedure
                            sends: #getBalance
                            to: #myAccount
                            result: #balance.
        calcDailyInterest := PrimitiveProcedure
                            sends: #calcDailyInterest:with:
                            with: #(balance interestRate)
                            to: #myAccount
                            result: #interest.
        depositInterest := PrimitiveProcedure
                            sends: #deposit:
                            with: #interest
                            to: #myAccount.
        newBalance := PrimitiveProcedure
                            sends: #getBalance
                            to: #myAccount
                            result: #balance.
        showBalance := PrimitiveProcedure
                            sends: #show:
                            with: #balance
                            to: #myAccount.
        thePrograme     :=     getInterestRate,     getBalance,     calcDailyInterest,
depositInterest, newBalance, showBalance.
        ^thePrograme
```

Figure 30 : Script SMALLTALK de la Figure 29 écrit en micro-workflow.

Dans ces conditions, le script de la Figure 30 s'exécute normalement. Cet exemple est donc représentatif des choses qui sont possibles avec le Micro-workflow. D'autres exemples détaillés sont fournis dans [MJ99c].

[77] Au niveau du code source, ce script est stocké dans la méthode `exampleStandardMFW01` de la méta-classe de `SavingsAccount class`.

4.2 Ce qui n'est pas possible avec le *Micro-workflow*

Supposons à présent que la classe `SavingsAccount` ne soit plus codée par les programmeurs, mais qu'elle soit définie dynamiquement par l'adaptation de la classe `Account`. Un problème majeur se pose alors tout de suite à l'exécution du micro-procédé de la Figure 30. En effet, les méthodes qui, dans l'exemple précédent, étaient implantées dans la classe `SavingsAccount` ne peuvent plus l'être, car cette classe n'existe plus en tant qu'une classe, mais un (simple[78]) complément de classe.

La Figure 31 permet de mieux illustrer ce problème[79]. Elle montre en effet, dans un premier temps l'adaptation de la classe `Account` en créant le complément de classe appelé `Savings Account` qui comporte deux descriptifs d'attributs `ownerId` et `interestRate`.

Ensuite une "instance" de cette nouvelle classe "`Savings Account`" est créée. Celle-ci comporte la valeur 1000 pour la variable d'instance balance et les valeurs Reza Razavi et 0.04 pour les attributs `ownerId` et `interestRate`.

La dernière étape consiste à écrire le micro-procédé de la Figure 30 et à l'exécuter dans le contexte de l'instance créée ci-dessus.

Toutefois, cette exécution ***échoue*** car parmi ces cinq méthodes `getInterestRate`, `getBalance`, `calcDailyInterest:with:`, `deposit:` et `show:` mentionnées ci-dessus, trois n'existent pas : `getInterestRate`, `getBalance`, `calcDailyInterest:with:`.

En effet, les deux méthodes `getInterestRate` et `getBalance` servent à accéder en lecture aux variables d'instances qui sont à présent définies dynamiquement comme des descriptifs d'attributs. Quant à la méthode `calcDailyInterest:with:`, elle sert ici à illustrer le fait que l'ajout des compléments de classe peut conduire à la nécessité d'ajouter de nouvelles fonctions primitives. Cela pose alors notamment le problème de la responsabilité d'implantation de ces primitives ainsi que la techniques déployée.

```
| dynamicSavingsAccount ownerIdPropertyTpe ratePropertyTpe myDynamicSavingsAccount
thePrograme |
      dynamicSavingsAccount := ComponentType named: 'Savings Account'.
      ownerIdPropertyTpe := PropertyType on: Core.String  named: 'ownerId'.
      ratePropertyTpe := PropertyType on: Double named: 'interestRate'.
      dynamicSavingsAccount
                  addPropertyType: ratePropertyTpe;
                  addPropertyType: ownerIdPropertyTpe.
      myDynamicSavingsAccount := Account onType: dynamicSavingsAccount.
      myDynamicSavingsAccount
            setProperty: 'balance' to: 1000;
            setValue: 'Reza Razavi' toProperty: 'ownerId';
            setValue: 0.04  toProperty: 'interestRate'.
      thePrograme := Cf. le micro-procédé de la Figure 30.
      thePrograme
            initialContextAt: #myAccount
            put: myDynamicSavingsAccount.
      ^thePrograme execute
```

Figure 31 : Script qui montre les limites du micro-workflow dans le contexte des AOMs.

Nous venons donc de constater les limites du Micro-workflow pour la définition et l'activation des micro-procédés dans le contexte de compléments de classe et des modèles objets adaptatifs de façon plus générale, ceci car cette architecture n'a pas été conçue pour être utilisée dans ce contexte. C'est ce que nous appelons le problème de couplage.

Notre solution à ce problème qui constitue la première partie de notre thèse est exposée dans le chapitre suivant.

---

[78] Puisque du point de vue de notre étude, les compléments de classes sont de nettement moins "bonnes qualités" que les classes. Pour plus de détails veuillez voir les chapitres IV et V.

[79] Ce script est stocké dans la méthode `exampleMFWLimitsInTheContextOfAOM` de la méta-classe de `Account class`.

Chapitre III : Premier outillage de l'Adaptation

Usage du schéma DOM des AOMs (DYCTALK)

Chapitre III : Premier outillage de l'Adaptation

- Usage du schéma DOM des AOMs (DYCTALK)

1 Introduction

Nous venons de montrer que la simple juxtaposition des deux technologies empruntées des travaux très récents de l'équipe de Ralph Johnson à *University of Illinois at Urbana-Champaign* (UIUC) sur l'outillage des modèles objets adaptatifs se heurte au problème de la co-évolution dynamique de structures et de procédures (problème de couplage, section 4, page 100 du chapitre II).

Ce chapitre est consacré à la résolution de ce problème et de façon plus générale à l'étude d'un premier outillage de l'adaptation sur la base du schéma DOM des AOMs (cf. la section 2.2, page 91 du chapitre II). Cela va concrètement se dérouler en deux temps. Tout d'abord nous exposons notre modèle d'analyse de la spécialisation dynamique ainsi que son modèle de conception détaillé. Ce travail nous conduit à un nouveau système de classes, appelé DARC[80] et le framework correspondant FDARC.

Le rôle principal de DARC est de montrer que la spécialisation dynamique est possible. Il apporte aussi une solution au problème de l'outillage de la dimension workflow et du lien causal (cf. section 1.2, page 17 de l'introduction).

DARC n'offre, toutefois, pas en soi une solution satisfaisante à l'outillage de l'adaptation. Aussi, nous montrons ensuite comment coupler DARC au système DART (défini dans le premier chapitre) afin d'aboutir à une solution satisfaisante, appelée le système de classes DYCRA et le framework correspondant DYCTALK.

DYCRA se caractérise par un choix subtil des éléments de ses deux composants DARC et DART. Il assure ainsi la création des langages d'experts munis de toutes les propriétés énumérées dans la section 1.2, page 17 de l'introduction, à l'exception du travail collaboratif et du choix local du type d'adaptation qui seront traités dans les deux chapitres suivants.

[80] Pour **A**rchitecture dédiée au **R**affinement **D**ynamique de **C**lasses.

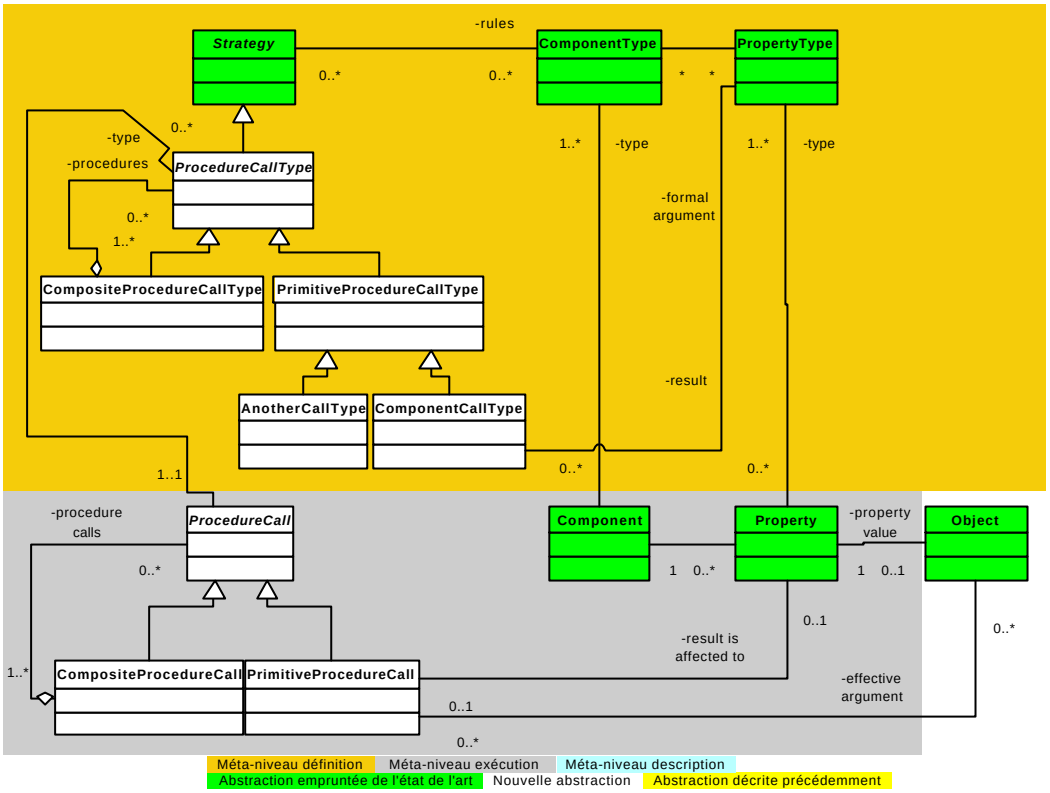


Figure 32 : Darc-I qui modélise la spécialisation dynamique (techniques standards).

Nous allons nous intéresser dans un premier temps au système DARC.

L'idée qui a guidé la définition de DARC est la suivante : comment peut-on modéliser la définition de nouveaux types d'objets, leur structure et comportements à l'aide des classes et leurs relations ? En d'autres termes, quel système de classes permet de mettre en œuvre la spécialisation dynamique ?[81]

Nous disposons déjà des éléments de solution à ce problème. C'est d'une part le schéma de conception DOM en ce qui concerne l'outillage de la définition dynamique de nouveaux types d'objets et leur structure, ainsi que leur instanciation. D'autre part le Micro-workflow outille, avec certaines limitations, la définition dynamique de procédures et leurs activations (cf. le chapitre II).

La Figure 32 ci-dessus illustre le modèle de notre proposition de couplage entre ces deux mécanismes. L'objectif premier de ce couplage est bien sûr de résoudre le problème de co-évolution dynamique de structures et de procédures, suivant la démarche que nous allons décrire.

1.1 DARC : modèle de définition de compléments de classe

En ce qui concerne la définition dynamique de nouveaux types d'objet, le modèle DARC (cf. partie haute de la Figure 32) s'appuient sur le schéma de conception DOM. La définition dynamique de procédures s'appuie sur le Micro-workflow.

Notre contribution ici consiste à modéliser la co-évolution dynamique de structure et de procédures de la façon suivante :

[81] Les deux chapitres suivants étudient l'importance de la prise en considération dans cette modélisation du fait qu'un tel mécanisme s'intègre dans un environnement de programmation par objet où il y a déjà des mécanismes existants dédiés à la mise en œuvre des objectifs similaires.

1. un *complément de classe* (instance de `ComponentType`) est modélisé par son nom, une collection de définitions d'attributs (instances de `PropertyType`), ainsi qu'une collection de définitions de procédures (instances de `ProcedureCallType`). Il sert à compléter la définition "statique" d'une classe `C` et permet ainsi de regrouper les instances de la classe `C` dans des sous-ensembles dont la structure et le comportement sont différents. Nous ajoutons donc au modèle standard DOM la "dimension comportement".
2. toute définition d'appel de service[82] (`PrimitiveProcedureCallType`) peut référencer un descriptif d'attribut comme un argument formel (lien `formal argument`) ;
3. toute définition d'appel de service peut spécifier un descriptif d'attribut comme le "receveur" du résultat de son exécution (lien `result`).

Cette modélisation permet aussi bien de transférer les valeurs des attributs dynamiques (définis à l'exécution) vers le contexte d'exécution des procédures dynamiques que le transfert du résultat d'exécution des procédures vers les objets du domaine.

1.2 DARC : modèle d'instanciation de compléments de classe

De la même façon que l'aspect définition décrit ci-dessus, DARC suit le schéma de conception DOM en ce qui concerne la modélisation de l'instanciation de nouveaux types d'objets et de leur structure. Par ailleurs, il suit le schéma de conception *Type Object* [JW97] inspiré du micro-workflow pour la modélisation de l'activation des procédures (cf. partie basse de la Figure 32). Plus de détails à ce sujet sont fournis dans le chapitre II. Il est donc inutile de revenir plus longuement ici sur ce sujet.

Notre contribution à cet égard se situe encore une fois au niveau de la modélisation de la co-évolution dynamique de structures et de procédures lors de l'instanciation. Suivant ce modèle, avant toute activation d'une procédure la valeur courante de chacun de ses arguments est calculée. Un argument non littéral est obtenu par l'évaluation de l'expression de calcul correspondant. Nous montrons ci-dessous (la section 2.5) dans quelle mesure la mise en œuvre de cette idée s'appuie sur des mécanismes empruntés du système DART.

Un cas d'usage de ces expressions est celui de l'accès en lecture aux attributs dynamiques. Celui-ci assure l'usage de la valeur courante d'un tel attribut lors de l'exécution d'une procédure. Un autre cas est celui de l'accès en écriture aux attributs dynamiques. Ceci assure l'affectation du résultat d'un calcul à un attribut dynamique.

2 Mise en œuvre de DYCRA à l'aide de techniques standard

Notre validation expérimentale du modèle DARC va se dérouler en deux grandes étapes.

La première étape se divise en deux sous-étapes. La première consiste à créer un framework, FDOM, dont on peut dire que le cahier des charges est défini par le schéma de conception DOM (cf. la section 2.1, page 110 ci-dessous). Nous construisons ce framework par une démarche progressive en quatre étapes. L'idée de cette approche progressive vient initialement du papier sur DOM [RTJ00]. Toutefois, la conception présentée ici est notre œuvre.

La seconde sous-étape consiste à créer un framework suivant les spécifications du micro-workflow que nous appelons DYCFLOW (cf. la section 2.2, page 119 ci-dessous)[83].

La phase finale de cette étape consiste à coupler ces deux composants ensemble afin d'obtenir une solution opérationnelle au problème de co-évolution (cf. la section 2.3, page 123 ci-dessous). Ce travail conduit au système de classes DARC et le framework correspondant FDARC.

[82] Ce qui représente une calcul/action dans une procédures/procédé.

[83] En effet, Dragos Manolescu spécifie en détail ce système dans sa thèse mais le code de son système n'est pas encore rendu public [MJ01]. Cela nous a conduit à développer ce framework en suivant scrupuleusement ses spécifications.

Dans la mesure où le système DARC n'offre pas une solution tout à fait satisfaisante, et notamment ne modélise pas la mise en œuvre par des experts, la seconde étape consiste à coupler DARC avec DART (décrit dans le chapitre I). Ce travail nous conduit enfin au système de classe DYCRA et son framework DYCTALK, qui permet de valider la première partie de notre thèse (cf. la section 1.5.2, page 25 de l'introduction).

DYCTALK est développé à l'aide du système VISUALWORKS/Cincom version NC 5i.3 [Cin01] et le langage SMALLTALK-80[84].

2.1 FDOM : un framework documenté par DOM

La définition dynamique de nouveaux types d'objets et de leur structure peut être conçue sous différentes formes. Nous allons ici en étudier cinq.

La Figure 33 ci-dessous illustre le modèle de conception du framework FDOM issu de ce travail que nous allons détailler dans cette section.

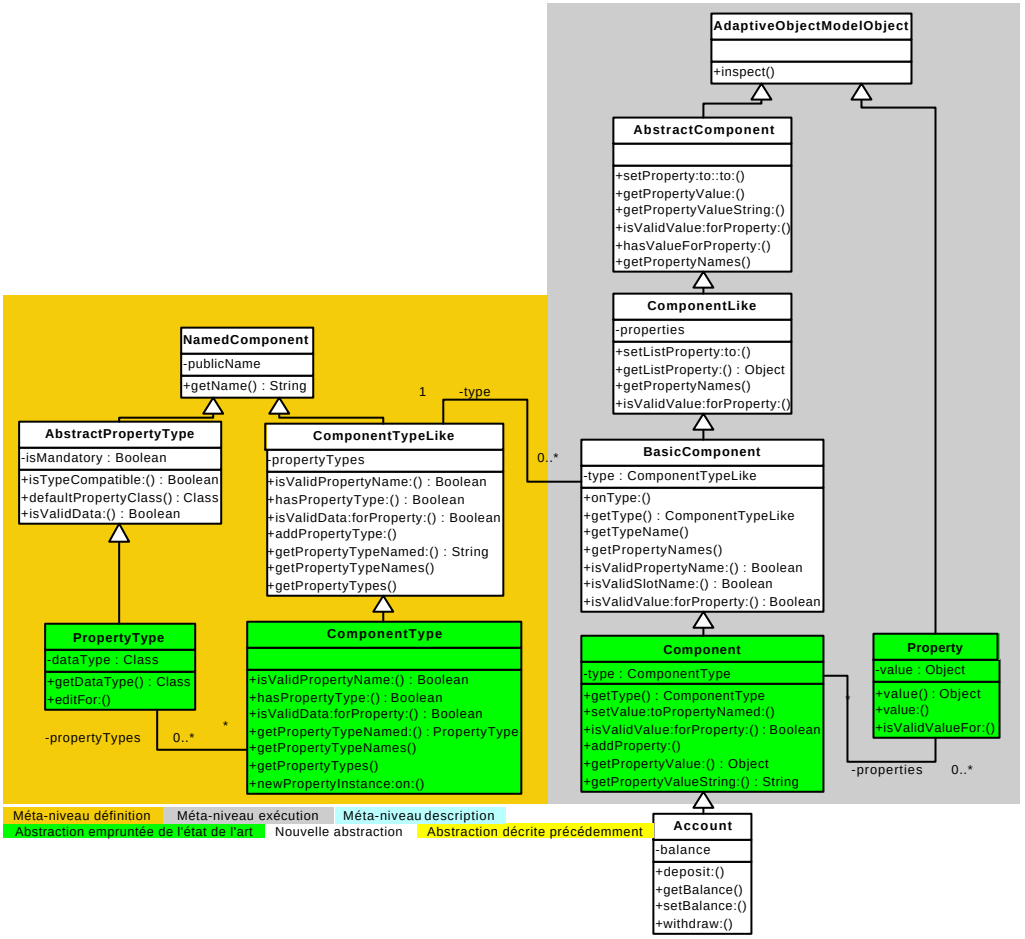


Figure 33 : Modèle de conception du framework FDOM.

[84] Le langage SMALLTALK-80 offre, en effet, des facilités de programmation qui permettent une mise en œuvre plus efficace. Nous estimons que cette caractéristiques est un plus important dans le cadre des travaux de recherche, où les ressources allouées au développement d'un système sont limitées et où la nature du problème traité requiert des expérimentations et modifications récurrentes au niveau de la conception et de l'implantation du système.

2.1.1 Un système élémentaire à deux classes

Un premier système élémentaire qui modélise la définition dynamique de nouveaux types d'objets est celui décrit par le schéma de conception *Type Object* [JW97]. Celui-ci s'applique à la conception des systèmes où il existe des objets qui possèdent une relation du type classe/instance entre eux.

Par exemple, chaque copie d'un livre disponible dans la réserve d'une bibliothèque peut être considérée comme une instance de ce livre. En effet, chaque copie dispose des informations qui lui sont propres, e.g. le nom de l'emprunteur. Le schéma *Type Object* propose de modéliser cette situation avec deux classes. Ici, la classe `Livre` qui modélise le livre et la classe `Exemplaire` qui modélise les copies[85]. Plusieurs instances de la classe `Exemplaire` sont associées à une même instance de la classe `Livre` (le livre).

Cette modélisation suppose, toutefois, que la structure et le comportement des abstractions concernées (ici `Livre` et `Exemplaire`) soient entièrement connus *a priori* (lors de la création du système). Il ne s'agit donc pas d'une technique réutilisable pour la définition systématique de nouveaux type d'objets quelconques. Cela pouvait, à titre d'exemple, permettre ici de modifier la structure ou le comportement de la classe `Livre`. On obtiendrait alors de nouveaux types de livre. Cela voudrait dire que chaque instance de la classe `Livre` joue le rôle d'un complément de classe pour la classe `Exemplaire`. Nous reviendrons sur cette question dans les sous-sections qui suivent.

Nous entamons ici la tournée progressive, promise eu début de ce chapitre, de construction du framework FDOM. Cette tournée est composée de quatre étapes et va nous conduire progressivement à un système de quatre classes. Celui-ci est conçu pour outiller la création de classes dont la structure peut être spécialisée à l'exécution et cela de façon contrôlée (nous verrons ici ce que ce contrôle représente précisément).

2.1.2 Retour provisoire à un système mono classe

Dans cette sous-section nous apportons une première réponse à cette question. Le but est de rendre la structure de chaque classe adaptable à l'échelle de chaque instance.

Cette solution consiste à créer une abstraction qui met en œuvre le schéma de conception *Property List* [FY98b, Rie97a]. Le but est de permettre d'ajouter des couples (nom d'attribut, valeur) au niveau de chaque objet. Le principe de cette solution consiste à utiliser un dictionnaire dont le rôle est de contenir des associations rajoutées lors de l'exécution du système. Chaque association est composée du nom d'un attribut et de sa valeur. Ainsi, à titre d'exemple, toute instance de la classe `Account` pourra avoir un nombre quelconque de couples d'attributs/valeurs. Pour ce faire, il faut que la classe `Account` hérite de la classe qui met en œuvre cette mécanique (cf. ci-dessous la classe `AbstractComponent` et ses sous-classes).

Cette conception nous conduit à la première classe concrète de notre framework, `ComponentLike` (cf. la Figure 33 ci-dessus). L'implantation de cette abstraction s'appuie sur d'autres classes : `AdaptiveObjectModelObject` et `AbstractComponent.` Celles-ci servent de racine commune à la hiérarchisation pratique des classes essentielles du framework FDOM.

[85] Une analogie semble ici possible avec les bases de données relationnelles. Il s'agit, là, de créer une table pour chaque catégorie d'instances, et inclure dans le schéma de la table une clé vers l'objet qui représente sa classe dans une autre table.

2.1.2.1 Principales abstractions

La classe `AdaptiveObjectModelObject`

La classe `AdaptiveObjectModelObject` est la super classe de toutes les classes du framework FDOM. Elle implante quelques fonctions utilitaires (`isValidName:`, `inspect` et `show:`).

La classe `AbstractComponent`

La classe `AbstractComponent` modélise les objets dont la structure est définie en deux temps. Une partie à l'aide des variables d'instances (par les programmeurs) et une partie à l'aide des descriptifs d'attributs (par des experts). Elle est une sous-classe de la classe `AdaptiveObjectModelObject`.

Suivant la terminologie déployée par le schéma de conception DOM, un slot défini à l'aide d'une variable d'instance est dit du type *field* et un slot défini à l'aide d'un descriptif d'attribut est dit du type *list*. Les classes concrètes doivent implanter les méthodes du protocole 'subclass responsibility'. Celles-ci concernent la gestion des slots du type *list*:

1. `getListProperty:` retourne la valeur associée à un attribut dont le nom est passé en argument.
2. `getPropertyNames` retourne la liste des noms de tout les attributs d'un objet.
3. `isValidValue:forProperty:` vérifie si la valeur du premier argument peut être affectée à l'attribut dont le nom est en second argument.
4. `setListProperty:to:` affecte l'objet reçu en premier argument à l'attribut dont le nom est en second argument. Un appel à `isValidValue:forProperty:` doit précéder celui-ci pour assurer la validité de l'argument.

Le protocole de cette classe est également composé des méthodes suivantes :

1. `setProperty:to:` affecte l'objet reçu en premier argument à slot dont le nom est passé en second argument. Ce slot peut être du type *field* ou *list*.
2. `hasValueForProperty:` teste si une valeur est déjà affecté à l'attribut dont le nom est passé en argument.
3. `getProperty:` (ou `getPropertyValue:`) retourne la valeur d'un slot dont le nom est passé en argument. Ce slot peut être du type *field* ou *list*.
4. `getPropertyValueString:` retourne la valeur d'un slot dont le nom est passé en argument sous forme d'une chaîne de caractères imprimable.

La classe `ComponentLike`

La classe `ComponentLike` met en oeuvre une première implantation concrète du protocole de gestion de descriptifs d'attributs, mis en place par sa super classe `AbstractComponent`. La Figure 34 illustre un exemple d'usage de cette classe.

```
ComponentLike new
             setProperty: 'amount' to: 1000;
             setProperty: 'amount' to: #($a $b $c);

             setProperty: 'name' to: 'Jabberwocky';
             inspect;
             showProperty: 'zork'
```

Figure 34 : Exemple d'usage de la classe `ComponentLike`.

L'exécution de cet exemple produit le résultat suivant (cf. la Figure 35) :

```
My property values are as follows:
amount = #($a "16r0061" $b "16r0062" $c "16r0063")
name = 'Jabberwocky'
zork = nil
```

Figure 35 : Le résultat de l'exécution de l'exemple de la Figure 34.

Cet exemple, tiré de [RTJ00], veut d'abord montrer le manque de contrôle sur l'ajout des attributs. En effet, les attributs `amount` et `name` sont ajoutés arbitrairement au niveau de l'objet. Le système ne proteste non plus pas à la demande d'affichage de la valeur de l'attribut `zork` qui n'est même pas ajouté. Il envoie simplement la valeur `nil`. De plus, des valeurs de types différents peuvent être affectées à un même attribut, ici `amount`.

La mise en œuvre du protocole d'accès aux attributs, défini ci-dessus, s'appuie ici sur une variable d'instance `properties`. Celle-ci référence une table de hachage (`IdentityDictionary` en SMALLTALK-80) qui contient des couples (nom d'attribut, valeur d'attribut). Les attributs ne peuvent pas ici être ordonnés.

2.1.2.2 Conclusion

La modification non contrôlée de la structure des objets n'est, du point de vue de la modélisation par objets, pas tolérable. En effet, dans une application réelle il n'est envisageable de modifier la structure des objets que dans des conditions très strictes. Par exemple, un compte anonyme dans une banque Suisse ne peut pas avoir un attribut nom du titulaire [RTJ00].

Cette modélisation n'est donc pas satisfaisante. En somme, l'existence de la classe `ComponentLike` se justifie par le fait qu'elle permet une meilleure appréciation de l'intérêt des compléments de classe. Ces derniers permettent de résoudre les problèmes cités ci-dessus, comme le contrôle d'ajout des attributs et du type ainsi que la gestion de l'ordonnancement des attributs (cf. la classe `ComponentType` décrite ci-dessous).

2.1.3 Un système à deux classes plus élaboré

Dans la sous-section 4.1 ci-dessus, nous avons exposé une première version d'un système à deux classes. Dans ce cas le schéma de conception *Type Object* modélisait une relation du type classe/instance comme celle qui existe entre un livre et ses exemplaires.

Nous allons ici utiliser une conception semblable qui, toutefois, sert à associer à chaque objet de façon *systématique* un complément de classe. Chaque complément de classe `CC` détient la liste des descriptifs d'attributs autorisés pour un ensemble d'instances de la classe `C` à laquelle elle est associée.

Cette conception requiert, en outre, que la classe `C` gère la valeur des descriptifs d'attributs. Pour ce faire `C` doit implanter ou hériter le protocole défini en sous-section 4.2 ci-dessus.

Nous avons à présent un système composé de deux classes. Il est conçu par l'application du schéma de conception *Type Object* et deux applications du schéma de conception *Property List*[86]. Les deux classes principales de ce système sont appelées `ComponentTypeLike` et `BasicComponent`.

A noter que le système mono-classe de la section 4.2 applique déjà le schéma de conception *Property List* pour permettre à la classe ainsi modélisée de détenir une liste quelconque d'attributs et de leur

[86] Le système Smalltalk-80 utilise une conception semblable pour son noyau classe/méta-classe qui sert à définir de nouvelles classes (spécialisations de la classe Object).

valeurs. Nous appliquons ici également ce schéma à la modélisation des compléments de classe. Ce choix permet de modéliser la définition de la liste des descriptifs d'attributs associés à un complément de classe.

En ce qui concerne le schéma *Type Object*, il sert ici à modéliser la relation entre une classe et le modèle de ses compléments de classe. C'est, à titre exemple, le cas dans un système qui associe une classe `AccountType` à la classe `Account`. Chaque instance de la classe `AccountType` sert à spécialiser la structure et le comportement d'un sous-ensemble d'instances de la classe `Account`.

2.1.3.1 Principales abstractions

La classe `NamedComponent`

La classe `NamedComponent` représente les objets nommés dont l'identifiant est une chaîne de caractères ayant une signification claire pour les utilisateurs. Ceci constitue une des différences entre un objet issu d'un discours tenu par un programmeur et destiné à un compilateur, et un discours tenu par un expert non informaticien et destiné aux utilisateurs non informaticiens. Même si ce rajout ne représente pas de difficulté majeure sur le plan technique, mais il est indispensable au bon fonctionnement des langages d'experts. Les classes `ComponentType` et `PropertyType` que nous décrirons ci-dessous sont des exemples d'objets nommés.

La variable d'instance `publicName` contient une chaîne de caractères. Celle-ci sert de clé discriminante pour identifier de façon unique un objet nommé parmi un ensemble d'objets nommés.

La méta-classe de cette classe propose la méthode `named:` pour créer une nouvelle instance en fournissant son nom. Elle redéfinie alors la méthode `new` afin de déclencher une exception et indiquer la nécessité de faire appel à la méthode `named:`.

La classe `ComponentTypeLike`

Chaque instance de la classe `ComponentTypeLike` correspond à un complément de classe[87] (encore un peu primitif). Elle stocke son nom ainsi que sa liste des descriptifs d'attributs[88]. Elle hérite la gestion du nom de sa super-classe `NamedComponent`. Elle ajoute à sa super-classe une variable d'instance `propertyTypes` qui stocke dans une collection ordonnée la liste des descriptifs d'attributs. C'est ainsi que l'ordonnancement des attributs est géré par ce système.

`ComponentTypeLike` permet ainsi de créer un nouveau type d'objet par son nom et par l'énumération de ses descriptifs d'attributs. Pour ce faire elle offre le protocole suivant :

1. `addPropertyType:` ajoute un nouveau descriptif d'attribut.
2. `getPropertyTypeNamed:` retourne le descriptif d'attribut dont le nom est passé en argument.
3. `newPropertyInstance:on:` crée une nouvelle instance d'attribut pour le descriptif d'attribut dont le nom est passé en premier argument. Cette instance va porter la valeur reçue en second argument.
4. `removePropertyType:` supprime un nouveau descriptif d'attribut.
5. `hasPropertyType:` vérifie si l'argument correspond à un descriptif d'attribut déjà défini.
6. `isValidPropertyName:` vérifie si l'argument est une propriété valide, c'est à dire s'il correspond au nom de l'un des descriptif d'attributs gérés par le receveur.
7. `isValidData:forProperty:` vérifie si la valeur reçu en argument peut être affecté à un descriptif d'attribut reçu en second argument.

[87] Notion comparable à celle de méta-objet au sens des langages à objets réflexifs [Raz00a].
[88] Plus bas dans ce document, la section 0, nous spécialisons cette classe afin de lui associer également la gestion de descriptifs de procédures.

Pour l'heure, la variable d'instance `propertyTypes` est une collection ordonnée (`OrderedCollection` en SMALTALK-80) des chaînes de caractères. Chaque chaîne de caractères désigne un descriptif d'attribut. De ce fait, cette modélisation ne solutionne pas le problème de la vérification du type de la valeur des attributs. Une solution plus complète (avec la gestion des types) est décrite plus bas dans les sous-sections 4.4 et 4.5.

La classe `BasicComponent`

La conception de compléments de classe requiert également l'ajout d'un lien qui associe chaque instance d'une classe adaptable (ici sous classe de `AbstractComponent` décrite ci-dessus) à son complément de classe. Ce lien permet à chaque instance d'une classe adaptable de se référer à son complément de classe afin de lui confier les contrôles nécessaires comme la possibilité d'ajout à un objet d'un attribut[89].

Ce lien est mis en œuvre par la création d'une sous-classe de la classe `ComponentLike`. C'est la classe `BasicComponent`. Celle-ci détient une variable d'instance `type` qui matérialise ce lien. Cela permet alors à tout objet de ce type de déléguer les messages comme `getPropertyNames`, `isValidPropertyName:` et `isValidValue:forProperty:` à son "type" (le complément de classe).

La méthode de classe *onType:* permet de créer une nouvelle instance et de lui affecter son complément de classe.

2.1.3.2 Conclusion

L'implantation de notre système à deux classes ne satisfait pas encore entièrement le cahier des charges décrit par le schéma de conception DOM. En effet, la notion de descriptif d'attribut n'est pas encore à ce niveau réifiée. Un descriptif d'attribut est pour l'instant représenté simplement par son nom (une chaîne de caractères).

A titre d'exemple, la Figure 36 ci-dessous montre un script écrit en SMALLTALK-80 (toujours basé sur [RTJ00]) qui définit dans un premier temps un complément de classe comme une instance de la classe `ComponentType` dont le nom est `My Special Account Type` (la variable locale aCompType). Celui-ci comporte deux descriptifs d'attributs : `amount` et `name`.

Ensuite, ce complément de classe est utilisé pour *adapter* la classe `BasicComponent` (ou une sous-classe comme `Account`). En effet, cette association permet d'ajouter à une instance (ici aComp) de cette classe des valeurs pour les deux attributs `amount` et `name`, alors que la classe concernée, ici `BasicComponent`, ne le prévoit pas.

```
| aComp aCompType |
aCompType := ComponentType named: 'My Special Account Type'.
aCompType
            addPropertyType: 'amount';
            addPropertyType: 'name'.
aComp := BasicComponent onType: aCompType.
aComp
      setProperty: 'amount' to: 1000;
      setProperty: 'name' to: 'Jabberwocky';
      inspect
```

Figure 36 : Exemple de complément de classe qui adapte la classe `BasicComponent`.

[89] A noter que chaque classe adaptable peut également avoir des variables d'instance. On peut dire que celles-ci décrivent la structure "statique" de ses instances.

L'exécution de cet exemple produit l'affichage illustré par la Figure 37 ci-dessous:

```
"My Special Account Type" has the following property types:
        -Property type named: "amount"
        -Property type named: "name"
My property values are as follows:
        -The value of String "amount", considered as a property, is equal to 1000
        -The value of String "name", considered as a property, is equal to 'Jabberwocky'
```

Figure 37 : Le résultat de l'exécution du script de la Figure 36.

Par ailleurs, comme permet de le constater la Figure 38 ci-dessous, le système actuel ne permet pas d'associer une valeur à un attribut dont le descriptif n'a pas été défini auparavant au sein du complément de classe (ici cas de l'attribut `zork`). Son inconvénient est qu'il ne permet pas de contrôler le types des valeurs affectées à un attribut (cas de l'attribut `amount` qui accepte comme valeur le tableau de caractères composé de $a $b et $c).

```
aComp
        setProperty: 'amount' to: 1000;
        setProperty: 'amount' to: #($a $b $c)

aComp
        setProperty: 'zork' to: #zork
```

Figure 38 : Inconvénient du système à deux classes : manque de contrôle du type de valeurs.

2.1.4 Un système à trois classes

Notre framework ne peut pas actuellement exercer un contrôle sur le type de la valeur affectée à un attribut. Une solution raisonnable à ce problème consiste à associer l'information sur le type à chaque descriptif d'attribut et de confier au complément de classe concerné la vérification des types. Par exemple, le descriptif d'attribut `amount` est, *a priori*, un nombre réel. Il est pertinent d'associer cette information au descriptif. Aussi, avant chaque affectation d'une valeur à un tel attribut, l'objet concerné s'adresse à son complément de classe pour qu'il effectue le contrôle de la compatibilité de type sur la valeur à affecter (cf. la méthode `isTypeCompatible:` ci-dessus).

Pour mettre cette idée en œuvre il faut remplacer l'usage des chaînes de caractères pour représenter les descriptifs d'attributs par un objet plus élaboré et susceptible de détenir une information sur le type des données autorisées. C'est lui qui va alors réaliser les vérifications nécessaires lors de l'affectation[90].

L'impact de cette solution sur notre framework est double. Elle nécessite :

1. la création d'une nouvelle classe `PropertyType`, qui représente la forme réifiée des descriptifs d'attributs ;
2. la création d'une nouvelle classe `ComponentType` qui gère la forme réifiée des descriptifs d'attributs, c'est-à-dire les `PropertyType`. `ComponentType` est une spécialisation de la classe `ComponentLikeType`.

[90] Ce système correspond à l'ajout au langage SMALLATLK-80 (conçu, en ce qui concerne la gestion des variables d'instances, suivant notre système à deux classe plus élaboré) d'un mécanisme de contrôle dynamique des valeurs affectées aux variables d'instances.

2.1.4.1 Principales abstractions

La classe `AbstractPropertyType`

La classe `AbstractPropertyType` implante le protocole des descriptifs d'attributs, conformément au schéma de conception *Dynamic Object Model*. Elle est implantée sous forme d'une classe abstraite pour permettre des raffinements supplémentaires (cf. sa sous-classe `PropertyType`).

`AbstractPropertyType` propose un protocole public composé de la méthode `isValidData:` qui contrôle la validité des valeurs à affecter à un attribut. Cette méthode utilise le schéma de conception *Template Method* [ABW98] afin de déléguer à ses sous-classes l'implantation effective des méthodes qui compose l'algorithme de vérification. Il s'agit des deux méthodes `isTypeCompatible:` et `satisfiesConstraintes:`. La première est une méthode abstraite et la seconde exige par défaut que la valeur à affecter soit non nulle si l'attribut est obligatoire.

`AbstractPropertyType` a également une variable d'instance `isMandatory` qui sert à fixer à la nature optionnelle ou obligatoire de l'attribut. La sémantique par défaut est qu'un type d'objet doit obligatoirement avoir une valeur pour tout attribut dont le type est obligatoire.

La classe `PropertyType`

La classe PropertyType ajoute à sa super-classe `AbstractPropertyType` une variable d'instance `dataType`. Celle-ci pointe sur la classe dont les instances sont des valeurs acceptable pour cet attribut. Cela permet à PropertyType de finaliser l'implantation du protocole des descriptifs d'attributs en donnant une implantation concrète à la méthode `isTypeCompatible:`. Cette méthode vérifie simplement que la classe de l'objet reçu en argument (la valeur à affecter) est celle référencée par la variable d'instance `dataType` ou une sous-classe de celle-ci.

La classe `ComponentType`

Le rôle de la classe `ComponentType` est d'implanter, enfin, les compléments de classes, tels qu'ils sont définis par le schéma de conception *Dynamic Object Model* [RTJ00]. Par rapport à son ancêtre, la classe `ComponentLikeType`, elle ajoute la réification du concept de descriptif d'attribut.

```
| aCompType |
aCompType := ComponentType named: 'My Special Account Type'.
^aCompType
      addPropertyType: (PropertyType on: Number named: 'amount');
      addPropertyType: (PropertyType on: String named: 'name').
```

Figure 39 : Instanicer `ComponentType` pour créer un complément de classe.

2.1.4.2 Conclusion

Nous disposons donc d'un système qui permet 1) de spécialiser la structure d'une classe à l'exécution; 2) d'effectuer un contrôle sur l'ajout et la suppression des attributs ; et 3) de contrôler le type de la valeur affectée à chaque attribut. Ce système apporte déjà une solution satisfaisante à notre problème.

Toutefois, la pratique montre, e.g., dans le cas du système CALIBRES [Raz00a], qu'il est aussi important d'avoir une représentation plus élaborée de la notion d'attribut. Cela permet, par exemple, de stocker la valeur de chaque attribut tel qu'elle a été saisit à l'écran (sous forme d'une chaîne de caractères) et de la transforme dans un format compatible avec le type de l'attribut à la demande. Nous allons donc procéder à une dernière étape de modélisation afin d'ajouter cette caractéristique à notre framework.

2.1.5 Un système à quatre classes

La mise en œuvre de cette dernière étape est très simple. Il s'agit d'ajouter une nouvelle classe, `Property`, qui réifie le concept d'attribut suivant le schéma de conception *Value Holder* [Rie97a]. Cet ajour nécessite aussi de spécialiser la classe `BasicComponent` afin qu'elle tienne compte de cette nouvelle conception des attributs.

2.1.5.1 Principales abstractions

La classe `Component`

La classe `Component` finalise notre processus de création progressive de la notion de classe adaptable, tel qu'elle est décrite par le schéma de conception DOM. Elle hérite de la classe `BasicComponent`. Elle redéfinit le protocole d'accès aux attributs décrits ci-dessous afin de tenir compte du changement de leur nature par la création de la classe `Property`.

La classe `Property`

La classe `Property` est une simple `ValueHolder` [Rie97a]. A ce titre, dans l'implantation actuelle en SMALLTLK-80, elle aurait pu hériter de la classe `ValueHolder`. Pour plus de commodité au niveau de la gestion du code du framework, nous l'avons fait hériter de la classe `ComponentObject`.

Chaque instance de `Property` permet d'accéder en lecture et en écriture à la valeur d'un d'attribut. `Property` ajoute, par ailleurs, un point d'extension au framework à travers la méthode `isValidValueFor:`. Celle-ci permet aux attributs d'avoir une possibilité de contrôle sur leur valeur.

2.1.5.2 Conclusion

Comme permet de l'illustrer la Figure 40, à présent l'instanciation d'un descriptif d'attribut est confiée au complément de classe auquel il appartient.

```
| aComp aCompType |
aCompType := ComponentType named: 'My Special Account Type'.
aCompType
      addPropertyType: (PropertyType on: Number named: 'amount');
      addPropertyType: (PropertyType on: String named: 'name').
aComp := Component onType: aCompType.
aComp
      setProperty: 'amount' to: (aCompType newPropertyInstance: 'amount' on: 1000);
      setProperty:   'name'   to:   (aCompType   newPropertyInstance:   'name'   on:
'Jabberwocky');
      inspect
```

Figure 40 : Exemple d'adaptation de la classe `Component`.

Le message `newPropertyInstance:on:` implantée dans classe `ComponentType` met en œuvre l'algorithme illustré par la Figure 41. L'essence de cet algorithme est dans le fait que le choix de la classe d'attribut à instancier dépend du descriptif d'attribut concerné (le message `defaultPropertyClass`). La valeur par défaut est la classe `Property`, mais les sous-classes peuvent fournir des classes plus spécialisées.

```
ComponentType >> newPropertyInstance: aString on: anObject
      | aPropertyType |
      (self isValidPropertyName: aString) ifFalse:
            [self notifyError: #'Property Not Defined'].
      aPropertyType := self getPropertyTypeNamed: aString.
      self mustBeValidValue: anObject forProperty: aPropertyType.
      ^aPropertyType defaultPropertyClass onValue: anObject
```

Figure 41 : Instanciation d'un descriptif d'attribut.

L'exécution de cet exemple produit l'affichage illustré par la Figure 42. Toute tentative d'affectation d'une valeur à un attribut non défini au sein du complément de classe déclenche une exception. Il en est de même en ce qui concerne l'affectation d'une valeur à un attribut avec un type incompatible.

```
"My Special Account Type" has the following property types:
        -Property type named: "amount"
        -Property type named: "name"
My property values are as follows:
        -The Property named "amount" is equal to 1000
        -The Property named "name" is equal to 'Jabberwocky'
```

Figure 42 : Exemple d'exécution du script de la Figure 40.

On dispose donc d'un framework, encore élémentaire, mais qui remplit bien son cahier des charges défini par le schéma de conception DOM. Pour créer une classe adaptable il suffit de la faire hériter de la classe `Component`.

Voici la vue d'ensemble des différents systèmes de classes définis dans cette section :

- Un système mono classe : Il gère l'évolution à l'exécution de la structure des objets. Il est composé de la classe `ComponentLike`.
- Un système à deux classes plus élaboré : il introduit la notion de complément de classe ce qui permet le contrôle de l'ajout et de la suppression (dynamique) des attributs. Il est composé des classes `BasicComponent` et `ComponentTypeLike`.
- Un système à trois classes : il ajout le contrôle sur le type de la valeur affectée à un attribut. Il est composé des classes `BasicComponent`, `ComponentType`, `PropertyType`.
- Un système à quatre classes : il ajout une représentation plus élaborée de la notion d'attribut. Il est composé des classes `Component`, `ComponentType`, `PropertyType`, `Property`.

Nous étudierons les inconvénients de l'approche promue par DOM dans le chapitre suivant. Pour l'heure nous allons montrer brièvement notre développement d'un framework conforme aux spécifications de micro-workflow.

2.2 DYCFLOW : framework conforme au Micro-workflow

Le micro-workflow dispose déjà d'une implantation sous forme d'un framework orienté-objet, développé en VISUALWORKS/Cincom version NC 3[91]. Toutefois, cette implantation n'a pas encore été rendue publique. Aussi, nous avions été amenés a procéder à une nouvelle implantation du noyau de ce

[91] Selon Dragos Manolescu ce framework a également été porté en JAVA par *Toshiba System Integration Technology Center*. Source http://micro-workflow.com/FAQ.phtml.

framework (les parties nécessaires à la réalisation de nos travaux), conformément à ses spécifications fournies dans [MJ99a, MJ99b, MJ99c, Man00].

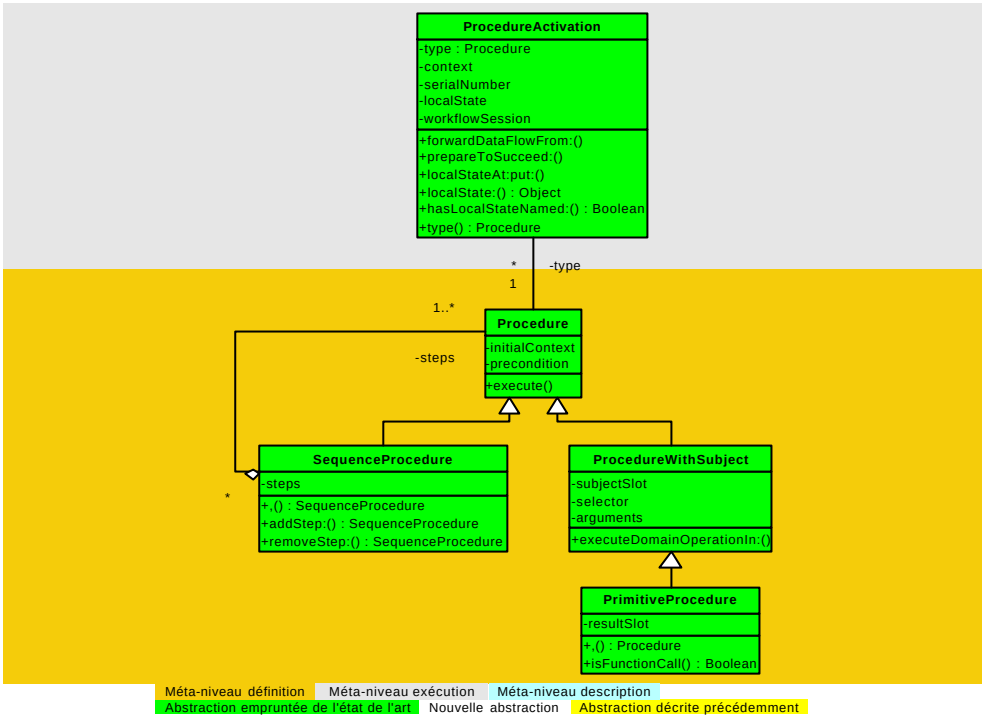


Figure 43 : Diagramme de classe UML de notre implantation du Micro-workflow (**DYCFLOW**).

La **Figure 43** illustre le diagramme de classe UML de notre implantation du framework Micro-workflow. Le but de cette implantation est de permettre de valider expérimentalement notre modèle d'adaptation (le système de classe issu du couplage DARC & DART).

Comme permet de le constater cette figure, le noyau de DYCFLOW est composé de deux sous-modèles de définition et d'exécution des micro-procédés. Notre présentation rapide suit ce découpage.

2.2.1 Définition de procédures

D. Manolescu spécifie plusieurs abstractions pour définir une procédure. `Procedure` est une classe abstraite qui agrège les différents types de procédure de ce modèle. Ses différentes sous-classes concrètes sont décrites dans le chapitre II. Nous allons nous focaliser sur le sous-ensemble des classes qui nous intéressent ici plus particulièrement :

1. `PrimitiveProcedure` permet de passer le contrôle aux objets du domaine et de leur permettre ainsi de réaliser une activité (relative au domaine).
2. `SequenceProcedure` permet la description d'une séquence d'activités "primitives".

L'usage d'autres abstractions du micro-workflow dans le contexte des modèles objets adaptatifs et l'adaptation nécessite une étude supplémentaire. En effet, la conception de Dragos Manolescu prévoit un mixage lors de la définition et de l'activation des procédures des mécanismes du langage d'implantation de micro-workflow, e.g. SMALLTALK, et le micro-wokrfow lui-même. Par exemple, il autorise l'écriture du code dans le langage d'implantation pour spécifier la condition d'activation d'une primitive. Lors de l'exécution c'est l'interprète du langage hôte qui exécute cette partie de la spécification d'une procédure. Or, les langages d'experts évite ce type d'usage et préconise de modéliser ces dimensions et de les intégrer de façon appropriée au sein des langages dédiés aux experts.

2.2.1.1 Principales abstractions

La classe `Procedure`

La classe `Procedure` est une sous-classe de la classe `FlowIndependentObject`. Cette dernière est une classe vide qui sert à agréger (dans le but pratique de faciliter la gestion du code) les différentes classes du composant micro-workflow.

La classe `Procedure` dispose de deux variables d'instances : `initialContext` et `precondition`.

La variable d'instance `initialContext` référence une table de hachage (`IdentityDictionary` en SMALLTALK-80). Celle-ci sert à stocker les objets nécessaires lors de l'exécution d'une procédure mais qui sont connus lors de sa définition. Ces objets sont copiés dans le contexte de chaque activation de cette procédure.

La variable d'instance `precondition` référence de façon optionnelle une instance de la classe `Precondition`. Celle-ci correspond à la condition d'activation de la procédure, décrite sous forme d'une fermeture (instance de la classe `BlockClosure` dans le cas de SMALLTALK-80). La première étape d'activation d'une procédure consiste à vérifier si cette condition (dans le cas où elle est définie) est vérifiée. Dans le cas contraire la procédure est mise dans une file d'attente et sa pré-condition est vérifié de façon cyclique. Dès que celle-ci est vérifiée la procédure est enlevée de la file d'attente est poursuit son exécution. Notre implantation ne gère pas cet aspect car il n'est pas nécessaire à la validation de notre thèse. De plus l'usage des fermetures pour décrire la pré-condition n'est pas une bonne solution dans le cas des langages d'expert car il nécessite la programmation.

La classe `Procedure` propose aussi un protocole composé principalement des méthodes suivantes :

1. executeProcedure: est une méthode abstraite. Son implantation concrète dépend du type de la procédure (cf., à titre d'exemple, ci-dessous les classes `PrimitiveProcedure` et `SequenceProcedure`)
2. `execute` permet de déclencher l'exécution d'une procédure. Le contexte initial d'exécution est alors vide.
3. `continueExecutionOf:` permet l'exécution d'une procédure dans le contexte de l'argument. C'est cette méthode qui met véritablement en oeuvre l'algorithme d'activation. Celui-ci consiste à attendre la vérification de la pré-condition quand celle-ci est définie (envoi du message `waitUntilFulfilledIn:` à la pré-condition avec en argument le contexte d'exécution). Ensuite, il crée une nouvelle instance de la classe `ProcedureActivation` (message `getNewProcedureActivation`). C'est elle qui matérialise l'activation courante de la procédure (cf. plus bas dans cette sous-section). L'étape suivante consiste à préparer le contexte d'exécution en copiant le contexte courant, reçu en argument, dans le contexte associé à la nouvelle instance de `ProcedureActivation` (message `prepareToSucceed:`). Enfin, la procédure procède à sa propre exécution en s'envoyant le message `executeProcedure:` qui reçoit en argument la nouvelle instance de `ProcedureActivation`. Nous estimons toutefois, que le déroulement de l'exécution devait à cette étape être confiée à cette instance et non pas à la procédure elle-même.

La classe `ProcedureWithSubject`

La classe `ProcedureWithSubject` est également une classe abstraite. Elle ajoute à sa super-classe `Procedure` le protocole nécessaire à la spécification de la délégation de l'exécution des actions aux objets du domaine ainsi que la récupération du résultat de cette exécution. `ProcedureWithSubject` met ce protocole en oeuvre avec une "vision orienté-objet", c'est à dire que la délégation de l'exécution d'un service prend la forme d'un envoi de message. Pour ce faire, elle gère trois variables d'instances :

1. `selector` désigne le message à envoyer.

2. `subjectSlot` pointe sur le nom de l'objet du domaine qui reçoit le message.
3. `arguments` comprend la liste de noms des arguments.

Cette classe redéfinie également la méthode `executeProcedure:` qui appelle à son tour une nouvelle méthode abstraite `executeDomainOperationIn:`, et de plus retourne systématiquement le résultat de l'exécution de cette dernière méthode.

La classe `PrimitiveProcedure`

La classe `PrimitiveProcedure` est la première sous-classe concrète de la classe `ProcedureWithSubject.` Elle ajoute une nouvelle variable d'instance `resultSlot`. Celle-ci va contenir la clé de stockage du résultat d'activation de la primitive (une chaîne de caractères) dans le contexte courant d'exécution.

De plus elle implante la méthode `executeDomainOperationIn:`de la façon suivante :

1. recherche le sujet dans le contexte d'exécution à l'aide de son nom. L'absence de cette clé signifie une erreur dans la définition de la procédure et conduit au déclenchement d'une exception.
2. s'il y a des arguments, les rechercher également dans le contexte courant à l'aide de leur nom.
3. envoi le message (à l'aide de la méthode `perform:` dans le cas de Smalltalk-80).
4. stocke le résultat dans le contexte d'exécution avec la clé fourni par la variable d'instance `resultSlot`.

Cette classe implante également la méthode `,` (virgule) qui sert à composer deux procédures en une séquence. Elle retourne donc une instance de la classe `SequenceProcedure`. Cette instance est composée de deux procédures : le receveurs du message et la procédure reçu en argument.

La classe `SequenceProcedure`

La classe `SequenceProcedure` est conçue suivant le schéma de conception *Composite.* Elle est sous-classe de la classe `Procedure` et comporte une variable d'instance `steps`. Celle-ci pointe sur une collection ordonnée qui contient la suite des étapes du calcul (au sens large du terme).

Elle implante la méthode `executeProcedure`: qui itère sur la collection des étapes et leur envoie successivement le même message *continueExecutionOf:* avec en argument le contexte d'exécution qui résulte de l'exécution de l'étape précédente.

Elle implante également la méthode `,` (virgule) qui procède à l'ajout dans le receveur de la procédure reçu en argument. La même fonction est remplie par la méthode `addStep:`. La méthode `removeStep:` permet de retirer une étape.

2.2.2 Exécution de procédures

La classe `ProcedureActivation`

Au niveau du noyau de micro-workflow, le module d'exécution des micro-procédés se résume à la classe `ProcedureActivation`. Celle-ci réalise l'exécution en collaboration très (voire trop[92]) étroite avec la classe `Procedure`.

[92] Cette conception met le Micro-workflow en défaut au niveau de l'extensibilité de son mécanisme d'exécution. En effet, la plupart des méthodes d'exécution sont implantées dans les classes dont le rôle est la définition des procédures. Notre proposition consiste à développer la hiérarchie des classes du niveau d'exécution de façon symétrique par rapport à celle du niveau définition. Cette approche permet de retracer complètement l'activation d'une procédure et de déléguer les traitements adéquats à chaque classe du niveau d'exécution (cf. la **Figure 43**). Le développement détaillé de ce sujet sort du cadre de ce mémoire.

Les deux variables d'instances principales de cette classe (de notre point de vu) sont `type` et `context`. La première point sur la procédure exécutée, et la seconde sur une table de hachage qui constitue le contexte d'exécution.

Cette classe offre principalement un protocole public pour accéder en lecture (la méthode `localState:`) et en écriture (la méthode `localStateAt:put:`) au contexte d'exécution.

2.2.2.1 Conclusion

L'implantation d'un framework suivant les spécifications de D. Manolescu ne pose pas de problème majeur. Ce qui a été moins évident, et qui restait jusqu'alors un problème ouvert, c'est la modélisation canonique de la co-évolution dynamique de procédures et structures. Notre expérience industrielle à travers le système CALIBRES a joué un rôle important dans aussi bien la mise en évidence de la nature des problèmes posés que dans la recherche d'une solution générique et réutilisable.

Les deux sections suivantes sont consacrées à la présentation de cette solution qui sera obtenue par deux couplages successifs.

2.3 FDARC: couplage du FDOM et du DYCFLOW

Les réalisations exposées ci-dessous nous permettent à présent de présenter et de valider expérimentalement une première solution (partielle) au problème de l'outillage de l'adaptation qui a l'avantage de s'appuyer sur des techniques standards. Cette solution est basée sur la résolution du problème de couplage décrit dans le chapitre précédent.

2.3.1 Modèle d'analyse

Le but du modèle du couplage du DOM et du Micro-workflow est de montrer comme définir dynamiquement des procédures qui comportent des accès en lecture et en écriture aux attributs dynamiques. Il faut aussi modéliser l'exécution de ces accès.

Un dernier problème est celui de l'ajout dynamique de méthodes "primitives".

Nous proposons pour l'heure la solution (provisoire) suivante :

1. *problème des accès en lecture* : quand le nom utilisé lors de la définition d'un appel de service correspond au nom d'un attribut dynamique, le moteur d'interprétation (ici la classe `BasicComponentCallDefinition`, cf. la Figure 44) le considère comme un accès en lecture à l'attribut concerné. Pour plus de convenance pour les experts, on peut fixer une convention de nommage, comme l'usage systématique du préfixe *Obtenir*. Par exemple, pour accéder en lecture à l'attribut `Compte-chèque associé`, l'expert ajoute à sa procédure une définition d'appel de service `Obtenir Compte-chèque associé` ;
2. *problème des accès en écriture* : quand le nom utilisé lors de la définition d'un appel de service correspond au nom d'un attribut dynamique suivi par le caractère ":" (suivant la convention de SMALLTALK-80), le moteur d'interprétation le considère comme un accès en écriture à l'attribut concerné. Ici aussi on ajout systématiquement un préfixe *Affecter*. Par exemple, pour accéder en écriture à l'attribut `Compte-chèque associé`, l'expert ajoute à sa procédure une définition d'appel de service `Affecter Compte-chèque associé:`;
3. *problème d'ajout dynamique de méthodes primitives* : notre couplage actuel n'offre pas de solution systématique à ce problème d'ajout dynamique de méthodes primitives. Il est toutefois envisageable de le solutionner ici avec des moyens *ad-hoc*. On peut par exemple créer une classe[93] dont le rôle est de contenir ce type de méthodes, développées par des programmeurs et chargées dynamiquement (ici la méta-classe `FlowIndependentComponentType class` dans la Figure 44)[94]. Une autre solution est de mettre en place des mécanismes d'appel des bibliothèques externes. A titre d'exemple, la fonction primitive `Cumuler les agios du jour ()` peut ainsi être ajoutée.

[93] Rappelons à nouveau que le type de compte-service est modélisé à l'aide d'une instance terminale.

[94] La possibilité de chargement dynamique de classes et méthodes offre donc ici une solution partielle, mais dans le contexte bien défini de l'outillage proposé ici où le système est conçu pour réceptionner ce chargement.

Ce manque de solution systématique satisfaisante constitue l'une des motivations de notre mise en œuvre de la spécialisation dynamique à l'aide des méta-clases (cf. les chapitres IV et V). L'usage récursif de procédures dynamiquement définies mise en œuvre par DART offre également une approche attirante. En effet, celle-ci a l'avantage de minimiser la nécessité d'intervention des programmeurs et d'augmenter la liberté d'action des experts. Cela réduit notamment le délai requis pour faire évoluer un logiciel. Nous reviendrons sur cette possibilité dans la section suivante, §7.

2.3.2 Modèle de conception

Comme permet de l'illustrer la Figure 44 ci-dessous, nous validons cette proposition en réalisant un couplage entre les deux frameworks FDOM et DYCFLOW.

Ce couplage nécessite une extension de ces deux implantations. Cette extension permet :

1. de valider, à l'aide des technologies standards, la faisabilité de l'outillage de la co-évolution dynamique de structures et de procédures ; et
2. de mettre en évidence ses limites (cf. alinéa n° 3 ci-dessus).

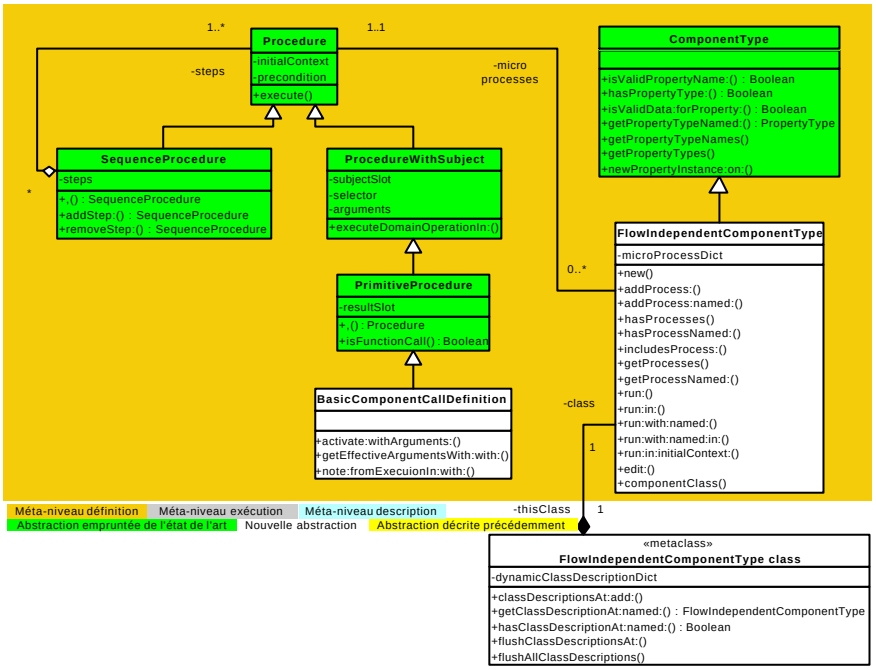


Figure 44 : Couplage du DOM et du Micro-workflow par des extensions de FDOM et DYCFLOW.

Aussi, une nouvelle classe est ajoutée à chacun de ces deux frameworks (cf. la Figure 44).

La classe `BasicComponentCallDefinition` est ajoutée au framework DYCFLOW et spécialise la classe `PrimitiveProcedure.`

La classe `FlowIndependentComponentType` est ajoutée au framework FDOM et spécialise la classe `ComponentType.`

La suite de cette section décrit avec plus de détails l'implantation de ces abstractions. Ensuite, nous montrons dans quelle mesure ces extensions permettent de réaliser ce qui n'était pas possible avant ce couplage (cf. le chapitre II, le paragraphe 4.2, page 102).

La classe `BasicComponentCallDefinition`

La conception de la classe `BasicComponentCallDefinition` s'appuie sur un mécanisme d'extension prévu par le micro-workflow. C'est la méthode abstraite `executeDomainOperationIn:` implantée initialement par la classe `ProcedureWithSubject`.

`BasicComponentCallDefinition` spécialise donc la classe `PrimitiveProcedure` et redéfinie la méthode `executeDomainOperationIn:` pour mettre en œuvre l'algorithme suivant :

1. recherche de l'objet qui va effectivement réaliser le service demandé.
2. calcule de la valeur effective des arguments.
3. activation du service.
4. stocker le résultat obtenu par l'exécution du service.

C'est lors de l'étape n° 2 que cette nouvelle implantation compare le nom de chaque argument avec le nom des descriptifs d'attributs du receveur. S'il y a correspondance, alors la valeur courante sera celle qui est affectée à cet attribut. A défaut la recherche continue dans le contexte courant d'exécution.

Ensuite, c'est lors de l'étape n° 3 que le système vérifie si le receveur sait répondre au message qui désigne le service demandé. Sinon, il considère qu'il s'agit d'une méthode primitive chargée dynamiquement au sein d'un objet qui sert de référentiel global des méthodes primitives. L'exécution du service est alors délégué à cet objet par la méthode `invoke:withArguments:`.

Enfin, lors de l'étape n° 4, il y a à nouveau une comparaison du nom du résultat (la valeur de la variable d'instance `resultSlot`) avec ceux des descriptifs d'attributs du receveur. Si à nouveau il y a une correspondance, le système considère alors que la valeur résultante devait être affectée à l'attribut concerné. Sinon cette valeur est mise dans le contexte courant d'exécution.

La classe `FlowIndependentComponentType`

La classe `FlowIndependentComponentType` modélise les compléments de classe qui gèrent de plus la dimension adaptation du comportement. En effet, celle-ci ajoute à sa super-classe `ComponentType` la gestion d'une table de hachage (la variable d'instance `microProcessDict`) qui stocke les procédures à l'aide de leur nom. Cet ajout lui permet d'offrir le protocole public suivant :

1. `addProcess:named:` ajoute une nouvelle procédure.
2. `hasProcessNamed:` vérifie si la procédure dont le nom est passé en argument est définie pour ce complement de classe.
3. `getProcessNamed:` retourne la procédure dont le nom est passé en argument.
4. `runProcess:in:initialContext:` exécute la procédure passée en premier argument dans le contexte du second argument (à l'aide de la méthode `continueExecutionOf:`). Avant le lancement de l'exécution les objets existants dans le contexte passé en troisième argument sont copiés dans le contexte d'exécution. Retourne le contexte résultant de l'exécution.

La classe `FlowIndependentComponentType` ajoute également une variable d'instance de classe `dynamicClassDescriptionDict`. Celle-ci sert d'un référentiel global pour stocker les compléments de classes. Elle contient une entrée par classe du système qui a besoin de stocker ses spécialisations dynamiques. Chaque entrée se réfère à nouveau à une table de hachage qui stocke l'ensemble des compléments de classes concernés.

Cette gestion se réalise à l'aide du protocole de classe suivant :

1. `classDescriptionsAt:add:` ajoute dans la table de hachage de la classe passée en première argument le complémente de classe passé en second argument.
2. `getClassDescriptionAt:named:` retrouve dans la table de hachage de la classe passée en première argument le complémente de classe dont le nom est passé en second argument.
3. `hasClassDescriptionAt:named:` vérifie dans la table de hachage de la classe passée en première argument s'il existe un complémente de classe dont le nom est passé en second argument.
4. `flushClassDescriptionsAt:` supprime les compléments de classe associés à la classe passée en argument.

Cette solution n'est pas tout à fait satisfaisante puisqu'elle ajoute parallèlement au langage qui dispose déjà d'un mécanisme de stockage des classes un autre pour le stockage des compléments de classe. La solution que nous mettons en œuvre dans les deux chapitres suivant résout également ce problème.

2.4 Validation expérimentale de notre solution au problème de couplage

Nous disposons à présent de l'outillage nécessaire pour la validation expérimentale de notre solution au problème de couplage. Il s'agit de montrer que DYCTALK permet de réaliser le contre exemple de la section 4.2, page 102 du chapitre II[95]. Celui-ci montrait en effet q'une simple juxtaposition de DOM et de micro-workflow ne permet pas la spécialisation dynamique et se heurte au problème de la co-évolution dynamique de procédures et structures.

Pour ce faire, nous proposons de modifier cet exemple de la façon suivante (cf. la Figure 46 ci-dessous) :

1. remplacer la classe `ComponentType` par la classe `FlowIndependentComponentType` ;
2. remplacer la classe `PrimitiveProcedure` par la classe `BasicComponentCallDefinition` dans les deux cas suivants :
   a. dans le cas de la primitive appelé `getInterestRate`. En effet, celle-ci décrit l'activation de la méthode `getInterestRate` dans le but d'importer dans le contexte courant d'activation la valeur courante du taux d'intérêt (`interestRate`). Or, cette méthode n'est pas défini dans notre complément de classe compte d'épargne. En remplaçant `getInterestRate` par le nom de du descriptif d'attribut qui détient cette valeur, c'est à dire `interestRate`, DYCTALK assimile cette définition à un accès en lecture à l'attribut dynamique du même nom.
   b. dans le cas de la primitive appelé `calcDailyInterest`. Cela correspond, en effet, à la seconde anomalie que nous avons décelée, c'est à dire manque d'implantation des méthodes primitives, ici `calcDailyInterest:with:`. Là encore notre mécanisme de gestion systématique des méthodes primitives dynamiquement chargées apporte une solution acceptable.

L'exécution du micro-procédé ainsi modifié retourne une instance de la classe `Account` dont l'attribut balance vaut 5250.0000037253d, pour un taux d'intérêt (defaultnterestRate) égal à 0.050000000745058d.

Nous validons ainsi notre solution de couplage entre DOM et Micro-workflow.

Ce qui suit fournit plus de détails sur cette mise en œuvre.

[95] Au niveau du code source de DYCTALK cette exemple est implanté par la méthode `exampleMFWLimitsInTheContextOfAOM`.

La Figure 45 ci-dessous montre un script écrit dans le langage SMALLTALK qui utilise notre framework pour créer, définir la structure et stocker un complément de classe de la classe `Account`. Cette adaptation de la classe `Account` est appelée `Savings Account`.

```
      | ownerIdPropertyTpe ratePropertyTpe dailyInterestPropertyTpe |
      ownerIdPropertyTpe := PropertyType newString: 'ownerId'.
      ratePropertyTpe := PropertyType newDouble: 'interestRate'.
      dailyInterestPropertyTpe := PropertyType newDouble: 'interest'.
      ^Account
            newClassDescriptionNamed: 'Savings Account'
            with: (Array with: ratePropertyTpe with: ownerIdPropertyTpe with:
dailyInterestPropertyTpe)
```

Figure 45 : La création, la définition de la structure et le stockage d'un complément de classe.

L'étape suivante consiste à ajouter un micro-procédé à ce complément de classe. La Figure 46 montre comment le complément de classe peut être rappelé et comment un nouveau micro-procédé peut lui être associé (à l'aide de la méthode `getClassDescriptionAt:named:`). Le micro-procédé est écrit comme toujours dans la syntaxe du Micro-workflow (cf. le chapitre II).

```
      |   dynamicSavingsAccount   calcDailyInterest   depositInterest   showBalance
thePrograme getInterestRate getBalance newBalance |
      "Définir un nouveau comportement."
      getInterestRate := BasicComponentCallDefinition
                        sends: #interestRate
                        to: #myAccount
                        result: #interestRate.
      getBalance := PrimitiveProcedure
                        sends: #getBalance
                        to: #myAccount
                        result: #balance.
      calcDailyInterest := BasicComponentCallDefinition
                        sends: #calcDailyInterest:with:
                        with: #(balance interestRate)
                        to: #myAccount
                        result: #interest.
      depositInterest := PrimitiveProcedure
                        sends: #deposit:
                        with: #interest
                        to: #myAccount.
      newBalance := PrimitiveProcedure
                        sends: #getBalance
                        to: #myAccount
                        result: #balance.
      showBalance := PrimitiveProcedure
                        sends: #show:
                        with: #balance
                        to: #myAccount.
      thePrograme     :=     getInterestRate,     getBalance,     calcDailyInterest,
depositInterest, newBalance, showBalance.
      "Rechercher la définition du descriptif des comptes d'épargne."
      dynamicSavingsAccount := FlowIndependentComponentType
                              getClassDescriptionAt: Account
                              named: #'Savings Account'.
      dynamicSavingsAccount
            addProcess: thePrograme
            named: #calcDailyInterest.
      ^dynamicSavingsAccount
```

Figure 46 : Ajout d'un micro-procédé à un descriptif de compte existant (compte d'épargne).

Le script de la Figure 47 ci-dessous montre une instanciation d'un complément de classe via l'exemple d'ouverture d'un compte d'épargne pour le client Reza Razavi, avec un montant de 5000 F et un taux d'intérêts de 0,04[96].

```
| dynamicSavingsAccount myDynamicSavingsAccount |

"Rechercher la définition du descriptif des comptes d'épargne."
dynamicSavingsAccount := FlowIndependentComponentType
                                getClassDescriptionAt: Account
                                named: #'Savings Account'.
"Créer une instance quelconque de ce descriptif."
myDynamicSavingsAccount := Account onType: dynamicSavingsAccount.
myDynamicSavingsAccount
      setValue: 'Reza Razavi' toPropertyNamed: 'ownerId';
      setProperty: 'balance' to: 5000;
      setValue: 0.04 toPropertyNamed: 'interestRate'.
"Exécuter le micro-procédé #calcDailyInterest."
^myDynamicSavingsAccount getType
      run: #calcDailyInterest
      with: myDynamicSavingsAccount
      named: #myAccount
```

Figure 47 : Exemple de création d'une instance de compte du type compte d'épargne.

2.5 DYCRA : couplage des systèmes DART et DARC

La dernière étape de la validation de la première partie de notre thèse sur l'outillage de l'adaptation consiste à mettre ensemble les deux solutions partielles DARC & DART afin de construire le système recherché DYCRA (cf. Figure 48 ci-dessous).

DART permet la composition dynamique de différentes formes de représentations de calcul. Mais, son inconvénient est que ces représentations agissent toutes sur des structures de données définies par des programmeurs. DART ne dispose en effet d'aucune notion d'évolution dynamique de structures ou d'ajout dynamique de nouveaux types d'objets.

Au contraire, DARC gère la définition dynamique de nouveaux types d'objets, leur structure et aussi leur comportement. Son inconvénient est que le modèle de programmation de procédures de DARC est un héritage de micro-workfow qui est conçu à l'usage des programmeurs. Or, nous recherchons une solution qui soit facile à apprendre par des experts.

Pour réaliser ce couplage, nous proposons de partir du système DART et de lui adjoindre les éléments de DARC de sorte que le système final assure la création des langages d'experts qui satisfont les caractéristiques énumérées dans l'introduction.

La mise en œuvre de ce couplage est possible grâce aux mécanismes d'extension que nous avons déjà prévu au sein du système DART. C'est notamment une conception suivant le schéma *Bridge* [GHJV95, ABW98] et aussi les notions de descriptif de service et de stratégie d'activation ainsi que la possibilité d'évolution dynamique du référentiel des descriptifs de service.

[96] Au niveau du code source de DYCTALK cette exemple est implanté par la méthode `exampleDyCS_05_C` de la méta-classe `Account class`.

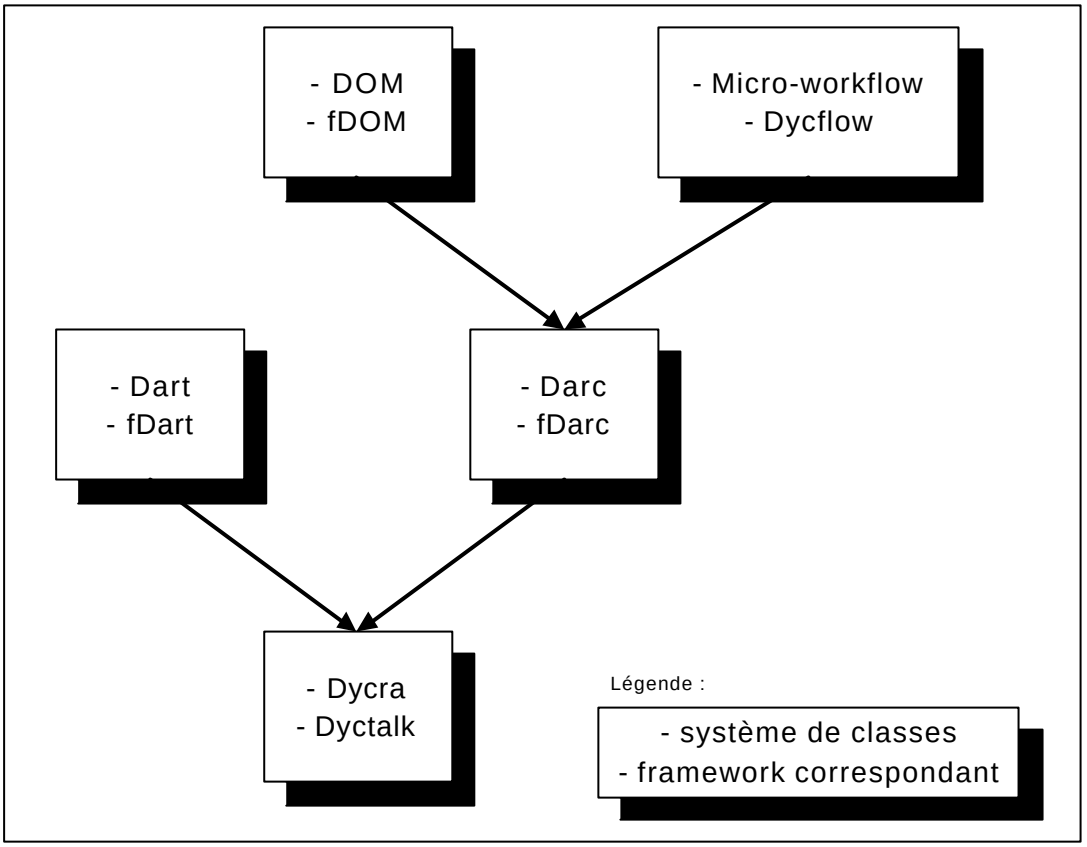


Figure 48 : Etapes successives de la validation de notre thèse.

DART modélise déjà les deux aspects apprentissage par des experts et le lien causal. Le procédé proposé ci-dessus va donc consister à intégrer à DART la "dimension workflow" et la spécialisation dynamique. L'outillage du travail collaboratif et le choix local du type d'adaptation seront étudiés lors des deux chapitres suivants.

Nous validons expérimentalement notre proposition à l'aide des deux frameworks FDOM et DYCFLOW, qui seront étendus par de nouvelles classes à cette occasion.

2.5.1 Intégrer la "dimension worflow" à DART

Comme permet de l'illustrer la Figure 49 ci-dessous, l'intégration à DART de la dimension workflow s'appuie sur la classe `ServiceEvaluation`. Celle-ci prévoit l'usage d'objets quelconques qui représentent un calcul (lien `action`). Cette conception suivant le schéma de conception *Bridge* permet de coupler à ce système une nouvelle implantation d'une telle représentation.

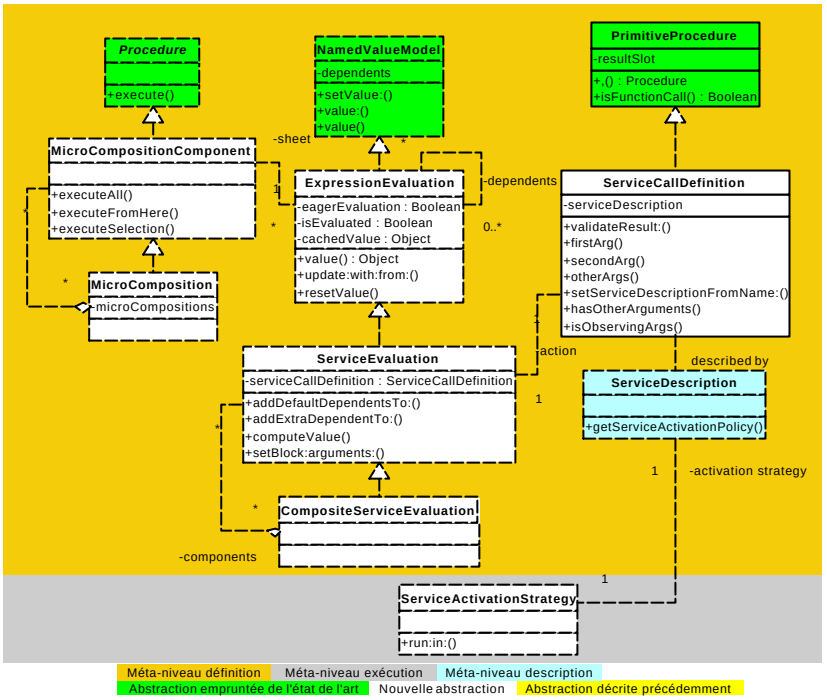


Figure 49 : Intégration à DART de la "dimension workflow".

Nous proposons d'utiliser cette possibilité et de coupler le Micro-workflow à DART comme une nouvelle représentation de calcul/procédures.

Il s'agit plus précisément de créer, dans un premier temps, une classe `ServiceCallDefinition` qui est une spécialisation de la classe `BasicServiceCallDefinition`. Celle-ci est à son tour une spécialisation de la classe `PrimitiveProcedure` issue du micro-workflow.

Ensuite, des instances de cette classe peuvent être utilisées comme des expressions de calculs, via le lien `action`, est donc intégrées dans la définition des procédures (du type ici micro-composition).

C'est là que le couplage de DART avec le micro-workflow a lieu et c'est ainsi que DYCRA outille la "dimension workflow".

La Figure 51 ci-dessous illustre ces spécialisations, que nous détaillons ici :

La classe BasicServiceCallDefinition

La classe `BasicServiceCallDefinition` ajoute à sa super-classe `PrimitiveProcedure` une variable d'instance `serviceDescription`. Celle-ci référence le descriptif de service instancié lors de la définition de l'appel de service concerné.

Cet ajout permet alors de redéfinir la méthode `executeDomainOperationIn:` dans le but de déléguer l'exécution du receveur à la stratégie d'activation dont la nature dépend du descriptif de service

(message `getServiceActivationPolicy`). Cette conception suit le schéma *Mediator* [GHJV95, ABW98].

Plus concrètement la méthode `executeDomainOperationIn:`est définie telle qu'elle est illsutrée par la Figure 50 ci-dessous.

```
BasicServiceCallDefinition >> executeDomainOperationIn: aProcedureActivation
        | result |
        result := self getServiceDescription getServiceActivationPolicy
                                    run: self
                                    in: aProcedureActivation.
        ^result
```

Figure 50 : Déléguer le choix de la stratégie d'activation au descriptif de service.

La classe ServiceCallDefinition

La classe `ServiceCallDefinition` hérite de `BasicServiceCallDefinition`. Elle ajoute à sa super-classe simplement la vérification du type du résultat à l'aide de la méthode `validateResult:`.

Le développement de cet aspect ne fait pas partie du cœur de notre thèse et n'est pas traité ici.

La classe ObserverServiceCallDefinition

La classe `ObserverServiceCallDefinition` ajoute à sa super-classe `ServiceCallDefinition` un contrôle sur la nature de ses arguments.

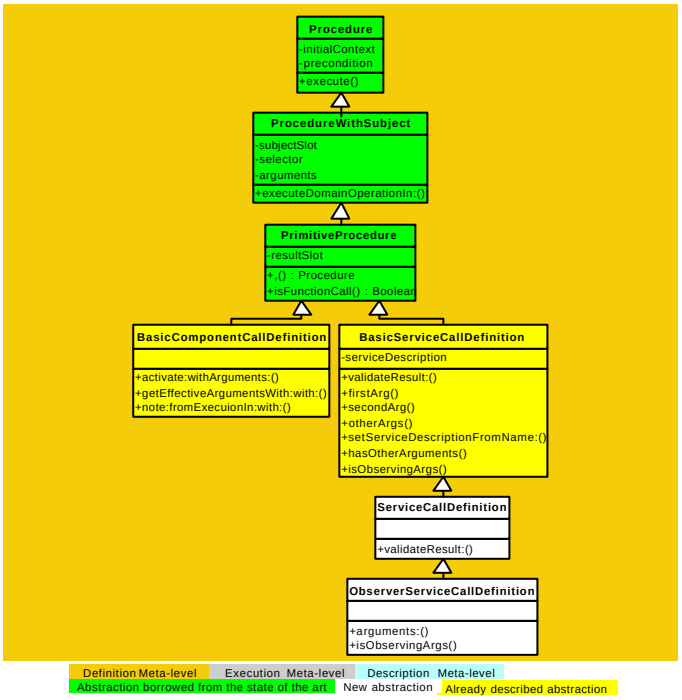


Figure 51 : Intégrer la "dimension workflow" à FDART.

En effet, les instances de `ObserverServiceCallDefinition` sont utilisées lors de la création de micro-compositions (à l'aide de DART) munies de la "dimension workflow". Dans la mesure où FDART associe ces instances à des instances de `ExpressionEvaluation` ou ses sous-classes (cf. chapitre I), alors les arguments reçus ici ne sont plus des chaînes de caractères mais des instances de `ExpressionEvaluation`.

C'est cette technique qui permet la gestion des dépendances entre une définition d'appel de service et ses arguments (cf. également le chapitre I).

2.5.2 Intégrer la spécialisation dynamique à DART

Le problème essentiel relatif à la spécialisation dynamique est qu'à l'heure actuelle la définition de micro-procédés se fait par l'écriture des scripts SMALLTALK. Or, nous disposons d'un mécanisme dédié aux experts qui est celui de DART.

L'idée principale de la technique d'intégration à DART de la spécialisation dynamique consiste à, d'une part, sauvegarder au sein des compléments de classe non pas les micro-procédés issu de DARC, mais des micro-compositions issus de DART, qui sont accessibles aux experts.

D'autre part, il s'agit d'utiliser la possibilité de faire évoluer dynamiquement le référentiel de descriptifs de services de DART, afin de transformer le problème de co-évolution dynamique de structures et de procédures en un problème de génération automatique de descriptifs de service d'accès en lecture et en écriture aux attributs dynamiques.

La mise en œuvre de cette idée s'appuie sur les possibilités d'extension prévues au sein du système DART. Ces extensions se déclinent sous forme d'ajout de deux méthodes (`asGetter` et `asSetter`) dans la classe `PropertyType` qui génère pour chaque descriptif d'attribut deux descriptifs de services, l'un pour l'accès en lecture à cet attribut et l'autre pour l'accès en écriture.

Par exemple, suite à l'ajout au sein d'un nouveau type de compte bancaire de l'attribut `taux d'intérêt` (`interestRate`), l'expert peut exprimer lors de la définition de nouvelles procédures des accès en lecture et en écriture à la valeur courante de cet attribut en instanciant les descriptifs de service auto-générés (`Affecter taux d'intérêt` et `Obtenir taux d'intérêt`).

En ce qui concerne les méthodes primitives, nous proposons également de les "habiller" à l'aide des descriptifs de service. Cette technique fonctionne également dans le cas des calculs décrits à l'aide des micro-procédés, micro-composition ou encore des structures conçues suivant les schémas de conception *Interpreter* et *Composite* [GHJV95, BW98][97].

La fonction "d'habillage" qui est réalisée par un expert consiste à choisir un nom pour le descriptif de service et, le cas échéant, à désigner et nommer ses paramètres en entrées. Le développement détaillé de ces aspects sort du cadre de notre démonstration.

Ces ajouts se concrétisent sous forme de classes ajoutées dans le framework DART que nous allons détailler davantage ici.

[97] Ce sont, en effet, des arbres de syntaxe abstraite créés directement (sans l'usage d'un compilateur) et qui sont associés à un interprète.

2.5.2.1 Extension des stratégies d'activation

Comme permet de l'illustrer la Figure 52 ci-dessous, la hiérarchie des stratégies d'activation est étendue par trois classes que nous allons décrire ci-dessous.

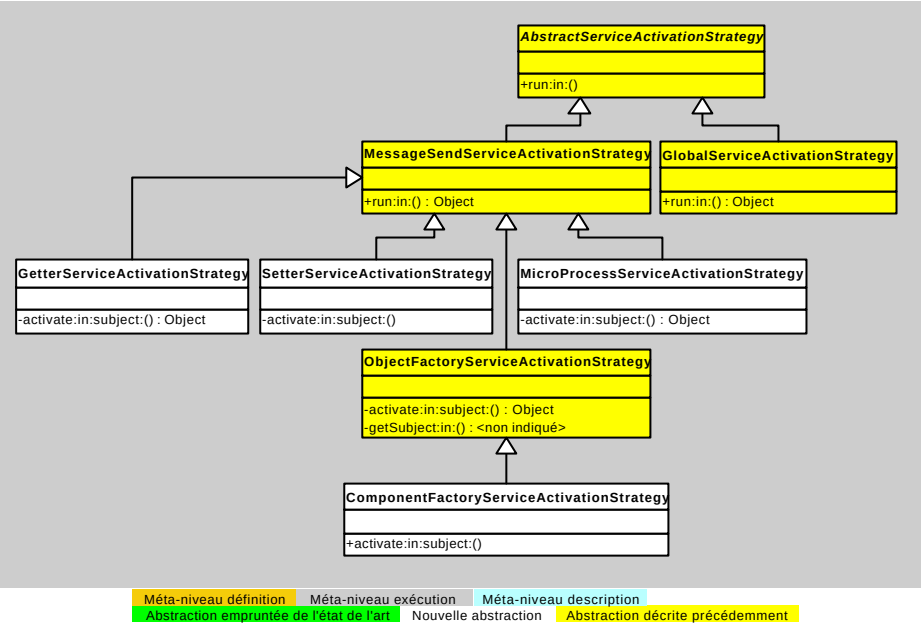


Figure 52 : Extension des stratégies d'activation de FDART.

La classe `MicroProcessServiceActivationStrategy`

La classe `MicroProcessServiceActivationStrategy` modélise une stratégie d'activation du type envoi de message. Il s'agit de pouvoir réutiliser des procédures définies dynamiquement dans la définition d'autres procédures en les considérant comme des primitives.

Pour ce faire, nous proposons d'encapsuler ces procédures dans des descriptifs de service. Cela nécessite alors la fourniture d'un nom pour ce nouveau service et également le nombre de ses arguments. Ce descriptif est alors ajouté au référentiel des descriptifs. Il peut donc être instancié par des experts lors de la composition de procédures.

L'encapsulation d'une procédure nécessite un type particulier de descriptif de service (cf. ci-dessous 3.4.2, page 72). La stratégie d'activation associée à ce type de descriptif est celle qui est présentée ici (la classe `MicroProcessServiceActivationStrategy`). On se retrouve donc à ce niveau lors de l'activation d'un tel service.

L'algorithme qui est alors appliqué (par la méthode `activate:in:subject:`) pour exécuter le service est le suivant :

1. le premier argument est le sujet ou le receveur du message. C'est lui qui sera stocké dans le contexte courant d'exécution sous la clé par défaut `#me`. Ce mot clé est équivalent du mot clé `self` dans le langage SMALLTALK ou `this` dans le langage JAVA.
2. s'il n'y a pas d'autres arguments, alors la méthode `run:with:named:in:` de la classe `FlowIndependentComponentType` est utilisée pour activer le service.
3. Sinon, la méthode `run:in:initialContext:` est activée en lui passant un dictionnaire qui comporte les arguments d'appel.
4. S'il s'agit d'un service du type fonctionnel, stocker alors le résultat dans le contexte courant d'exécution.

Il est important de noter que cet algorithme s'appuie sur l'hypothèse que le sujet est l'instance d'un complément de classe et qu'on peut donc lui déléguer l'exécution de la procédure par l'envoi du message `run:with:named:in:`.

Cette hypothèse est fondée sur le principe que l'expression de l'activation d'une procédure n'a de sens que s'il existe déjà une instance du complément de classe qui comporte la définition de cette procédure. C'est cette instance qui va jouer le rôle du sujet.

La classe `GetterServiceActivationStrategy (et Setter)`

Un autre type d'opération dont les experts ont besoin dans l'écriture de procédures sont les accès en lecture et en écriture aux attributs qu'ils ont défini lors de la définition des compléments de classe. Par exemple, si l'expert ajout l'attribut *nom du client* à la classe compte, il va avoir besoin d'exprimer un accès en lecture à celui-ci pour, par exemple, afficher le nom du client à l'écran.

Le micro-workflow n'a pas prévu une telle opération car il requiert l'existence d'une méthode pour formuler toute sorte d'opérations. Or, ici l'attribut étant défini à l'exécution, il n'y a pas de méthode d'accès à celui-ci. Il n'est donc pas possible d'exprimer explicitement un accès en lecture ou en écriture aux attributs.

Nous proposons une solution basée sur la génération automatique de descriptifs de service. Il s'agit de descriptifs d'un type particulier qui seront décrits dans la sous-section suivante. Pour lors nous allons exposer les deux classes qui mettent en œuvre les stratégies d'activation d'instances de tels descriptifs.

Il s'agit d'une part de la classe `GetterServiceActivationStrategy` qui réimplante la méthode `activate:in:subject:` afin de contrôler que le l'attribut relatif au descriptif auto-généré est bien un attribut du receveur du message. Ensuite, il utilise la méthode `getPropertyValue:` pour lire et retourner la valeur courante de cet attribut.

D'autre part, la classe `SetterServiceActivationStrategy` réimplante la même méthode `activate:in:subject:`afin d'opérer cette fois l'opération d'affectation à un attribut dynamique. La valeur à affecter est supposé être fournie en second argument.

La classe `ComponentFactoryServiceActivationStrategy`

La dernière extension de ce composant de FDART correspond à la classe `ComponentFactoryServiceActivationStrategy`. Celle-ci assure l'exécution des expressions relatives à la création d'instances des compléments de classes.

Pour ce faire, cette classe réimplante également la méthode `activate:in:subject:` afin de rechercher le complément de classe concerné et procéder à son instanciation. C'est alors que le lien `type` entre la nouvelle instance et son complément de classe sera renseigné.

Nous avons, par défaut, prévu une méthode d'instanciation et d'initialisation de compléments de classe. Il s'agit de la méthode `fetchInstanceOfRefinedClass:`. Celle-ci est envoyée à la classe adaptée avec le complément de classe concerné en argument.

Par exemple, pour créer une nouvelle instance du complément de classe `SavingsAccount`, le message `fetchInstanceOfRefinedClass:` est envoyé à la classe `Account` avec comme argument ce complément de classe. Ce qui est important à rappeler ici est que 1) c'est la classe adaptée, ici `Account`, qui sera effectivement instanciée ; et 2) c'est le complément de classe `SavingsAccount` qui dispose des descriptifs d'attributs et de procédures ajoutés par l'expert.

2.5.2.2 Extension des descriptifs de service

Comme permet de l'illustrer la Figure 53 ci-dessous, la hiérarchie des descriptifs de service est également étendue par trois classes.

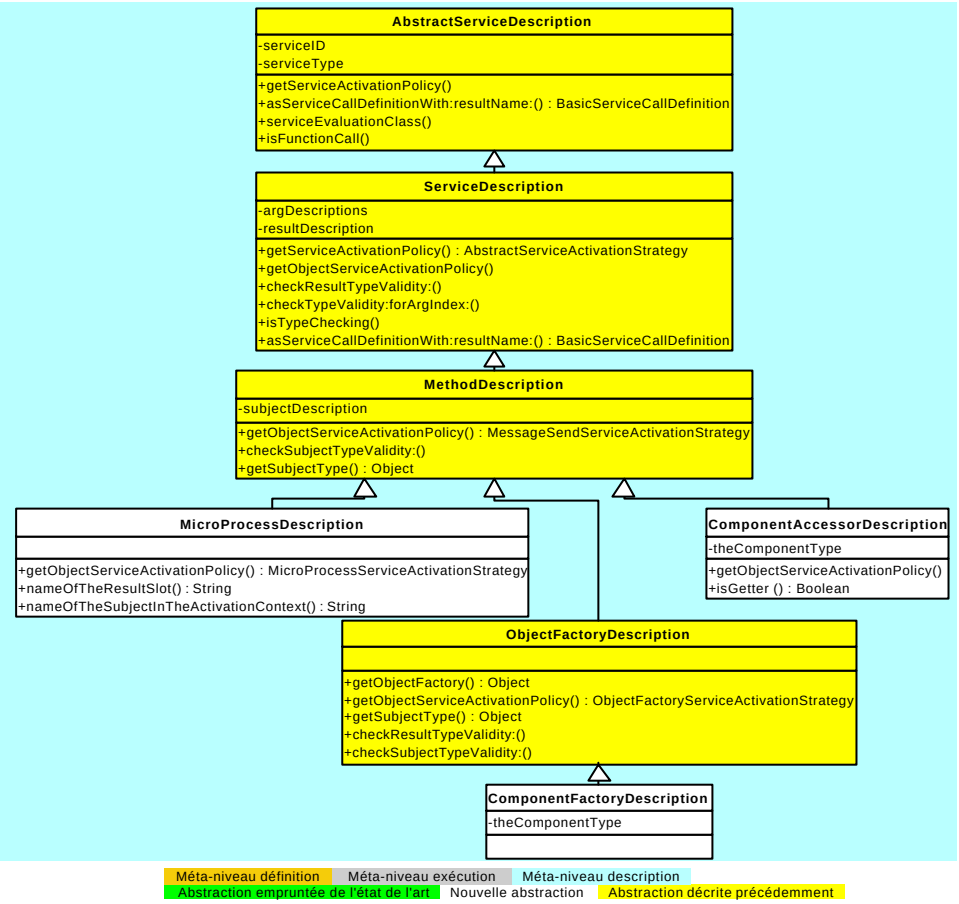


Figure 53 : Extensions des descriptifs de service de FDART.

Comme nous l'avons précisé lors de la description du modèle initial des descriptifs de service dans le paragraphe 3.4.2, page 72, le rôle essentiel de ces trois classes `MicroProcessDescription`, `ComponentAccessorDescription` et `ComponentFactoryDescription` est de redéfinir la méthode `getObjectServiceActivationPolicy` afin de retourner la stratégie d'activation adéquate. Le Tableau 7, page 72 fournit la correspondance entre les descriptifs de classe et leur stratégie d'activation.

3 Exemple : adaptation de comptes bancaires

A présent nous disposons, théoriquement, de tout le matériel nécessaire à la mise en œuvre de notre exemple illustratif d'adaptation de comptes, présenté dans le paragraphe 2, page 31 de l'introduction. L'édition des adaptations et le choix local du type d'adaptation ne rentrent pas encore dans ce cadre.

Toutefois, sur le plan pratique, comme le décrivent Dirk Riehle, Michel Tilman et Ralph Johnson dans leur publication DOM [RTJ00], la conception que nous venons d'outiller conduit très vite à des situations dont l'explication s'avère très difficile. Il y a constamment une confusion entre classes et compléments de classes, les instances terminales et les instances des compléments de classes, etc.

Aussi, nous allons encore retarder notre illustration de la mécanique complète pour la mettre enfin en œuvre à la fin du chapitre V, le paragraphe 3, page 158, lorsque nous disposerons enfin d'une représentation acceptable des adaptations.

Afin de permettre de comparer les deux démarches, nous tenons simplement à exposer ici la mise en œuvre, sur la base de l'exemple illustratif de l'introduction, de l'élément essentiel qui changement entre DYCTALK et MIDCTALK (cf. le prochain chapitre), c'est à dire la technique utilisée pour créer de nouveaux types d'objets.

Avant cet exposé nous devons décrire la création du langage d'experts l'objet de l'adaptation.

3.1 Question de méthodologie de création de langages d'experts

La première phase de travail consiste, de toute évidence, à créer le langage d'experts. La mise en œuvre méthodique de cette réalisation nécessite une étude systématique du sujet, que nous évoquons, par ailleurs, dans les perspectives, paragraphe 2.3, page 217.

Pour l'heure notre démarche s'appuie sur des expériences personnelles, mais aussi des études sur les évolutions de frameworks [RJ97, RJ98] qui se recoupent parfaitement avec nos propres expériences.

Nous estimons donc qu'en règle générale le modèle objet d'un système quelconque (pas nécessairement un langage d'experts) prend forme au fil des années. Cette évolution se concrétise grâce à un effort de capitalisation par l'application des techniques comme le *refactorings*.

Aussi, nous ne rentrons pas ici dans ce débat qui sort du cadre de notre étude et supposons simplement qu'il existe un langage d'experts, basé ici sur une spécialisation DYCTALK du langage à objets SMALLTALK-80, et qui comporte les trois classes abstraite `ObjetBancaire`, `ObjetCompte`, et `CompteBancaire`, illustrée par la Figure 5,page 40.

De plus, et cela en raison de la conception suivant DOM outillée par DYCTALK, pour que la classe `CompteBancaire` soit adaptable, il faut :

1. l'existence d'une classe `CompteBancaireType` dont les instances servent à représenter les nouveaux types de comptes qui spécialisent la classe `CompteBancaire`.
2. que la classe `CompteBancaire` hérite de la classe `Component`.

Nous supposons également ici que ces deux conditions sont remplies.

3.2 Ajouter de nouvelles adaptations (suivant DOM)

La première étape de l'adaptation d'un langage d'experts consiste à ajouter de nouveaux types d'objets. Cette fonction est ici assurée par le composant FDOM (cf. le paragraphe 2.1, page 110 du même chapitre) du framework DYCTALK.

Par rapport à notre exemple en cours, il s'agit de créer les trois types de compte : *Compte-Service Equilibre*, *Confort* et *Privilège*. Le script de la Figure 62 ci-dessous montre le cas de la création du type *Compte-Service Equilibre*.

```
CompteBancaireType
                    named: 'Compte-Service Equilibre'
```

Figure 54 : Script de création de nouveaux types d'objet suivant DARC-I.

Le script de la Figure 54 ci-dessus illustre le fonctionnement interne de notre framework lors d'ajout d'un nouveau type d'objet, ici Compte-Service Equilibre. Celui-ci décrit l'envoi du message `named:` à la classe `CompteBancaireType`. Ce message crée une nouvelle instance (*terminale*) du type de compte bancaire, dont le nom est également passé en argument (*Compte-Service Equilibre*).

Les deux autres types de compte sont également créés de la même façon.

Il convient d'ajouter ici que cette modélisation de l'ajout de nouveaux types d'objets ne permet pas d'obtenir le résultat que nous avions annoncé dans l'introduction, §2.4.1, page 41. En effet, les nouveaux types de compte créés ainsi ne seront pas sous-classe, au sens des langages à objet [Per92] de la classe `CompteBancaire`.

Nous reprendrons et poursuivrons cet exposé à la fin du chapitre suivant, le paragraphe 3, page 158, sur la base de l'outillage plus performant offert par le framework MIDYCTALK. A cette occasion nous montrons comment il est possible de remédier à ce problème et atteindre les objectifs annoncés.

4 Conclusion : apports du framework DYCTALK

4.1 Propriétés assurées par notre premier outillage de l'adaptation

Le système de classes DYCRA et le framework DYCTALK ainsi créés outillent la création de langages d'experts munis des propriétés suivantes :

4.1.1 Spécialisation dynamique

La spécialisation dynamique est outillée par le système DYCRA de façon assez subtile.

En effet, ce dernier "hérite" la solution au problème d'ajout dynamique de nouveaux types d'objets de DARC. Il "hérite", par ailleurs, la solution au problème d'ajout dynamique de procédures de DART. Le choix de ce dernier se justifie par le fait que c'est le modèle de composition dynamique de procédures de DART qui avait été étudié à l'usage des experts[98]. Par conséquence, la fonction du Micro-workflow au sein de l'outillage final DYCRA/DYCTALK est surtout d'assurer la dimension workflow.

Cette fonctionnalité est obtenue grâce à l'usage du schéma *Bridge* [GHJV95, ABW98] dans la conception de DART. Cette technique a permis d'y introduire les mécanismes de worklow comme un cas d'implémentation concrète d'une représentation quelconque de calcul, qui est supportée par cette conception.

[98] Il est par ailleurs, possible de montrer que le modèle de composition de Micro-workflow est un cas particulier de celui de DART.

Il est important de noter que nous avons également emprunté du Micro-workflow sa conception suivant le schéma *Type Object* [JW97] de la définition et de l'exécution de procédures. C'est ce même schéma que nous avons, en effet, appliqué à notre conception du système DART afin d'outiller le lien causal entre la définition d'une procédure et ses activations. Comme le précisent Manolescu et Johnson, cette conception établit une relation du type classe/instance entre ces deux concepts[99] [MJ99c].

Par ailleurs, au contraire de ce que l'on puisse attendre *a priori*, DYCRA remplace la solution obtenue au problème de la co-évolution dynamique de structures et de procédures mise en œuvre par DARC, par une solution plus élégante basée sur l'extension du système DART.

En effet, en utilisant le mécanisme d'évolution dynamique du référentiel de descriptifs de services de DART, DYCRA procède à la génération automatique des descriptifs d'accès en lecture et en écriture aux attributs dynamiques.

La même technique est également utilisée pour l'ajout dynamique de primitives, méthodes et procédures. Ce choix permet la description de ces accès sous forme d'envois de message lors de la définition de procédures, et cela de façon accessible aux experts.

4.1.2 Apprentissage par les experts

Le système ainsi produit est facile à apprendre et à utiliser par les experts. En effet, ceux-ci n'ont pas d'autres tâches à réaliser que l'ajout de nouveaux types d'objets, leurs structures et procédures.

Les deux premières activités ne posent pas de problèmes particuliers et peuvent trouver des solutions satisfaisantes par le développement des interfaces graphiques appropriées. La troisième, c'est à dire l'ajout des procédures, constitue la question centrale pour laquelle DYCRA "hérite" une solution appropriée de son composant DART.

4.1.3 La dimension *workflow*

La dimension workflow est également garantie en intégrant les mécanismes de micro-workflow dans DART, comme une forme particulière de définition et d'exécution de calculs.

4.1.4 Lien causal (Causal connection)

Notre traitement du problème d'outillage du "lien causal" se concrétise ici par la considération systématique dans nos modèles de *l'instanciation des définitions*, par l'application du schéma de conception *Type Object* [JW97]. Aussi, à chaque définition peut être associée l'ensemble de ses exécutions ou instanciations.

Aussi, on peut ici conclure que le couplage DARC & DART donne lieu à une première version du système de classes recherché, que nous appelons DYCTALK et le framework associé DYCTALK. Il est important de noter ici que les extensions réalisées lors de ce couplage au sein des systèmes DARC & DART montrent la réutilisabilité de ces systèmes [JF88].

Nous validons ainsi la première partie de notre thèse.

Il nous reste, toutefois, d'apporter une réponse au problème de l'outillage de l'accessibilité des adaptations aux programmeurs et également celui du choix local du type d'adaptation. Ces deux questions seront respectivement l'objet des développements des deux chapitres suivants.

---

[99] Two characteristics form the basis for the parallel between an object model and this process model.
First, in an object model a class creates instances. In the framework's process model, `Procedure` creates `ProcedureActivation` instances. Second, an object model splits state and behavior between the instance side and the class side. Likewise, the workflow framework splits procedure execution between Procedure and `ProcedureActivation`. From an analysis viewpoint, procedure instances govern the execution of procedure activations. The former reside on the knowledge level and the latter on the operational level [MO98, Fow97]. From a design perspective, they implement the *Type Object* [MRB97] pattern. [MJ99c]

4.2 Résultat complémentaire: MOP de DYCRA

Il convient ici de synthétiser les résultats obtenus jusqu'au présent à travers la présentation du protocole pour la création de compléments de classe. Nous proposons d'appeler ce protocole le MOP de DYCRA dans la mesure où il s'agit d'un protocole d'accès aux objets de méta-niveaux.

4.2.1 Création de compléments de classes

Le processus d'adaptation d'une classe commence par l'ajout dans le système d'une entité nommée qui représente le concept métier. Par exemple, Savings Account. Cela requiert le protocole de méta-classe suivant :

1. `named:`
2. `on:named:` le premier argument (optionnel) est la collection des descriptifs d'attributs.

4.2.2 Création d'instances

Un complément de classe ne peut pas être instancié à proprement parler. C'est alors le message classique `new` qui est utilisé et qui est envoyé à la classe qui a été adaptée.

4.2.3 Gestion de descriptifs d'attributs

Le processus d'adaptation d'une classe comprend la définition de la structure d'objets métier par l'ajout de descriptifs d'attributs :

1. `addPropertyType:`
2. `getPropertyTypeNamed:`
3. `getPropertyTypeNames`
4. `getPropertyTypes`

5. `isValidPropertyName:`
6. `isValidData:forProperty:`
7. `isValidData:forPropertyNamed:`
8. `hasPropertyType:`
9. `hasPropertyTypeNamed:`

4.2.4 Gestion de micro-procédés

L'adaptation dynamique de comportement est assurée par le protocole suivant :

1. `addProcess:`
2. `addProcess:named:`

3. `hasProcesses`
4. `hasProcessNamed:`

5. `getProcesses`
6. `getProcessNamed:`

7. `run:`
8. `run:in:`
9. `run:with:named:`
10. `run:in:initialContext:`

4.2.5 Accès au niveau "méta"

La méthode `getType` permet à un objet d'accéder à son complément de classe.

4.2.6 Gestion des attributs

Le protocole suivant assure la gestion des attributs au niveau des instances :

1. `addProperty:`
2. `setValue:toPropertyNamed:`
3. `setProperty:to:`
4. `getPropertyValue:`
5. `getPropertyValueString:`
6. `hasValueForProperty:`
7. `edit`
8. `inspect`
9. `printStringProperty:`

La fonction de chacune de ces méthodes est décrite dans la section correspondante.

Chapitre IV : Deuxième outillage de l'adaptation

Usage de méta-classes standard (MIDYCTALK)

Chapitre IV : Deuxième outillage de l'adaptation

- Usage de méta-classes standard (MIDYCTALK)

1 Introduction

Nous disposons à présent d'un système de classes DYCRA et d'un framework DYCTALK qui fournissent un premier outillage dédié à la création de langages d'experts. Ce dernier satisfait la plupart des propriétés énumérées dans la section 1.2 de l'introduction : la spécialisation dynamique, la facilité d'apprentissage par des experts, la dimension *workflow* ainsi que le lien causal.

Cette solution n'assure, toutefois, pas le travail collaboratif entre les experts et les programmeurs. Les compléments de classes créés à l'aide de cette technique ne sont, en effet, pas éditables par des programmeurs à travers leurs outils habituels. Cette propriété devait leur procurer la possibilité de mener aussi bien des interventions classiques comme l'ajout de variables d'instances et de méthodes d'instances, que des activités plus spécifiques comme l'ajout des micro-procédés ou le refactoring des adaptations.

L'objet de ce premier volet de la seconde partie de nos travaux consiste à apporter une réponse satisfaisante à ce problème. Le second volet correspond à l'outillage du choix local du type d'adaptation qui sera exposé lors du chapitre suivant (chapitre V).

Nous exposons ici en détail le problème de l'édition des adaptations par les programmeurs. Nous décrivons notre analyse de ce problème, suivi par notre modèle de solution (DARC-II). Nous validons ensuite expérimentalement ce modèle à l'aide d'une implantation en langage SMALLTALK-80.

1.1 Problème du statut des adaptations

Avant d'exposer notre solution, il convient ici de mieux détailler les problèmes posés qui peuvent être classifiés en trois catégories.

1.1.1 Problèmes liés à DOM

L'usage du modèle DARC présenté dans le chapitre précédent conduit en pratique à une situation qui présente plusieurs inconvénients du point de vue des programmeurs. Ceux-ci sont répertoriés par les auteurs de ce schéma de conception de la façon suivante [RTJ00, page 6] :

1. Cette conception est complexe[100] : une classe, en tant qu'une entité logique est à présent conçue sous forme d'instances d'une multitudes de classes. Une classe est représentée par une instance de `ComponentType` et plusieurs instances de `PropertyType`. Cette complexité diminue l'accessibilité des modèles objets adaptatifs aux programmeurs qui sont habitués aux modèles objets conçus sous forme de hiérarchies de classes.
2. Cette conception engendre plus de complexité à l'exécution [101]: l'état d'un système conçu suivant ce modèle est plus complexe qu'un logiciel à objet classique car un objet métier est représenté ici à travers une multitude d'objets, instances des classes `ComponentType, PropertyType, Component, Property` et de leurs relations.
3. Cette conception conduit à la nécessité de nouveaux outils de développement[102] : les programmeurs ne peuvent pas se servir de leurs outils habituels pour visualiser et éditer les adaptations.

1.1.2 Problèmes engendrés par le couplage du DOM & du Micro-workflow

Le système Micro-workflow de Dragos Manolescu est conçu dans le but de permettre aux programmeurs objets de définir explicitement les collaborations entre les objets dans la syntaxe du langage à objet d'implantation. Un micro-procédé selon Micro-workflow est donc une chaîne de caractères acceptable par le compilateur du langage, e.g. SMALLTALK-80, et qui est stockée comme une méthode (cf. à titre d'exemple la Figure 46, page 128).

La particularité de cette méthode est que son invocation produit un objet qui décrit la procédure en question sous forme d'un ensemble d'instances d'objets du modèle de Micro-workflow (la classe `Procedure` et ses sous-classes).

Ce sont donc les classes qui sont chargées par le Micro-workflow des fonctions d'édition et de stockage des micro-procédés. C'est pourquoi le système DYCTALK ne permet actuellement pas de mettre en œuvre le procédé ci-dessus. Celui-ci utilise, en effet, des instances terminales pour représenter des adaptations. Une adaptation n'est donc pas de la même nature qu'une classe (du point de vue du langage d'implantation) et ne peut pas assurer les charges qui lui sont confiées par le Micro-workflow.

---

[100] *Increased design complexity*. One logical class is now represented as a set of instances from a larger set of classes. A class is represented by one instance of `ComponentType` and several instances of `PropertyType`. For a programmer, this design is more complex and harder to understand than a traditional class inheritance hierarchy.

[101] *Increased runtime complexity*. One logical object now consists of several objects, together with their relationships. An instance of a domain class is now represented by instances of `ComponentType`, `PropertyType`, `Component`, `Property` and values/value objects. The runtime state of a system becomes harder to understand than the state of a traditional system.

[102] *New development tools*. Programmers can no longer rely on their familiar development tools, such as browsers to edit and view types of components. Other traditional tools break down because they are not effective anymore. Debuggers and inspectors, for instance, still work, but they are much harder to use: type objects appear as any other field in an inspector, whereas we should be able to view components as instances of their component type. You need to provide new tools that replace or enhance the existing tools. This need has been captured by many, for example as the Visual Builder pattern of Roberts and Johnson [Roberts+1998].

Notre solution actuelle ne conserve donc pas toutes les fonctionnalités du Micro-workflow. Or, les travaux de Dragos Manolescu apporte une solution remarquable au problème de création de logiciels objets flow-independent et des systèmes de gestion de workflow intégrés aux logiciels objets. L'un de nos objectifs premiers est donc ici de rétablir la fonctionnalité prévue par D. Manolescu.

Ainsi faisant, nous améliorons également le Micro-workflow. En effet, il sera alors possible de *conjuguer* l'intervention des programmeurs *et* les experts dans le cadre du développement des workflow adaptatifs [MJ99b]. Nous reviendrons sur ce point important dans la sous-section 3.11, page 171.

De plus, selon le Micro-workflow toute primitive utilisée lors de la définition des micro-procédés correspond à une méthode. Cette fonctionnalités n'est non plus pas disponible dans le système DYCTALK. Celle-ci pose, en effet, à nouveau le problème d'édition d'instances terminales (adaptations).

Ce second problème sera aussi résolu par les travaux présentés dans ce chapitre.

1.1.3 D'autres problèmes

La solution actuelle présente d'autres inconvénients :

1. Cette implantation rend complexe la mise en œuvre de l'héritage entre les adaptations (cf. le cas du système OBJECTIVA [AJ98], que nous avons également évoqué brièvement dans l'introduction, §1.1.1, page 15).
2. Cette conception requiert la mise en place de mécanismes ad-hoc pour le stockage des adaptations (cf. le chapitre III).
3. Cette conception nécessite que toute classe adaptable héritent de la classe `Component`. Une telle classe ne peut alors pas hériter de sa super-classe "naturelle".

Cette liste n'est sûrement pas exhaustive, mais elle est suffisante pour permettre de constater que l'interprétation d'instances terminales comme des classes engendre des problèmes qui méritent la recherche d'une solution adéquate.

Les facilités réflexives [FJ89, KdRB91, Riv96b] de certains langages à classes fournissent un moyen efficace pour résoudre ces problèmes. L'idée consiste à faire de sorte que la classe d'un compléments de classe soit une méta-classe, du point de vue du langage à objets d'implantation, et non pas une classe.

1.2 Notre analyse du problème

Le modèle actuel conduit à la situation complexe suivante, aussi bien du point de vue des évolutions dynamiques du modèle objet, que de ses exécutions. Nous décrivons cette situation à l'aide de l'exemple de `Comptes-Service` (cf. la section 2, page 31 de l'introduction).

1.2.1 Ajout dynamique de nouveaux types d'objets

Supposons que le langage d'expert comporte une classe adaptable `CompteBancaire`. Le modèle actuel implique que ce système comporte une seconde classe `CompteBancaireType`.

L'ajout du nouveau type `Compte-Service`, comme une adaptation de la classe `BankAccount` (`CompteBancaire`) conduit à l'évolution dynamique du modèle objet de ce langage d'experts de la façon suivante :

Une instance de la classe `CompteBancaireType` est créée. Le nom de celle-ci est `Compte-Service`. Elle va comprendre par ailleurs les définitions des attributs et procédures décrits dans le paragraphe 2.3 de l'introduction. Cet objet est une instance terminale qui joue le rôle d'une classe (par le code écrit de façon appropriée dans notre outillage pour interpréter cette instance comme une classe). Toutefois, de ce fait, les programmeurs ne peuvent pas l'éditer à travers leurs outils habituels. Cela conduit aux problèmes n°1 et n° 3 énoncés ci-dessus.

1.2.2 Instanciation de types d'objets ajoutés dynamiquement

Supposons à présent vouloir instancier le nouveau type de `Compte-Service`. Dans la mesure où ce type est décrit à l'aide d'une instance terminale, il est impossible de l'instancier (au sens de la programmation par objets [Per98]). Aussi, la technique mise en œuvre par DOM consiste à créer une instance de la classe `CompteBancaire` qui, associée à l'instance de la classe `CompteBancaireType` qui décrit le type `Compte-Service`, est interprétée comme une instance de l'hypothétique classe `CompteService`.

C'est cette situation qui explique la confusion des programmeurs face aux exécutions des modèles objets adaptatifs (problème n° 2 ci-dessus).

En somme, notre analyse est que la source de ces problèmes est dans le fait que cette conception considère une adaptation comme une instance terminale, interprétée comme une classe. Ce choix d'implantation complexifie considérablement la compréhension du modèle objet issu de l'adaptation dynamique par les programmeurs. Il faut alors créer des outils d'édition spécifiques, différents de ceux dont se servent habituellement les programmeurs.

1.3 Solution à l'aide de méta-classes

Cette section est dédiée à la description détaillée de notre solution aux problèmes exposés ci-dessus et sa mise en œuvre à l'aide du langage SMALLTALK-80, le langage à objets réflexif actuellement le plus répandu.

1.3.1 Rapprochement de la représentation des adaptations et spécialisations

Le rapprochement de la représentation des adaptations et celle des spécialisations veut dire que des structures semblables devaient être utilisées pour modéliser la définition de classes dans les langages à objets et celle des adaptations dans le cas des langages d'experts. En effet, dans les deux cas le système doit permettre l'ajout de nouveaux concepts, la définition de leur structure et comportement.

Le but de ce rapprochement est de permettre la création d'outils d'édition (e.g. flâneurs) communs des spécialisations et des adaptations. De plus, dans ce cas une adaptation peut être instanciée au même titre qu'une classe. Cette approche facilite alors le travail collaboratif des experts et des programmeurs en rendant les adaptations accessibles aux programmeurs via leurs outils habituels.

La validation expérimentale de cette idée peut prendre deux formes :

1. créer un nouveau langage à objets dont la conception prend en considération ce nouveau besoin. Cette possibilité sort du cadre de cette recherche (cf. le mot de la fin, page 220) ;
2. modifier la conception des langages actuels pour y intégrer la gestion appropriée de ce nouveau besoin. C'est cette seconde possibilité qui est utilisée ici grâce notamment à la présence des méta-classes dans certains langages à objets réflexif.

Il s'agit plus précisément du langage SMALLTAK-80 [GR83] qui met en œuvre le choix implicite de la méta-classes [BC89b] et le langage METACLASSTALK [Bou99b] qui met en œuvre le choix explicite de méta-classes [BC87a, Coi87a].

1.3.2 DARC-II : le nouveau modèle d'adaptation

Nous proposons comme modèle de solution une nouvelle version du système DARC, que nous proposons d'appeler DARC-II. Celui-ci offre une nouvelle modélisation de la spécialisation dynamique, ou plus précisément de l'ajout dynamique de nouveaux types d'objets, qui résout les problèmes exposés ci-dessus.

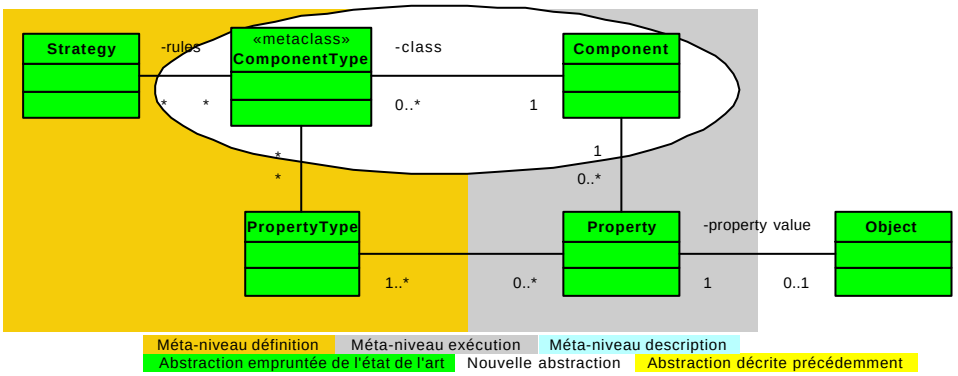


Figure 55 : DARC-II : prise en cosidération de la *relation* avec les langage à objets.

La structure du système DARC-II est illustrée par le diagramme UML [FS97] de la Figure 55 ci-dessus. Ce modèle est conçu suivant l'idée que toute adaptation (représentée par l'abstraction `Component`) est l'instance d'une méta-classe qui la modélise (représentée par l'abstraction

`ComponentType`). De ce fait la représentation des classes et des adaptations se rejoignent au niveau de la représentation des classes, e.g. la classe `Class` dans le système SMALLTALK-80.

Hormis ce point de différence, les autres aspects de ce modèle sont identiques à son prédécesseur, le modèle DARC-I décrit dans le chapitre III.

Il est ici important de noter que ce problème d'intégration du modèle des adaptations dans le langage à objets est également valable pour la représentation des structures et des procédures. En somme, il s'agit d'un problème de mise en œuvre de langages à objets dont la discussion sort du cadre de cette étude. En ce qui concerne ce problème particulier, dans cette thèse nous nous intéressons uniquement à la modélisation de la relation entre la représentation d'un langage d'experts d'une adaptation et celle des langages à objets d'une spécialisation.

1.4 Différents types d'adaptation

Le type d'une adaptation détermine, sur le plan pratique, le comportement offert aux experts lors de la mise en œuvre de l'adaptation. Nous considérons ici plusieurs types d'adaptation, que nous avons observé lors de cette étude, que nous proposons de classer ainsi :

1. *adaptation de nom* : permet d'adapter le nom d'un concept à un cas d'usage particulier.
2. *adaptation de structure* : permet d'adapter le nom et la structure d'un concept à un cas d'usage particulier.
3. *adaptation de comportement* : permet d'adapter le nom, la structure et le comportement d'un concept à un cas d'usage particulier.
4. *adaptation* : conjugue les trois fonctions ci-dessus avec une gestion du référentiel de descriptifs de service pour chaque adaptation.
5. *adaptation avec stratégie d'héritage* : permet d'adapter le nom, la structure et le comportement d'un concept tout en appliquant une certaine stratégie d'héritage de la structure et du comportement des super-classes. La stratégie d'héritage par défaut correspond à celui des langages à objet à l'héritage simple.
6. *adaptation prototypique* : ce type d'adaptation considère chaque "instance terminale" comme un *être* adaptable (cf. la nation de ***classe autonome*** ci-dessous) et permet d'adapter son nom, sa structure et son comportement avec une certaine stratégie d'héritage. Nous reviendrons sur ce type particulier plus longuement dans les perspectives, page 207.

Pour des raisons pratiques et afin de préciser le type d'adaptation associé à chaque classe dans les diagrammes de classe UML, nous utilisons différents stéréotypes de classe qui sont récapitulés par le Tableau 8 ci-dessous :

Le type d'adaptation	Le stereotype de classe utilisé	Abréviation
Adaptation de nom	*Name refinement*	*name ref.*
Adaptation de structure	*Structural refinement*	*strutural ref.*
Adaptation de comportement	*Behavioral refinement*	*beh. refinement*
Adaptation	*Refinement*	*refinement*
Adaptation avec stratégie d'héritage	*Refinement with inheritance strategy*	*ref. with delegation*
Adaptation prototypique	*Prototype*	*prototype*

Tableau 8 : Stéréotypes de classe pour préciser le type d'adaptation dans les diagrammes UML.

Il est important de noter que la création de langages d'experts peut conduire à la création de nouveaux types d'adaptation qui sont propres au domaine d'application. Par exemple, lorsqu'un concept du domaine gère des valeurs qui sont communes à l'ensemble de ses instances, chaque valeur peut être stockée dans une variable d'instance de classe au sens du langage SMALLTALK-80.

C'est à titre d'exemple, le cas pour le taux d'intérêt pour un type de compte rémunéré (`Savings Account`). L'attribut `taux` peut, en effet, être une variable d'instance de la méta-classe qui modélise ce type adaptation. Concrètement cela veut dire que tout type de compte rémunéré est créé par l'expert comme une instance de ce type d'adaptation.

1.5 Modèle SMALLTALK-80 de la programmation par spécialisation

Le but de cette section est de déterminer, par une analyse précise du mode de fonctionnement de ce noyau, la technique appropriée pour l'intégration de l'outillage de l'adaptation dans le noyau classe/méta-classe du système SMALLTALK-80.

1.5.1 Trois principales hiérarchies de classes

Par ses nombreuses facilités réflexives [FJ89, Riv96a, KdRB91] SMALLTAK-80 [Kra83] offre aux outilleurs des moyens de programmation souples et puissants. L'une de ces facilités se concrétise à travers des méta-classes [Ing78, Coi87, Per98].

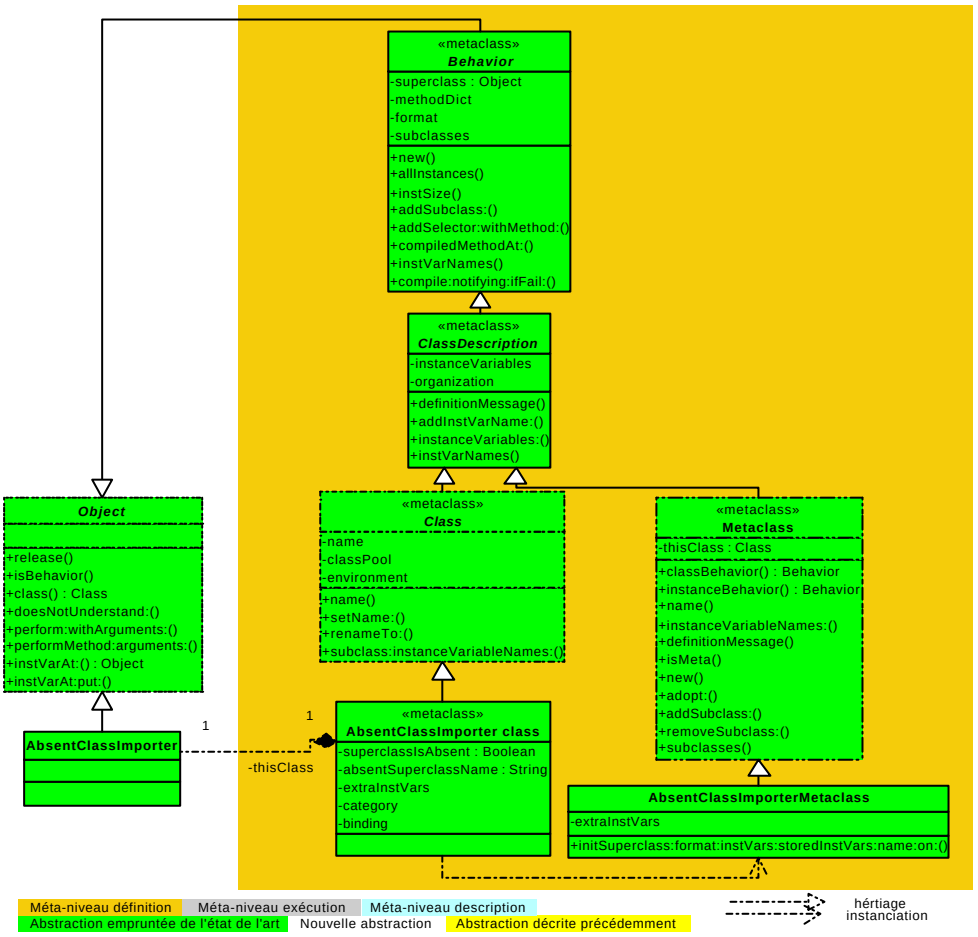


Figure 56 : Modèle Smalltalk-80 de la programmation par spécialisation de classes.

Nous tirons profit de cette possibilité pour valider l'idée du rapprochement des structures de représentation des spécialisations et adaptations.

Comme l'illustre la Figure 56 ci-dessus, le noyau du langage SMALLTAK-80 est principalement composé de trois classes `Metaclass`, `Class` et `Object`. Ce noyau constitue un cadre opérationnel pour la programmation par spécialisation.

Nous allons d'abord rappeler rapidement les principes qui régissent la conception de ce noyau ainsi que son extension lors de la programmation. Ensuite nous montrons dans quelle mesure cette conception permet une intégration aisée de l'outillage dédié à l'adaptation, toute en conservant toutes les propriétés du langage cible.

1.5.2 Evolution des trois hiérarchies lors de la programmation

Le modèle SMALLTAK-80 de la programmation par spécialisation est fondé sur la contribution de trois hiérarchies :

1. *hiérarchie de classes* : ce sont des classes qui modélisent la structure et le comportement des instances terminales (comme des concepts métier).
2. *hiérarchie de méta-classes* : ce sont des classes qui modélisent la structure et le comportement des classes de la première hiérarchie.
3. *hiérarchie de méta-méta-classes* : ce sont des classes qui modélisent la structure et le comportement de méta-classes.

Mise à part les subtilités relatives à la réalisation réflexive de ce noyau[103], nous pouvons raisonnablement considérer que chacune des trois classes `Metaclass, Class` et `Object` se trouve à la racine de l'une de ces trois hiérarchies[104].

L'ajout d'une nouvelle classe correspond alors à une extension systématique de *deux* de ces trois hiérarchies. La première extension est faite de façon explicite et correspond à la hiérarchie à laquelle appartient la super-classe de la classe ajoutée. La seconde extension est faite de façon, *implicite* et correspond systématiquement à celle de la hiérarchie des méta-classes[105].

La méta-classe `Class` est, en effet, une classe abstraite qui n'est jamais instanciée[106]. Elle transmet sa structure et son comportement à chaque nouvelle classe par le fait que la méta-classe de celle-ci (instance de la classe `Metaclass`) héritent directement ou indirectement de la classe `Class`. Ce mécanisme est entièrement *automatique*.

La seule façon dont le programmeur peut intervenir dans ce processus, et cela de façon indirecte, c'est *le choix de la super-classe de la méta-classe de sa nouvelle classe.* C'est alors cette possibilité que nous allons exploiter pour intégrer harmonieusement notre outillage dédié à l'adaptation au noyau clase/méta-classe de SMALLTALK-80.

La nature des classes ajoutées est alors déterminée de façon implicite et cela par le système[107]. En effet, contrairement aux premières intuitions que l'on pouvait avoir, l'outillage réflexif de l'adaptation ne correspond pas à spécialiser la méta-classe primitive `Class` qui est le modèle de toutes les classes du système Smalltalk-80[108]. En effet, une telle spécialisation même si elle est correcte sur le plan conceptuel, n'a pas d'effet au niveau de la nature des dasses créées par les programmeurs qui spécialisent la classe `Object` et qui n'ont pas droit au choix de leur méta-classes.

---

[103] Notamment le fait que la classe `Object` se trouve à la racine de ces trois hiérarchies.

[104] On peut ici également noter que ce modèle conduit à un système composé d'objets qui peuvent être répartis dans quatre niveaux conceptuels. En effet, toutes les classes de chacune de ces trois hiérarchies sont issues de l'instanciation de leur méta-classes. Plus précisément, les classes de la hiérarchie de classes sont des instances uniques d'une sous-classe de la classe `Class`. Notamment, la première sous-classe de la classe `Class` est la classe `Object class`, dont l'instance unique est la classe `Object`[104], qui comporte le comportement par défaut de toutes les instances terminales. Les sous-classes de la classe `Class` sont appelées des méta-classes et sont toutes, à leur tour, instances de la classe `Metaclass`. Par ailleurs, la classe `Class` est elle même une instance de sa méta-classe, laquelle est une instance de la classe `Metaclass`. Cette dernière est à son tour instance de sa méta-classe qui est aussi une instance de la classe `Metaclass`, ce qui permet d'éviter la régression infinie. Par conséquence, c'est la classe `Metaclass` qui est la racine de l'arbre d'instanciation de toutes les classes et méta-classes[104]. C'est donc la classe `Metaclass` qui est à l'origine de ces deux hiérarchies de classes et de méta-classes. Le quatrième niveau de ce modèle est composé d'instances terminales.

[105] Cette évolution parallèle est l'une des propriétés principales du système Smalltalk-80. Celle-ci assure notamment la compatibilité ascendante et descendante [Gra89].

[106] Au contraire de la méta-méta-classe `Metaclass` qui est instanciée une fois, lors d'ajout de chaque nouvelle classe. A noter aussi des exceptions comme le cas de la méthode `nilSubclassImporter`.

[107] A noter qu'en ce qui concerne le langage METACLASSTALK, il offre plus de souplesse à ce niveau en permettant au programmeurs de choisir à volonté (au prix de quelques complications) aussi bien la super-classe que la méta-classe de la classe à ajouter. Nous étudions l'impact de cette différence sur le raffinement dynamique dans la section sur l'implantation de DYCRA à l'aide de méta-classes explicites.

[108] y compris elle-même.

2 Mise en œuvre réflexive de DYCRA à l'aide du langage SMALLTALK

Après un bref rappel des systèmes utilisant déjà la possibilité du langage SMALLTALK-80 que nous venons d'exposer, afin d'intégrer de nouveaux systèmes dans son noyau, nous exposons ici notre implantation du second outillage de l'adaptation qui conduit au framework MIDYCTALK[109].

2.1 Modélisation des adaptations avec les méta-classes standard

L'analyse ci-dessus montre que le *choix d'un type de méta-classe doit se faire de façon indirecte,* par le choix de l'instance unique d'une méta-classe implicite qui spécialise la classe `Class`. La solution à notre problème consiste alors à :

1. concevoir l'outillage de l'adaptation comme une spécialisation de la méta-classe de la classe `Object` (`Object class`); et
2. "exiger" le choix de faire hériter une classe de l'instance unique de cette méta-classe afin de la rendre adaptable.

Cette approche de spécialisation de la méta-classe de la classe `Object` est déjà utilisée dans la mise en œuvre d'autres systèmes. NEOPUS l'utilise pour mettre en œuvre les bases de règles [Pac92, PP94, Pac94, Pac95]. Benny Sadeh fait, par ailleurs, appel à cette technique pour proposer une mise en œuvre en SMALLTALK-80 des interfaces à la JAVA [Sad99].

Nous allons ici étudier de plus près la technologie des composants (Parcels) du système VISUAWORKS [Mir98]. Celle-ci spécialise les méta-classes `Object class` et aussi `Metaclass` (cf. la Figure 56). Le but de cette spécialisation est d'assurer le chargement dans une image SMALLTALK de classes dont la super-classe est absente. Les principes de la mise en œuvre de ce mécanisme sont les suivants.

Lorsque le chargeur ne trouve pas dans l'image la super-classe d'une classe à charger, dans un premier temps il crée à sa place une instance de la classe `AbsentClassImport`. Cet objet conserve les informations relatives à la classe absente qui ont été stockéés dans le fichier en cours de chargement. Cela comprend à titre d'exemple, le `format` de la classe. De plus, il prend en charge le chargement de ses "sous-classe" directes. Pour se faire, il procède à la création d'une vraie classe, instance de la méta-classe `AbsentClassImporter class`, qui s'installe dans la hiérarchie des classes à la place de la classe absente et qui va poursuivre le chargement des sous-classes.

La méta-classe `AbsentClassImporter class` est une spécialisation de la méta-classe de la classe `Object`. Elle définit les variables d'instances et les méthodes nécessaires à ses instances, qui sont des classes de nature particulière, c'est à dire les classes absentes. Cela comprend, à titre d'exemple, la variable d'instance `extraInstVars` qui contient le nom des variables d'instances héritées des super-classes absentes. En effet, les variables d'instances des classes absentes ne peuvent pas être considérées comme de "vraies" variables d'instances puisqu'elles n'ont pas toute à fait la même sémantique et requièrent un traitement particulier.

Cette situation est analogue au cas des descriptifs d'attributs dans DYCRA. D'autres données qui définissent de telles classes sont `superclassIsAbsent`, `absentSuperclassName`, `category` et `binding`.

A noter que conformément au problème de la propagation des effets d'une méta-classe dans toute la hiérarchie [BC87b], les sous-classes "présentes" d'une classe absente, c'est à dire celles qui seront effectivement chargée après la création d'une classe absente, vont hériter de œs variables d'instances et comportements. C'est ce phénomène qui explique l'existence de la variable d'instance `superclassIsAbsent` et les méthodes associées comme `isSubclassOfAbsentClass` pour pouvoir départager les classes et déterminer laquelle dans une hiérarchie est véritablement la représentante d'une classe absente.

[109] Acronyme pour implantation en Small**talk**-80 de **DYC**RA basée sur le choix **I**mplicite du **M**éta-classe.

Par ailleurs, la méta-classe `AbsentClassImporter class` est elle-même instance de la méta-méta-classe `AbsentClassImporterMetaclass`. Cette conception permet à la méta-classe `AbsentClassImporter class` de prendre notamment en charge la gestion des variables d'instances de classe de la méta-classe d'une classe absente.

Cette mise en œuvre permet tout particulièrement de charger sans difficulté les instances d'une classe dont la super-classe est absente ou de créer de nouvelles instances. De plus, lors d'un chargement ultérieur, le système se rend compte du chargement effective d'une classe qui était auparavant absente, il fait le nécessaire pour assurer le remplacement nécessaire.

2.2 Implantation réflexive de l'adaptation en SMALLTALK-80

La Figure 57 ci-dessous montre comment nous avons mis en œuvre cette nouvelle conception des adaptations à l'aide du langage SMALLTALK-80.

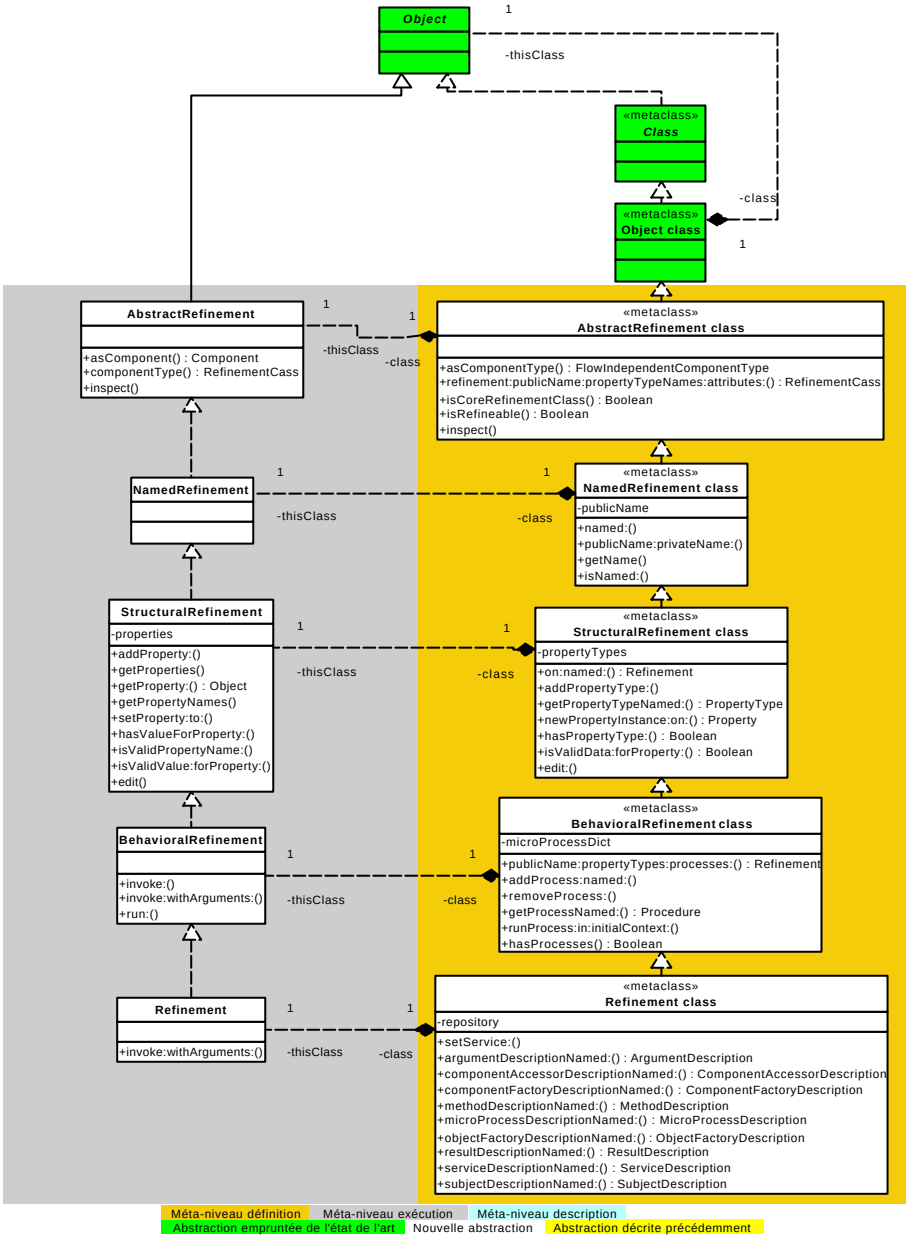


Figure 57 : DYCRA implanté dans le contexte du choix implicite de méta-classes par SMALLTALK.

2.2.1 Principales abstractions

Le modèle ci-dessus est composé de classes et de méta-classes. En raison de la mise en œuvre particulière du langage SMALLTALK-80 que nous venons d'évoquer ci-dessus, chaque classe va de pair avec une méta-classe. En ce qui nous intéresse ici, c'est à dire l'adaptation, c'est la méta-classe qui détermine les caractéristiques de son instance unique, ainsi que ses sous-classes.

A noter également que nous avons ici considéré que la méta-méta-classe `Metaclass` convient comme la classe de nos méta-classes. En effet, l'adaptation ne concerne que les classes et non pas leur

classes (les méta-classes). Aussi, jusqu'alors la spécialisation des méta-méta-classes ne s'est pas avérée nécessaire.

La structuration des méta-classes qui mettent en œuvre notre nouvelle solution suit notre définition des différents types d'adaptation (cf. la section 1.4 du même chapitre). Notre modèle est composé de cinq méta-classes et de leurs instances respectives. Le rôle de la plupart des classes est uniquement de représenter leur méta-classe. La classe `StructuralRefinement` fait toutefois exception et implante le protocole d'instances d'accès aux attributs. Les protocoles de classes et de méta-classes sont ici identiques à ceux présentés dans le cas de notre première implantation (cf. le chapitre III, le système DYCTALK). Voici une description succincte de chacune des abstractions de ce modèle.

La méta-classe AbstractRefinement class

Le rôle de cette méta-classe est de mettre en œuvre le comportement commun à tous les raffinements. Ce comportement peut être regroupé en plusieurs catégories :

1. les méthodes pour l'intégration des adaptations dans l'environnement de programmation du système VISUALWORKS [Cin01].
2. les méthodes pour la création d'une nouvelle adaptation.
3. la méthode `asComponentType` qui permet de transformer une adaptation (selon MIDYCTALK) en un complément de classe *équivalent* (selon DYCTALK). A noter que `asComponentType` est une méthode abstraite et requiert de ses implantations concrètes de fournir le nom des deux classes de base et type, suivant DOM (e.g. `CompteBancaire` et `CompteBancaireType`), qui sont nécessaires à la mise en œuvre de cette transformation.
4. les méthodes pour la libération d'une adaptation (`localObsolete`).
5. d'autres méthodes utilitaires.

La méta-classe NamedRefinement class

La méta-classe `NamedRefinement class` ici correspond à la classe `NamedComponent` dans le cas du système DYCTALK. Elle implante toutes les méthodes de la classe `NamedComponent` dont le but est de permettre d'associer un "nom métier" à une classe..

La méta-classe StructuralRefinement class

L'étape suivante dans la mise en œuvre de MIDYCTALK est l'ajout d'une méta-classe qui met en œuvre le protocole du DOM [RTJ00]. Ce rôle est confié à la méta-classe `StructuralRefinement class`. Celle-ci correspond à la classe `ComponentType` de DYCTALK. En effet, nous n'avons pas estimé nécessaire l'implantation d'une méta-classe équivalente à la classe `ComponentTypeLike` qui n'avait, a priori, qu'un intérêt pédagogique en raison de son implantation naïve des mécanismes de DOM.

Rappelons également ici que le protocole de la méta-classe `StructuralRefinement class` reste identique, du point de vu d'implantation, à celui de la classe `ComponentType.` Seulement, tous les protocoles d'instance deviennent des protocoles de classe, à l'exception du protocole de classe appelé `instance creation` qui reste un protocole de classe mais qui change de nom en `creating class hierarchy`.

La classe StructuralRefinement

Nous utilisons la classe `StructuralRefinement,` l'instance unique de la méta-classe `StructuralRefinement class,` pour servir de super-classe par défaut à toutes les adaptations. Notons que les propriétés de celle-ci seront héritées par les sous-classes de la méta-classe `StructuralRefinement class,` qui complèteront cette implantation notamment par l'ajout de la gestion des procédures.

Cette classe synthétise le protocole des quatre classes `AbstractComponent`, `ComponentLike`, `BasicComponent` et `Component`. La seule différence entre les deux implantation se situe au niveau de la variable d'instance `type` de la classe `BasicComponent`. En effet, celle-ci n'est plus utile car dans cette nouvelle implantation le *type* d'une instance d'un raffinement est sa classe qui peut être obtenu par l'envoi du message `class`.

La méta-classe BehavioralRefinement class

Le rôle de la méta-classe `BehavioralRefinement class` est d'implanter le protocole de gestion de la définition et de l'interprétation des procédures. Elle correspond à la classe `FlowIndependentComponentType` du système DYCTALK. Elle comprend un protocole identique à son homologue, mais implanté comme un protocole de classe. Il existe, toutefois, deux différences :

1. La variable d'instance de classe `dynamicClassDescriptionDict` et les méthodes associées perdent leur utilité. En effet, dans la mesure où les adaptations ont à présent le statut de classe, la gestion de leur référentiel est confié au système VISUALWORKS [Cin01].
2. le protocole d'instance `instance creation` n'est plus nécessaire car les adaptations sont des classes et la méthode `new` est implantée dans leur classe (au "niveau méta").

La méta-classe Refinement class

Notre nouvelle outillage de l'adaptation s'achève par l'ajout à cet ensemble de la classe `Refinement class` dont le rôle est de permettre la gestion locale et personnalisée de descriptifs de service, qui jouent un rôle essentiel dans la mise en œuvre de langages d'experts (cf. le chapitre I, le système DART).

En effet, dans notre framework DYCTALK ce rôle avait était confié à la classe de chaque type de descriptif de service (la hiérarchie de classe `EnityDescription`). Or, ce choix n'offre qu'un référentiel global à l'ensemble des adaptations. Il nous semble plus pratique de proposer au niveau de chaque adaptation une possibilité de gestion de descriptifs de services qui lui sont associés. C'est pourquoi cette méta-classe prévoit une variable d'instance `repository` est les méthodes nécessaires à cette gestion.

2.2.2 Règles générale de correspondance entre DYCTALK et MIDYCTALK

Cette nouvelle modélisation rehausse, du point de vue de langage SMALLTALK-80, d'un "grade" le niveau des classes qui mettent en œuvre les adaptations. L'impact de ce changement au niveau de l'implantation est que les protocoles d'instances deviennent systématiquement des protocoles de classe. De même, les variables d'instance deviennent des variables d'instance de classes. Le code des méthodes correspondantes reste intacte.

Cette règle ne s'applique, toutefois, pas aux protocole de classe qui assurent la création des instances. Ces protocoles contiennent en effet, des méthodes comme `named:` dont le rôle est de créer de nouvelles adaptations. Pour se faire, les experts activent une fonction qui envoie le message d'instanciation, ici `named:`, à la classe qui représente le type d'adaptation souhaité, par exemple `Refinement`, avec en argument le nom de l'adaptation. Aussi, le message de création d'une nouvelle adaptation n'est pas envoyé à la méta-classe, mais à une classe (la super-classe de la nouvelle adaptation). De ce fait, les protocole de création d'instance restent des protocoles de classe[110].

Par ailleurs, il est possible d'établir une correspondance entre les abstractions des diagrammes de classe qui décrivent les deux implantations DYCTALLK et MIDYCTALK. Les méta-classes `NamedRefinement class`, `StructuralRefinement class`,

[110] D'autres schémas d'instanciation sont également possibles, y compris s'adresser directement à la méta-méta-classe. Le système `VisualWorks` propose aussi de s'adresser à l'objet qui représente l'espace de nommage (*Naming Space*) de la classe à créer et de passer sa super-classe en argument.

`BehavioralRefinement class` et `Refinement class` correspondent ici aux classes `NamedComponent`, `ComponentTypeLike`, `ComponentType` et `FlowIndependentComponentType` de DYCTALLK.

En ce qui concerne le niveau "base" des deux modèles, on peut globalement considérer que la classe `StructuralRefinement` du nouveau modèle correspond à la classe `Component` du framework DYCTALLK.

3 Exemple : adaptation de comptes bancaires

Dans l'exposé qui suit, nous illustrons, à travers l'exemple introductif d'adaptation de comptes, le fonctionnement de l'outillage le plus élaboré que nous avons pu construire jusqu'alors, le framework MIDYCTALK. Lors de cet exposé nous décrivons les activités des experts, mais aussi celles des programmeurs, dans l'ordre de leur occurrence.

Nous comparerons, le cas échéant, le mode de fonctionnement de MIDYCTALK à celui de DYCTALK[111]. En effet, trois des principales fonctions assurées par MIDYCTALK le sont aussi par DYCTALK. Il s'agit de l'ajout de nouvelles abstractions, définition de leur structures et procédures. Cela couvre également l'instanciation des structures, l'activation des procédures et toutes les autres dimensions que nous avons exposés dans les chapitres I et III.

3.1 Le modèle objet initial

Avant de rentrer dans le vif du sujet, nous évoquons ici brièvement la création du langage d'experts, l'objet de l'adaptation, et plus particulièrement son modèle objet initial (tel qu'il est lors de la livraison aux experts du langage d'experts).

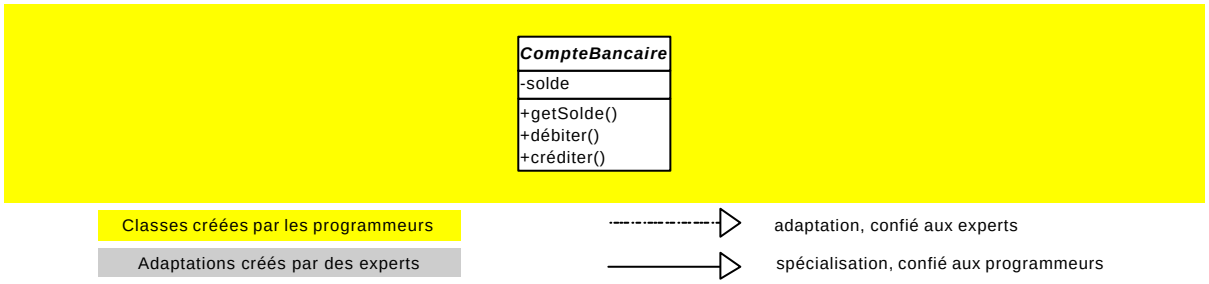


Figure 58 : Modèle objet initial du langage à objet dédié à la gestion de comptes.

[111] Il est important d'insister ici sur le fait que notre nouvelle implantation du système de classes DYCRA conserve toutes les fonctionnalités de la première implantation. En effet, les exemples présentés dans le cas de DYCTALLK peuvent ici être réimplantés de façon quasi identiques. Le seul changement à y appliquer consiste à remplacer les références aux classes de DYCTALLK par une référence à la classe qui représente le type d'adaptation choisi dans "l'univers" de MIDYCTALK. Celle-ci doit être l'instance d'une des méta-classes de MIDYCTALK.
Par exemple, il faut changer la classe `CompteBancaireType` par la classe `CompteBancaire`. Nous supposons alors que la `CompteBancaire` est sous-classe direct ou indirecte de la classe `AbstractRefinement` (en l'occurrence `Refinement`).
Rappelons que, en gros, c'est alors la méta-classe `CompteBancaire class` qui joue le rôle de la classe `CompteBancaireType`.
Rappelons aussi que la méthode `asComponentType` (cf. ci-dessus, paragraphe 2.2.1, page 155) permet de transformer une adaptation en son équivalent complément de classe.

En effet, dans le paragraphe 3.1, page 136 du chapitre IV nous avons évoqué rapidement que la question de la méthodologie de développement des langages d'experts reste une question ouverte.

Pour l'heure, nous évitons donc ce débat et, pour les besoins de notre validation expérimentale, supposons l'existence d'un langage d'experts muni d'un modèle objet composé d'une classe : `CompteBancaire` (comme permet de l'illustrer la Figure 58 ci-dessus).

Cette classe hérite de la classe `Refinement`. Comme nous l'avons décrit dans ce chapitre et notamment dans le paragraphe 1.5, page 151 consacré au modèle de la programmation par spécialisation de SMALLTALK-80, dans le cas de MIDYCTALK c'est ainsi (par l'héritage) qu'une classe devient adaptable.

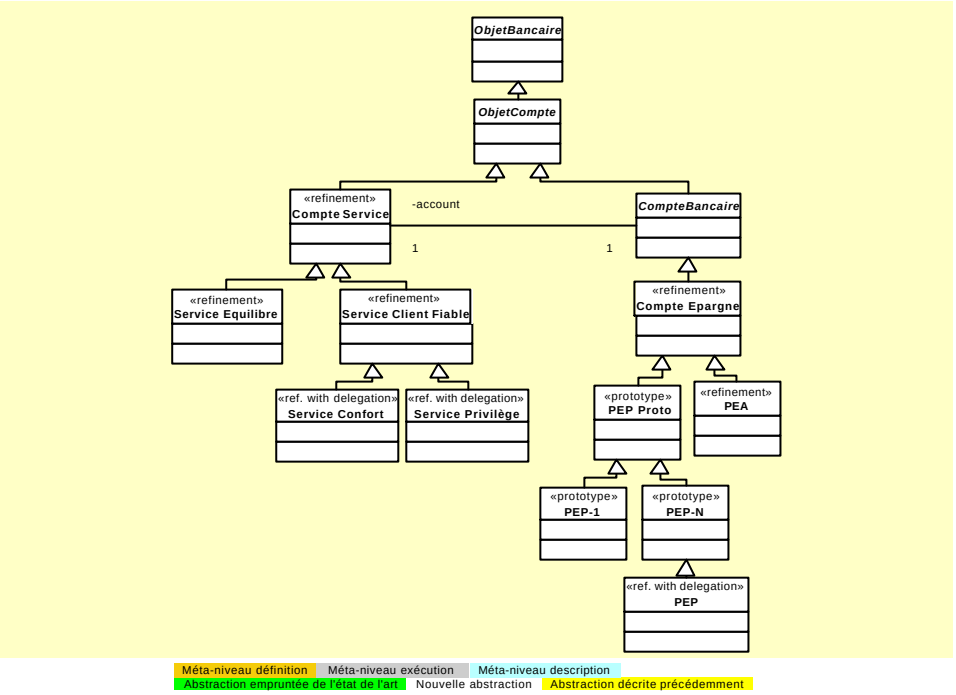


Figure 59 : Modèle objet visé par notre exemple d'adaptation de comptes bancaires.

La Figure 59 ci-dessus illustre, par ailleurs, le modèle objet visé. Nous allons montrer lors de cet exemple, ainsi que sa suite dans le chapitre V, comment ce modèle sera obtenu par une collaboration entre les experts et les programmeurs.

En somme, la classe `CompteBancaire` de la Figure 58, qui correspond ici à la classe `BankAccount`, est d'une part spécialisée dans un premier temps par des experts, par la création des trois types de `Compte-Service` exposés dans l'introduction. Ensuite, les programmeurs *refactorent* (restructurent) le modèle obtenu à l'issu des interventions des experts afin de produire le modèle des Comptes-Service présenté ci-dessus.

D'autre part, les experts procèdent à la modélisation du compte PEP par prototypage. De plus, il est nécessaire que le type d'adaptation de chaque abstraction lui soit propre (cf. les annotations sur les rectangles qui correspondent aux stéréotypes récapitulés par le Tableau 8, 150). Ces deux derniers aspects seront discutés lors du prochain chapitre.

3.2 Ajouter de nouvelles adaptations

Comme nous l'avons évoqué également dans le paragraphe 3.2, page 137, la première étape de l'adaptation d'un langage d'experts consiste à ajouter de nouveaux types d'objets. Dans le cas de notre nouveau framework MIDYCTALK cette fonction s'appuie sur les méta-classes. Les adaptations sont de même nature que les classes.

Aussi, grâce à MIDYCTALK, et comme nous l'avions annoncé dans l'introduction, §2.4.1, page 41, il devient à présent possible de faire évoluer le modèle initial vers le modèle de la Figure 60 ci-dessous. L'adaptation `Compte-Service Equilibre` est une classe, du point de vue du langage à objets spécialisé, et hérite effectivement de la classe `CompteBancaire`[112].

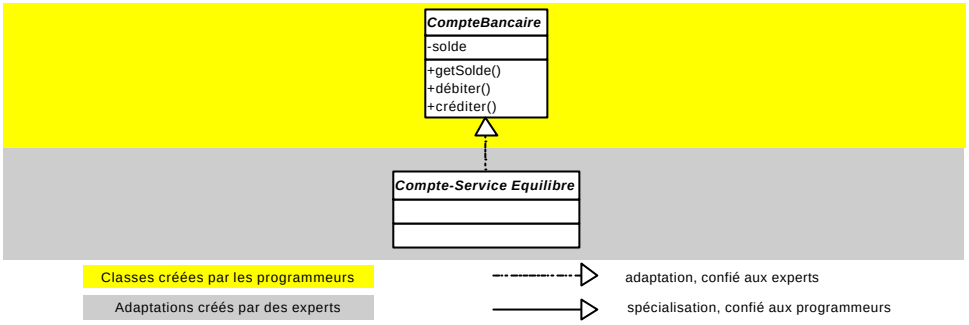


Figure 60 : Evolution du modèle objet par l'ajout du type `Compte-Service Equilibre`.

De ce fait, il y a donc ici une différence notable par rapport à ce que nous avons pu illustrer dans le cas du paragraphe 3.2, page 137.

Le script[113] de la Figure 61 ci-dessous illustre l'envoi de message qui réalise la création de l'adaptation souhaitée. Le résultat de l'exécution de ce script est, en effet, une classe.

Or, le résultat du script de la Figure 62[114], qui est sur le plan fonctionnel équivalent celui de la Figure 61, est une instance terminale. En effet, dans le cas de ce dernier cette fonction s'appuyait sur le composant FDOM du framework DYCTALK, qui a une conception tout à fait différente de l'adaptation (cf. paragraphe 2.1, page 110 du chapitre III).

```
CompteBancaire
            publicName: 'Compte-Service Equilibre'
```

Figure 61 : Script de création de nouveaux types d'objets suivant DARC-II.

Il importe de préciser ici que, sur le plan pratique, cette différence reste invisible aux yeux des experts. En effet, ils doivent simplement fournir le nom de l'adaptation et décider de la classe à adapter. Comme nous venons de l'exposer ci-dessus, le seule changement se situe au niveau de la classe qui reçoit le message de la création du nouveau type.

```
CompteBancaireType
                  named: 'Compte-Service Equilibre'
```

Figure 62 : Script de création de nouveaux types d'objets suivant le schéma DARC-I.

Ce changement dans la nature des adaptations est source de nombreux bénéfices. Par exemple, la Figure 63 ci-dessous illustre une vue sur cette classe à travers le flâneur du langage SMALLTALK-80. Elle

[112] Les deux autres type de compte peuvent également être créés suivant la même démarche.

[113] En raison de manque d'interface graphique, nous sommes amenés dans ce qui suit décrire les interventions des experts à travers des scripts décrits dans la syntaxe du langage SMALLTALK-80. Il est important de noter qu'il ne s'agit en aucun cas de demander à l'expert de procéder à l'écriture de ce type de scripts. En principe, il devait exister des interfaces graphiques chargées de la saisie des informations nécessaires et de l'exécution de ces scripts sur la base d'informations recueillies.

[114] Déjà exposé dans le paragraphe 3.2, page 137.

permet de constater l'ajout par des programmeurs d'une variable d'instance `dateDeTransaction` et des méthodes (dans les protocoles *accessing* et *printing*). Ils pourront également modifier le nom auto-généré `CompteBancaire1` par un nom interne plus approprié comme `CompteServiceEquilibre`.

Nous reviendrons sur ces considérations dans le paragraphe §3.5, page 164, ci-dessous.

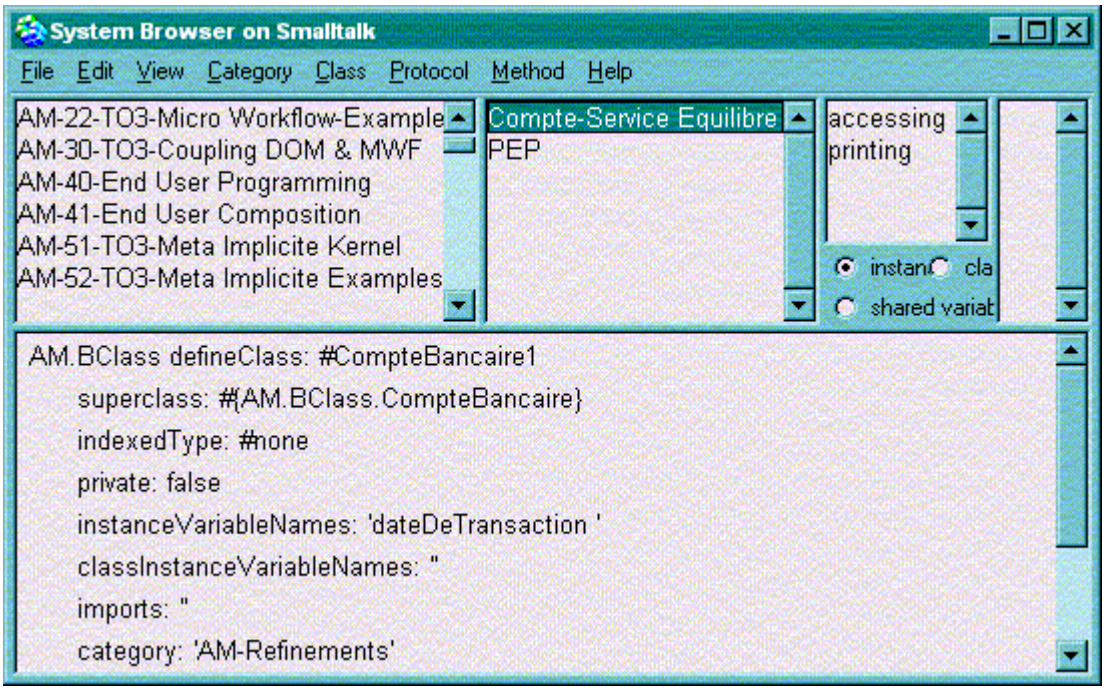


Figure 63 : Adaptation `Compte-Service Equilibre`, vue par le flâner de VISUALWORLKS.

3.3 Définir la structure

Il faut à présent définir (dynamiquement) la structure de nouvelles adaptations, ici le `Compte-Service Equilibre`. Cette fonction est toujours assurée par des éléments que nous avons empruntés du framework FDOM (§2.1, page 110 du même chapitre).

Plus précisément, pour outiller cet aspect nous faisons appel au schéma de conception DOM qui préconise l'usage de deux classes `Property` et `PropertyType` (§2.2, page 91, chapitre II).

```
| typeCompteServiceEquilibre attTypeCarte |
typeCompteServiceEquilibre := CompteBancaire
                                  publicName: 'Compte-Service Equilibre'.
attTypeCarte := PropertyType
      on: String
      named: 'Type de carte bancaire associé'.
typeCompteServiceEquilibre addPropertyType: attTypeCarte.
^typeCompteServiceEquilibre
```

Figure 64 : Script d'ajout d'un attribut à un nouveau type d'objets (cas de MIDYCTALK).

Le script de la Figure 64 ci-dessus illustre la création suivant ce modèle du descriptif d'attribut `Type de carte bancaire associé.` Celui-ci est créé par l'envoi du messgae `on:named:` à la classe `PropertyType`, avec en argument le nom et le type de l'attribut. Il est ensuite ajouté à l'adaptation `Compte-Service Equilibre` par l'envoi du message `addPropertyType:`.

Pour les besoins de notre exemple, nous supposons l'ajout des attributs `Cumul d'agios`, `Compte-chèque associé`, `Taux préférentiel de calcul d'agios` et `Montant de découvert forfaitaire`, suivant le même procédé[115]. Cela fait effectivement évoluer le modèle objet initial vers celui illustré par la Figure 65 ci-dessous.

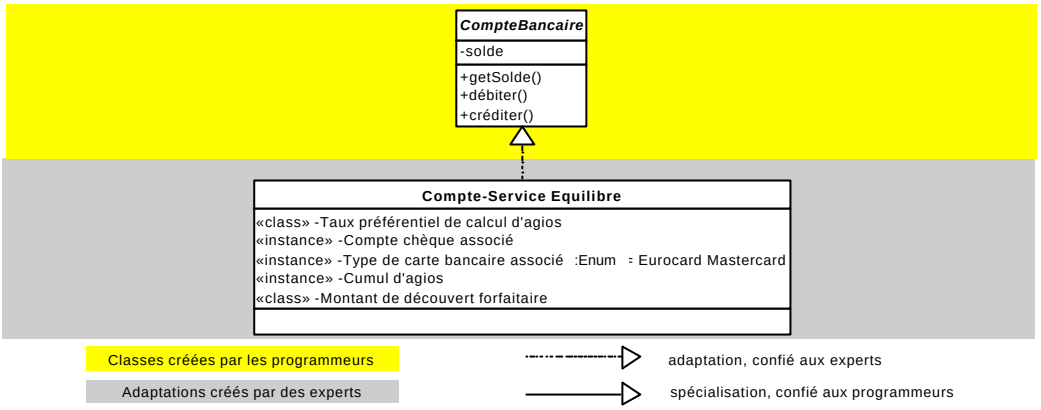


Figure 65 : Définition de la structure du `Compte-Service Equilibre`.

Afin de permettre une comparaisons des deux approches, nous illustrons par la Figure 66 ci-dessous, le même script que celui de la Figure 64 ci-dessus écrit dans le cas de DYCTALK.

```
| typeCompteServiceEquilibre attTypeCarte |
      typeCompteServiceEquilibre := CompteBancaireType
                                          named: 'Compte-Service Equilibre'.
      attTypeCarte := PropertyType
            on: Core.String
            named: 'Type de carte bancaire associé'.
      typeCompteServiceEquilibre addPropertyType: attTypeCarte.
      ^typeCompteServiceEquilibre
```

Figure 66 : Script d'ajout d'un attribut à un nouveaux type d'objets (cas de DYCTALK)

On peut observer que la seule différence entre ces deux approches se situe au niveau du statut, vis à vis du langage à objet spécialisé, de celui qui reçoit le message d'ajout de l'attribut. Dans le premier cas il s'agit de la classe `CompteBancaire` et dans le second d'une instance terminale de la classe `CompteBancaireType`.

Le développement plus poussé de cette dimension ne rentre pas dans le cadre de notre travail. Joseph W. Yoder, Federico Balaguer et Ralph Johnson propose une étude systématique d'un outillage plus élaboré, appliqué à la modélisation des observations médicales par des cadres hospitaliers [YBJ99, YB99].

3.4 Génération automatique de *Getters* et *Setters*

Pour permettre l'expression des accès en lecture et en écriture aux attributs définis dynamiquement, lors de la composition des procédures, nous proposons d'associer automatiquement deux descriptif de service à chaque attribut.

[115] Nous fournissons en annexe II, page 258 une description plus détaillée de la structure de ces types de compte.

Par exemple, lorsque l'expert ajoute l'attribut `Cumul d'agios`, notre outillage génère automatiquement deux descriptifs de services appelés `Obtenir Cumul d'agios` et `Affecter Cumul d'agios`.

```
Refinement class >> addPropertyType: aPropertyType

      super addPropertyType: aPropertyType.
      self generatePropertyServiceDescriptions: aPropertyType
```

Figure 67 : Génération automatique de descriptifs de service en cas d'ajout d'attributs (phase 1).

Comme permet de l'illustrer la Figure 67 ci-dessus, la première phase de la génération de ces descriptifs est mise en œuvre par la méthode `addPropertyType:`, redéfinie dans la méta-classe `Refinement class`. Celle-ci procède à la génération des deux descriptifs à l'aide de la méthode `generatePropertyServiceDescriptions:` (cf. Figure 68 ci-dessous). Cette dernière méthode se charge également du stockage de ces deux descriptifs dans le référentiel de descriptifs de service de l'adaptation, ici `Compte-Service Equilibre`, par l'envoie du message `setService:`.

```
Refinement class >> generatePropertyServiceDescriptions: aPropertyType
      ""
      | getterSD setterSD aServiceDescriptionClass |

      aServiceDescriptionClass := self componentAccessorDescriptionClass.
      getterSD := aPropertyType asGetter: aServiceDescriptionClass.
      getterSD setTheRefinedClass: self.
      setterSD := aPropertyType asSetter: aServiceDescriptionClass.
      setterSD setTheRefinedClass: self.

      self
            setService: getterSD;
            setService: setterSD.
```

Figure 68 : Génération automatique de descriptifs de service en cas d'ajout d'attributs (phase 2).

La Figure 69 ci-dessous illustre le script de code écrit en SMALLTALK-80 qui comprend les responsabilités confiées à ce sujet à la classe `PropertyType`. Il s'agit de créer deux instances de la classe `ComponentAccessorDescription`, qui modélise ce type de descriptifs de service (cf. §2.5.2.2, page 135 du chapitre III). Cela comprend également le calcul de la valeur des arguments requis.

```
PropertyType >> getterNamePrefix
	^'Obtenir '

PropertyType >> setterNamePrefix
	^'Affecter '

PropertyType >> getGetterName
	^self getterNamePrefix, self getName


PropertyType >> getSetterName
	^self setterNamePrefix, self getName


PropertyType >> asGetter: aServiceDescriptionClass
	^aServiceDescriptionClass
		named: self getGetterName
		serviceID: self getName
		serviceType: #attributGet
		resultType: dataType.


PropertyType >> asSetter: aServiceDescriptionClass
	^aServiceDescriptionClass
		named: (self getSetterName, ':')
		serviceID: self getName
		serviceType: #attributSet
		resultType: dataType
```

Figure 69 : Génération automatique de descriptifs de service en cas d'ajout d'attributs (phase 3).

3.5 Définir manuellement des descriptifs de service

L'étape suivante de l'adaptation consiste à préparer les descriptifs de services nécessaires à la composition.

Les trois figures ci-dessous illustrent le mode d'emploi de l'outillage que nous avons présenté dans le chapitre I, paragraphe 3.4, page 70 et qui permet de créer (ici manuellement) les descriptifs de service nécessaires à la définition des procédures.

En l'occurrence afin de composer les deux procédures `Cumuler les agios du jour()`et `Traiter les agios du jour()` il y a besoin de trois descriptifs.

Le script a la Figure 70ci-dessous montre l'ajout d'un descriptif du type *primitive externe* (cf. le Tableau 2, page 36) au `Compte-Service Equilibre`. Celui-ci s'appelle `Additionner deux nombres` et comporte deux arguments numériques. Le service appelé s'appelle `additionnerA:etB:`, qui prend en entrée ces deux arguments et qui retourne leur somme.

```
CompteServiceEquilibre class >> ajouterDescriptifDeServiceAdditionnerDeuxNombres
      | args aServiceDescription |
      args := ArgumentDescriptionCollection
                  with: (ArgumentDescription name: 'A'  type: Number )
                  with: (ArgumentDescription name: 'B'   type: Number ).

      aServiceDescription := ServiceDescription
            named: 'Additionner deux nombres'
            serviceID: #additionnerA:etB:
            resultType: Number.
      aServiceDescription setArgDescriptions: args.

      uneAdaptation setService: aServiceDescription.
      ^uneAdaptation
```

Figure 70 : Exemple de descriptif de service du type *primitive externe*.

Le script de la Figure 71 montre l'ajout d'un descriptif du type *méthode statique*. Celui-ci s'appelle `Calculer les agios journaliers()` et comporte quatre arguments : un du type compte bancaire et trois autres des nombres. Le service appelé s'appelle `calculerLesAgiosJournaliersDuCompte:soldeActuel:tauxAgios:decouvertForfaitaire:`, qui prend en entrée ces arguments et qui retourne leur le montant des agios journaliers suivant un algorithme approprié.

```
CompteServiceEquilibre                          class                          >>
ajouterDescriptifDeServiceCalculerLesAgiosJournaliers
      | args aServiceDescription |
      args := OrderedCollection new.
      args
      add: (ArgumentDescription name: 'Compte'  type: CompteBancaire);
      add: (ArgumentDescription name: 'Solde actuel'  type: Number);
      add: (ArgumentDescription name: 'Taux d''gios'  type: Number);
      add: (ArgumentDescription name: 'Découvert forfaitaire'  type: Number).
      args := ArgumentDescriptionCollection withArgCollection: args.

      aServiceDescription := self objectFactoryDescriptionClass
            named: 'Calculer les agios journaliers'
            serviceID:
#calculerLesAgiosJournaliersDuCompte:soldeActuel:tauxAgios:decouvertForfaitaire:
            resultType: Number.
      aServiceDescription setArgDescriptions: args.

      uneAdaptation setService: aServiceDescription.
      ^uneAdaptation
```

Figure 71 : Exemple de descriptif de service du type *primitive statique*.

Enfin, le script de la Figure 72 montre l'ajout d'un descriptif du type *méthode*. Celui-ci s'appelle `Obtenir Solde` et comporte un seul argument : le compte bancaire concerné. Le service appelé s'appelle `getBalance`, qui retourne le solde du compte reçu en argument.

```
CompteServiceEquilibre class >> ajouterDescriptifDeServiceObtenirSolde

      | aServiceDescription |
      aServiceDescription := self methodDescriptionClass
            named: 'Obtenir Solde'
            serviceID: #getBalance
            resultType: Number.
      aServiceDescription setArgDescriptions:  (ArgumentDescription  name:  'Compte'
type: CompteBancaire).

      uneAdaptation setService: aServiceDescription.
      ^uneAdaptation
```

Figure 72 : Exemple de descriptif de service du type *méthode*.

3.6 Définir des procédures

Notre expert peut à présent procéder à la composition de ses procédures. La première s'appelle `Cumuler les agios du jour`. Sa composition fait intervenir l'outillage présenté dans le paragraphe 3.1, page 62 du chapitre I.

```
CompteServiceEquilibre class >> ajouterProcedureCumulerLesAgiosDuJour
      "AM.BClass.CompteBancaire procedureCumulerLesAgiosDuJour"

      | composition leCompte leMontant cumulAgios somme |
      composition := ListMicroCompositionComponent default.
      leCompte := composition addInPin: 'Le Compte'.
      leMontant := composition addInPin: 'Le montant'.
      cumulAgios := composition
                        addGetter: 'Obtenir Cumul d''agios'
                        of: self
                        with: leCompte.
      somme := composition
                        addExternalPrimitive: 'Additionner deux nombres'
                        of: self
                        with: leMontant
                        with: cumulAgios.
      composition
            addSetter: 'Affecter Cumul d''agios:'
            of: self
            with: (OrderedCollection with: leCompte with: somme).
      self
            addProcess: composition
            named: 'Cumuler les agios du jour'.
      ^composition
```

Figure 73 : Exemple de composition dynamique de procédure.

3.7 Composer dynamiquement des procédures définies à l'exécution

3.7.1 Habiller

Notre expert souhaite utiliser la procédure qu'il vient de définir comme une macro-procédure. Suivant la démarche que nous avons décrite dans le paragraphe 2.4.2, page 61 du chapitre I, il faut d'abord habiller cette procédure à l'aide d'un descriptif de service.

La Figure 74 ci-dessous montre l'application de l'outillage que nous avons présenté au paragraphe 3.5.2, page 74 du chapitre I afin de réaliser cet habillage. Il s'agit de rappeler la procédure concernée et de lui envoyer le message `asServiceDescriptionNamed:`. Une meilleure solution serait de stocker le nom de la procédure en son sein pour éviter ce passage de paramètre.

Cela montre, toutefois, que la génération de ces descriptifs de service peut être entièrement automatisée.

```
CompteServiceEquilibre                          class                          >>
habillerProcedureCumulerLesAgiosDuJourEnDescriptifDeProcedure
      "Préparer l'usage de la procédure Cumuler les agios du jour dans la
composition d'autres procédures"
      | uneProcedure aServiceDescription |

      uneProcedure := self getProcessNamed: 'Cumuler les agios du jour'.
      aServiceDescription := uneProcedure asServiceDescriptionNamed: 'Cumuler les
agios du jour'.
      self setService: aServiceDescription.
      ^aServiceDescription
```

Figure 74 : Exemple d'habillage d'une procédure et création d'une macro-procédure.

3.7.2 Appeler une sous-procédure

Nous disposons à présent de tout le matériel nécessaire à la démonstration de la technique d'appel de sous-procédures. La Figure 75 ci-dessous montre l'usage de l'outillage dédié à la composition et toujours présenté dans le paragraphe 3.1, page 62 du chapitre I.

Plus particulièrement il s'agit d'utiliser le protocole de création d'instances de descriptifs de service présenté par le Tableau 6, page 62. Une méthode qui nous intéresse en particulier, et `addProcedure:of:with:`. C'est elle qui intègre l'appel de la sous-procédure dans la composition en cours.

Il est important de noter ici l'homogénéité obtenue au sujet du mode d'emploi de la composition de procédures. Nous avons ainsi également rempli notre engagement de considérer dans la conception de notre outillage la *facilité d'apprentissage.*

```
CompteServiceEquilibre class >> ajouterProcedureTraiterLesAgiosDuJour

	|	composition	leCompte	leCompteCheque	soldeCompteCheque	leTauxAgios
montantDecouvert agiosJour |
	composition := ListMicroCompositionComponent default.
	leCompte := composition addInPin: 'Le Compte'.
	leCompteCheque := composition
				addGetter: 'Obtenir Compte-chèque associé'
				of: self
				with: leCompte.
	soldeCompteCheque := composition
				addMethode: 'Obtenir Solde'
				of: self
				with: leCompteCheque.
	leTauxAgios := composition
				addGetter: 'Obtenir Taux préférentiel de calcul d''agios'
				of: self
				with: leCompte.
	montantDecouvert := composition
				addGetter: 'Obtenir Montant de découvert forfaitaire'
				of: self
				with: leCompte.
	agiosJour := composition
				addStatiqueMethode: 'Calculer les agios journaliers'
				of: self
				with:	(OrderedCollection	with:	leCompte	with:
soldeCompteCheque with: leTauxAgios with: montantDecouvert).
	composition
		addProcedure: 'Cumuler les agios du jour'
		of: self
		with: (OrderedCollection with: leCompte with: agiosJour).

	self
		addProcess: composition
		named: 'Traiter les agios du jour'.
	^composition
```

Figure 75 : Exemple de procédure avec appel de sous-procédure.

3.8 Instancier les structures et activer les procédures

La Figure 76 ci-dessous montre comment il est possible d'instancier les nouveaux types d'objets, ici le Comptes-Servie et de leur appliquer les procédures définies dynamiquement. Cette mise en œuvre fait intervenir les outillage relatifs chacun de ces deux aspects présentés dans les paragraphes 2.2, 2.4.4, 3.3, 3.5.4, 2.1 et 2.2.2 respectivement pages 59, 61, 67, 75, 110 et 122. Ces paragraphes couvrent aussi bien l'instanciation des adaptation et l'activation des procédures.

```
CompteServiceEquilibre class >> exemple05
      | typeCompteServiceEquilibre unCompteService unCompteCheque |
      typeCompteServiceEquilibre := self exemple04.
      unCompteService := typeCompteServiceEquilibre new.
      unCompteCheque := typeCompteServiceEquilibre new. "Afin de simplifier"
      unCompteService
            setProperty: 'balance' to: 5000;
            setValue: 0 toPropertyNamed: 'Cumul d''agios';
            setValue: #Visa toPropertyNamed: 'Type de carte bancaire associé';
            setValue: unCompteCheque toPropertyNamed: 'Compte-chèque associé';
            setValue: 0.4 toPropertyNamed: 'Taux préférentiel de calcul d''agios';
            setValue: 3000 toPropertyNamed: 'Montant de découvert forfaitaire'.
      unCompteService getType
            run: 'Traiter les agios du jour'
            in: nil
            initialContext: (IdentityDictionary with: #'1@1' -> unCompteService).
      unCompteService getType
            run: 'Traiter les agios du jour'
            in: nil
            initialContext: (IdentityDictionary with: #'1@1' -> unCompteService).
```

Figure 76 : Exemple d'instanciation d'adaptations et de l'activation de procédures.

La Figure 77 ci-dessous illustre le résultat des deux exécutions successives de la procédure `Traiter les agios du jour`, suivant le script de la Figure 76. Ces deux exécutions montrent, par la différence de la valeur de l'attribut `Cumul d'agios` (0, puis 28, puis 56) que le montant des agios a bien été calculé et affecté à cet attribut dynamique.

```
"Compte-Service Equilibre"
 has the following property types:
      -Property type named: "Cumul d'agios"
      -Property type named: "Type de carte bancaire associé"
      -Property type named: "Compte-chèque associé"
      -Property type named: "Taux préférentiel de calcul d'agios"
      -Property type named: "Montant de découvert forfaitaire"
My property values are as follows:
      -The Property named "Cumul d'agios" is equal to 28
      -The Property named "Type de carte bancaire associé" is equal to Visa
      -The Property named "Compte-chèque associé" is equal to a
R*AM.BClass.CompteBancaire16 c/o
      -The Property named "Taux préférentiel de calcul d'agios" is equal to 0.4
      -The Property named "Montant de découvert forfaitaire" is equal to 3000
"Compte-Service Equilibre"
 has the following property types:
      -Property type named: "Cumul d'agios"
      -Property type named: "Type de carte bancaire associé"
      -Property type named: "Compte-chèque associé"
      -Property type named: "Taux préférentiel de calcul d'agios"
      -Property type named: "Montant de découvert forfaitaire"
My property values are as follows:
      -The Property named "Cumul d'agios" is equal to 56
      -The Property named "Type de carte bancaire associé" is equal to Visa
      -The Property named "Compte-chèque associé" is equal to a
R*AM.BClass.CompteBancaire17 c/o
      -The Property named "Taux préférentiel de calcul d'agios" is equal to 0.4
      -The Property named "Montant de découvert forfaitaire" is equal to 3000
```

Figure 77 : Résultat d'exécution du script de la Figure 76.

3.9 Edition des adaptations

Etant donné que dans le cas de MIDYCTALK les adaptations et les spécialisations sont de même natures, alors lorsque l'expert ajoute une nouvelle adaptation, e.g. le `Compte-Service Equilibre`, le langage d'expert crée en réalité une nouvelle classe. De ce fait, les programmeurs peuvent les éditer à travers leurs outils habituels.

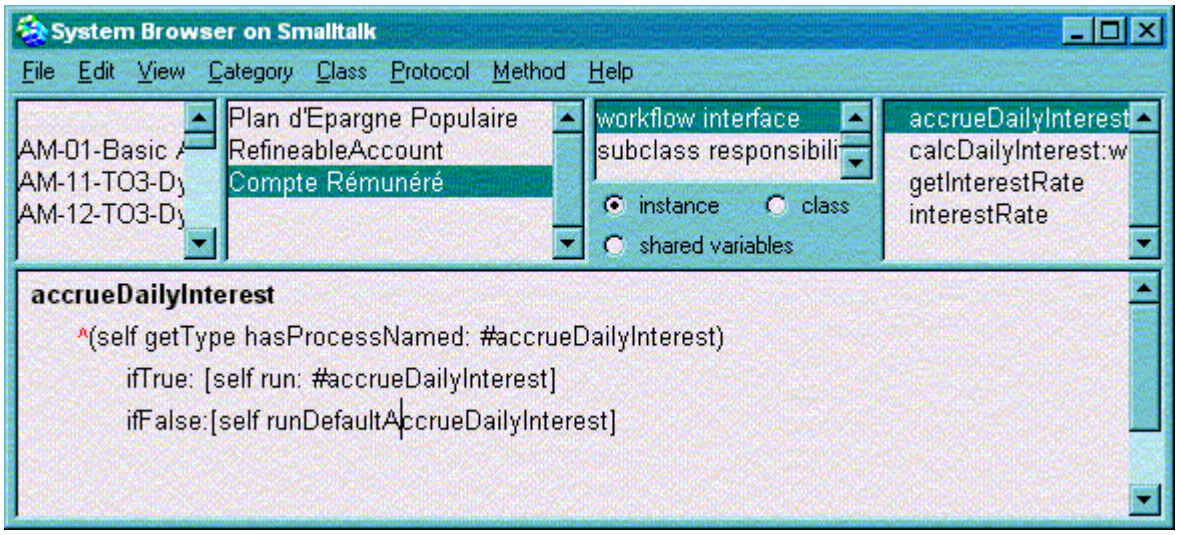


Figure 78 : MIDYCTALK permet aux programmeurs de collaborer avec les experts.

La Figure 78 ci-dessus montre l'édition d'une adaptation appelée `Compte Rémunéré` par un programmeur SMALLTALK à travers le flâneur (*browser*) du système VISUALWORKS NC 5.i3 [Cin01]. Nous avo,ns fourni une autre cliché par la Figure 63, 161ci-dessus.

3.10 *Refactoring* des adaptations

L'objet du *refactoring* est de réorganiser et améliorer le code d'un système sans remettre en cause ses fonctions [Opd92, Rob99]. De notre point de vue, l'application de cette technique conserve tout à fait son intérêt dans le cas des langages d'experts. Cela permet en effet, aux programmeurs comme les professionnels garants du bon fonctionnement du système, son intégrité et sa maintenabilité à superviser et le cas échéant modifier le modèle objet issu des interventions des experts.

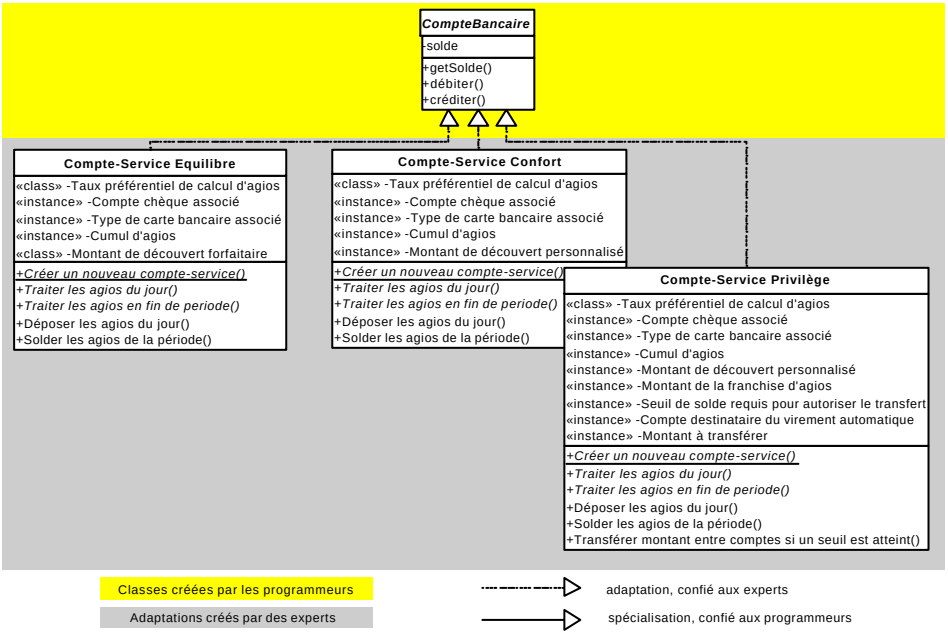


Figure 79 : Modèle objet après la définition par l'expert des trois Comptes-Service.

La nouvelle implantation réflexive de notre modèle de spécialisation dynamique, DYCRA, *rend possible* la création d'outils dédiés au *refactoring* des adaptations.

Cette nouvelle possibilité permet, à titre d'exemple, aux programmeurs de transformer le modèle objet de la Figure 79 ci-dessus, issu de l'ajout par des experts des trois types de Comptes-Service, vers le modèle de la Figure 80 ci-dessous.

En effet, les experts ne sont pas supposés être familier avec des notions comme classes abstraite (ici `Compte-Service Client Fiable`) et donc reproduisent (copient) la définition des attributs et procédures au niveau de chaque adaptation.

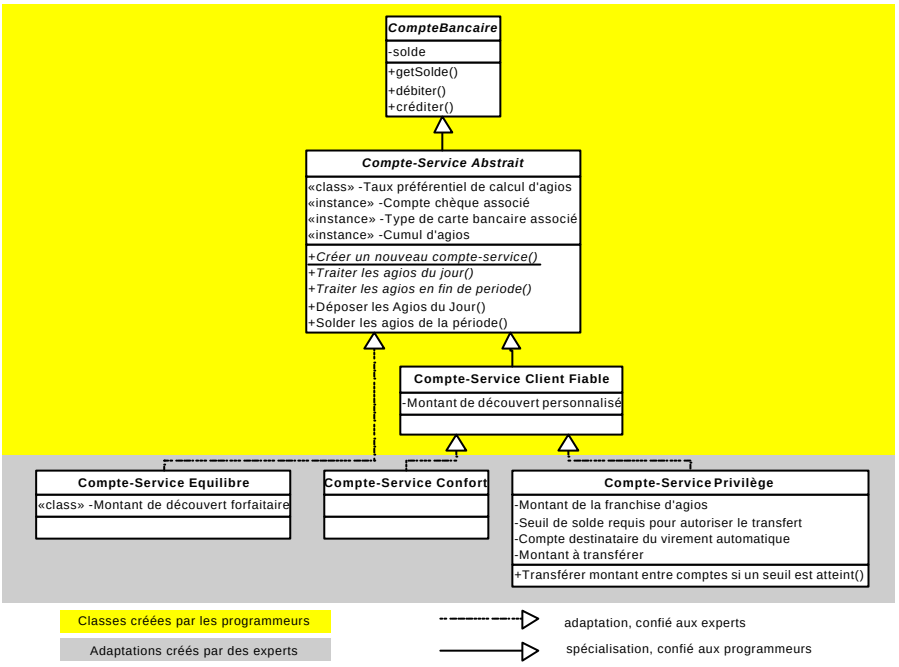


Figure 80 : Modèle objet de Figure 79 après le *refactoring* par des programmeurs.

3.11 Effort conjugué pour la création de *workflow* adaptatifs

Un apport majeur de cette implantation à l'aide de la réflexion est la réhabilitation entière de la démarche proposée par D. Manolescu dans le nouveau contexte des modèles objets adaptatifs.

En effet, ce dernier considère que la mise en œuvre réussit de workflows requiert la possibilité d'une intervention de plus bas niveau (au niveau des objets et non pas d'application). Il propose alors le Micro-workflow comme outil dédié à ce type de mise en œuvre. Celui-ci considère que les collaborations entre les objets seront codées sous forme de méthodes, mais dans la syntaxe du Micro-workflow. Autrement dit, chaque méthode correspond à un script dont l'exécution retourne un objet du type procédure (au sens du Micro-workflow [Man00]).

Comme nous l'avons décrit au début de ce chapitre, notre premier outillage de l'adaptation à travers le système DYCTALK ne permet pas de mettre en œuvre ce procédé. En effet, dans la mesure où dans cette implantation ce sont des instances terminales qui jouent le rôle de classes, il n'est pas possible de les éditer comme une classe.

Cette mise en œuvre réflexive rend notre outillage *compatible* avec le Micro-workflow. En effet, à présent les programmeurs peuvent ajouter aux adaptations des procédés au sens du Micro-workflow. En raison de cette compatibilité, le Micro-workflow qui est à l'origine conçu pour les programmeurs, se trouve très avantageusement également accessible aux experts et cela dans le contexte des modèles objets adaptatifs.

Plus concrètement, supposons que tout compte rémunéré, une adaptation de la classe `RefineableSavingsAccount`, doit répondre au message `accrueDailyInterest`. Celui-ci calcule le montant des intérêts journaliers et ajoute la somme calculée au solde du compte. De toute évidence, la méthode de calcul n'est pas identique d'un type de compte rémunéré à un autre. Aussi, comme l'illustre la Figure 81 ci-dessous, les programmeurs prévoient deux cas :

1. une implantation par défaut de ce calcul (méthode `defaultAccrueDailyInterestProcess`). Celle-ci est utilisée si les experts ne procèdent pas à la définition d'un micro-procédé plus adapté. Le micro-procédé associé est déclenché à l'aide de la méthode `execute.`
2. une possibilité pour la prise en compte prioritaire du micro-procédé défini par l'expert. Celui-ci doit ici s'appeler `accrueDailyInterest`. Celle-ci est activée par l'intermédiaire de la méthode `run:`.

```
RefineableSavingsAccount >> accrueDailyInterest
      ^(self getType hasProcessNamed: #accrueDailyInterest)
            ifTrue: [self run: #accrueDailyInterest]
            ifFalse:[self runDefaultAccrueDailyInterest]

RefineableSavingsAccount >> runDefaultAccrueDailyInterest
      ^self defaultAccrueDailyInterestProcess execute
```

Figure 81 : Personnalisation du calcul des intérêts journalier à l'aide des micro-procédés.

Notons que, comme le décrit plus en détail D. Manolescu [Man00], l'ajout de cet algorithme sous forme d'un micro-procédé a l'avantage d'offrir plus de souplesse dans le changement de la logique applicative et des règles métier. Cela s'inscrit dans une logique de système de workflow adaptative [MJ99b].

De plus, le fait d'avoir écrit le micro-procédé sous forme d'une méthode permet le transfert aisé de données (ici `self`) entre le contexte d'exécution du langage d'implantation (ici SMALLTALK-80) et le Micro-workflow. En effet, ici l'instance courante du compte rémunéré s'ajoute au contexte global du micro-procédé comme étant le compte courant (clé `myAccount`).

Nous contribuons ainsi à l'extension du Micro-workflow vers un usage effectif dans le contexte des AOMs et cela par des experts.

3.12 Implantation de primitives

Les programmeurs peuvent aussi à présent intervenir au sein des adaptations afin d'ajouter des méthodes primitives. Rappelons que de telles primitives servent à la création de descriptifs de service.

Par exemple, si l'on reconsidère le concept de compte rémunéré, les programmeurs peuvent implanter les méthodes telles que `getInterestRate` et `calcDailyInterest:with:` dans la classe `RefineableSavingsAccount`. De telles primitives servent, à titre d'exemple, dans l'écriture de micro-procédés tel que `defaultAccrueDailyInterestProcess`.

4 Conclusion : apports du framework MIDYCTALK

Ce chapitre a principalement été consacré à l'étude des problèmes posés par l'outillage de l'adaptation basée sur les techniques standard (DOM des AOMs, cf.[RTJ00]), que nous avons mise en œuvre lors des chapitres précédents.

Afin de proposer une solution appropriée à ces problèmes, nous avons alors considéré l'usage de la Réflexion [Smi84, Maes87, Pit90, Kic92, Kic94, Pit95] et plus particulièrement des méta-classes standard du langage SMALLTALK-80.

4.1 Pas décisif vers la validation définitive de notre thèse

Les travaux présentés dans ce chapitre constituent une étape intermédiaire vers la validation complète de notre thèse par l'outillage d'une propriété importante des langages d'experts, c'est à dire le travail collaboratif.

Ces travaux montrent ainsi essentiellement l'intérêt du rapprochement de la représentation des adaptations à celle des spécialisations. Cette direction sera poursuivie, lors du chapitre suivant, en s'intéressant cette fois à un cas très particulier de la mise ne œuvre de ce rapprochement à travers le système METACLASSTALK.

La solution obtenue par ce rapprochement s'appuie sur les méta-classes et nous procure les avantages suivants :

1. une adaptation C^r de la classe/adaptation C est à présent de même nature qu'une classe. Cela permet l'usage par les programmeurs de leurs outils habituels pour éditer les adaptations.
2. la compréhension des adaptations par les programmeurs est plus aisée en raison de leur ressemblance aux classes.
3. il n'y a plus de nécessité pour gérer des référentiels *ad-hocs* pour stocker les adaptations. Cette tâche est à présent pris en charge par le système, au même titre que les classes.

Ces travaux conduisent à la création d'une nouvelle version du framework orienté-objet DYCTALK, appelé MIDYCTALK. Notre nouveau framework outille, tout comme son prédécesseur :

1. l'adaptation du nom d'une classe.
2. l'adaptation de la structure par ajout de descriptifs d'attributs.
3. l'adaptation du comportement d'une classe par l'ajout de procédures.
4. la composition dynamique de procédures définies dynamiquement

Il fournit, de plus, des services d'aide à la mise en œuvre pratique de l'adaptation :

1. le *refactoring* des adaptations
2. l'ajout des des méthodes qui codent des micro-procédés (cf. Manolescu [Man00]).
3. l'ajout des des méthodes qui implantent des primitives.
4. l'ajout des des variables d'instances et des variables d'instance de classe.
5. la gestion propre à chaque adaptation d'un référentiel de descriptif de services.

L'apport principal du framework MIDYCTALK par rapport à notre thèse (cf. l'introduction, le paragraphe 1.5.2) est de montrer la relation entre les méta-classes et l'outillage de l'adaptation. En effet, MIDYCTALK utilise les méta-classes comme le point d'articulation pour la mise en œuvre du rapprochement de la représentation des spécialisations et celle des adaptations.

Comme le précise de nombreux auteurs et notamment Briot & al. [BGL98], grâce à la puissance de la réflexion nous avons pu relativement facilement procéder à l'adaptation du langage SMALLTALK-80 aux besoins particuliers de l'outillage de l'adaptation.

MIDYCTALK constitue, toutefois, uniquement une étape intermédiaire vers la validation de la seconde et la dernière partie de notre thèse (cf. ci-dessous, §9.2). En effet, il n'est pas encore une solution tout à fait satisfaisante et présente une certaine rigidité.

Ce sujet sera exploré dans le chapitre suivant, par l'usage du système METACLASSTALK qui met en œuvre une politique différente en matière du choix de la méta-classe. Ce système devait nous permettre de résoudre la dernière série des problèmes de base que nous avons pu recenser jusqu'alors.

4.2 Résultat complémentaire : *la réalité n'existe qu'à travers des relations*

L'un des apports indirects de ces travaux, et sur le plan personnel le plus important, consiste à fournir un exemple concret d'implantation d'un système aussi bien au "niveau base" qu'au "niveau méta" et de nous permettre ainsi de conclure sur la *non existence d'une réalité absolue.*

En effet, on peut ici observer que le matériel textuel du code de nos deux implantations DYCTALK et MIDYCTALK reste le même. Nous avons les mêmes structures de données et les mêmes méthodes qui mettent ensemble en place un mécanisme qui interprète les jeux d'instanciation d'objets issus de ce modèle comme des adaptations.

Le changement réside dans sa distribution (via le browser) entre classe et méta-classe et donc dans la nature de *relation* établie entre ce texte et le langage d'implantation (ici SMALLTALK).

Aussi, la distinction entre les niveaux dits "base" et "méta" et purement d'ordre contextuel et interprétatif. En effet, ici le même texte intégré au niveau des "classes" engendre un effet (création d'instances terminales) tout à fait différent par rapport au même texte intégré au niveau des "méta-classes" (création de classes).

Ce ne sont véritablement que des effets produits par une certaine interprétation qui n'a de sens que du point de vue des individus engagés dans les relations en question.

Quel est alors la réalité du code écrit ?[116]

[116] Cela dit, comment peut-on décrire la non existence de la réalité par des outils qui deviennent alors inexistants (en l'occurrence l'écriture) ? Mes cher lecteurs associeront aux traces visiblement noir de l'objet visiblement papier qu'ils auront en face, une sémantique qui est propre au type de *relation* qu'ils auront établi avec l'écriture, les modèles de pensé, etc.

Chapitre V : Troisième outillage de l'adaptation

Usage de méta-classes explicites (MxDyctalk)

Chapitre V : Troisième outillage de l'adaptation

- Usage de méta-classes explicites (MXDYCTALK)

1 Introduction

Lors des précédents chapitres nous avons montré l'importance du *statut* vis à vis du langage d'implantation de la représentation des adaptations. En effet, nous avons jusqu'alors exploré deux statuts possibles pour les adaptations.

Le premier cas est basé sur l'application des techniques standards DOM [RTJ00] et du Micro-workflow [Man00]. Celles-ci conduisent à interpréter une instance terminale comme une classe (le framework DYCTALK, le chapitre III). Cette solution est assez critiquable, en particulier en ce qui concerne le travail collaboratif entre les programmeurs et les experts ainsi que le choix local du type d'adaptation.

Le second cas est basé sur l'usage des facilités réflexives du langage SMALLTALK-80 qui permettent aux adaptations de disposer du même statut que celui des classes. Ce choix conduit alors à de meilleurs résultats car il rapproche la représentation des adaptations à celle des classes (chapitre IV). Le framework MIDYCTALK issu de ce travail permet l'adaptation dynamique et collaborative de classes et résout plusieurs problèmes posés par l'approche classique de l'outillage de l'adaptation.

Par ailleurs, il confirme à nouveau l'importance de la présence des méta-classes dans un langage à objet.

Cette seconde implantation peut néanmoins être aussi critiquée, en particulier en ce qui concerne sa "rigidité" lors de l'usage. Nous évoquons ici dans un premier temps plusieurs cas de telles situations. Ensuite, nous montrons en quoi et pourquoi notre implantation actuelle est inapte pour apporter une réponse satisfaisante à ces interrogations. Cette analyse nous permet de diagnostiquer le problème qui se situe au niveau de la mise en œuvre limitée des méta-classes dans le cas du langage SMALLTALK-80. Nous proposons alors une solution à ce problème basée sur l'usage des méta-classes explicites et le système METACLASSTALK. Enfin, nous montrons comment cette solution peut être mise en œuvre et dans quelle mesure elle répond aux questions posées. Nous terminons ce chapitre sur des conclusions, ainsi que des perspectives de recherche sur ce sujet.

1.1 Limites de MIDYCTALK : manque du choix local du type d'adaptation

Il existe de nombreuses situations où le framework MIDYCTALK[117] ne peut pas apporter une réponse satisfaisante.

Tout d'abord, comme permet de l'illustrer la Figure 82 ci-dessous, MIDYCTALK ne permet pas de rendre une classe adaptable, indépendamment de sa situation aux sein des hiérarchies de classes. En effet, dans ce cas une classe devient adaptable uniquement par l'héritage d'une des classes dont la méta-classe met en œuvre un certain type d'adaptation. Autrement dit, il n'est pas possible de choisir localement le type d'adaptation.

Par exemple, la classe `SavingsAccount` peut devenir adaptable si elle hérite de la classe `Prototype` dont la méta-classe `Prototype class` outille l'adaptation prototypique. Cela conduit alors à la perte de l'héritage "naturel". Ici, `SavingsAccount` ne peut pas hériter de sa super-classe "naturelle", la classe `BankObject` (par l'intermédiaire de la classe `AccountObject`).

Deuxièmement, MIDYCTALK ne permet pas le choix libre des classes adaptables d'un système. En effet, dès qu'une classe devient adaptable (par la technique d'héritage décrite dans le paragraphe ci-dessus), alors toutes ses sous-classes deviennent également adaptables. Autrement dit, MIDYCTALK propage le choix relatif au type d'adaptation d'une classe à toutes ses sous-classes.

Ici, toutes les sous-classes de la classe `SavingsAccount` seront adaptables. De plus, comme l'illustre à nouveau la Figure 82, le type d'adaptation de toutes ces classes est l'adaptation prototypique.

Troisièmement, MIDYCTALK ne permet pas de revenir dynamiquement (et de façon contrôlée) sur le choix du type d'adaptation. En effet, celui-ci est fixé par le choix initial de la super-classe (réalisé par des programmeurs) et n'est pas conçu pour une modification lors de l'exécution.

Ici, le type d'adaptation de la classe `SavingsAccount` et ses sous-classes est l'adaptation prototypique et le restera sauf par l'intervention des programmeurs et une révision de la conception du système. Les nouveaux choix propageront également leurs effets sur toute la hiérarchie.

[117] Le système DYCTALK présente également des problèmes similaires, mais pas tout à fait identiques. Dans la mesure où nous avons déjà montré de nombreux autres problèmes avec cette solution (cf. chapitre IV, section 1.1, page 146), nous ne le considérons plus et poursuivons ici notre analyse seulement dans le cas du framewok MIDYCTALK, le successeur de DYCTALK.

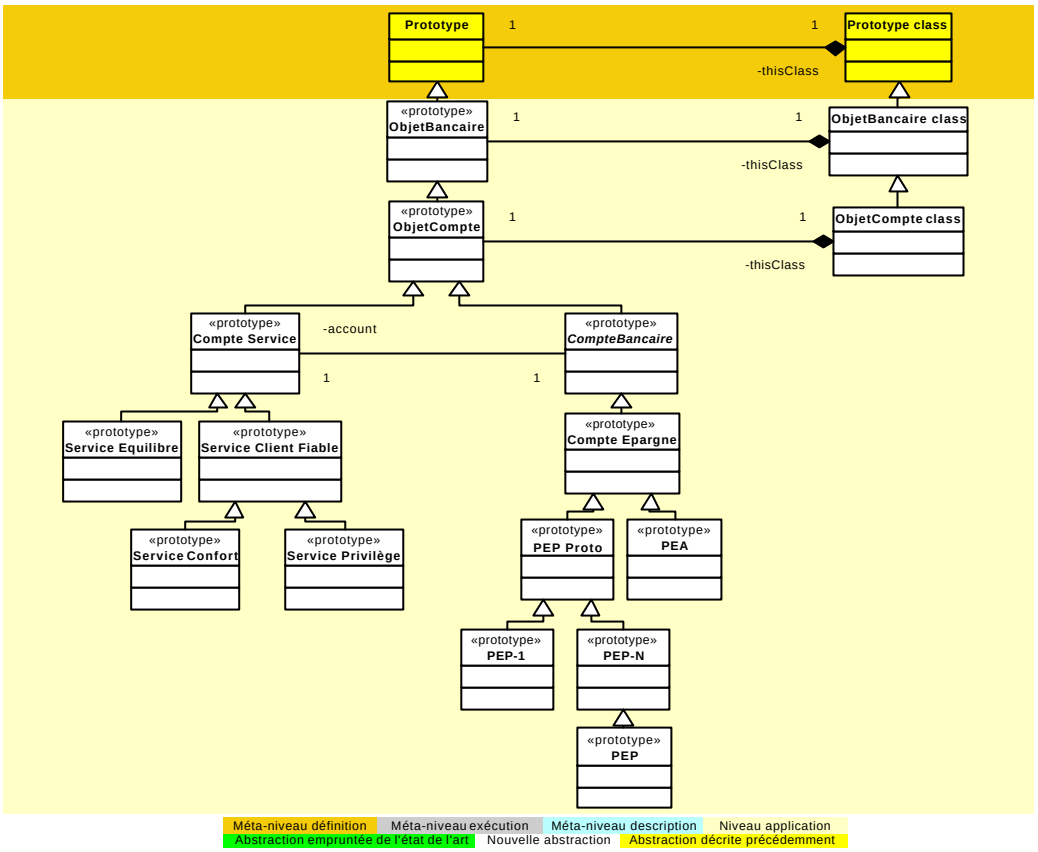


Figure 82 : Rigidité de MIDYCTALK concernant le choix du type d'adaptation.

1.2 Critique de l'approche SMALLTALK-80

SMALLTAK-80 a la particularité de prendre en charge la création et la suppression automatique de méta-classes. Cette technique permet de rendre les méta-classes avantageusement "invisibles" aux yeux des programmeurs SMALLTALK, tout en conservant leurs avantages pratiques, comme les méthodes et les variables d'instances de classe.

Ces choix de SMALLTALK-80 qui visent à privilégier les programmeurs, se trouvent limitatifs en ce qui concerne l'outillage de l'adaptation[118]. Rappelons, en effet, que le fonctionnement du système SMALLTALK-80 en ce qui concerne les méta-classes suit les deux principes suivants :

1. le choix implicite de la nature des classes (de leur méta-classe) par le système SMALLTALK-80 lui-même (c'est une instance de la classe `Metaclass` et sous-classe de la méta-classe de la super-classe d'une classe).
2. la propagation de caractéristiques des classes par l'héritage.
3. le choix du type d'adaptation basé sur l'héritage, et donc seulement accessible aux programmeurs.

Ce sont ces choix qui expliquent, en effet, la situation problématique décrite dans la sous-section précédente.

[118] A ce propos, Jaques Malenfant constate que 'SMALLTALK n'a pas été conçu avec l'objectif de rendre cette organisation (réflexive) si manifeste qu'elle inciterait les utilisateurs à l'utiliser et à la modifier... " [Mal97, page 81]. Le travail communiqué dans ce mémoire donne un exemple de l'intérêt de techniques génie logiciel pour documenter ce types d'architectures réflexives et complexes. Cette approche par définition vise à inciter les programmeurs à "l'utiliser et à la modifier".

1.3 Solution basée sur l'usage des Propriétés de Classes

Les méta-classes sont des réifications qui servent à faire varier la sémantique par défaut des classes. Chaque variation ainsi mise en œuvre constitue une *propriété de classe* [LC96]. Noury Bouraqadi précise que "dans certains systèmes ou langages comme CLOS [Kee89], CLASSTALK [BC89, Coi90, Coi93], SOM [FCDR95] ou NEOCLASSTALK [Riv97], les méta-classes sont explicites et peuvent être manipulées par les programmeurs notamment pour définir des *propriétés de classes réutilisables*." [Bou99, page 58]

La règle proposée par N. Bouaqadi est la suivante : "*afin d'attribuer une propriété* **P** *à une classe* **C**, **C** *doit être défini comme instance de la méta-classe* **Mp** *qui défini la propriété* **P** ." [Bou99, page 76]

Toutes ces bonnes caractéristiques laissent penser que si l'on considère *l'adaptabilité comme une propriété de classe il sera alors possible de résoudre les problèmes mentionnés ci-dessus.* C'est cette hypothèse que nous allons vérifier à présent.

Avant d'enter dans le vif du sujet, nous donnons ici un rapide aperçu du système METACLASSTALK. La section 4.2, page 196 de ce chapitre ainsi que la section 2.2, page 213 sur les perspectives fournissent également un complément d'information sur ce système.

1.4 Le système METACLASSTALK de N. Bouraqadi

Le système METACLASSTALK est une extension réflexive de SMALLTALK-80. Il résulte d'une succession de travaux dont le but est l'étude des méta-classes et leur utilisation pour définir des propriétés de classes. "METACLASSTALK reprend les méta-classes explicites de son ancêtre CLASSTALK de Pierre Cointe [BC89] et hérite des possibilités de changement dynamique de classe et de contrôle de l'application des méthodes de son ascendant direct NEOCLASSTALK de Fred Rivard [Riv97].

A ces caractéristiques héritées, METACLASSTALK ajoute un MOP permettant le contrôle des structures et des comportements des objets. Ce contrôle consiste à décrire explicitement les mécanismes d'envoi de messages, de construction des instances et d'accès à leur structure. Ces mécanismes décrits dans les méta-classes permettent de définir diverses propriétés de classe, moyennant la résolution de problèmes de la *composition* [MMC95, BS99]et de la *compatibilité* [Gra89, BSLR98] des méta-classes. " [Bou99, page 10]

Le système METACLASSTALK est par ailleurs, un descendant du système NEOCLASSTALK [Riv96d]. Ce dernier hérite de SMALLTALK la syntaxe et la philosophie réflexive, de CLASSTALK [Coi90] le noyau des classes et de MOOSTRAP [MDC92, MC93] la réification (de l'application) des envois de message. A cela, NEOCLASSTALK ajoute la mise en œuvre du protocole de changement dynamique de classe ainsi que celui de changement dynamique de superclasse. NEOCLASSTALK s'appuie sur la machine virtuelle SMALLTALK-80 en tant que support d'exécution.

METACLASSTALK se présente actuellement sous forme d'une extension de la version 2.7 du système SQUEAK [IKMWK97]. Celui-ci est le dernier descendant en date du système SMALLTALK-80.

Il est important de noter que nous n'avons pas pu pousser plus loin notre étude sur ce sujet en raison du portage en cours du système METACLASSTALK de sa plate-forme initiale VISUALWORKS 2.5 (qui est à présent obsolète) sur le nouveau système SQUEAK. Le système actuellement disponible n'offre pas encore la composition des méta-classes. C'est elle qui nous intéresse tout particulièrement dans le cadre de l'exposé du paragraphe 4.2, page 196 du même chapitre.

2 Mise en œuvre réflexive à l'aide du langage METACLASSTALK

Dans cette section nous exposons une nouvelle implantation du système DYCRA. Celle-ci correspond à une adaptation à l'univers de méta-classes explicites et le système METACLASSTALK de l'implantation en SMALLTALK-80 de DYCRA (le framework MIDYCTALK).

Nous appelons MXDYCTALK le framework issu de ce nouveau travail. Celui-ci comporte une implantation complète des deux systèmes DARC et DART. Cette implantation est, toutefois, en grande partie identique à celle du framework MIDYCTALK. En effet, ce qui change ici est uniquement la partie de l'implantation de ces deux systèmes qui détermine le statut des adaptations vis à vis du langage d'implantation. Plus précisément, c'est l'implantation de la hiérarchie des méta-classes dont la racine est `AbstractRefinment class` qui change.

Afin d'offrir un exposé plus synthétique et dans la mesure où nous avons déjà détaillé l'implantation de ce système dans le cas du framework MIDYCTALK (cf. le chapitre IV), nous décrirons ici uniquement le différentiel entre ces deux implantations sous forme des règles de portage de MIDYCTALK vers METACLASSTALK.

Nous fournissons également le code source complet de ce système à l'URL suivante : *http://www-poleia.lip6.fr/~razavi/xps*.

2.1 Outiller le choix explicite du type d'adaptation

La création d'une adaptation s'appuie aussi bien dans le cas de MIDYCTALK que celui de MXDYCTALK sur les mécanismes de création de classes par instanciation de méta-classes. Toutefois, ici nous avons, grâce au système METACLASSTALK, l'avantage de pouvoir spécifier la nature d'une adaptation en fournissant *explicitement*, lors de sa création, la méta-classe qui met en œuvre la fonctionnalité souhaitée.

C'est cette possibilité qui permet d'outiller le choix local du type d'adaptabilité. Nous l'avons concrétisé dans notre système par l'ajout d'un argument supplémentaire à la méthode de définition de nouvelles adaptations, `refinement:publicName:`, qui devient alors `refinement:publicName:metaclass:`.

Rappelons que cette méthode crée une nouvelle classe qui est sous-classe du receveur de ce message. Le nom système (nom habituellement fourni par des programmeurs) est soit passé en argument, soit calculé à partir du nom de la super-classe[119]. Le second argument correspond au nom de l'adaptation tel qu'il est connu des experts ("nom métier").

L'ajout de ce nouvel argument permet le choix explicite du type d'adaptation par la fourniture de la méta-classe appropriée[120]. En l'absence de valeur pour cet argument, c'est la méta-classe de la super-classe qui est utilisée.

La section 3.1, page 189 ci-dessous fournit des exemples d'utilisation de cette nouvelle technique de création des adaptations.

[119] Ce calcul est confié à la méthode `computeNextPrivateRefinementNameWith:` de la méta-classe `AbstractRefinementClass`. Il consiste à construire une chaîne de caractères composée du nom de la super-classe et d'un nombre entier. Celui-ci est incrémenté jusqu'à ce que la chaîne de caractères issue de la concaténation ne correspond plus au nom d'une classe existante dans le système.

[120] De toute évidence, les experts ne se réfèrent pas aux méta-classes directement. Ces dernières sont "habillées" par des programmeurs et proposées aux experts suivant une terminologie qui leur est familière.

2.2 D'autres apports notables des méta-classes "explicites"

La gestion explicite des méta-classes par le langage METACLASSTALK conduit à trois autres différences notables entre les deux systèmes MIDYCTALK et MXDYCTALK.

2.2.1 Nommage des méta-classes

Comme permet de le constater la Figure 83, le premier élément qui distingue MXDYCTALK de MIDYCTALK est le fait que le nom des méta-classes n'est plus déduit de leur instance unique.

Rappelons que dans le cas de SMALLTALK-80, et par conséquence le framework MIDYCTALK, le nom d'une méta-classe est obtenu par l'ajout du suffixe `class` au nom de son instance unique qui la représente "officiellement" dans le système. C'est, à titre d'exemple, pour cela que la méta-classe qui implante le modèle des adaptations structurelles s'appelle `StructuralRefinement class`.

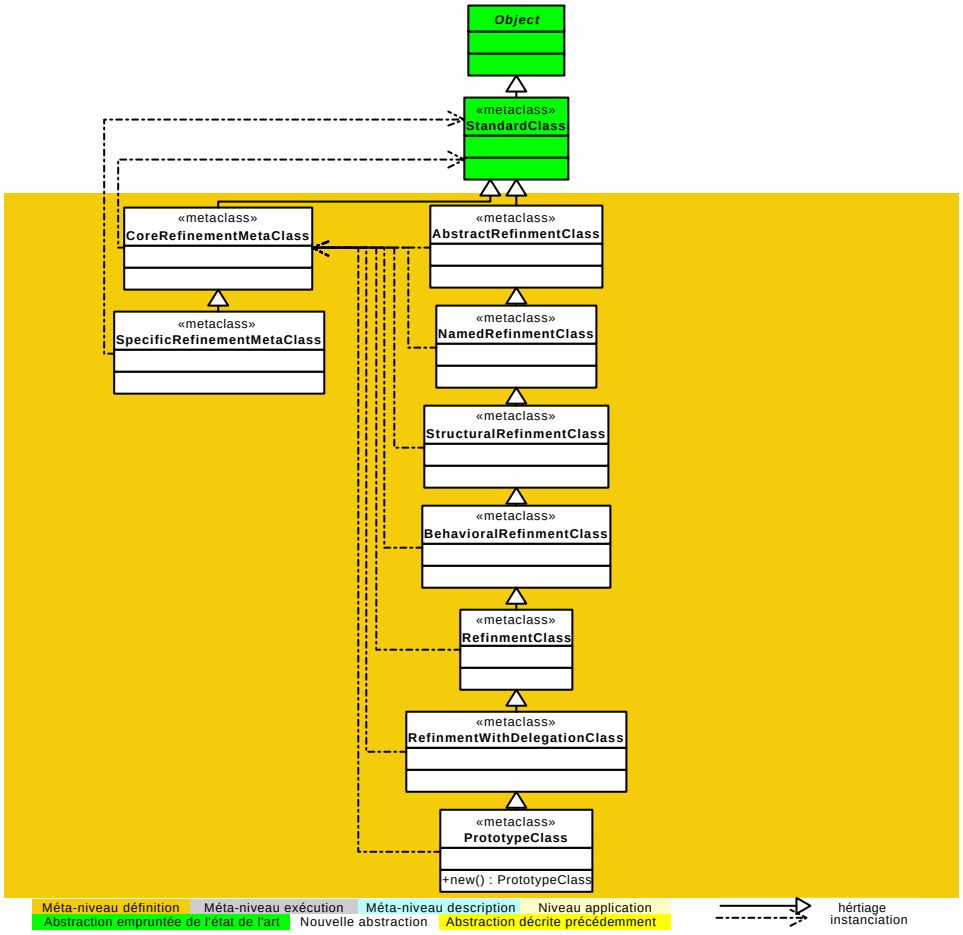


Figure 83 : Diagramme de classe UML du noyau du framework MXDYCTALK.

Au contraire, dans le cas de MXDYCTALK chaque méta-classe a son identité propre et peut être instanciée autant de fois que cela peut s'avérer nécessaire. En effet, en quelque sorte on peut dire qu'ici les classes et les méta-classes *se différencient seulement du point de vue fonctionnel* : les instances des méta-classes sont à leur tour instanciables et servent à décrire la structure et le comportement d'autres classes, alors que les instances des classes sont des instances terminales, c'est à dire des objets qui n'ont pas la propriété de définir la structure et le comportement d'autres objets et d'être instanciables.

Aussi, à chaque méta-classe du cœur de MIDYCTALK est associée une méta-classe "explicite" dans MXDYCTALK. Le Tableau 9 ci-dessous décrit la correspondance entre ces méta-classes:

A cette méta-classe dans MIDYCTALK	Correspond celle-ci dans MXDYCTALK
`AbstractRefinement class`	`AbstractRefinementClass`
`NamedRefinement class`	`NamedRefinementClass`
`StructuralRefinement class`	`StructuralRefinementClass`
`BehavioralRefinement class`	`BehavioralRefinementClass`
`Refinement class`	`RefinementClass`
`RefinementWithDelegation class`	`RefinementWithDelegationClass`
`Prototype class`	`PrototypeClass`

Tableau 9 : Correspondance entre les méta-classes dans MIDYCTALK et MXDYCTALK.

Comme il est illustré par la Figure 83, la méta-classe `AbstractRefinementClass` est la super-classe de toutes les méta-classes de notre implantation. Elle est, par ailleurs, sous-classe de la classe `StandardClass`. Les sous-classes suivantes de la méta-classe `AbstractRefinementClass` mettent chacune en œuvre un *type d'adaptation* (cf. le chapitre IV, section 1.4, page 150) :

1. La classe `NamedRefinementClass` met en œuvre le changement dynamique du nom des classes.
2. La classe `StructuralRefinementClass` met en œuvre le changement dynamique de la structure des classes.
3. La classe `BehavioralRefinementClass` met en œuvre le changement dynamique du comportemlent des classes
4. La classe `PrototypeClass` met en œuvre l'adaptation continue.

La classe `RefinementClass` met en œuvre la gestion des descriptifs de services. Par ailleurs, il nous a semblé raisonnable de faire hériter chacune de ces classes de celle qui la précède dans la liste ci-dessus. Il n'y a toutefois pas d'obligation sur ce sujet et les outilleurs, voire les programmeurs, sont libres d'adopter d'autre stratégies d'implantation.

2.2.2 Cas des instances uniques de méta-classes dans SMALLTALK-80

Chaque méta-classe du cœur de MIDYCTALK est représentée à travers son instance unique. C'est par ailleurs, ainsi que SMALLTALK-80 assure la gestion aisée du problème de *compatibilité* [Gra89]. La situation est tout à fait différente dans le cas de METACLASSTALK où le programmeur peut choisir à volonté non seulement la super-classe de sa classe, mais aussi sa méta-classe. Comme la Figure 83 peut ici en témoigner, les deux arbres d'instanciation et d'héritage peuvent de ce fait devenir par ailleurs, beaucoup plus complexes.

MIDYCTALK utilise cette facilité de SMALLTALK-80 pour implanter au sein des instances uniques des méta-classes de son noyau la structure et le comportement relatifs aux instances des adaptations. Cela comprend tout particulièrement la gestion des valeurs des attributs qui est implantée au sein de la classe `StructuralRefinement`, instance unique de la méta-classe `StructuralRefinement class`. Quelques méthodes utilitaires sont également implantées dans les classes `AbstractRefinement` et `BehavioralRefinement`. Notre choix est ici approprié car les programmeurs sont obligés de faire hériter une classe à adapter de l'une des instances uniques des méta-classe du cœur de MIDYCTALK

En l'absence de ces instances uniques, nous proposons l'ensemble de ces méthodes d'instance sous forme d'une interface, au sens du langage Java [Java]. Cette interface est décrite dans le Tableau 10. Toute classe adaptable (instance d'une des méta-classes de MXDYCTALK) doit soit implanter soit hériter l'implantation de ce protocole. Une telle implantation consiste à ajouter une variable d'instance, `properties`, et à implanter le protocole minimalisé ci-dessous.

Nom de la méthode	Son rôle au sein de l'interface
`getType`	Retourne le méta-objet qui contrôle la structure et le comportement de l'objet. Se confond ici avec la classe de l'objet.
`getProperties`	Retourne l'ensemble des couples (nom d'attribut, valeur).
`getProperty: aString`	Retourne la valeur associée à un slot du nom aString. Ce slot peut être une variable d'instance ou un attribut.
`setProperty: aString to: anObject`	Affecte la valeur `anObject` à un slot du nom `aString`. Ce slot peut être une variable d'instance ou un attribut.
`removeProperty: aString`	Supprime la valeur associée à l'attribut dont le nom correspond à l'argument `aString`.
`hasValueForProperty: aString`	Retourne vrai si le receveur a déjà une valeur pour l'attribut dont le nom est passé en argument (`aString`), sinon retourne faux.
`isValidValue: aProperty forProperty: aString`	Retourne vrai si le type de la valeur stockée dans l'argument aProperty est compatible avec le type défini pour l'attribut dont le nom est passé en argument (`aString`), sinon retourne faux.
`run: aProcessName`	Exécute, dans le contexte du receveur, le micro-procédé dont le nom est passé en argument.
`invoke: selector withArguments: argArrayOrNil`	Si l'argument `selector` correspond au nom d'une méthode implantée dans le receveur, alors lance l'exécution de cette méthode et retourne le résultat. Si l'argument `selector` correspond au nom d'un attribut, alors retourne la valeur courante de celui-ci. Sinon, il s'agit d'un appel à une procédure externe avec le receveur en premier argument.

Tableau 10 : Protocole d'instance des adaptations.

Les classes adaptables dont la méta-classe est `PrototypeClass` (adaptation du type prototype) font *exception* à cette règle. En effet, dans ce cas il n'y a *jamais la création d'instance terminale.* L'interface ci-dessus est implantée par cette méta-classe elle même, dont les instances sont aussi bien des classes que des instances terminales (cf. Perspectives : Langages d'experts et langages à prototypes (Classes Autonomes), page 207).

2.2.3 Problème de repérage des méta-classes de DARC

L'un des problèmes que les outilleurs doivent ici solutionner consiste à permettre de distinguer les méta-classes qui outillent l'adaptation suivant le système de classes DARC, par rapport aux autres méta-classes du système. A titre d'exemple, comment distinguer la méta-classe `PrototypeClass` par rapport aux méta-classes `StandardClass` ou `SavnigsAccountClass` ?

C'est ce que nous entendons par le problème de repérage des méta-classes de DARC.

La gestion SMALLTALK-80 des méta-classes rend impossible l'automatisation d'une telle distinction. C'est pour cela que nous avions été obligé d'ajouter dans chacune des méta-classes du cœur du système MIDYCTALK le message `isCoreRefinementClass` dont l'implantation consiste à tester « en dur » si le receveur du message est équivalent à la méta-classe concernée. A titre d'exemple, la méta-classe `PrototypeClass` implante cette méthode de la façon suivante : `^self == PrototypeClass`. Cela permet en effet, de distinguer `self` par rapport à ses sous-classes (ici `SavingsAccountClass`).

METACLASSTALK permet une solution élégante à ce problème. Celle-ci consiste simplement à créer une méta-classe qui est la méta-classe des classes du cœur du système MXDYCTALK, et qui répond systématiquement `true` au message `isCoreRefinementClass`.

Nous avons ici confié ce rôle à la classe `CoreRefinementMetaClass`. Celle-ci hérite de la classe `StandardClass` et est aussi une instance de celle-ci. Elle implante principalement trois prédicats :

1. la méthode `isCoreRefinementClass` qui retourne `true`, pour indiquer que toute instance de cette méta-classe est une classe qui implante le cœur du système MXDYCTALK. C'est par exemple, la classe `Refinement`.
2. la méthode `isRefineable` qui retourne `false`, pour indiquer que toute instance de cette méta-classe est une classe qui ne peut pas être adaptée. Il s'agit d'un choix arbitraire pour éviter des complications inutiles.
3. la méthode `isRefinement` qui retourne également `false`, pour indiquer que toute instance de cette méta-classe n'est pas une adaptation. Il s'agit là aussi d'un choix arbitraire pour éviter des complications inutiles relatives à une application réflexive de l'adaptabilité aux classes qui mettent en œuvre l'adaptation.

La Figure 84 illustre un usage de cette méta-classe. Elle montre la définition de la méta-classe `AbstractRefinementClass` comme une sous-classe de la méta-classe `StandardClass` et instance de la méta-classe `CoreRefinementMetaClass.`

```
StandardClass subclass: #AbstractRefinementClass
      instanceVariableNames: ''
      category: 'AM-51-TO3-Meta Kernel'
      metaclass: CoreRefinementMetaClass
```

Figure 84 : Implantation en METACLASSTALK de la méta-classe `AbstractRefinementClass.`

La méta-classe `CoreRefinementMetaClass` implante d'autres méthodes qui permettent de trier les classes adaptées et adaptables comme `allCoreRefinementClassesDo`:, `allRefineablesDo`: et `allRefinementsDo`: Elle est, par ailleurs, spécialisée par la méta-classe `SpecificRefinementMetaClass` qui redéfinie le prédicat `isCoreRefinementClass` pour retourner systématiquement `false`. Celle-ci est alors utilisée comme méta-classe des classes applicatives comme `SavingsAccountClass`. (pour une explication de l'intérêt de cette méta-classe cf. le chapitre IV, section 1.4, page 150).

Cette amélioration conduit à la suppression du message `isCoreRefinementClass` dans les classes suivantes de MIDYCTALK lors du portage vers MXDYCTALK : `NamedRefinement class`, `StructuralRefinement class`, `BehavioralRefinement class`, `Refinement class` et `Prototype class` . A présent elle est uniquement implantée par la méta-classe `AbstractRefinementClass`, qui retourne systématiquement `false`.

2.3 D'autres cas notables

2.3.1 Protocoles de classe deviennent des protocoles d'instance

Il nous semble également important de préciser la disparition dans le cas de MXDYCTALK des protocoles dits de classe qui existent dans le cas de MIDYCTALK au profit des protocoles d'instances. En effet, les méta-classes explicites sont des classes qui comportent la définition de la structure et du comportement de leur instances (qui se trouvent être également interprétées comme des classes).

Nous reviendrons également dans la section 2.1, page 207 sur la complexité engendrée par ce schéma basé sur trois "niveaux" d'objets : instances terminales, classes et méta-classes et les perspectives offertes par les ***classes autonomes*** pour la réduire.

2.3.2 Disparition des protocoles d'exemples

Chaque méta-classe de MIDYCTALK comporte un protocole d'exemples qui sert à illustrer le mode d'usage du type d'adaptation qu'elle modélise et met en œuvre à travers des instances de son instance unique. Cette configuration méta-classe/instance unique étant absente dans le cas de MXDYCTALK, les protocoles d'exemples ne peuvent plus être implantée de la même façon.

Aussi, dans la mesure où tous ces exemples ne sont pas nécessaires pour mettre en évidence l'intérêt des méta-classes explicites, nous détaillons ici uniquement un exemple qui montre dans quelle mesure les meta-classes explicites permettent de valider nos hypothèses (cf. ci-dessous)[121].

2.3.3 Modifications imposées par METACLASSTALK

Cas de la méthode `getType`

La méthode `getType` est un héritage du schéma de conception DOM [RTJ00] et son rôle est de retourner le méta-objet qui contrôle la structure et le comportement d'un objet. Cet objet correspond à une instance de la classe `ComponentType` dans le cas de l'implantation standard de notre framework, mais il se confond avec la classe de l'objet dans le cas de l'implantation à l'aide de SMALLTALK-80. Aussi, son implantation dans ce cas se résume à `^self class`. Or, cette implantation ne convient pas à METACLASSTALK qui exige de remplacer le message `class` par `metaobject`.

Cas de la première variable d'instance

METACLASSTALK exige que toute classe issue directement ou indirectement de la méta-classe StandardClass dispose d'une variable d'instance appelée `body`. Celle-ci sert dans la mise en œuvre du changement dynamique de classe. C'est, à titre d'exemple, pourquoi la classe `Refinement` dispose d'une variable d'instance `body`.

A noter que cette approche, qui correspond au schéma de conception PropertyList [FY98b, Rie97a], est également utilisée dans nos architectures. Elle se concrétise au sein du protocole d'instance des adaptations, décrit par le Tableau 10.

2.4 Etendre le framework MXDYCTALK

Un des avantages de MXDYCTALK est qu'il permet au programmeurs de créer de nouveaux types d'adaptation. Pour ce faire, il suffit de spécialiser la méta-classe `AbstractRefinementClass,` ou l'une de ses sous-classes. Dans ce cas, la méta-classe `SpecificRefinementMetaClass` doit être la méta-classe d'une telle méta-classe (ou l'une de ses spécialisations).

En effet, comme nous venons de l'expliquer ci-dessus (cf. le paragraphe §2.2.3, page 186), ce choix permet de distinguer cette spécialisation par rapport aux méta-classes livrées par défaut par MXDYCTALK.

A titre d'exemple, la Figure 85 montre la création par des programmeurs d'un nouveau type d'adaptation, appelé `SavingsAccountClass.` Comme permet de le constater ce listing, la méta-classe `SavingsAccountClass` est une instance de `SpecificRefinementMetaClass` est sous-classe de `PrototypeClass`. Elle comporte une variable d'instance `interestRate`. Celle-ci est utilisé pour stocker le taux d'intérêt qui est un attribut commun à toutes les instances d'une adaptation de ce type.

```
PrototypeClass subclass: #SavingsAccountClass
```

[121] Au niveau du code source, le protocole d'exemples des méta-classes `PrototypeClass` et `SavingsAccountClass` offre d'autres exemples.

```
instanceVariableNames: 'interestRate '
category: 'AM-52-TO3-Meta Examples'
metaclass: SpecificRefinementMetaClass
```

Figure 85 : Etendre le système MXDYCTALK par création de méta-classes "métier".

Cette méta-classe est utilisée lors de l'illustration de notre framework, à travers l'exemple d'adaptation de comptes bancaires (cf. ci-dessous, §3, page 189).

3 Exemple : adaptation de comptes bancaires

Le but de cette section est de montrer dans quelle mesure l'outillage que nous venons d'exposer ici apporte une solution convenable aux problèmes mentionnés ci-dessus (cf. §1.1, page 180).

Il est important de noter que le framework MXDYCTALK conserve toutes les propriétés des langages d'experts que nous avons exposé dans le chapitre précédent (chapitre IV). Cela comprend notamment la dimension workflow, le refactoring des adaptations et le travail collaboratif, la facilité d'apprentissage et le lien causal. En effet, le code de MXDYCTALK est en grande partie identique à celui de MIDYCTALK.

Ce qui change ici relève du choix local du type d'adaptation, une particularité de la spécialisation dynamique et plus particulièrement de la technique utilisée pour assurer l'ajout dynamique de nouveaux types d'objets. C'est cette particularité qui est absente dans le cas de la technique utilisée par MIDYCTALK.

Aussi, afin d'offrir un exposé plus synthétique, nous ne répétons pas ici les propriétés identiques de MIDYCTALK. Nous illustrons uniquement les propriétés nouvellement assurées par MXDYCTALK. Autrement dit, nous montrons que les méta-classes "explicites" permettent bien de considérer l'adaptabilité comme une propriété de classe et par conséquence assurent le *choix local* du type d'adaptation.

Nous rappelons toutefois à la fin de cette section quelques propriétés déjà écrites et dont l'exposé représente, toutefois, un intérêt dans cette nouvelle situation.

3.1 Assurer le choix local du type d'adaptation

Par rapport au framework MIDYCTALK, le framework MXDYCTALK offre principalement un nouvel outil. Il s'agit de la possibilité de préciser lors de la définition le type d'une adaptation, en passant en argument de la méthode de création de l'adaptation, la méta-classe qui modélise le type d'adaptation souhaité.

Nous allons utiliser cette possibilité pour montrer que MXDYCTALK permet bien le choix local du type d'adaptation et, par conséquence, est en mesure de nous conduire, à titre d'exemple, au modèle objet final de notre langage d'experts dédié à la gestion de comptes bancaires (cf. l'introduction, §2, page 31).

Pour plus de commodité, nous rappelons ici par le Tableau 11 ci-dessous les stéréotypes utilisés pour désigner les différents types d'adaptation (cf. le chapitre IV, §1.4, page 150).

Le type d'adaptation	Le stereotype de classe utilisé	Abréviation
Adaptation de nom	*Name refinement*	*name ref.*
Adaptation de structure	*Structural refinement*	*strutural ref.*
Adaptation de comportement	*Behavioral refinement*	*beh. refinement*
Adaptation	*Refinement*	*refinement*
Adaptation avec stratégie d'héritage	*Refinement with inheritance strategy*	*ref. with delegation*
Adaptation prototypique	*Prototype*	*prototype*

Tableau 11 : Rappel des stéréotypes utilisés pour désigner les types d'adaptation.

3.1.1 Ajouter une adaptation du type *refinement*

La première étape de notre démonstration consiste à ajouter une adaptation de la classe `BankAccount` du type *refinement* (cf. le Tableau 11 ci-dessus).

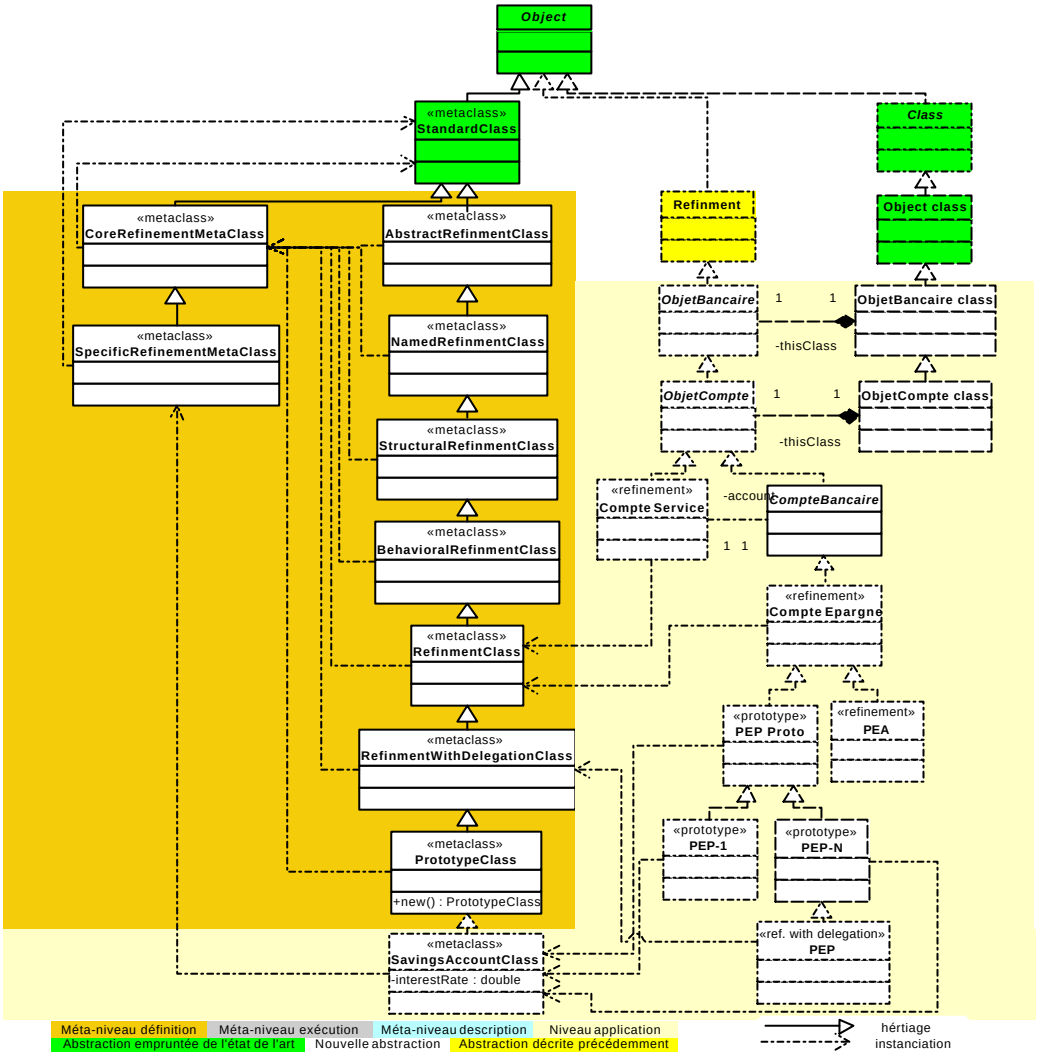


Figure 86 : Choix explicite de méta-classes permet un meilleur outillage de l'adaptation.

Comme permet de l'illustrer la Figure 86 ci-dessus, grâce à la nouvelle possibilité offerte par le framework MXDYCTALK, l'expert peut ajouter une adaptation de la classe `BankAccount`, appelée `SavingsAccount`, alors que `BankAccount` n'est pas une classe adaptable et hérite "naturellement" de la classe `AccountObject`[122].

[122] Le langage METACLASSTALK est compatible avec SMALLTALK-80. Aussi, `AccountObject` peut indifféremment être une classe issue de SMALLTALK-80 ou de METACLASSTALK. Dans ce dernier cas, elle peut ici être simplement une instance de la (méta-)classe `StandardClass`. Comme permet de l'illustrer la Figure 86, la méta-classe `StandardClass` est la racine de l'arbre d'héritage (et aussi d'instanciation et méta-objets) des méta-classes de METACLASSTALK. Cette classe comporte la définition de la structure et du comportement requis à toutes les (méta-)classes du système. Elle met, par ailleurs, en œuvre une solution au problème de la régression infinie du lien d'instanciation qui est initialement proposée dans le système SMALLTALK -76 [Ing78] est explicitée dans les travaux autour de OBJVLISP [Coi87].
Dans notre implantation de cet exemple, `AccountObject` est toutefois, une classe issue de SMALLTALK-80.

Pour ce faire, l'expert fournit le nom de la nouvelle adaptation (ici `SavingsAccount`), la classe à adapter (ici `BankAccount`) ainsi que la méta-classe qui modélise le type d'adaptation souhaité (ici `RefinementClass`)[123].

La Figure 87 montre une vue à travers le flâneur du système SQUEAK/MXDYCTALK du modèle objet du langage d'experts dédié à la gestion de comptes bancaires après l'ajout de cette adaptation.

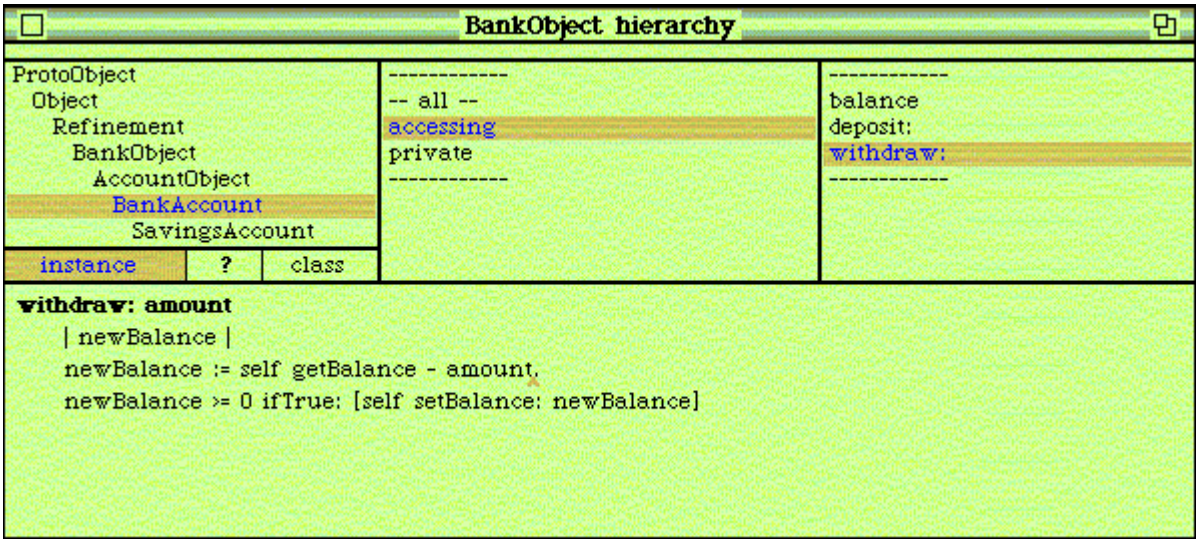


Figure 87 : Adaptation assurée par MXDYCTALK et vue à travers le flâneur de SQUEAK.

3.1.2 Spécialiser l'adaptation du type *refinement* par une du type *prototype*

L'étape suivante de notre démonstration consiste à adapter à nouveau l'adaptation `SavingsAccount`. L'expert souhaite cette fois une adaptation du type "prototype". Ce type est donc choisi localement pour cette nouvelle adaptation et indépendamment du type d'adaptation de `SavingsAccount`.

La raison du choix type "prototype" est la volonté de l'expert de procéder à une série de prototypage avant de se fixer sur un choix de nouveau type de compte `PEP` (Plan d'Epargne Populaire).

Pour ce faire, nous procédons de la même façon que dans le cas de `SavingsAccount`, décrite ci-dessus. Nous devons ici simplement changer les paramètre fournis lors de la création de l'adaptation.

La Figure 88 montre ces paramètres. Le nom de l'adaptation est `Proto PEP`. Sa super-classe est `SavingsAccount` et sa méta-classe est `SavingsAccountClass` (définie ci-dessus §2.4, page 188). Rappelons que la méta-classe `SavingsAccountClass` spécialise la méta-classe `PrototypeClass`.

Cette création est réalisée par l'envoi du message[124] `refinement:publicName:metaclass:` à la classe adaptée, ici `SavingsAccount`.

```
^SavingsAccount
      refinement: nil
      publicName: 'Proto PEP'
      metaclass: SavingsAccountClass.
```

Figure 88 : Spécifier explicitement (et localement) le type d'adaptation.

[123] De toute évidence, l'expert ne désignera pas ces entités par des nom, plutôt familier pour des informaticiens, que nous déployons ici. Le langage d'expert les leur proposera par des appellations et d'habillages plus appropriés par rapport au métier concerné.

[124] Ce script peut être retrouvé dans la méthode `example00` de la méta-classe `PrototypeClass class`.

Il est important de noter que l'expert n'est pas supposé fournir le nom système de cette adaptation. C'est pourquoi le premier argument de cet appel est `nil`. Cela conduit à la génération automatique d'un nom défaut par MXDYCTALK, suivant le principe décrit ci-dessus. A titre d'exemple, cette génération produit ici la chaîne de caractères `SavingsAccount1`.

La Figure 89 montre le résultat de l'exécution de l'envoi de message de la Figure 88, tel qu'il apparaît dans le flâneur du système SQUEAK. Comme nous venons de l'exposer, on peut constater la création d'une classe appelée `SavingsAccount1`, sous-classe de la classe `SavingsAccount` et l'instance de la méta-classe `SavingsAccountClass`.

```
SavingsAccount subclass: #SavingsAccount1
      instanceVariableNames: ''
      category: 'AM-Refinements'
      metaclass: SavingsAccountClass
```

Figure 89 : Résultat de l'exécution de l'envoi de message de la Figure 88.

3.1.3 Phases de prototypage

Comme le veut notre cahier des charges, l'expert qui dispose d'un premier prototype de compte `PEP`, va pouvoir en définir d'autres. La procédure est identique à celle déjà décrite ci-dessus. Cela nous conduit aux adaptations appelées `PEP`-1 à `PEP`-N de la Figure 86.

3.1.4 Spécialiser une adaptation du type *prototype* par une du type *ref. with delegation*

Le dernier élément de notre cahier des charges consiste à fixer le type d'adaptation de `PEP`-N. En effet, c'est elle qui est choisie par l'expert comme le "prototype" approprié pour représenter le type de compte `PEP`.

Pour ce faire, il suffit de créer une nouvelle adaptation (suivant toujours la même procédure) dont le nom est `PEP`, la super-classe est `PEP`-N, la méta-classe est `RefinementWithDelegationClass`. Il s'agit d'une classe/adaptation vide dont le rôle est uniquement d'assurer un retour à une instanciation "ordinaire", au sens des langages à objets.

Cela permet à cette dernière adaptation de `Proto PEP` d'hériter des définitions de `PEP`-N, tout en répondant au message `new` par la création d'instances terminales.

Nous obtenons ainsi le modèle objet de la Figure 90. Celui-ci correspond à celui que nous avions promis de faire produire par des experts dans l'introduction (cf. §2, page 31).

Rappelons que nous avions montré lors du chapitre précédent (chapitre IV, §3, page 158) l'ajout dynamique des adaptations relatives aux `Comptes-Service` (partie gauche de la Figure 90). Nous venons de monter ici l'ajout dynamique des adaptations concernant l'ajout dynamique du type de compte `PEP` (partie droite de la Figure 90).

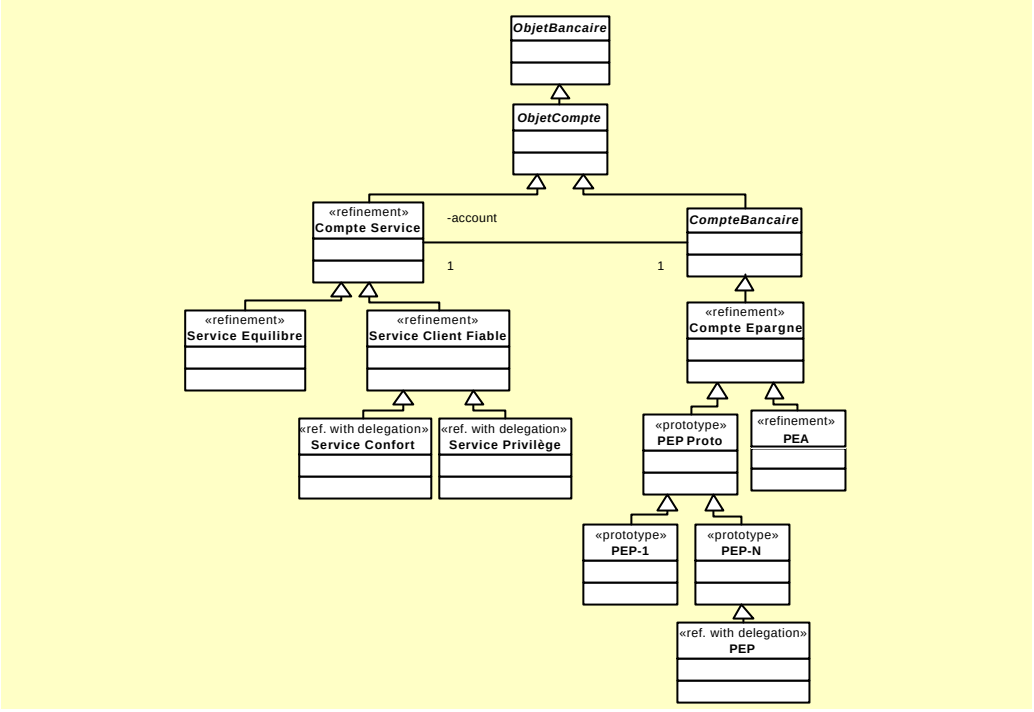


Figure 90 : Obtention du modèle objet visé par l'adaptation des comptes bancaires.

Changer le type d'adaptation d'une classe est aussi simple que de le choisir initialement. Il s'agit en effet, de créer une nouvelle adaptation vide de cette classe qui aura comme méta-classe celle qui modélise le type d'adaptation souhaité. Cette technique est identique à celle que nous avons expliquée ci-dessus pour fixer le type d'adaptation des types de comptes PEP.

Une autre possibilité consiste à utiliser la technique de changement dynamique de classe mise en œuvre par le système METACLASSTALK.

Il est important de noter ici que le choix initial ou le changement du type d'adaptation correspond à instancier une méta-classe. Cette opération ne devait pas, *a priori*, solliciter le compilateur. Or, pour des besoins propres au système METACLASSTALK, il y a ici l'usage d'un compilateur spécifique [Bou99b, page 57]. Celui-ci met en œuvre la technique d'encapsulation de méthodes (*method wrapper*) proposée par [BFJR98].

Nous reviendrons sur cette question plus tard dans les perspectives, section 2.2, page 213.

3.2 Rappel sur d'autres propriétés des langages d'experts

Nous tenons à rappeler ici brièvement que les programmeurs peuvent, de la même façon que nous l'avons décrite dans le chapitre IV, intervenir sur les adaptations. Cette intervention peut prendre diverses formes : simple ajout de variables d'instances et de méthodes ou encore de l'ajout de procédures à la micro-workflow ou le r*efactoring* [Opd92, Rob99] des interventions des experts. Un cas primitif d'intervention consiste ici à changer le nom auto-généré par un nom plus pertinent, par exemple ici `ProtoPEP.`

Il est important de noter également que les principes qui régissent la mise en œuvre de l'adaptation structurelle et comportementale par l'ajout de descriptifs d'attributs et de procédures ne changent pas ici par rapport à ce que nous avons décrit dans les chapitres précédents.

La Figure 91 montre, à titre d'exemple, l'ajout dynamique de descriptifs d'attributs à la nouvelle adaptation `Proto PEP`. Il s'agit ici d'un attribut du type numérique appelé *Interest Amount* (intérêts cumulés), et d'un autre attribut du type chaîne de caractères appelé *Last nam*e (nom de famille).

```
PrototypeClass >> example01
      "PrototypeClass new example01"

      ^ProtoPEP
            addPropertyType: (PropertyType on: Number named: 'Interest Amount');
            addPropertyType: (PropertyType on: String named: 'Last name')
```

Figure 91 : Ajout dynamique de descriptifs d'attributs à la nouvelle adaptation Proto PEP.

Comme notre nouvelle adaptation est du type *prototype*, elle peut elle-même contenir des valeurs pour ces attributs. Ainsi, on peut lui envoyer le message `setValue: 'Razavi' toPropertyNamed: 'Last name'` afin d'affecter la chaîne de caractères 'Razavi' à l'attribut 'Last name'. L'exécution de cet envoi de message conduit à l'ajout dans la table de hachage `properties` d'une clé composé de couple ('Last name', 'Razavi').

Si on interroge à présent cette *classe autonome* (cf. Perspectives, page 207) par l'envoi de message getProperty: `'Last name'`, elle retourne une instance de la classe `Property` dont la valeur courante est la chaîne de caractères 'Razavi'.

Une autre possibilité pour "instancier" cette *classe autonome* consiste à lui envoyer le message `new`. Ce dernier donne alors lieu à l'appel du message `autoRefine` qui crée une sous-classe de cette classe anonyme de `Proto PEP` dont la méta-classe est une nouvelle instance de la méta-classe de self (ici `Proto PEP`). On peut alors procéder à l'affectation de valeur à cette nouvelle instance (cf. également Perspectives, page 207).

4 Conclusion: apports du framework MXDYCTALK

Les travaux présentés dans ce chapitre nous conduisent à deux conclusions. La première concerne la validation de notre thèse concernant le choix local du type d'adaptation. La seconde est une tentative de rapprochement entre les deux systèmes MXDYCTALK et METACLASSTALK en vue de la création d'un système qui intègre leurs traits intéressants, en ce qui concerne l'outillage de la création des langages d'experts.

Cela comprend d'une part le choix explicite des propriétés de classes et leur *composition* offert par METACLASSTALK et d'autre part l'outillage modulaire, personnalisable et dédié à l'adaptation offert par MXDYCTALK.

4.1 Validation de notre thèse

Dans ce chapitre nous avons montré que la solution mise en œuvre par le système MIDYCTALK n'est pas tout à fait satisfaisante. Elle conduit, en effet, au choix obligatoire par des programmeurs du type d'adaptabilité au niveau de toute une hiérarchie de classe. Cette situation engendre de nombreux problèmes que nous avons également exposés ici.

Nous diagnostiquons l'origine de ces problèmes comme étant le choix implicite de la méta-classe par le système SMALLTALK-80. Nous proposons alors une solution basée sur le choix explicite de la méta-classe, que nous mettons en œuvre à l'aide du système METACLASSTALK de Noury Bouraqadi. Cette solution ramène, en effet, notre problème à celui déjà connu et résolu de la représentation des propriétés de classe à l'aide de méta-classes et le choix de la nature de classes par le *choix explicite de leurs propriétés* (au sens de [LC96]).

Afin de pouvoir valider notre solution, nous avons procédé à une implantation des deux systèmes DART et DARC dans la dernière version du langage METACLASSTALK. Cette implantation donne lieu au nouveau framework MXDYCTALK, qui nous a permis de montrer la faisabilité d'attribuer un type d'adaptation à toute classe `C`, indépendamment du type d'adaptation attribué à d'autres classes du système et notamment à la super-classe de `C`.

En s'appuyant toujours sur le système METACLASSTALK, MXDYCTALK permet aussi de revenir sur ces décisions lors de l'exécution et changer dynamiquement le type d'adaptation.

Aussi, nous pouvons affirmer :

1. que les méta-classes explicites et le système METACLASSTALK offrent de meilleures performances aux outilleurs des langages d'experts ;
2. qu'un langage d'experts basé sur MXDYCTALK (d'une manière générale, d'une implantation basée sur le choix explicite de méta-classes de DYCRA) offre plus de souplesse aux programmeurs ;
3. qu'un langage d'experts basé sur MXDYCTALK offre plus de souplesse aux experts.

Nous validons ainsi la seconde et la dernière partie de notre thèse (cf. le paragraphe 1.5.2 de l'introduction, page 25).

4.2 Résultats complémentaires : langages d'experts et AOP

Ce troisième outillage de l'adaptation nous conduit également à des résultats complémentaires que nous souhaitons commenter ici. Nous rappelons qu'en raison de la non-disponibilité de la fonction de composition des méta-classes dans la version actuellement en cours de portage sous SQUEAK du système METACLASSTALK, les aspects développés ci-dessus n'ont pas pu être mis en œuvre.

4.2.1 Rendre adaptable l'Aspect de base

Le but de cette section est de montrer, à travers des exemples précis, la complémentarité des mécanismes mis en œuvre par les deux systèmes METACLASSTALK et MXDYCTALK.

En effet, METACLASSTALK apporte déjà à MXDYCTALK le choix local du type d'adaptation. Nous estimons que cet apport peut être enrichi et permettre de traiter automatiquement les aspects techniques des adaptations (cf. §4.2.1.1, page 196). Par ailleurs, MXDYCTALK peut apporter à METACLASSTALK l'adaptabilité de "l'aspect de base" (cf. §4.2.1.2, page 197).

Cette situation peut être résumée de la façon suivante : METACLASSTALK s'intéresse à l'adaptation dynamique des aspects techniques alors que MXDYCTALK s'intéresse à l'adaptation de l'aspect de base.

Nous étudions ici le rapprochement de ces deux systèmes pour outiller la création de systèmes dont *aussi bien les aspects techniques que l'aspect de base seront adaptables*.

4.2.1.1 Traitement automatique des Aspects Techniques des adaptations

En apportant une solution au problème de composition de méta-classes, METACLASSTALK permet d'attribuer plusieurs propriétés à une même classe. Pour ce faire, METACLASSTALK adopte une solution basée sur l'héritage multiple de mixins et la coopération de méta-classes. Les propriétés de classe sont définies dans des méta-classes-mixins qui coopèrent pour résoudre les conflits entre les méthodes du MOP de METACLASSTALK.

Cette fonctionnalité nous semble permettre l'usage de METACLASSTALK pour résoudre un autre problème posé pour la création de langages d'experts, c'est le problème du traitement automatique des aspects techniques des adaptations. A titre d'exemple, cela devait permettre de spécifier lors de l'exécution que l'adaptation Proto PEP de la Figure 82, page 181 synchronise les accès en écriture à ses attributs. Nous reviendrons sur cette question dans le paragraphe suivant.

4.2.1.2 Rendre également adaptatif "l'aspect de base"

La séparation des aspects[125] [HL95] s'inscrit dans un cadre génie logiciel et vise à améliorer la maintenabilité et l'évolutivité des logiciels. Elle est promue par le paradigme de la programmation par aspects [KLMMLI97] qui vise notamment à palier aux "limites de la programmation par objets lorsqu'elle est utilisée pour définir des logiciels qui nécessitent des mécanismes d'exécution particuliers".

Une application est alors décomposée de la façon suivante (source [Bou99, page 157]):

1. *un aspect de base* (appelé également tâche) : il est constitué des définitions des objets métier de l'application et de leurs interactions dont la somme représente les services (i.e. "fonctionnalités") réalisés par l'application.
2. *plusieurs aspects techniques* : ils représentent la manière de réaliser les services définis dans l'aspect de base. Ils correspondent aux mécanismes d'exécution de l'application.

Comme le précise N. Bouraqadi, cette approche s'inscrit dans la ligne des travaux qui préconisent l'usage de la réflexion pour séparer les "besoins non-fonctionnels[126]" des services réalisées par les logiciels [WY88, KP94, McA95b, GC96, SW96]. Il semble qu'un consensus se dégage, c'est celui qui considère que "un langage réflexif offre deux niveaux de programmation : le niveau de base et le méta-niveau. Le niveau de base permet de décrire les services réalisés par une application donnée (i.e. le "Quoi") alors que le méta-niveau permet de décrire la manière de l'exécuter (i.e. le "Comment"). … . Autrement dit, l'aspect de base est décrit dans les classes alors que les aspects techniques sont définis dans les méta-classes." [Bou99, page 170].

N. Bouraqadi constate également une analogie entre la séparation des aspects [HL95] des applications et la séparation des collaborations des objets telle qu'elle est mise en œuvre dans les travaux sur la programmation par contextes [VCD97, Van98] ou la programmation adaptative [Lie96, ML98]. Il observe, en effet, que de façon analogue aux aspects, les collaborations sont également transversales à différentes classes [Bou99, page 195].

Aussi, il propose d'étendre dans un premier temps la notion d'aspect en introduisant les *aspects fonctionnels* qui correspondent à l'explicitation des collaborations des objets. Dans une application donnée, chaque aspect fonctionnel décrit une collaboration entre différents objets en vue de réaliser un service donné. Ensuite, il montre que la distinction entre les aspects techniques et les aspects fonctionnels n'est pas toujours pertinente. Alors, il propose de généraliser la notion d'aspect pour couvrir non-seulement les aspects techniques et les aspects fonctionnels, mais aussi les aspects *hybrides*.

Un aspect hybride, selon N. Bouraqadi, réunit des collaborations et des mécanismes d'exécution qui sont étroitement liés. Dans ce cadre, les aspects techniques et les aspects fonctionnels sont considérés comme des cas particuliers des aspects hybrides. Enfin, il propose l'usage de la réflexion et des méta-classes comme support de la programmation par aspects généralisée [Bou99, page 178].

Dans le contexte réflexif de METACLASSTALK, cette proposition a conduit à un schéma de représentation unique des différents aspects techniques, fonctionnels et hybrides. Ce schéma décompose un *aspect généralisé* en trois parties distinctes :

1. Partie *collaboration*: décrit les interactions entre certains objets pour réaliser un service donné.
2. Partie *mécanisme d'exécution*: décrit à l'aide des méta-classes, la manière de réaliser les collaborations.
3. Partie *configuration*: permet d'attribuer aux classes de l'application les propriétés définies dans les méta-classes de la partie *mécanisme d'exécution*. Les méta-classes servent donc ici à

[125] seperation of concerns.
[126] Non-functional reqirements [SW96].

définir le mécanisme d'exécution et aussi de point de départ à la composition des aspects généralisés [Bou99, page 189].

Notre étude confirme cette généralisation et ce schéma de représentation. En effet, nous avons montré dans ce chapitre dans quelle mesure l'adaptabilité dynamique pouvait être considérée comme une propriété de classe, et donc décrite entre autres[127], à l'aide des méta-classes. C'est la partie *mécanisme d'exécution.*

En s'appuyant sur les mécanismes de composition de méta-classes mise en œuvre par METACLASSTALK il serait alors possible d'associer à chaque classe l'adaptabilité parmi d'autres propriétés comme par exemple, la synchronisation et la persistance. C'est donc la partie *configuration*. De plus, nous explicitons les collaborations entre les objets sous forme de procédés, c'est la partie *collaboration.*

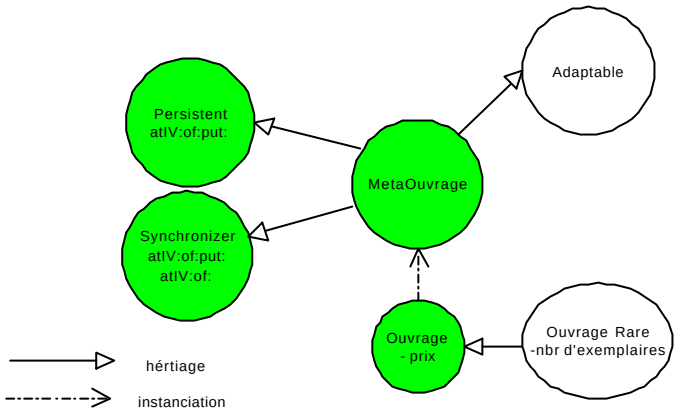


Figure 92 : Associer MXDYCTALK à METACLASSTALK permet de rendre l'Aspect de base adaptable.

Toutefois, notre étude montre la nécessité de préciser davantage la nature des techniques déployées pour expliciter les collaborations. En effet, nous avons montré ici que l'explicitation des collaborations entre les objets peut prendre une dimension autre que la séparation des aspects et la "définition des fragments des classes dont les instances collaborent pour réaliser un service donné" [Bou99, page 183].

C'est aussi intégrer dans un langage à objet un langage spécialisé dans la description explicite de collaborations (avec son propre mécanisme d'interprétation), qui conjugué avec d'autres mécanismes, permet de rendre l'aspect de base dynamiquement adaptable et cela par des experts.

Pour mieux illustrer la situation, nous reprenons ici l'exemple présenté par N. Bouraqadi d'un système de gestion d'une librairie électronique. Le modèle objet d'un tel système est composé des abstractions comme `Librairie`, `Client`, `Ouvrage`, `Commande`, `CarteDeCredit`, `CompteBancaire` et `Banque`. N. Bouraqadi identifie la nécessité de la prise en charge de trois aspects techniques : la distribution, la persistance et la synchronisation. Il montre alors dans quelle mesure il est possible de définir les aspects de base (services) de l'application sans toutefois "polluer" le code avec le traitement des aspects techniques, par la réutilisation de méta-classes. La partie gauche de la Figure 92 illustre la solution proposée par N. Bouraqadi qui s'applique au cas de la classe `Ouvrage`.

La partie droite de cette figure montre que l'usage de l'adaptation comme une propriété de classe permet de rendre la classe `Ouvrage` dynamiquement adaptable. Cela permet, à titre d'exemple, à un bibliothécaire de créer une nouvelle classe/adaptation `Ouvrage Rare` qui comportera la structure et le comportement que l'expert jugera nécessaire pour gérer ce type de livres.

[127] En effet, les adaptations s'appuient également sur d'autres objets de méta-niveau qui réifient en partie la représentation des collaborations entre les objets ainsi que leur exécution. Les méta-classes sont plus particulièrement le point d'articulation entre ces mécanismes et aussi avec ceux du langage d'implantation.

En somme, en surpassant l'usage d'un langage de programmation unique (rappelons qu'un langage d'experts est programmé à deux niveaux) pour la définition des aspects de base et technique nous obtenons de nouvelles fonctionnalités et en l'occurrence *l'adaptabilité de l'aspect de base*.

4.2.2 Rendre modulaire l'architecture de METACLASSTALK

Nous avons donc ici montré l'importance d'un système comme METACLASSTALK pour la mise en œuvre de langages d'experts. Toutefois, l'approche monolithique actuelle de METACLASSTALK pose le problème général du choix optionnel des mécanismes qu'il propose.

En effet, à travers MXDYCTALK nous montrons l'exemple d'un système qui requiert l'usage de certains mécanismes fondamentaux de METACLASSTALK comme le choix explicite des méta-classes et leur composition, mais qui ne fait pas, à titre d'exemple, l'usage du mécanisme d'explicitation implicite des envois de messages (cf. les Perspectives, §2.2, page 213).

L'appel systématique de METACLASSTALK à ce mécanisme basé sur la compilation devient d'autant plus problématique qu'il peut empêcher le bon fonctionnement de langages d'experts.

De manière générale, nous estimons que la modularisation de l'architecture de METACLASSTALK et un couplage approprié avec le framework MXDYCTALK constituent une étape fondamentale dans les recherches sur l'outillage de la création systématique des langages d'experts.

Les deux situations décrites ci-dessus permettent de mieux rendre compte de l'importance d'une telle association qui peut conduire à un outillage qui assure aussi bien l'adaptation des aspects techniques que l'aspect de base.

Conclusions et Perspectives

Conclusions et Perspectives

1 Conclusions générales

1.1 Bilan

La *fonction* de création (lors de l'exécution) de nouveaux types d'objets, leur structure et comportement fait partie du cahier des charges de nombreuses catégories de logiciels à objets. Des exemples familiers sont les langages à objets écrits eux-mêmes sous-forme d'une application objet (sans tenir compte ici de l'aspect réflexif), mais encore des environnements de (méta-)modélisation comme METAGEN du LIP6 [RSBP95, LSGBP99, RBP00], la machine virtuelle UML de la *start-up* américaine SKYVA [RFBO01] ainsi que les langages d'experts présentés dans ce mémoire.

Toutefois, on constate que chacune de ces catégories de logiciel utilise des techniques de conception et d'implantation différentes par rapport à celles utilisées dans les langages à objets pour assurer cette fonction. En effet, la création d'une nouvelle classe par un langage à objets ne fait pas appel au même système de classes qu'un *méta-individu* au sens du système METAGEN, un complément de classe dans le cas des AOMs[128] ou une *adaptation* au sens des langages d'experts.

Pourtant, sur le plan conceptuel, la création d'une classe, d'un *méta-individu*, d'un complément de classe ou d'une adaptation correspond au raffinement d'une même représentation informatique, sous forme d'une hiérarchie de classes, de *l'Univers*. Dans ce cadre, la création (voire l'exécution) d'un logiciel à objets peut être assimilée à l'acte de raffiner successivement la représentation par défaut des langages à objets de *l'Univers* : la classe `Object`.

Ce raffinement peut être mis en œuvre à l'aide de techniques différentes, notamment celles énumérées ci-dessus. Toutefois, de notre point de vue, son outillage approprié nécessite de considérer au moins deux règles :

1. Compatibilité
2. Continuité

La *compatibilité* vise à assurer le travail collaboratif. En effet, différentes catégories d'intervenants peuvent participer au cycle de raffinement, chacune utilisant des modèles et outils qui lui sont plus adaptés. La compatibilité entre les représentations adoptées dans chacun de ces cas est nécessaire pour

[128] La machine virtuelle UML de SKYVA utilise également le schéma de conception DOM des AOMs.

permettre un suivi dans de bonnes conditions par des programmeurs des évolutions de chaque modèle ainsi produit.

Sur le plan technique, sont en premier lieu concernés par la règle de compatibilité les langages à objets, car c'est par eux que commence la création de la représentation *cible*, laquelle inclut le modèle mis au point lors de la création elle-même de ces environnements.

A titre d'exemple, le système SMALLTALK-80 permet la création de logiciels suivant une telle vision : l'implantation du langage lui-même (en très grande partie) n'est pas différentiable de celle des applications implantées avec son aide, l'ensemble constituant un tout, une représentation en *perpétuelle* raffinement de *l'Univers*.

Toutefois, SMALLTALK-80 représente deux inconvénients. Tout d'abord, il n'offre qu'un seul modèle de programmation. Deuxièmement, il fait appel à *l'instanciation* qui engendre une coupure irréversible[129] dans le cycle du raffinement.

En ce qui concerne, le premier inconvénient, nous venons de montrer à travers le système de classes DYCRA-II (DART + DARC-II) et le framework MXDCYTALK qu'il était possible d'aller plus loin et d'outiller l'usage de modèles de programmation différents tout en conservant la compatibilité et donc en assurant le travail collaboratif.

Il s'agit d'une nouvelle solution documentée, validée et testée pour la création systématique de langages d'experts qui satisfont les propriétés que nous avons énumérées dans la section 1.2 de l'introduction.

Cette solution est mise en œuvre par l'extension des langages à objets réflexifs (SMALLTALK-80 et METACLASSTALK). Toutefois, une meilleure approche serait de considérer son cahier des charges dans la conception elle-même de langages à objets de façon à mieux assurer la compatibilité.

En ce qui concerne la règle de *continuité*, elle vise à assurer la continuité du cycle de raffinement et d'éviter la situation mentionnée ci-dessus dans le cas du langage SMALLTALK-80. Nous avons réalisé un travail préliminaire dans ce sens (adaptations prototypiques) et reviendrons sur cet aspect plus en détails dans la suite de ce chapitre, lors de la présentation des *classes autonomes*.

[129] Si l'on veut respecter en même temps la règle de compatibilité, sinon on peut faire appel aux techniques comme celle proposée par DOM, qui, comme nous l'avons montré dans le chapitre IV ne considère pas la compatibilité des représentations adoptées.

1.2 Synthèse des contributions

Les éléments du travail présenté dans ce mémoire peuvent être synthétisés de la façon suivante:

1.2.1 Systèmes de classes

Voici en résumé chacun des systèmes de classes présentés dans ce mémoire, son origine et rôle :

1. *DOM* : l'œuvre de l'équipe de Johnson ([RTJ00,YBJ01b]), il décrit comment concevoir un système pour que ses classes puissent être adaptées lors de l'exécution. Cette modélisation comprend une solution systématique au problème d'ajout lors de l'exécution de nouveaux types d'objets ainsi que la définition lors de l'exécution de la structure de tels types d'objets (la notion de complément de classe). Par contre, en ce qui concerne la définition lors de l'exécution du comportement des compléments de classe, les auteurs reconnaissent qu'il n'existe pas actuellement de solution systématique (implantable sous forme d'un framework) et suggèrent l'usage, au cas par cas, des schémas de conception *Strategy* ou *Interpreter*.
2. *Micro-workflow* : l'œuvre également de l'équipe de Johnson (thèse de D. Manolescu [Man00]), il est dédié à la création de logiciels à objets *flow-independent*. Il s'agit d'un outil destiné aux programmeurs.
3. *DARC* : notre proposition de couplage des deux systèmes DOM et Micro-workflow pour servir de base à l'outillage de la co-évolution dynamique de structures et de procédures (outiller la création de logiciels objets qui permettent, non seulement, de faire évoluer lors de l'exécution la hiérarchie des classes sur le plan structurel, comme le propose DOM, mais aussi assurent la définition de procédures qui utilisent dans leurs calculs la structure des objets définis lors de l'exécution). DARC n'est toutefois, pas en pratique très utile s'il n'est pas couplé avec un autre système comme par exemple DART, afin de rendre son usage pratique.
4. *DART* : également notre proposition, inspiré du modèle de langage de feuilles de calcul et des travaux de Bonnie A. Nardi. Il modélise un langage dont l'apprentissage est, *a priori*, facile pour des experts. DART peut servir indépendamment (de DARC, par exemple) comme un modèle pour la création de frameworks qui assurent la composition de services par des experts. DART n'a, toutefois, aucune notion relative à l'adaptation, d'où l'intérêt de son couplage avec DARC qui va suivre.
5. *DYCRA* : également notre proposition, qui résulte du couplage de DARC & DART. Il sert à créer des frameworks qui assurent la création de langages d'experts munis de toutes les propriétés énumérées dans la section 1.2, page 17 de l'introduction, à l'exception du travail collaboratif et du choix local du type d'adaptation.
6. *DOM-II* (et donc DARC-II et DYCRA-II) : également notre proposition, qui consiste à réviser DOM en considérant que les compléments de classe devaient être de même nature que les classes (donc instances de méta-classes dans le cas de langages à objets réflexifs) pour être éditables par des programmeurs. Le couplage de DOM-II avec le Micro-workflow donne naturellement lieu à DARC-II et le couplage de DARC -II avec DART donne lieu à DYCRA-II. Celui-ci est le système de classes le plus élaboré que nous proposons dans ce mémoire. Il participe (avec le framework MXDYCTALK) à l'outillage de la création de langages d'experts qui assurent toutes les propriétés énumérées dans la section 1.2, page 17 de l'introduction.

1.2.2 Frameworks

Voici en résumé chacun des frameworks présentés dans ce mémoire, son origine et rôle :

1. *FDOM* : notre contribution, framework dédié à la création d'applications qui assurent l'adaptation à l'aide des compléments de classes suivant DOM.
2. *DYCFLOW* : notre contribution, framework réalisé suivant les spécifications de D. Manolescu du système de classes Micro-worflow. Notre implantation ne couvre pas les modules optionnels de Micro-worflow. A noter que Manolescu a lui même validé son système par une implantation sous forme de framework. Cette implantation n'a pas encore été rendue publique.
3. *FDARC* : notre contribution, permet de valider le système de classes DARC par la conception et l'implantation effective d'un framework orienté-objets.
4. *FDART* : notre contribution, permet de valider le système de classes DART.
5. *DYCTALK* : notre contribution, permet de valider le système de classes DYCRA.
6. *MIDYCTALK* : notre contribution, permet de valider le système de classes DYCRA-II. Celui-ci utilise le modèle classe/méta-classe du langage SMALLTALK-80 et ne peut pas outiller le choix local du type d'adaptation.
7. *MXDYCTALK* : notre contribution, permet de valider le système de classes DYCRA-II. Celui-ci utilise le modèle classe/méta-classe du langage METACLASSTALK, ce qui lui permet d'outiller également le choix local du type d'adaptation.

1.2.3 Domaines

Par ailleurs, cette thèse est de nature pluridisciplinaire et contribue à chacune des technologies standard déployées dans sa mise en œuvre de la façon suivante :

1. *Modèles objets adaptatifs* : cette étude complète le *cœur* des AOMs par une solution de conception dédiée à l'outillage de la co-évolution dynamique de structures et de procédures. Elle montre également comment ce modèle peut être implanté sous forme d'un framework orienté-objet.
2. *Architectures workflow* : cette étude a permis de compléter le système Micro-workflow (à notre connaissance l'unique architecture conçue pour la création par les programmeurs objets de systèmes de gestion de workflow et des logiciels *flow-independent*) afin qu'il permette avantageusement la définition dynamique et l'activation de procédés par des experts.
3. *Programmation par des experts* : cette thèse contribue aussi aux travaux sur la programmation par des experts en outillant la création systématique d'un certain type de logiciels conçus pour être programmables par des experts (langages d'experts).
4. *Réflexion dans les langages à objets* : cette thèse contribue également aux travaux sur la réflexion dans les langages à objets en montrant que les langages réflexifs permettent une meilleure mise en œuvre de l'adaptation.
5. *Programmation par composants* : cette thèse contribue aux recherches sur la programmation par composant en fournissant un système de classes, DART, qui utilise un formalisme visuel dans la conception elle-même de la composition. De plus, il permet la composition dynamique de procédures qui elles-mêmes peuvent être définies dynamiquement. Finalement, il intègre de façon systématique la composition dans le contexte de l'adaptation, que nous montrons par ailleurs comme une technique dédiée à la création de langages d'experts avec les avantages que cela procure (adaptation dynamique de logiciels par des experts).
6. *Langages à prototypes* : et enfin, à travers la nouvelle notion de *classe autonome*, notre travail ébauche une technique qui permet une réconciliation entre prototypes et abstractions (au sens de Jacques Malenfant dans [Mal97]). Plus de détails à ce sujet sont fournis dans la section suivante (Perspectives).

2 Perspectives

La recherche présentée dans ce mémoire peut être poursuivie suivant plusieurs axes. Nous décrivons ici ceux qui nous semblent les plus importants et plus particulièrement les classes autonomes (cf. §2.1, cette même page), les rapports entre les techniques AOMs et la Réflexion (cf. §2.2, page 213) ainsi que la méthodologie de développement des langages d'experts (cf. §2.3, page 217).

2.1 Langages d'experts et langages à prototypes (Classes Autonomes)

Le but de cette section est d'approfondir le concept d'adaptation protoypique et l'outillage de sa mise en œuvre à travers les classes autonomes, dans le but de tenter une généralisation de son usage à travers une comparaison avec les prototypes [Lie86, MC93, DMB98].

2.1.1 Motivations

Dans son analyse du problème de *particularisation des énoncés généraux*, ou la *généricité paramétrique* connue aussi sous le nom de polymorphisme en théorie des types, Jean-François Perrot [Per98] précise qu'il s'agit de trouver le moyen d'exprimer des informations à caractère général et de les mettre en œuvre dans un cas particulier sans avoir à les reformuler.

Il observe ensuite que la réponse des langages à objets à ce problème est le choix de la *classe* comme un énoncé général, dont les *instances* sont autant de particularisations ; de même le lien *d'héritage* permet de passer d'une classe plus générale à une autre plus particulière.

Jean-François Perrot affirme que *dans nos langues "naturelles" ces deux mécanismes ne se distinguent pas* : nous utilisons indifféremment les deux expressions *un chien est un mammifère* et *Médor est un chien*. Or, la *distinction stricte* entre classes et instances nous impose de traduire ces deux énoncés par deux mécanismes différents, le premier par un lien de spécialisation entre les classes chien et mammifère, le second par un lien d'instanciation : l'individu *Médor* est instance de la classe *chien*[130].

Il conclue alors que devoir séparer dans la programmation (par objets) des énoncés que la langue naturelle confond est une contrainte qui peut être jugée salutaire ou insupportable suivant les buts poursuivis ; l'approche objet l'accepte.

Or, le choix de la technique de particularisation basée sur la création d'instance s'avère ne pas être *toujours* approprié dans le cas de langages d'experts. En effet, ce choix conduit à la création d'instances terminales qui sont dépourvues de la capacité d'adaptation. Il y a donc une rupture potentielle dans le cycle de raffinement du modèle objet, situation que la règle de continuité proposée au début de ce chapitre nous suggère d'éviter.

Aussi, nous avons montré dans ce mémoire, à travers l'étude de l'adaptation prototypique, que lorsque ce choix des langages à objets est remis en cause, les langages d'experts se procure une propriété intéressante, celle de permettre "d'instancier" une classe tout en assurant la *continuité* dans le cycle de raffinement du modèle objet concerné.

Nous décrivons ici comment nous avons mis en œuvre sur le plan technique cette remise en cause du choix des langages à objets. Nous en étudions ensuite des conséquences par rapport au sujet de la "réconciliation entre les abstractions et les prototypes".

2.1.2 Mise en œuvre

De notre point de vue, la clé de la solution à ce problème se trouve également dans l'analyse ci-dessus de Jean-François Perrot. En effet, dans la mesure où le choix des langages à objets s'avère ne pas être de caractère obligatoire, une solution serait alors de changer ce choix et d'adopter, en quelque sorte, la voie choisie par les langues naturelles, c'est-à-dire ne pas utiliser deux mécanismes distincts d'instanciation

[130] Les soulignements qui figurent sur le texte de ces citations ne font pas partie du texte original.

et d'héritage pour la particularisation. Plus précisément, puisque c'est l'instanciation qui cause de problèmes, il s'agit de la supprimer et de rendre l'héritage systématique.

De plus, ce choix doit pouvoir s'opérer localement, c'est-à-dire *uniquement* au niveau des abstractions où le besoin s'en fait sentir. Il faut également que ce mécanisme soit réversible, c'est à dire pouvoir revenir à tout moment vers le mode de fonctionnement "normal", celui des langages à objets.

Comme permet de l'illustrer la Figure 93 ci-dessous, ce cahier des charges peut relativement facilement être mis en œuvre grâce au frameworks MXDYCTALK.

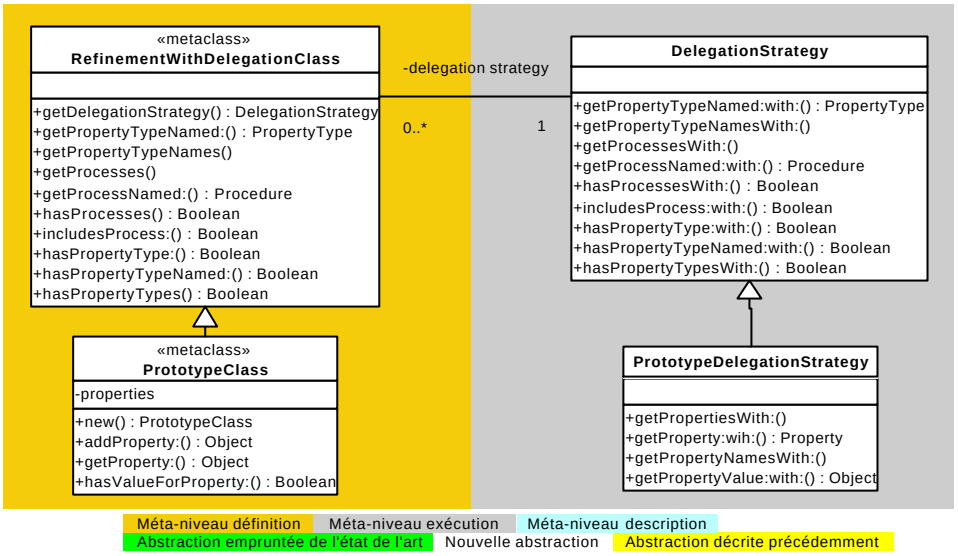


Figure 93 : Implantation des Classes Autonomes en MXDYCTALK.

En effet, on peut produire le comportement souhaité en spécialisant la méta-classe `RefinementClass` de la façon suivante :

1. redéfinir la méthode `new` afin de remplacer la création d'instances terminales par une particularisation basée sur l'héritage.
2. ajouter la variable d'instance de classe `properties` ainsi que les méthodes de gestion associées (cf. Tableau 10, page 186) afin de permettre à chaque classe issue de cette méta-classe d'avoir ses propres valeurs d'attributs.

C'est la méta-classe `PrototypeClass` qui met en œuvre ces deux changements. Sa particularité réside donc dans le fait qu'elle comporte la structure et le comportement nécessaire à la gestion des *valeurs*. Elle surcharge, par ailleurs, la méthode `new` afin de procéder à la création d'une classe, sous-classe du receveur de ce message, au lieu de créer une instance terminale au sens de langages à objets. Elle gère également une stratégie de délégation que nous décrivons à la fin de cette sous-section.

Cette mise en œuvre remplit le cahier des charges prévu dans la mesure où :

1. l'instanciation classique au sens des langages à objets disparaît et laisse sa place à une nouvelle forme de création d'instance basée sur l'héritage. Cette technique a l'avantage que les "instances" ainsi produites sont adaptables.
2. le choix de ce type de comportement peut s'effectuer localement. En effet, `PrototypeClass` étant intégrée au sein de notre outillage de l'adaptation et le framework MXDYCTALK, elle est proposée comme un type d'adaptation parmi d'autres, dont le choix peut être réalisé de façon locale à chaque classe ou adaptation (cf. le chapitre V).
3. cette intégration dans MXDYCTALK implique également que lorsque ce mode de fonctionnement n'est plus souhaité il est possible d'en changer et notamment de revenir

sur celui des langages à objets, en jouant tout simplement sur le type d'adaptation (cf. également le chapitre V)[131].

Ces travaux nous conduisent vers un nouveau concept, celle de la classe autonome. Nous définissons une ***classe autonome*** comme un objet qui se comporte en même temps comme une méta-classe, une classe et une instance terminale :

1. une classe autonome est une instance terminale dans la mesure où elle comporte des valeurs pour ses attributs[132] ;
2. une classe autonome est une classe car elle comporte la définition de la structure et du comportement de ses instances, c'est-à-dire elle-mêmes ainsi que ses sous-classes (dépend aussi de la stratégie d'héritage choisie, cf. le paragraphe ci-dessous)[133].
3. une classe autonome est une méta-classe car elle peut créer des classes (en fait ses instances !).

Plus concrètement, être une classe autonome signifie ici être instance de la méta-classe `PrototypeClass`[134]. Aussi, quand une adaptation `A` est du type prototypique, alors toutes les instances de `A` sont, par défaut, également des classes autonomes et sous-classes (directe ou indirecte) de `A`. L'adaptation peut donc se réitérer à l'échelle de chaque instance terminale.

Les classes autonomes suppriment, par ailleurs, le décalage de niveau entre la représentation d'un concept et son instanciation. Ici, chaque objet dispose de sa propre représentation et donc le programmeur décrit et expérimente l'objet lui-même et non pas ses instances[135].

De plus, comme permet de le constater aussi la Figure 93 ci-dessus, nous avons décidé d'aller plus loin et de munir également les classes autonomes d'un mécanisme de choix des propriétés héritées de leurs super-classes. C'est pourquoi la méta-classe `PrototypeClass` hérite de la méta-classe `RefinementWithDelegationClass`, laquelle hérite de la méta-classe `RefinementClass` décrite dans le chapitre V.

La méta-classe `RefinementWithDelegationClass` ajoute à cet ensemble la gestion d'une stratégie de délégation, suivant le schéma de conception *Strategy*. La stratégie par défaut est celle des langages à objets. Elle est mise en œuvre par la classe `DelegationStrategy`. Le rôle de celle-ci est de déterminer la façon dont chacune des instances de la méta-classe `RefinementWithDelegationClass` (qui sont donc des classes) hérite les définitions des attributs et procédures de leurs super-classes.

---

131 Notre outillage permet donc de passer d'une représentation *concrète* d'un concept à une représentation *abstraite* et *vis versa*. Cela veut plus concrètement dire qu'une classe autonome, qui correspond plutôt à une représentation *concrète* d'un concept, peut devenir sur le plan pratique, la super-classe d'une représentation dite *abstraite* ! C'est par exemple, la situation dans le modèle de la Figure 90, page 193, où l'adaptation `PEP` hérite de la classe autonome `PEP-N`. Une question qui nous semble ici se poser consiste à savoir si cette distinction stricte entre les représentations *concrète* et *abstraite* d'un concept a toujours du sens ? Existe-il en pratique une frontière si nette entre un *être* et ses particularisations et par conséquence entre la représentation informatique d'un *être* et celle de ses particularisations ? Ne résulte pas cette distinction de nos difficultés techniques d'assumer une représentation informatique adéquate de *l'Univers* ? Cette étude semble montrer (et elle n'est sûrement pas la seule) que le modèle de la programmation par objets n'est pas en mesure d'apporter une réponse satisfaisante à cette difficulté majeure. Le modèle des langages d'experts, incluant les classes autonomes, nous semble, toutefois, contribuer à réduire ses rigidités et manquements.

132 Les adaptations se rapprochent ainsi sur le plan structurel des *frames* [Min75] dans la mesure où chaque adaptation peut être vue comme un ensemble de couples "nom d'attribut-ensemble de facettes". La facette valeur de l'attribut est stockée dans la variables d'instance de classe `properties` et les autres facettes comme le fait que l'attribut est effectivement défini, ou son type, etc. sont stockées dans la variable d'instance de classe `propertyTypes`.

133 Mais en fait ces *choses*-là ne sont jamais instanciées car elles sont leur propre instance. La méthode `new` est redéfinie pour donner lieu à une nouvelle sous-classe au lieu de créer une instance terminale.

134 Il convient de noter que UML ne dispose pas de représentation graphique pour les classes autonomes. En effet, la sémantique des classes n'est pas ici identique à celle des langages à objets et donc celle d'UML.

135 Dans la mise en œuvre présentée ici, le *lookup* se fait en réalité dans la méta-classe de la classe autonome, mais du point de vue de programmeur/expert c'est l'objet lui-même qui comporte ses propres définitions.

La classe `PrototypeDelegationStrategy` s'associe à la méta-classe `PrototypeClass` et vient enrichir sa super-classe `DelegationStrategy` par la gestion de la stratégie d'héritage des valeurs des attributs de chaque classe autonome.

Comme nous allons le montrer dans la sous-section suivante, l'ajout de la notion de stratégie de délégation rend les classes autonomes comparables aux prototypes[136].

2.1.3 Rapprochement entre langages d'experts et langages à prototypes

Ch. Dony, J. Malenfant et D. Bardou [DMB98] définissent un *prototype* comme "un représentant typique (distingué) d'un concept, d'une famille ou d'une catégorie d'entités" [DMB98]. Dans le même article, il précisent également que : "la notion de prototype apparaît d'abord dans les travaux sur la représentation de connaissances. Ces derniers évoquent la "nécessité de formalismes de représentation en mesure de prendre en compte des connaissances qui se décrivent mal (dans d'autres formalismes) comme la typicalité, les valeurs par défaut, les exceptions, les informations incomplètes ou redondantes [BW77] [MNCLT89]".

Ces auteurs ajoutent que :

> La caractérisation très générale et informelle des langages à prototypes est relativement aisée : ce sont des langages dans lesquels on trouve en principe une seule sorte d'objets dotés d'attributs et de méthodes, trois primitives de création d'objets : création *ex nihilo*, *clonage* et *extension* (ou *description/création différentielle*), un mécanisme de calcul, l'envoi de message, intégrant un mécanisme de *délégation*.
>
> La délégation correspond à l'héritage entre objets. Elle est différente de l'héritage entre classes et a ses propres applications. La délégation entre objet induit un partage des attributs ou des valeurs de ces attributs entre objets. Ces formes de partage, absentes en programmation par classes, sont à la base de possibilités originales de représentation. Elles sont aussi la cause de problèmes d'inter-dépendance entre objets, ou de gestion d'entités représentées par des ensembles d'objets. [DMB98]

Nous estimons qu'il y a ici un rapprochement possible entre les concepts et outils développés dans ce mémoire et ceux relatifs aux langages à prototypes.

En effet, la notion d'adaptation qui, d'ailleurs, cherche ses origines dans la notion de complément de classe, *alias* objet-typique[137] (cf. le chapitre II), et plus particulièrement la notion d'adaptation prototypique, sont comparables à celle de prototype.

Tout d'abord, sur le plan objectif, les deux cherchent à apporter une réponse au besoin de la représentation informatique "fine" de connaissances. De plus, une classe autonome remplit le cahier des charges des prototypes de la manière suivante :

1. une classe autonome est par définition une sorte de classe et donc un objet doté d'attributs et de méthodes.
2. une classe autonome peut aussi bien être un objet concret ou représentant *moyen* d'un concept. En effet, toute classe autonome peut comprendre des valeurs qui lui sont propres pour ses attributs. Dans ce cas, il est un objet concret. De plus, il peut être raffinée davantage par l'envoi de message `new`. Nous disposons donc en même temps d'un moyen

[136] L'essence de la programmation par prototypes se résume principalement au partage de valeurs entre objets concrets, notion absente du modèle à classes. Comme nous l'avons démontrer, ce genre de partage, dans la mesure où il est domestiqué par une notion d'objet morcelé, résout des problèmes réels de façon efficace et conceptuellement irréprochable. L'objet morcelé devient l'unité de représentation indivisible, seule détentrice du moi (*self*), alors que les morceaux, entre lesquels existent des liens de délégation, représentent des perspectives ou points de vue sur cette entité. [Mal97, page 138]

[137] De l'anglais *type object* [RTJ00].

qui sert naturellement à "exprimer que deux concepts partagent certaines caractéristiques" et aussi à exprimer des spécificités d'un "objet concret".

3. une classe autonome permet facilement de mettre en œuvre la copie différentielle à l'aide d'une stratégie d'héritage appropriée.

2.1.4 Conséquences : vers une "réconciliation entre les abstractions et prototypes"

Les résultats décrits ci-dessus sont obtenus grâce à deux éléments :

1. en reconsidérant la "distinction stricte entre classe et instance" suivant les observations de Jean-François Perrot [Per98] ;
2. en utilisant le framework MXDYCTALK et les mécanismes comme le choix local du type d'adaptation et la stratégie de délégation.

DYCRA semble donc être un système de classes qui est fondamentalement basé sur deux modèles de représentation de connaissances : les classes comme modèles de description par *intention* et les prototypes comme modèles de description par *extension*[138]. Les classes interviennent pour la représentation du cœur de connaissances métier. Les adaptations de façon générale et les classes autonomes plus particulièrement sont ensuite utilisées afin de particulariser ce modèle pour des cas d'application particuliers et des besoins locaux.

Nous sommes donc *en présence des abstractions mais aussi des prototypes*. Cela nous permet notamment d'éviter un problème majeur qui se pose dans le cas des langages à prototypes, c'est celui de manque de moyen pour le *partage de propriétés commune à une famille d'objets* lié à la disparition de la possibilité de description en compréhension des concepts, tout en assurant un cycle de raffinement continu du modèle objet du domaine concerné.

[138] Selon Ch. Dony & al. [DMB98], différents modèles de la notion de *concept* ont été proposés :
"Un de ces modèles, fondé sur la théorie des ensembles considère qu'à chaque concept correspond une collection d'entités (extension) et que chaque concept admet une définition qui caractérise son "essence" et définit les conditions nécessaires et suffisantes à l'appartenance d'une instance à ce concept (intension). La relation qui lie une instance à un concept s'y apparente à la relation ensembliste d'appartenance. Par ailleurs, la relation qui lie un concept plus spécifique à un concept plus général s'y apparente à la relation d'inclusion. Ces auteurs concluent que ce modèle de concepts conduit à une mise en œuvre basée sur les classes, par ailleurs très critiquées en raisons notamment de leur rigidité à l'égard de leurs instances.

Un autre modèle (développé en linguistique) permet de ne pas valuer systématiquement toutes les caractéristiques d'une instance. Il y a toujours des conditions nécessaires et suffisantes pour l'appartenance à un concept, mais on s'accorde la possibilité de ne pas savoir : on sait qu'une instance appartient à un concept, qu'elle n'y appartient pas, ou bien on n'en sait rien. Ces auteurs considèrent que la "théorie des prototypes" est une extension de cette approche dans laquelle la relation d'appartenance est une certaine relation de ressemblance plus ambiguë. Dans cette théorie, *les concepts ne sont décrits ni en intension ni en extension mais indirectement au travers de prototypes de concepts*, c'est-à-dire d'exemples. Cette théorie découle du principe selon lequel l'humain se représente mentalement un concept, identifie une famille d'objets et mène des raisonnements sur ses membres en faisant référence, au moins dans un premier temps, à un objet précis, typique de la famille. Selon ces auteurs, on trouve aussi dans la théorie des prototypes la notion de description différentielle qui désigne la possibilité de décrire un nouveau représentant du concept via l'expression de ses différences par rapport à un représentant existant."

Il semble donc qu'il y ait ici un pas vers un nouveau modèle de langage qui *réconcilie prototypes et abstractions*, un thème estimé comme crucial de la recherche à venir par Jaques Malenfant :

> La programmation centrée sur les objets concrets vient du refus de s'enfermer dans l'alternative prototypes/classes. Elle propose au contraire de réconcilier prototypes et abstraction dans de nouveaux types de langages dont le modèle de programmation reste basé principalement sur la création d'objets concrets. Bien qu'incluant des abstractions, ces langages ne forcent pas le programmeur à penser exclusivement en leurs termes. Nous croyons que l'élargissement du domaine d'application de la programmation par prototypes passe par la réintroduction de l'abstraction sans toutefois lui redonner toute la place. Un thème crucial de la recherche à venir sur cette approche sera l'expérimentation systématique de ces nouveaux modèles de programmation. [Mal97, page 137]

Le modèle de programmation issue des travaux présentés dans ce mémoire nous semble s'inscrire dans ce cadre et mériterait donc une exploration plus approfondie.

2.2 AOMs et Réflexion

Les travaux présentés dans ce mémoire nous conduisent à observer des ressemblances entre les techniques "réflexives" et les AOMs. Cette observation a également été confirmée par des résultats d'un premier workshop sur le sujet [YR00a, YR00b]. Nous estimons donc important d'explorer davantage cette dimension.

Nous présentons ici un rapprochement préliminaire réalisé sur l'exemple concret du système METACLASSTALK que nous comparons avec le framework MXDYCTALK. Nous nous intéressons aux techniques utilisées pour rendre les envois de messages explicites.

2.2.1 Exemple de l'explicitation des envois de messages

Les deux systèmes METACLASSTALK et MXDYCTALK rendent explicite les envois de message. Cette explicitation constitue une caractéristique fondamentale de chacun de ces deux systèmes. Toutefois, dans le cas de METACLASSTALK cette fonction est remplie de façon implicite, par l'usage d'un compilateur spécialisé. En ce qui concerne MXDYCTALK, il préfère l'usage d'un mécanisme dédié à la création d'applications objets *flow-independent* (le Micro-workflow [Man00] et aussi DART).

Nous comparons ici brièvement ces deux techniques et concluons sur les avantages, *a priori,* de celle mise en œuvre dans le cas des AOMs.

2.2.1.1 Cas de MetaclassTalk

Comme le précise Noury Bouraqadi, "les travaux de Pattie Maes [Mae87a, Mae87b] ont contribué à introduire la réflexion dans les systèmes et langages à objets." [Bou99, page 17]

Un des principes fondamentaux du fonctionnement de la réflexion dans ce contexte est de créer les systèmes dits à méta-niveaux où un niveau N est muni de mécanismes qui lui permettent d'observer (*introspection*) le fonctionnement du niveau N-1 et, le cas échéant, d'intervenir (*intersession*) sur la définition ou le déroulement des activités de ce niveau [BGW93].

L'application de ce principe dans le cas du système METACLASSTALK prend la forme d'usage d'un MOP conjugué à un compilateur spécifique. Ce dernier transforme *implicitement* le code écrit par les programmeurs en des appels *explicites* aux méthodes du MOP. C'est ainsi que ce système offre au niveau N le droit de regard sur ce qui se passe au niveau N-1. Cet ensemble constitue alors un certain mécanisme (primaire) *d'auto représentation* [MNCLT89, page 495].

Le code ainsi généré se caractérise par la re-direction vers le niveau "méta" des envois de message à l'aide des méthodes du MOP de METACLASSTALK.

A titre d'exemple, la méthode SMALLTALK présentée par la Figure 94 ci-dessous et écrite par un programmeur, est automatiquement transformée par le compilateur spécifique de METACLASSTALK en code illustré par la Figure 95.

```
accrueDailyInterest
      | interest interestRate |
      interestRate := self getType getInterestRate.
      interest := self interestCalculator
                        calcDailyInterest: self getBalance
                        with: interestRate.
      self deposit: interest
```

Figure 94 : Exemple d'une méthode SMALLTALK écrite par un programmeur.

```
DoIt
      | t1 t2 |
      t2 _ self metaobject
                        send: #getInterestRate
                        from: self
                        to: (self metaobject
                                    send: #getType
                                    from: self
                                    to: self
                                    arguments: #()
                                    superSend: false
                                    originClass: nil)
                        arguments: #()
                        superSend: false
                        originClass: nil.
      t1 _ self metaobject
                        send: #calcDailyInterest:with:
                        from: self
                        to: (self metaobject
                                    send: #interestCalculator
                                    from: self
                                    to: self
                                    arguments: #()
                                    superSend: false
                                    originClass: nil)
                        arguments: (Array with: (self metaobject
                                          send: #getBalance
                                          from: self
                                          to: self
                                          arguments: #()
                                          superSend: false
                                          originClass: nil)
                                    with: t2)
                        superSend: false
                        originClass: nil.
      self metaobject
            send: #deposit:
            from: self
            to: self
            arguments: (Array with: t1)
            superSend: false
            originClass: nil
```

Figure 95 : METACLASSTALK : transfert *implicite* de contrôle au niveau "méta".

N. Bouraqadi résume son approche de la façon suivante : "METACLASSTALK utilise les méta-classes explicites comme support de la réflexion comportementale. En effet, les méta-classes définissent les propriétés des classes et en particulier l'utilisation des classes comme méta-objets. Ainsi, les méta-classes redéfinissent en particulier les méthodes du cœur du MOP de METACLASSTALK. Ces méthodes sont appelées implicitement pour contrôler l'exécution des programmes." [Bou99]

2.2.1.2 Cas de MXDYCTALK

MXDYCTALK met en œuvre un mécanisme que nous estimons comparable, mais utilisé dans un but différent. La démarche de MXDYCTALK est comparable à celle de METACLASSTALK car, là aussi on peut distinguer une forme d'explicitation de collaborations entre les objets.

A titre d'exemple, l'algorithme codé par la méthode de la Figure 94 s'écrit à l'aide du composant Micro-workflow de MXDYCTALK sous forme de la conséquence illustrée par la Figure 96.

```
      getInterestRate := PrimitiveProcedure
                         sends: #getInterestRate
                         to: #myAccount
                         result: #interestRate.
      getBalance := PrimitiveProcedure
                         sends: #getBalance
                         to: #myAccount
                         result: #balance.
      calcDailyInterest := PrimitiveProcedure
                         sends: #calcDailyInterest:with:
                         with: #(balance interestRate)
                         to: #myAccount
                         result: #interest.
      depositInterest := PrimitiveProcedure
                         sends: #deposit:
                         with: #interest
                         to: #myAccount.
      newBalance := PrimitiveProcedure
                         sends: #getBalance
                         to: #myAccount
                         result: #balance.
      showBalance := PrimitiveProcedure
                         sends: #show:
                         with: #balance
                         to: #myAccount.
      ^getInterestRate, getBalance, calcDailyInterest, depositInterest, newBalance,
showBalance.
```

Figure 96 : MXDYCTALK : transfert *explicite* de contrôle au niveau "méta".

Nous estimons qu'il y a là une certaine équivalence entre le code de la Figure 95 et celui de la Figure 96. En effet, les instances de la classe `PrimitiveProcedure` qui servent à la *description explicite* de ce comportement, se transforment lors de l'exécution du micro-procédé en des appels au MOP de MXDYCTALK[139].

Ce protocole comporte, à titre d'exemple, des méthodes comme `run:` et `run:withArguments:` pour assurer le contrôle des envois de message et les méthodes comme `getProperty:` et `setProperty:to:`, pour le contrôle des accès aux attributs.

[139] Rappelons que ce MOP est composé des protocoles publics des trois types d'objets méta-niveaux de DART et DARC, à savoir niveau de description, de définition et d'exécution (cf. le paragraphe § 4.2, page 139 du chapitre III : le MOP de DYCRA).

2.2.2 Avantage aux AOMs

La démarche de MXDYCTALK se différencie par rapport à celle de METACLASSTALK par le fait que ce mécanisme "d'auto-représentation" est utilisé ici pour outiller la description explicite et par les experts, des objets, leur structure et comportements.

Il n'y a pas d'intérêt, *a priori*, pour généraliser une telle approche, en vue, par exemple de la substituer à celles proposées par des langages de programmation. Les langages d'experts sont étudiés pour étendre et compléter les fonctions des langages à objets.

Toutefois, il nous semble que la technique utilisée par MXDYCTALK présente des avantages qui devraient être considérés lors de la conception des langages de programmation (réflexifs) :

1. le programmeur décide des situations où les envois de message devaient être rendus explicites. Dans ce cas il ne code plus les méthodes à l'aide du langage de programmation, mais plutôt à l'aide du mécanisme de *flow-independency*, ici le Micro-workflow ou DART.
2. les outilleurs et programmeurs peuvent adapter à leurs besoins (suivant des techniques classiques de la programmation par objets) le framework qui met en œuvre cette technique. Cette adaptation est, toutefois, contrôlée par la conception du framework qui ne l'autorise qu'à travers des points d'extension précis.
3. le programmeur maîtrise la description réifiée des collaborations et peut notamment les éditer à l'aide d'outils spécialisés. A notre connaissance ceci n'est actuellement pas le cas des méthodes encapsulées de METACLASSTALK.

Nous fournissons donc ici un exemple concret de l'intérêt d'une telle implantation de systèmes réflexifs (considérant MXDYCTALK assimilable à un système réflexif au sens des architectures à méta-niveaux) "qui inciterait les utilisateurs à l'utiliser et à la modifier" [Mal97, page 81] (cf. également le mot de la fin, le §3, page 220 du même chapitre).

2.3 Méthodologie de développement

La mise en œuvre de la solution que nous proposons ici conduit à une nouvelle méthodologie de développement d'applications objets qui met en œuvre de façon conjuguée la spécialisation et l'adaptation et qui introduit des niveaux supplémentaires d'intervention. Celle-ci est schématisée par la Figure 97 ci-dessous.

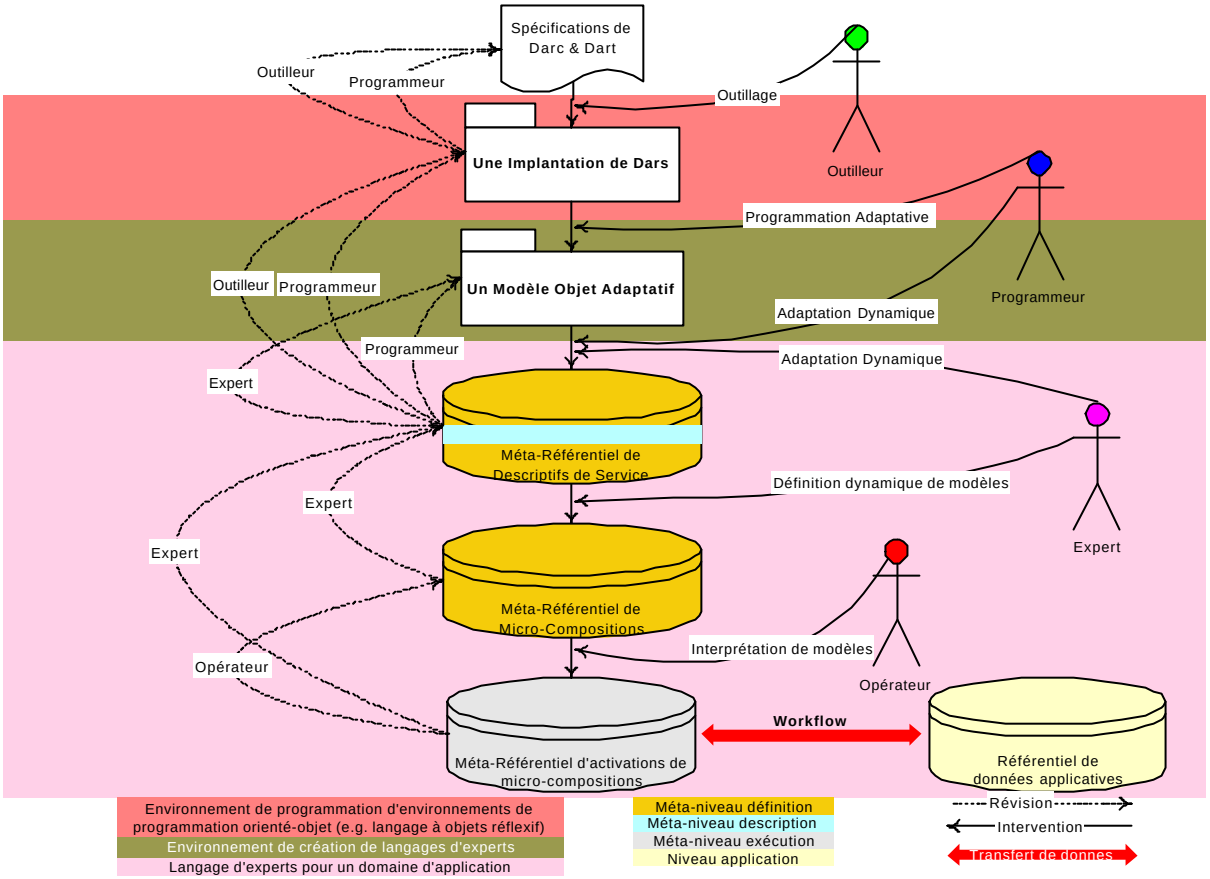


Figure 97 : Méthodologie de développement de logiciels par un langage d'experts.

Le processus de développement commence ici par l'intégration dans un langage à objets réflexif (rectangle rouge sur la Figure 97) d'une implantation appropriée des spécifications du système de classes DYCRA que nous fournissons ici[140]. Implanter ces spécifications relève de la responsabilité de programmeurs expérimentés, outilleurs. Cette réalisation conduit à la création d'un environnement dédié à la création de langages d'experts (rectangle vert sur la Figure 97).

La suite des travaux, en ce qui concerne les programmeurs, se réalise dans ce contexte et consiste, principalement, à développer le logiciel suivant la démarche habituelle. Toutefois, les concepts dédiés à l'adaptation doivent suivre un mode d'emploi particulier. Nous avons montré dans ce mémoire que ce mode d'emploi peut être plus ou moins contraignant, suivant les facilités mise à la disposition des

[140] Un usage du type réflexif d'un framework dédié à l'adaptation pour rendre lui-même adaptable est théoriquement envisageable. Nous ne l'avons toutefois pas étudié de près car nous estimons qu'un tel usage rendrait une implantation de nos spécifications moins accessibles aux programmeurs désireux de l'adapter à un besoin particulier suivant les techniques habituelles de la spécialisation de frameworks.

outilleurs par le langage à objets hôte (cas de DYCTALK, MIDYCTALK et MXDYCTALK). Ce travail donne lieu à la création d'un langage d'experts dédié à un domaine d'application (rectangle rose sur la Figure 97).

Lors de cette implantation les programmeurs doivent notamment veiller à fournir aux experts les descriptifs de service nécessaires à la composition. Ensuite, c'est la troisième catégorie d'intervenants qui entre en jeu. Il s'agit des experts qui procèdent en deux temps :

1. compléter le référentiel de descriptifs de service par l'ajout de nouvelles adaptations, leur structure ;
2. créer des procédures pour les adaptations par la composition dynamique d'instances des descriptifs de services. Cette composition peut à son tour donner également lieu à l'enrichissement du référentiel de descriptifs de service.

Ces actions conduisent à la création d'une famille de logiciels similaires et/ou à l'adaptation de logiciels existants.

Enfin, les utilisateurs (ou les opérateurs, selon la Figure 97) créent des objets et exécutent les procédures définies par des experts.

En résumé, sur le plan méthodologique, les éléments suivants semblent jouer un rôle important dans la réalisation de ce type de projets :

1. un contact régulier avec les utilisateurs afin de recueillir leurs observations et évaluer de façon objective la progression des travaux.
2. le *refactoring* [FBBOR99] systématique du code.
3. une architecture à base de frameworks et un effort de généralisation en regard des exemples concrets.
4. des développements itératifs et incrémentaux.

L'explicitation des propriétés de cette nouvelle méthodologie nécessite également une étude systématique. Nous estimons que ce travail entre dans le cadre des travaux sur les méthodologies "légères" de développement et notamment l'*eXtreme Programming* [Bec99, BF00].

2.4 D'autres pistes de recherche

Ce travail fournit également un cadre technique précis, documenté et validé pour de futures explorations au sujet du rapprochement des structures de représentation et d'exécution de programmes par les langages à objets et les langages d'experts.

Nous avons ici étudié longuement la représentation des adaptations par rapport à celle des spécialisations comme l'élément central d'une solution qui assure le travail collaboratif entre les experts et les programmeurs. Il reste tout de même à en faire autant en ce qui concerne les différentes formes de définition de structures et de comportements.

Améliorer l'expressivité de langages d'experts constitue une autre piste de recherche importante. En effet, nous avons pour l'heure outillé uniquement l'expression de procédures. L'intégration dans DYCRA d'un moteur d'inférence comme NEOPUS [Pac92, Pac94, PP94, Pac95] devrait faciliter l'expression par des experts de certaines règles métier complexes.

Ce système peut également et surtout servir à générer automatiquement des adaptations (structures et procédures) à partir d'un ensemble de règles qui décrivent les principes d'une telle génération. Un tel usage peut permettre la création de logiciels ayant un potentiel *d'adaptation* structurelle et comportementale *automatique.*

Il est important de noter que dans de telles situations la base de règles en question peut-elle même être définie ou générée dynamiquement. Cela donc permet d'atteindre un niveau d'adaptabilité conséquent et surtout contrôlé par un framework dédié.

Dans cet ordre d'idées, le framework exposé dans ce mémoire nécessite également un effort plus important au sujet de la gestion du *lien causal.* En effet, nous fournissons ici des systèmes de classes qui modélisent la définition et aussi l'instanciation de structures et de procédures. Il faut à présent exploiter cette fondation en mettant en œuvre des outils qui assurent la gestion synchronisée des modifications des types d'objets et leur instanciations.

Une dernière piste que nous souhaitons évoquer ici concerne l'étude comparative des mécanismes proposés dans ce mémoire et la notion de *Web Services*, promue par *World Wide Web Consortium* (W3C). Un modèle comme DYCRA nous semble être en mesure d'apporter une réponse aux préoccupations comme la création d'applications par la composition dynamique de services[141].

[141] Cf. notamment *Web Services Description Language (WSDL)* (http://www.w3.org/TR/wsdl),

3 Mot de la fin

Jaques Malenfant constate que

> " SMALLTAK n'a pas été conçu avec l'objectif de rendre cette organisation (réflexive) si manifeste qu'elle inciterait les utilisateurs à l'utiliser et à la modifier... " [Mal97, page 81].

Sur la base de cette étude, nous estimons que ce constat est valable dans le cas général de la création de langages de programmation où la définition de la sémantique des langages n'a pas su véritablement mettre en exergue le principe fondamental qui régit le fonctionnement des ordinateurs, c'est à dire le *déroulement des traitements sur des structures de données*. S'il y a un consensus sur l'idée générale de ce principe, sa mise en œuvre n'a pas été suivie de rigueur.

Les efforts de la communauté "réflexion", malgré leur importance pour mettre en évidence l'intérêt de l'accessibilité des structures qui définissent la sémantique des langages, n'ont pas su véritablement contribuer à *expliciter et documenter les* systèmes de classes *qui régissent la mise en œuvre de cette sémantique*.

De ce fait, la nature des concepts mis en jeu dans *l'implantation* de la sémantique de langages et les relations entre eux restent dans la plupart des cas un *mystère*. Il y a eu, en effet, à notre connaissance très peu de travaux dans ce domaine et cela de façon isolée [Lor97, Yea97, Nob98]. Or, le travail communiqué dans ce mémoire fournit un nouvel exemple de l'intérêt de techniques de génie logiciel pour documenter ce types d'architectures réflexives et complexes. Nous estimons que c'est une meilleure approche pour la définition et la documentation des implantations de langages qui vise, par définition, à inciter les programmeurs à "l'utiliser et à la modifier".

Aussi, il nous semble important de poursuivre de façon plus systématique les travaux sur *la définition des schémas de conception des implantations des langages de programmation.*

Références

Références bibliographiques

[AB01] G.Arevalo et I.Borne.
Architectural Description of Object-Oriented Frameworks: An Approach.
Actes de LMO'01, l'Objet vol. 7 n. 1-2/2001, éditions Hermes, 2000.

[ABW98] Sherman R. Alpert, Kyle Brown et Bobby Woolf.
The Design Patterns Smalltalk Companion.
Software Patterns Series. Addison Wesley, 1998.

[AJ98] Francis Anderson et Ralph Johnson.
The Objectiva telephone billing system.
MetaData Pattern Mining Workshop, Urbana, IL, May 1998.
URL: *http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html*.

[And98] Francis Anderson.
A collection of history patterns.
In Proc. 5th Pattern Languages of Programming, Monticello, Illinois, August 1998.
URL! http://st-www.cs.uiuc.edu/~plop/plop98.

[Ars00] Ali Arsanjani.
Rule Object: A Pattern Language for Pluggable and Adaptive Business Rule Construction.
Proceedings of PLoP2000. Technical Report #wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science, October 2000.
URL: *http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf*.

[Ars01] Ali Arsanjani.
Using Grammar-oriented Object Design to Seamlessly Map Business Models to Component–based Software Architectures.
Proceedings of The International Association of Science and Technology for Development, 2001, Pittsburgh, PA.
URL: *http://www.mum.edu/cs_dept/aarsanjani*.

[ASM99] Guide d'utilisation de Prélude INSPECTION --- Atelier de Spécialisation Métier.
Matra Datavision, 1999.
URL: *http://www.matra-datavision.com/*.

[Bec99] Kent Beck.
Extreme Programming Explained: Embrace Change.
Addison-Wesley Pub Co; ISBN: 0201616416
URL: *http://www.xprogramming.com/*

[BF00] Kent Beck et Martin Fowler.
Planning Extreme Programming.
Addison-Wesley. ISBN: 0201710919.

[BLR98] Noury Bouraqadi-Saadani, Thomas Ledoux et Fred Rivard.
Safe Metaclass Programming.
Proceedings of OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 84-96, October 1998.

[Bos00] Jan Bosch.
Design and Use of Software Architectures, Adopting and evolving a product-line approach.
Addison-Wesley, Software Engineering/Software Architecture Series. Addison-Wesley 2000. ISBN 0-201-67494-7.

[Bou99a] Noury Bouraqadi-Saadani.
Un cadre Réflexive pour la Programmation par Aspects.
Publication à LMO'99, Hermés. Janvier 1999, Villefranche sur Mer, France.

[Bou99b] Noury Bouraqadi-Saadani.
Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.
Thèse de l'Université de Nantes. Ecole des Mines de Nantes, le 13 juillet 1999.

[BC87a] Jean-Pierre Briot et Pierre Cointe.
A Uniform Model for Object-Oriented Langauges Using the Class Abstraction.
In Proceedings of the Tenth Internationl Joint Conference on Artificial Intelligence (IJCAI'87), Vol. 1, pages 40-43,Milan, Italy, August 1987.

[BC87b] Jean-Pierre Briot et Pierre Cointe.
Implementing the ObjVlisp model in Smalltalk-80 or making the Smalltalk-80 metaclasses explicit.

[BO87] A. Borning et T. O'Shea.
Deltatalk : An empirically and aesthetically motivated simplification of the smalltalk-80 language.
In Proceedings of ECOOP'87, number 276, pages 1--10. LNCS, June 1987.

[BR99] Isabelle Borne et Nicolas Revault
Comparaison d'outils de mise en oeuvre de Design Patterns.
Revue L'Objet, Volume 5, n° 2 - numéro spécial sur les "patrons de conception", pp. 243-266, 1999.
URL : *http://www.emn.fr/borne/publi.html*.

[Briot84] Jean-Pierre Briot.
Instanciation et Héritage dans les Langages Objets.
Thèse de 3ème cycle. Université Paris 6. 15 Déc. 1984. Rapport de Recherche LITP 85-21, Mai 1985.

[BSLRC96] Noury Bouraqadi-Saadani, Thomas Ledoux, Fred Rivard, et Pierre Cointe.
Providing Explicit Metaclasses In Smalltalk.
In OOPSLA'96 workshop "Extending the Smalltalk Language", October 1996.

[CG01] Claus Gittinger,
Smalltalk/X.
URL: *http://www.exept.de/*

[Cin01] Cincom visualworks.
Cincom Systems, Inc.
URL: *http://www.cincom.com/visualworks/*.

[Coi87a] Pierre Cointe.
Metaclasses are First Class : the ObjVlisp Model.
OOPSLA'87 Special Issu of SIGPLAN Notices, Vol. 22, N° 12, pp. 156-167, Orl., Fl., USA. Oct. 87.

[Coi87b] Pierre Cointe.
The ObjVLisp Kernel: A Reflexive Lisp Architecture to Define a Uniform Object-Oriented System.
In P. Maes and D. Nardi, editors, Meta-Level Architectures and Reflection, pages 155--176. North-Holland, Amsterdam, 1987.

[Coi90] Pierre Cointe.
The ClassTalk System : a Laboratory to study reflection in Smalltalk.
In Informal Proceedings of the First Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90, October 1990.

[DDHL96] H. Dicky, C. Dony, M. Huchard et T. Libourel.
On automatic class insertion with Overloading.
ACM Sigplan Notice - Proceedings of ACM OOPSLA'96, Object-Oriented Programming Languages, Systems and Applications. Volume 31, numéro 10, pages 251 à 267, San Jose, California, USA, oct. 1996.

[DES00] Frédéric Duclos, Jack Estublier et Rémy Sanlaville.
Architectures Ouvertes pour l'Adaptation des Logiciels.
Publié dans les actes de la 13ème Journées Internationales (ICSSEA 2000): Génie Logiciel & Ingénierie de Systèmes et leurs Applications. 6-8 Décembre 2000. CNAM --- Paris, France. Volume 1. Software & Systems Engineering and Applications.

[DMB98] Christophe Dony, Jacques Malenfant et D. Bardou.
Les Langages à Prototypes.
In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, Langages et Modèles à Objets : Etats des recherches et perspectives, chapter 9, pages 227-256. INRIA - Collection Didactique, 1998.

[DT98] Martine Devos et Michel Tilman.
A repository based framework for evolutionary software development.
MetaData Pattern Mining Workshop, Urbana, IL, May 1998.

[ECOOP2000] Joseph W. Yoder et Reza Razavi.
Metadata and Active Object-Model Pattern Mining Workshop.
URL: *http://www.joeyoder.com/Research/metadata/ECOOP2000/*.

[ECOOP2001] Nicolas Revault, Joseph W. Yoder et Ali Arsanjani.
Adaptive Object-Models and Metamodeling Techniques.
URL: *http://www.joeyoder.com/Research/metadata/ECOOP2001/*.

[FBBOR99] Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke et don Roberts.
Refactoring : Improving the Design of Existing Code.
Addison-Wesley Object Technology Series, August 1999.

[FBHS01] Xiang Fu, Tevfik Bultan, Richard Hull et Jianwen Su.
Verification of Vortex Workflows.
Proc. of Conf. on Tools and Algorithms for the Construction and Analysis of Systems, February, 2001.

[FD99] Ira Forman, Scott Danforth.
Putting Metaclasses to Work.
Addison-Wesley, 1999, ISBN: 0 201 43305 2.

[Fer95] Jacques Ferber.
Les Systèmes Multi-Agents.
Interéditions, 1995.

[FJ89] Brian Foote et Ralph E. Johnson.
Reflective facilities in Smalltalk–80. In Proceedings of OOPSLA'89. ACM, 1989.
URL: *http://www.acm.org/pubs/citations/proceedings/oops/74877/p327-foote/*.

[Foote88] Brian Foote.
Designing to facilitate change with object-oriented frameworks.
Master's thesis, University of Illinois at Urbana-Champaign, 1988.
URL: *ftp://www.laputan.org/pub/foote/DFC.pdf*.

[Foote90] Brian Foote.
Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?
OOPSLA/ECOOP '90 Workshop on Reflection and Metalevel Architectures. Mamdouh Ibrahim, Brian Foote, Jean-Pierre Briot, Gregor Kiczales, Satoshi Matsuoka, and Takuo Watanabe, organizers.

[Foote92] Brian Foote.
Objects, Reflection, and Open Languages.
Workshop on Object-Oriented Reflection and Metalevel Architectures (ECOOP '92).
URL: *www.laputan.org*.

[Foote91] Brian Foote.
Flexible Foundations and Movable Walls.
OOPSLA '91 Workshop on Reflection and Metalevel Architectures. Phoenix, AZ. Mamdouh Ibrahim, Brian Foote, Pierre Cointe, Gregor Kiczales, Satoshi Matsuoka, and Takuo Watanabe, organizers.

[Fow97] Martin Fowler.
Analysis Patterns--Reusable Object Models.
Addison-Wesley Object-Oriented Software Engineering Series. Addison-Wesley, 1997.

[Frost97] Frost
ObjectShare, 1997.
URL: *http://wiki.cs.uiuc.edu/VisualWorks/Frost*.

[FS97] Martin Fowler et Kendall Scott.
UML Distilled—Applying the Standard Object Modeling Language.
Object Technology Series. Addison-Wesley, June 1997.

[FSJ99] Mohamed Fayad, Douglas C. Schmidt et Ralph E. Johnson, editors.
Implementing Application Frameworks: Object-Oriented Frameworks at Work.
John Wiley & Sons, 1999.

[FY98a] Brian Foote et Joseph Yoder.
Metadata and active object-models.
Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.
URL: *http://jerry.cs.uiuc.edu/~plop/plop98*.

[FY98b] Brian Foote et Joe Yoder.
Metadata.
In Technical Report #WUCS-98-25 (PLoP '98). Dept.of Computer Science, Washington University: 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.
Design Patterns---Elements of Reusable ObjectOriented Software.
AddisonWesley, 1995.

[GHV95] Erich Gamma, Richard Helm et John Vlissides.
Design Patterns Applied, tutorial notes from OOPSLA'95.

[Gra89] Nicolas Graube.
Metaclass Compatibility.
In proceedings of OOPSLA'89,pages 305-315, New Orleans, Louisina, USA, October 1989.

[Gin90] Vincent Ginot.
Modélisation de l'évolution nychtémérale de l'oxygène dissous en étang.
Thèse de doctorat de Université Lyon-1, spécialité Biométrie, juillet 1990 N°128-90.

[GLP98] Vincent Ginot, Christophe Le Page.
Mobidyc, a Generic Multi-Agents Simulator for Modeling Populations Dynamics.
Angel P. Del Pobil, Jose Mira, Moonis Ali (Eds.): Tasks and Methods in Applied Artificial Intelligence, 11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE-98, Castellón, Spain, June 1-4, 1998, Proceedings, Volume II. Lecture Notes in Computer Science, Vol. 1416, Springer, 1998, ISBN 3-540-64574-8. Pages 805-814.

[GR83] Adele Goldberg et David Robson.
Smalltalk–80: The Language and its Implementation.
Addison-Wesley, Reading, Massachusetts, 1983.

[GS93] David Garlan et Mary Shaw.
An Introduction to Software Architecture.
Advances in Software Engineering and Knowledge Engineering, Volume I, edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993.

[HJ98] Wai Ming Ho et Jean-Marc Jézéquel.
Object-Oriented Frameworks for Distributed Systems : A Survey.
Rapport de recherche INRIA, N° 3590, Décembre 1998. ISSN 0249-6399.

[HL95] Wlater L. Hürsch et Cristina Viedeira Lopes.
Separation of Concerns.
Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[HLD99] M.Huchard and T.Libourel and C. Dony.
Evolution de hiérarchies par approches algorithmiques.
Génie des Objets, éditeur C. Oussalah, Hermes, 1999.

[IKMWK97] Dan Ingalls Ted Kaehler John Maloney Scott Wallace et Alan Kay.
Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself.
Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'97), Atlanta, GA, October 1997.
URL: *http://users.ipa.net/~dwighth/squeak/oopsla_squeak.html*.

[Ing78] Daniel H. H. Ingalls.
The Smalltalk-76 Programming System Design and Implementation.
5th ACM Symposium on POPL, pp. 9-15. Tucson, AZ, USA, January 1978.

[Ing81] Daniel H. H. Ingalls.
Design Principles Behind Smalltalk.
BYTE Magazine, August 1981.
URL: *http://users.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html*.

[Java] *http://java.sun.com/*.

[JF88] Ralph E. Johnson et Brian Foote.
Designing reusable classes.
Journal of Object-Oriented Programming, 1(2), June/July 1988.

[JO98] Ralph E. Johnson et Jeff Oakes.
The User-Defined Product Framework.
Unpublished manuscript ,1998.
URL: *http://st.cs.uiuc.edu/pub/papers/frameworks/udp*.

[Joh92] Ralph E. Johnson.
Documenting frameworks using patterns.
ACM SIGPLAN Notices, 27(10):63–76, Oct. 1992. OOPSLA'92 Proceedings, Andreas Paepcke (editor).
URL: *http://st.cs.uiuc.edu/pub/papers/HotDraw/documenting-frameworks.ps*.

[Joh97a] Ralph E. Johnson.
Components, Frameworks, Patterns.
Proceedings of the 1997 Symposium on Software Reusability (keynote address) May 1997, pages 10-17.

[Joh97b] Ralph E. Johnson.
Frameworks = (components + patterns).
Communications of the ACM, V40 N10, October 1997, pages 39-42.

[Joh98] Ralph E. Johnson.
Dynamic object model. (Work in progress).
URL: *http://st-www.cs.uiuc.edu/users/johnson/DOM.html*.

[JW97] Ralph E. Johnson et Bobby Woolf.
The Type Object Pattern.
Chapter 4, in Martin et al. [MRB97], October 1997, pp. 47-66.

[KC00] Kennedy Carter.
I-OOA Modelling Tool Technical Overview.
Kennedy Carter, 2000.
URL: *www.kc.com*.

[KdRB91] G. Kicsales, J. des Rivières et D. G. Bobrow.
The Art of Metaobjet Protocol.
MIT Press, 1991.

[Kra83] Glen Krasner, editor.
Smalltalk-80 --- Bits of History --- Wors of Advice.
Addison-Wesley, 1983.

[Kri90] Philippe Krief.
M.Pv.C - Un système interactif de construction d'environnements de prototypage de multiples outils d'interprétation de modèles de représentation,
Thèse de doctorat, Université Paris 8, Paris, 1990.

[Kri91] Philippe Krief.
ACKPRO (2.5), atelier de conception et de kits pour le prototypage.
Rapport interne ACKIA, décembre 1991.

[Kri96] Philippe Krief.
Utilisation des langages objets pour le prototypage.
MASSON, 1992. Traduction anglaise : Prototyping with Objects. Prentice Hall, 1996.

[LC96] Thomas Ledoux et Pierre Cointe.
Explicit Metaclasses As a Tool for Improving the Design of Class Libraries.
In Proceedings of ISOTAS'96, Kanazawa, Japan, March 1996. JSSST-JAIST, Springer-Verlag.

[Lie86] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in ObjectOriented Systems.
In Proceedings of 1st OOPSLA Conference, pp. 214-223, Portland, USA, 1986.
URL: *http://lieber.www.media.mit.edu/people/lieber/Lieberary/OOP/OOP.html*.

[LSGBP99] Bruno Lesueur, Gerson Sunyé, Zahia Guessoum, Gilles Blain et Jean-François Perrot.
La métaphore du dossier. INFORSID'99, 1999, France.

[Lor97] David H. Lorenz.
Tiling Design Patterns - {A} Case Study Using the Interpreter Pattern.
Proceedings of OOPSLA:97, pages 206—217, Atlanta, Georgia, USA, 1997.
URL: *http://www.ccs.neu.edu/home/lorenz/papers/oopsla97/tiling.html*.
URL: *http://www.cs.technion.ac.il/~david/Papers/Tech_Reports/tools95.PS.gz*.

[LR00] Leymann, F. et Roller, D.
Production Workflow—Concepts and Techniques.
Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[Maes87] Pattie Maes.
Concepts and experiments in computational reflection.
In Proceedings of OOPSLA'87, volume 22, pages 147--155. SIGPLAN Notices, ACM Press, Oct. 1987.

[Mal97] Jacques Malenfant.
Abstraction, encapsulation et réflexion dans les langages à prototypes.
Mémoire d'Habilitation à Diriger les Recherches, soutenu le 21 avril 1997 à l'Université de Nantes.

[Man00] Dragos Manolescu.
Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development.
PhD thesis, Computer Science Technical Report UIUCDCS-R-2000-2186. University of Illinois at Urbana-Champaign, October 2000, Urbana, Illinois.
URL: *http://micro-workflow.com/PhDThesis/*.

[Mar67] James Martin.
Design of Real-time Computer Systems.
Englewood Cliffs, NJ: Prentice-Hall. 1967.

[MC93] P. Mulet et P. Cointe.
Definition of a reflective kernel for a prototype-based language.
In Shojiro Nishio and Akinori Yonezawa, editors, First International Symposium on Object Technologies, volume 742 of Lecture Notes in Computer Science, pages 128--144, Kanazawa, Japan, November 1993. JSSST-JAIST, Springer-Verlag.

[McA95a] Jeff McAffer.
A Meta-level Architecture for Prototyping Object Systems.
PhD thesis, University of Tokyo, Japan, Spetember 1995.

[McA95b] Jeff McAffer.
Meta-level Programming wih CodA.
In the Proceedings of ECOOP'95, volume LNCS 952, page 190-214. Spring-Verlg, 1995.

[MDC92] Jacques Malenfant, Christophe Dony et Pierre Cointe.
Behavioral Reflection in a Prototype-Based Language.
In Proceedings of Int'1 Workshop on reflection and Meta-Level Architectures (Tokyo, Nov. 1992), A. Yonezawa and B. Smith, Eds., RISE and IPA (Japan) + ACM SIGPLAN, JSSST & IPS, pp. 143--153.

[Mir98] Eliot Miranda.
Meta-programming in a Flexible Component Architecture.
Metadata and Dynamic Object-Model Pattern Mining Workshop OOPSLA '98.
URL: *http://www-poleia.lip6.fr/~razavi/aom/papers/*.

[MJ98a] Dragos-Anton Manolescu et Ralph E. Johnson.
A proposal for a common infrastructure for process and product models.
In OOPSLA Mid-year Workshop on Applied Object Technology for Implementing Lifecycle Process and Product Models, Denver, Colorado, July 1998.
URL: *http://www.uiuc.edu/ph/www/manolesc/Workflow/OOPSLA98/*.

[MJ98b] Dragos-Anton Manolescu et Ralph E. Johnson.
Patterns of workflow management facility.
URL: *http://www.uiuc.edu/ph/www/manolesc/Workflow/PWFMF/*.

[MJ99a] Dragos-Anton Manolescu et Ralph E. Johnson.
A Micro Workflow Framework for Compositional Object-Oriented Software Development.
OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II, November 1999.
URL: *http://st.cs.uiuc.edu/users/manolesc/Workflow/PDF/oopsla99.pdf*

[MJ99b] Dragos-Anton Manolescu et Ralph E. Johnson.
Dynamic Object Model and Adaptive Workflow.
Metadata and Active Object-Model Pattern Mining Workshop. OOPSLA'99, Denver, USA.
URL: *http://www.uiuc.edu/ph/www/manolesc/Workflow*

[MJ99c] Dragos-Anton Manolescu et Ralph E. Johnson.
Dynamic Object-Model Workflow Framework for Developers.
URL: *http://st.cs.uiuc.edu/users/manolesc/Workflow/PDF/Framework.pdf*

[MJ00] Dragos-Anton Manolescu et Ralph E. Johnson.
A micro-workflow component for federated workflow.
OOPSLA2000 Workshop on Implementation and Application of Object-Oriented Workflow Management Systems III, October 2000.
URL: *http://micro-workflow.com/*.

[MM00] MetaModeling et Model Engineering.
URL: *http://www.metamodel.com*.

[MNCLT89] G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre.
Les langages à objets.
InterEditions, Paris, 1989.

[MRB97] Robert C. Martin, Dirk Riehle et Frank Buschmann, editors.
Pattern Languages of Program Design 3.
Software Patterns Series. Addison-Wesley, October 1997.

[MO98] James Martin et James J. Odell.
Object-Oriented Methods—A Foundation.
Prentice Hall, second edition, 1998.

[Nar93] Bonnie A. Nardi.
A Small Matter of Programming: Perspectives on End User Computing.
Cambridge. Cambridge, 1993. MIT Press.

[NJ98] Hiroaki Nakamura et Ralph E. Johnson.
Adaptive Framework for the REA Accounting Model.
OOPSLA'98 Workshop on Business Object Component Design and Implementation IV: From Business Objects to Complex Adaptive Systems.
URL: *http://jeffsutherland.com/oopsla98/nakamura.html*.

[NFX07] Norme française NF X 07 – 010. Métrologie, la fonction métrologique dans l'entreprise.
AFNOR, Octobre 1986.

[Nob98] James Noble.
Classifying relationships between object-oriented design patterns.
In Australian Software Engineering Conference (ASWEC), 1998.

[OJ98] Jeff Oakes et Ralph Johnson.
The Hartford insurance framework.
MetaData Pattern Mining Workshop, Urbana, IL, May 1998.
URL: *http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html*.

[OHE97] Robert Orfali, Dan Harkey et Jeri Edwards.
Instant CORBA.
John Wiley & Sons, 1997.

[OMG98] OMG.
Action Semantics for the UML RFP. OMG Document 98-11-01. OMG, 1998.
URL: *www.omg.org*.

[Opd92] William F. Opdyke.
Refactoring Object-Oriented Frameworks.
PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[OOPSLA98] Joseph W. Yoder, Michel Tilman, Dirk Riehle, Martin Fowler et Brian Foote.
Metadata and Active Object-Models. Workshop OOPSLA'98.
URL: *http://www.joeyoder.com/Research/metadata/OOPSLA98MetaDataWkshop.html*.

[OOPSLA99] Joseph W. Yoder, Brian Foote, Dirk Riehle, Martin Fowler et Michel Tilman.
Metadata and Active Object-Models Workshop; OOPSLA, 1999.
URL: *http://www.adaptiveobjectmodel.com/OOPSLA99*.

[OOPSLA2000] Ali Arsanjani et Joseph W. Yoder.
Best–practices in Business Rule Design and Implementation; OOPSLA, 2000.
URL: *http://www.mum.edu/cs_dept/aarsanjani/oopsla2000/business-rules.html*.

[Pac92] François Pachet.
Représentation de connaissances par objets et règles : le système NéOpus.
Thèse de l'université Paris VI. Rapport LAFORIA n. 92/30. Septembre 92. get a compressed tar directory with postscript files

[Pac94] François Pachet.
Vers un modèle du raisonnements dans les langages à objets.
Colloque "Langages et Modèles à Objets", Grenoble, octobre 1994, pp. 111-123.
URL : *http://www-poleia.lip6.fr/~fdp/NeOpus-papers.html*.

[Pac95] François Pachet.
On the embeddability of production rules in object-oriented languages.
Journal of Object-Oriented Programming, 1995. Vol. 8, N. 4, pp. 19-24.

[Per92] Jean-François Perrot.
Langages à objets et programmation par objets.
Rapport interne LAFORIA 92/34. Université Paris 6, 1992, Paris, France.

[Per98] Jean-François Perrot.
Objets, classes, héritage : définitions.
In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, Langages et Modèles à Objets : Etats des recherches et perspectives, chapter 1, pages 3-31. INRIA - Collection Didactique, 1998.

[Pit90] Jacques Pitrat.
Métaconnaissance, futur de l'intelligence artificielle.
Hermès, Paris, 1990,

[Pit95] Jacques Pitrat.
What does "itself" mean ? LAFORIA 95/28. Novembre 1995.
URL : *http://www-apa.lip6.fr/META/Itself.doc*.

[PP94] François Pachet et Jean-François Perrot.
Rule Firing with Metarules.
Software Engineering and Knowledge Engineering - SEKE '94, Jurmala, Lettonie. Knowledge System Institute Ed. pp. 322-329, 21-23 juin 1994.

[PT00] Project Technology.
BridgePoint Tutorial.
Project Technology, 2000.
URL: *www.projtech.com*.

[Raz93] Reza Razavi
TTxTalk : définition et réalisation d'un protocole de pilotage des logiciels de bureautique, pour la génération de documents techniques.
Rapport de stage de fin d'études supérieurs spécialisées. DESS Génie des Logiciels Applicatifs.
Laforia - Université Paris 6. 1993.

[Raz99] Reza Razavi.
Building an End-user-oriented Application Framework by Meta-programming -- A Case Study.
Position Paper to OOPSLA'99 Metadata and Dynamic Object-Model Pattern Mining Workshop. Nov. 1999, Denver, USA.
URL : *http://www-poleia.lip6.fr/~razavi/oopsla99/*.

[Raz00a] Reza Razavi.
Active Object-Models et Lignes de Produits -- Application à la création des logiciels de Métrologie.
OCM'2000 18 - May 2000, Nantes, France. Actes de l'OCM, pages 130-144.
URL : *http://www-poleia.lip6.fr/~razavi/ocm-2k/*.

[Raz00b] Reza Razavi.
Coupling The Core of Active Object-Models and Micro Workflow -- Foundations of a Framework for Developing End User Programming Environments.
Position paper to ECOOP '2000 workshop on Metadata and Active Object-Model Pattern Mining, June 2000, Cannes, France.
URL: . *http://www-poleia.lip6.fr/~razavi/ecoop2000/*.

[Raz00c] Reza Razavi.
Type Cube: Foundation for an Architectural Style aimed at Building Adaptive and Flow-Independent Software.
OOPSLA'2000 First Workshop on Best-practices in Business Rule Design and Implementation. October 2000 at Minneapolis, MN, USA.
URL: *http://www-poleia.lip6.fr/~razavi/oopsla2000/*.

[Raz01a] Reza Razavi.
Reusable Designs for Building Dynamically Programmable and Workflow-enabled Object-Oriented Software.
In Companion Papers of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). ACM Press, 2001. 14-18 October 2001 at Tampa, Florida, USA.
URL: *http://www-poleia.lip6.fr/~razavi/oopsla2001/poster/*.

[Raz01b] Reza Razavi.
Why Object-Oriented Languages Should Support Building Tools for Adaptive Object-Models?
Submission to ESUG'2001 1st Doctoral Symposium. Tuesday, August 28th - Friday, August 31st, Essen, Germany.
URL: *http://www-poleia.lip6.fr/~razavi/esug2001/*.

[Raz01c] Reza Razavi.
Concepts and Tools to Support Building Expert-Programmable Software.
Submission to the third Workshop on *Best-practices in Business Rule Design and Implementation OOPSLA'2001*, October 2001, Tampa, FL, USA. 14-18 October 2001 at Tampa, Florida, USA.
URL: *http://www-poleia.lip6.fr/~razavi/oopsla2001/ws/*.

[RB93] Reza Razavi et Jean-Marie Bonne.
ACKPRO-GLA : conception générale et documentation technique.
Rapports de projet DESS GLA, Laforia - Université Paris 6, 1993.

[RBP00] Nicolas Revault, Xavier Blanc et Jean-François Perrot.
On Meta-Modeling Formalisms and Rule-Based Model Transforms.
Communiction to Ecoop'2000 workshop Iwme'00, Sophia Antipolis & Cannes, France, June, 2000.

[RDREM00] Rouvellou, I.; Degenaro, L.; Rasmus, K.; Ehnebuske D. et McKee, B.
Extending business objects with business rules.
Proceedings on Technology of Object-Oriented Languages, 2000. On page(s): 238 – 249.

[RFBO01] Riehle, D., Fraleigh, S., Bucka-Lassen, D. et Omorogbe, N.
The Architecture of a UML Virtual Machine.
In Proceedings OOPSLA '01. 14-18 October 2001. Tampa, USA. ACM Press, 2001.
URL: *http://www.riehle.org/papers/2001/oopsla-2001.html*.

[Rev96] Nicolas Revault.
Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MétaGen et les frameworks).
Thèse de doctorat de l'Université P. et M. Curie, TH 96-16, pp. 315, Laforia, Université P. et M. Curie (Paris 6), Paris, France, 1996
URL : *http://www-poleia.lip6.fr/~revault/publications.html*.

[RY01] N. Revault & J. W. Yoder.
Adaptive Object-Models and Metamodeling Techniques.
In Ecoop'01 Workshop Reader, Ákos Frohner (ed), LNCS, 2001, Springer-Verlag -- Report of workshop at Ecoop'01, University Eötvös Loránd, Budapest, Hungary .
URL: *http://www-poleia.lip6.fr/~revault/references.html*.

[Riv96a] Fred Rivard.
Dynamic Instance-Class Link.
In Technical Report, Ecole des Mines de Nantes, febuary 1996.

[Riv96b] Fred Rivard.
Smalltalk: a Reflective Language.
In REFLECTION'96, pages 21--38, San Franscico, USA, April 21-23 1996. Edited by Gregor Kiczales.

[Riv96c] Fred Rivard.
A New Smalltalk Kernel Allowing Both Explicit And Implicit Metaclass Programming.
Submission to OOPSLA'96 Workshop : Extending the Smalltalk Language.

[Riv96d] Fred Rivard.
Réflexion & Langages à classes.
PhD thesis, Université de Nantes, Laboratoire Jules Verne, Ecole des Mines de Nantes & Object Technology International Inc., France, 1996.

[RJ97] Don Roberts et Ralph E. Johnson.
Evolving Frameworks—A Pattern Language for Developing Object-Oriented Frameworks.
In Martin et al. [MRB97], chapter 26, October 1997.
URL: *http://st-www.cs.uiuc.edu/users/droberts/evolve.html*.

[RJ98] Don Roberts et Ralph E Johnson.
Patterns for Evolving Frameworks.
In Pattern Languages of Program Design 3. Addison-Wesley, 1998. Page 471-486.

[Rie97a] Dirk Riehle.
A Role-Based Design Pattern Catalog of Atomic and Composite Patterns
Structured by Pattern Purpose. Ubilab Technical Report 97-1-1. Zürich, Switzerland: Union Bank of Switzerland, 1997.
URL: *http://www.riehle.org/papers/1997/ubilab-tr-1997-1-1.html*.

[Rie97b] Dirk Riehle.
Composite Design Patterns.
In Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97). ACM Press, 1997. Page 218-228.
URL: *http://www.riehle.org/papers/1997/oopsla-1997.html*.

[Rie98] Dirk Riehle et Erica Dubach.
Why a Bank Needs Dynamic Object Models?
Position Paper for OOPSLA '98 Workshop 15 on Metadata and Active Object Models. UBS AG, 1998.
URL: *http://www.riehle.org/papers/1998/oopsla-1998-ws-15.html*.

[Rie99] Dirk Riehle.
Framework Design, A Role Modeling Approach.
PhD dissertation No. 13509, ETH Zürich.
URL: *http://www.inf.ethz.ch/publications/abstract.php3?no=dissertations/th13509*

[Rob99] Donald Bradley Roberts.
Practical Analysis for Refactoring.
PhD thesis, University of Illinois at Urbana-Champaign, April 1999. Available as Computer Science Technical Report #2092.
URL: *ftp://ftp.cs.uiuc.edu/pub/dept/tech_reports/1999/UIUCDCS-R-99-2092.pdf.%gz*.

[RSBP95] Nicolas Revault, , Houari A. Sahraoui, Gil Blain et Jean.-François. Perrot.
A Metamodeling Technique : The METAGEN system.
Proc. TOOLS 16 (127-139). Versailles, March1995.

[RTJ00] Dirk Riehle, Michel Tilman et Ralph E. Johnson.
Dynamic Object Model.
In Proceedings of the 2000 Conference on Pattern Languages of Programming (PLoP 2000). Washington University Technical Report number WUCS-00-29. Washington University, 2000.
URL: *http://www.riehle.org/papers/2000/plop-2000-dom.html*.

[Sad99] Benny Sadeh.
Reifying Interfaces in Smalltalk.
Metadata and Dynamic Object-Model Pattern Mining Workshop OOPSLA '98.
URL: *http://www-poleia.lip6.fr/~razavi/aom/papers/*.

[Sah95] Sahraoui H.A..
Application de la méta-modélisation à la génération d'outils de conception et de mise en oeuvre de bases de données.
Thèse de doctorat de l'Université P. et M. Curie (Paris 6), Juin 1995.

[TD99] Michel Tilman et Martine Devos.
A Reflective and Repository-Based Framework.
Implementing Application Frameworks (M.E. Fayad, D. C. Schmidt, R. E; Johnson ed.), p. 29-64, Wiley Computer Publishing 1999.

[Til99] Michel Tilman.
Active Object-Models and Object Representations.
Position Paper for the OOPSLA '99 MetaData and Active Object-Model Pattern Mining Workshop.
URL : *http://users.pandora.be/michel.tilman/Publications*.

[UIUC98] Ralph E. Johnson et Joseph W. Yoder.
MetaData Pattern Mining Workshop, Urbana, IL, May 1998.
URL: *http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html*.

[Win95] Terry Winograd.
From Programming Environments to Environments for Designing.
Communications of the ACM, June 1995/Vol. 38, No. 6, pages 65 -74.

[Wuy98] Roel Wuyts.
Declarative Reasoning about the Structure of Object-Oriented Systems
Proceedings TOOLS USA'98, IEEE Computer Society Press, pages 112-124, 1998.
URL: *http://prog.vub.ac.be/Research/ResearchPublicationsDetail2.asp?paperID=58*.

[YBJ99] Joseph W. Yoder, Federico Balaguer et Ralph E. Johnson.
From Analysis to Design of the Observation Pattern.
Metadata and Active Object-Model Pattern Mining Workshop. OOPSLA'99, Denver, USA.
URL: *http://www.joeyoder.com/Research/metadata/*.

[YB99] Joseph W. Yoder et Frederico Balaguer.
Using Metadata and Active Object-Model to Implement Fowler's Analysis Patterns.
Tutorial notes #76, OOPSLA'99, Denver, USA.
URL: *http://www.uiuc.edu/ph/www/manolesc/Workflow*

[YBJ01a] Joseph Yoder, Federico Balaguer et Ralph Johnson.
The Architectural Style of Adaptive Object-Models.
Presented for ECOOP 2001 Workshop on Adaptive Object-Models and Metamodeling Techniques.
URL: http://www.adaptiveobjectmodel.com/OOPSLA2001/

[YBJ01b] Joseph Yoder, Federico Balaguer et Ralph Johnson.
Architecture and Design of Adaptive Object-Models.
OOPSLA '01 Intriguing Technical Session - *Paper*. 14-18 October 2001. Tampa, USA.
URL: *http://www.adaptiveobjectmodel.com/OOPSLA2001/AOMIntriguingTechPaper.pdf*.

[YBJ01c] Joseph Yoder, Federico Balaguer et Ralph Johnson.
Architecture and Design of Adaptive Object-Models.
OOPSLA '01 Intriguing Technical Session – *Extended Abstract*. 14-18 October 2001. Tampa, USA.
URL: *http://www.adaptiveobjectmodel.com/OOPSLA2001/AOMExtendedAbstract.pdf*.

[YBJ01d] Joseph Yoder, Federico Balaguer et Ralph Johnson.
Architecture and Design of Adaptive Object-Models.
OOPSLA '01 Intriguing Technical Session - *Talk*. 14-18 October 2001. Tampa, USA.
URL: *http://www.adaptiveobjectmodel.com/OOPSLA2001/OOPSLA2001IntriguingTalk.pdf*.

[Yea97] S. A. Yeates.
Design of a garbage collector using design patterns.
In TOOLS Pacific 25, 1997.

[YFRT98] Joseph W. Yoder, Brian Foote, Dirk Riehle et Michel Tilman.
Metadata and Active Object-Models.
Workshop Results Submission OOPSLA'98 Addendum, Vancouver, Canada.
URL: *http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata*

[YR00a] Joseph W. Yoder et Reza Razavi.
Metadata and Adaptive Object-Models.
ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. no. 1964; Springer Verlag 2000.
URL: *http://www-poleia.lip6.fr/~razavi/ecoop2000/report/ecoop2000-workshop-report.pdf*.

[YR00b] Joseph W. Yoder et Reza Razavi.
Adaptive Object Models.
OOPSLA'2000 Poster Session Abstract. OOPSLA'2000 Companion, pages 81-82, MN, Minnesota.
URL: *http://www-poleia.lip6.fr/~razavi/oopsla2000/poster/oopsla2000-poster.pdf*.

[Zim96] Chris Zimmermann (editor).
Advances in Object-Oriented Metalevel Architectures and Reflection.
CRC Press, 1996.

Bibliographie

[AB92] M. Aksit and L. Bergmans.
Obstacles in object-oriented software development.
In Proceedings of OOPSLA'92, volume 27 of ACM SIGPLAN Notices, pages 341--358. ACM, Oct 1992.

[AMT97] M. Aksit, F. Marcelloni, and B. Tekinerdogan.
Developing object-oriented frameworks using domain models.
email: aksit, bedir@cs.twente.nl and france@iet.unipi.it, 1997.

[Ber90] Lucy Berlin.
When objects collide: Experiences with reusing multiple class hierarchies.
In ECOOP/OOPSLA'90 Proceedings, pages 181–193, October 1990.

[BGKLRZ97] D. Baumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zulighoven.
Framework development for large systems.
Communications of the ACM, 40(10):52--59, oct 1997.

[BJ94] K. Beck & R. Johnson.
Patterns generate architectures.
In Proc. of ECOOP '94. Bologna, Italy. July 1994. M. Tokoro & R. Pareschi (Eds.). LNCS 821. Springer-Verlag. pp. 139-150.

[BM98] Pauline M. Berry and Karen L. Myers.
Adaptive process management: An ai perspective.
CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998.
URL : *http://ccs.mit.edu/klein/cscw-ws.html*

[BMRSS96] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal.
Pattern-Oriented Software Architecture—A System of Patterns.
John Wiley & Sons, July 1996.

[BP89] Ted J. Biggerstaff and Alan J. Perlis, editors.
Software Reusability, volume 2 of Frontier Series.
Addison-Wesley, 1989.

[Bog95] Douglas Paul Bogia.
Supporting Flexible, Extensible Task Descriptions in and Among Tasks.
PhD thesis, University of Illinois at Urbana-Champaign, 1995.
URL : *ftp://ftp.cs.uiuc.edu/pubs*

[Bol98] Gregory Alan Bolcer.
Flexible and Customizable Workflow on the WWW.
PhD thesis, University Of California, Irvine, 1998.

[BW77] D.G. Bobrow et T. Winograd.
An overview of KRL, a Knowledge Representation Language.
Cognitive Science, 1(1):3-46, 1977.

[Car97] Steinar Carlsen.
Conceptual Modeling and Composition of Flexible Workflow Models.
PhD thesis, Department of Computer and Information Science, Faculty of Physics, Informatics and Mathematics, Norwegian University of Science and Technology, 1997.

[CCPP96] F. Casati, S. Ceri, B. Pernici, and G. Pozzi.
Deriving active rules for workflow enactment.
In Proc. 7th International Conference on Database and Expert Systems Applications, Lecture Notes in Computer Science, pages 94–110. Springer-Verlag, 1996.

[CHRW98] Andrzej Cichocki, Abdelsalam (Sumi) Helal, Marek Rusinkiewicz, and Darrell Woelk.
Workflow and Process Automation—Concepts and Technology.
Kluwer Academic Publishers, 1998.

[CHSV97] W. Codenie, K. Hondt, P. Steyaert, and A. Vercammen.
From custom applications to domain specific frameworks.
Communications of the ACM, 1997.

[DCKL98] Dickson K. W. Chiu, Kamalakar Karlapalem, and Qing Li.
Exception handling with workflow evolution in ADOME-WfMS: a taxonomy and resolution techniques. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998.
URL : *http://ccs.mit.edu/klein/cscw-ws.html*

[Deu89] L. Peter Deutsch.
Design Reuse and Frameworks in the Smalltalk–80 System.
Chapter 3, pages 57–72. Volume 2 of Biggerstaff and Perlis [11], 1989.

[DGSZ94] Guido Dinkhoff, Volker Gruhn, Armin Saalmann, and Michael Zielonka.
Entity-Relationship Approach–ER'94, Business Modelling and Re-engineering, chapter Business Process Modeling in the Workflow Management Environment Leu, pages 46–63.
Number 881 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[DMNS97] S. Demeyer, T.D. Meijler, O. Nierstraasz, and Steyaert P.
Design guidelines for tailorable frameworks.
Communications of the ACM, 40(10):60--64, oct 1997.

[EH98] David Edmond and Arthur H. M. ter Hofstede.
Achieving workflow adaptability by means of reflection.
CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998.
URL : *http://ccs.mit.edu/klein/cscw-ws.html*

[FBHS01] Xiang Fu, Tevfik Bultan, Richard Hull et Jianwen Su.
Verification of Vortex Workflows.
Proc. of Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), February, 2001.

[FS97] Mohamed Fayad and Douglas C. Schmidt.
Object-oriented application frameworks.
Communications of the ACM, 40(10):32--38, October 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom.
Architectural mismatch or why it's hard to build systems out of existing parts.
In Proc. 17th International Conference on Software Engineering, Seattle, WA, April 1995.

[GHS95] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth.
An overview of workflow management: From process modeling to workflow automation infrastructure.
Distributed and Parallel Databases, an International Journal, 3:119–153, 1995.
URL : *ftp://ftp.gte.com/pub/dom/reports/GEOR95a.ps*

[GPW99] Dimitrios Georgakopoulos, Wolfgang Prinz, and Alexander L. Wolf, editors.
Proceedings of the Joint Conference on Work Activities Coordination and Collaboration (WACC), volume 24 of Software Engineering Notes. ACM, March 1999.
URL: *http://www.cs.colorado.edu/wacc99*.

[GT98] Dimitrios Georgakopoulos and Aphrodite Tsalgatidou.
Technology and Tools for Comprehensive Business Process Lifecycle Management, pages 356–395. Volume 164 of Do˘ gaç et al. [GPW99], August 1998.

[Hag99] Claus Johannes Hagen.
A Generic Kernel for Reliable Process Support.
PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1999.

[HHKW77] Michael Hammer, W. Gerry Howe, Vincent J. Kruskal, and Irving Wladawsky.
Very high level programming language for data processing applications.
Communications of the ACM, 20(11):832–840, November 1977.

[HLKZDS00] Richard Hull, Francois Llirbat, Bharat Kumar, Gang Zhou, Gouzhu Dong et Jianwen Su.
Optimization Techniques for Data-intensive Decision Flows.
Appears in IEEE Intl. Conf. on Data Engineering (ICDE), San Diego, March, 2000.

[HLSDKZ98] Richard Hull, Francois Llirbat, Jianwen Su, Guozhu Dong, Bharat Kumar et Gang Zhou.
Adaptive execution of workflow: Analysis and optimization.
Bell Labs working paper, October 1998.
URL: *http://www-db.research.bell-labs.com/projects/vortex/*.

[HLSDKZ99] Richard Hull, Francois Llirbat, Jianwen Su, Guozhu Dong, Bharat Kumar et Gang Zhou.
Efficient Support for Decision Flows in E-Commerce Applications.
Appears in 2nd Intl. Conf. on Telecommunications and Electronic Commerce (ICTEC), pp. 109-123, October, 1999.

[HLSSDKZ99] Richard Hull, Francois Llirbat, Eric Simon, Jianwen Su, Guozhu Dong, Bharat Kumar et Gang Zhou.
Declarative worflows that support easy modification and dynamic browsing.
In Georgakopoulos et al. [GPW99], pages 69–78.

[HS98] Richard Hull et Jianwen Su.
The Vortex Approach to Integration and Coordination of Workflows.
Position paper to the Workshop on Cross-Organisational Workflow Management and Co-ordination, San Francisco, February, 1999, held in conjunction with the International Joint Conference on Work Activities Coordination and Collaboration (WACC) San Francisco, 2/99.
URL: *http://www-db.research.bell-labs.com/projects/vortex/papers/vortex-wacc99-cross-org-workshop.html*.

[HSB98] Yanbo Han, Amit Sheth, and Christoph Bussler.
A taxonomy of adaptive workflow management.
CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998.
URL : *http://ccs.mit.edu/klein/cscw-ws.html*

[Hol95] David Hollingsworth.
The Workflow Reference Model.
Workflow Management Coalition, Avenue Marcel Thiry 204, 1200 Brussels, Belgium, 1995.
URL ! *http://www.aiim.org/wfmc*.

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay.
Contracts: Specifying behavioural compositions in object-oriented systems.
In Proceedings of ECOOP/OOPSLA'90, pages 169--180, 1990.

[Jac00a] Daniel Jackson
Object Models as Heap Invariants.
A chapter in: Essays on Programming Methodology, edited by Carroll Morgan and Annabelle McIver. Springer Verlag, 2000.
URL: ***http://sdg.lcs.mit.edu/~dnj/pubs/om-heap-tex.pdf***.

[Jac00b] Daniel Jackson
Lecture Notes on Software Design.
Lecture notes from 6.170: Laboratory in Software Engineering, MIT, 2000.
URL: ***http://sdg.lcs.mit.edu/~dnj/pubs/****fall00-lectures.pdf***.**

[JB96] Stefan Jablonski and Christoph Bussler.
Workflow Management—Modeling Concepts, Architecture and Implementation.
International Thomson Computer Preess, 1996.

[JF01] Daniel Jackson and Alan Fekete
Lightweight Analysis of Object Interactions
Fourth International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan, October 2001.
URL: *http://sdg.lcs.mit.edu/~dnj/pubs/tacs-01.pdf*

[Joh94] Ralph E. Johnson.
How to Develop Frameworks.
Tutorial notes (10), 8th European Conference on Object-Oriented Programming (ECOOP'94), Bologna, July 1994.

[JR91] Ralph Johnson and Vince Russo.
Reusing Object-Oriented Design.
University of Illinois, Technical Report UIUCDCS 91-1696, 1991.

[JT97] Michael Jackson and Graham Twaddle.
Business Process Implementation–Building Workflow Systems.
Addison-Wesley, 1997. ISBN 0-201-177684.

[KL92] G. Kiczales and J. Lamping.
Issues in the design and specifications of class libraries.
In Proceedings of OOPSLA '92, pages 435--451. ACM/SIGPLAN, Oct 1992.

[KRSR99] Gerti Kappel, Stefan Rausch-Schott, and Werner Retshitzegger.
A Framework for Workflow Management Systems Based on Objects, Rules and Roles, chapter TBP.
In Fayad et al. [FS97], 1999.
URL : *ftp://ftp.ifs.unilinz.ac.at/pub/publications/1998/1698.ps.zip*

[Kov99] Zsolt Kováks.
The Integration of Product Data with Workflow Management Through a Common Data Model.
PhD thesis, Faculty of Computer Studies and Mathematics, University of the West of England, Bristol, April 1999.

[LH87] M. D. Lubars & M. T. Harandi.
Knowledge-Based Software Design Using Design Schemas.
In Proc. of the 9th Int. Conf. on Software Engineering. Monterey, CA. March-April 1987. IEEE Computer Society Press. pp. 253-262.

[Min75] M. Minsky.
A framework for representing knowledge.
In The Psychology of Computer Vision, editors P. Winston, pages 211-281. McGraw-Hill, New York (NY), USA, 1975.

[MIT] The MIT process handbook project.
URL : *http://ccs.mit.edu/ph*

[Moh97] C. Mohan.
Recent trends in workflow management products, standards and research. In Proc. NATO
Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability, Istanbul, Turkey, August 1997. Springer-Verlag.
URL: *http://www.almaden.ibm.com/cs/exotica/wfnato97.ps*.

[MT00] Nenad Medvidovic and Richard N. Taylor.
A Classification and Comparison Framework for Software Architecture Description Languages.
IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, January 2000.

[MT97] Nenad Medvidovic and Richard N. Taylor.
Exploiting Architectural Style to Develop a Family of Applications.
IEE Proceedings Software Engineering, vol. 144, no. 5-6, pp. 237-248 (October-December 1997).

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor.
A Language and Environment for Architecture-Based Software Development and Evolution.
In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp. 44-53, Los Angeles, CA, May 16-22, 1999.

[MR99] Nenad Medvidovic and David S. Rosenblum.
Assessing the Suitability of a Standard Design Method for Modeling Software Architectures.
In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pp. 161-182, San Antonio, TX, February 22-24, 1999.

[Nut96] Gary J. Nutt.
The evolution toward flexible workflow systems.
Distributed Systems Engineering, 3(4):276–294, December 1996.

[OGTHJMQRW99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf.
An Architecture-Based Approach to Self-Adaptive Software.
IEEE Intelligent Systems and Their Applications, vol. 14, no. 3, pp. 54-62 (May/June 1999).

[OMG98a] Joint workflow management facility—revised submission.
OMG Document Number bom/98–06–07, 1998.
URL : at *ftp://ftp.omg.org/pub/docs/bom/98-06-07.pdf*

[OMG98b] Workflow management facility specification.
OMG Document Number bom/98–03–01, 1998.
URL : *ftp://ftp.omg.org/pub/docs/bom/98-03-01.pdf*

[OW98] Aris M. Ouksel and Jr. James Watson.
The need for adaptive workflow and what is currently available on the market—perspectives from an ongoing industry benchmarking initiative.
CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998.
URL : *http://ccs.mit.edu/klein/cscw-ws.html*

[PDBH97] Mike Papazoglou, Alex Delis, Athman Bouguettaya, and Mostafa Haghjoo.
Class library support for workflow environments and applications.
IEEE Transactions on Computers, 46(6):673–686, June 1997.

[Pam72] David L. Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM, 15(12):1053–1058, December 1972.

[PC86] David Lorge Parnas and Paul C. Clements.
A rational design process: How and why to fake it.
IEEE Transactions on Software Engineering, SE-12(2):251–7, 1986.

[PS00] Charles Petrie and Sunil Sarin.
Controlling the flow.
IEEE Internet Computing, 4(3):34–36, May–June 2000.

[Smi84] Brian Cantwell Smith.
Reflection and semantics in LISP.
In Proceedings 11th ACM Symposium on Principles of Programming Languages, pages 23-- 35, 1984.

[Szy97] Clemens Szyperski.
Component Software—Beyond Object-Oriented Programming.
Addison-Wesley, 1997.

[VJK96] Vijay Vaishnavi, Stef Joosten, and Bill Kuechler.
Representing workflow management systems with smart objects.
In Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, May 1996.
URL : *http://www.cis.gsu.edu/~bkuechle/allsec3.html*

[WJ90] Wirfs-Brock R. J. & Johnson R. E.
Surveying current research in Object-Oriented design.
CACM Vol. 33, No. 9, pp. 105-124, Sept. 1990.

[WMC99] The Workflow Management Coalition.
Process definition model and interchange language, October 1999.
Document WfMC-TC-1016P v1.1.

[WMW] Jeanine Weissenfels, Peter Muth, and Gerhard Weikum.
Flexible worklist management in a light-weight workflow management system.
URL : *http://www-dbs.cs.unisb.de/~mlite*

[You94] Patrick Scott Chun Young.
Customizable Process Specification and Enactment for Technical and Non-Technical Users.
PhD thesis, University Of California, Irvine, 1994.

[Zis97] M.D. Zisman.
Representation, Specification and Automation of Office Procedures.
PhD thesis, University of Pennsylvania, Warton School of Business, 1977.

Annexes

Annexe I: Tableaux & Figures

1 Liste des tableaux

2 Liste des figures

Annexe II : le Compte-Service

1 Description commerciale du produit Compte-Service

Ce qui suit correspond à la description commerciale du produit bancaires compte-service, tel qu'il est proposé par le Crédit Agricole[142]. Ce descriptif nous a servi dans l'élaboration de notre exemple illustratif présenté dans l'introduction, paragraphe 2, page 31.

CRÉDIT AGRICOLE DE LA VENDÉE

Dès maintenant, découvrez les avantages que vous réserve votre Compte-Service, associé à votre carte bancaire

Avec un Compte-Service et une carte bancaire du Crédit agricole, vous bénéficiez de tous les atouts pour vous rendre la banque plus facile, plus claire et plus efficace.

Vous souhaitez un peu de souplesse certains mois mais aussi que les excédents de votre compte soient placés, consulter vos comptes à distance... nos conseillers vous orienteront vers **LA solution Compte-Service** qui vous convient le mieux.

[142] Ce descriptif a été emprunté du site Web du Crédit Agricole de Vendée.
URL : *http://www.ca-vendee.fr/libre.asp?id=10495*.

Au Crédit agricole, il y a forcément un Compte-Service qui répond à vos attentes !

Le Compte-Service Equilibre : pour gérer facilement votre argent au quotidien

Le Compte-Service Equilibre du Crédit agricole, c'est la souplesse et la sécurité dont vous avez besoin pour gérer votre budget efficacement.

Avec le Compte-Service Equilibre,

- vous disposez d'un découvert forfaitaire à taux préférentiel,
- vous protégez vos moyens de paiement,
- vous gérez vos comptes grâce à "*Crédit agricole en ligne*" et son service Essentiel Plus,
- vous bénéficiez de trois gratuits à la revue "Dossier Familial",
- vous profitez de l'exonération de services (chèques de banque,...).

Le Compte-Service Confort : une gestion simple et efficace de votre budget

Le Compte-Service Confort du Crédit agricole, c'est la souplesse et la sécurité dont vous avez besoin pour gérer votre budget efficacement.

Avec le Compte-Service Confort,

- vous bénéficiez d'un découvert personnalisé à taux préférentiel,
- vous protégez vos moyens de paiement,
- vous gérez vos comptes grâce à "*Crédit agricole en ligne*" et son service Essentiel Plus,
- vous bénéficiez de trois mois gratuits à la revue "Dossier Familial",
- vous profitez de l'exonération de services (chèques de banque,...).

Le Compte-Service Privilège : une gestion dynamique et performante de vos comptes

Le Compte-Service Privilège du Crédit agricole, c'est la souplesse et la sécurité dont vous avez besoin pour gérer vos comptes efficacement. C'est aussi une série d'avantages exclusifs qui vous assure tranquillité et performance.

Avec le Compte-Service Privilège,

- vous disposez d'une formule dynamique et performante pour épargner l'argent qui dort sur votre compte,
- vous recevez un relevé trimestriel complet de votre épargne et de vos emprunts,
- vous protégez vos moyens de paiement,
- vous disposez d'une autorisation de découvert avec une franchise d'agios en cas d'imprévus,
- vous gérez vos comptes grâce à "*Crédit agricole en ligne*" et son service Essentiel Plus,
- vous bénéficiez de trois mois gratuits à la revue "Dossier Familial",
- vous profitez de l'exonération de services (chèques de banque,...).

2 Caractéristiques du Comptes-service

Le but de cette section est de fournir le matériel sur laquelle s'appuie notre illustration du cahier des charges de l'outillage des langages d'expert, à travers l'exemple de l'adaptation de comptes bancaire (cf. l'introduction, paragraphe 2.4, page 41).

Ce matériel est issu de notre analyse de la fiche descriptive du produit Compte-Service fournie dans le paragraphe précédent (page 256).

2.1 Le produit

Tout compte-service est attaché à un compte-chèque. Il sert à contractualiser les relations entre le client titulaire du compte-chèque et sa banque en matière de quelques règles de fonctionnement de ce compte. Globalement, un client s'engage à payer une certaine somme à sa banque pour en contre partie bénéficier des quelques "services" suivants :

1. une protection des moyens de paiement (par un contrat d'assurance),
2. une autorisation de découvert à taux préférentiel,
3. un accès à "Crédit agricole en ligne" et son service Essentiel Plus,
4. trois mois d'abonnement gratuits à la revue "Dossier Familial",
5. une exonération de services (chèques de banque,...).

En outre, les concepteurs de ce produit ont prévu plusieurs types de compte-service : le Compte-Service Equilibre, le Compte-Service Confort et le Compte-Service Privilège. Cette variété permet à la banque de proposer à chaque client un compte-service qui lui convient mieux, suivant les critères décidés par les banques[143]. En plus des services communs décrits ci-dessous, chaque compte a des caractéristiques qui lui son propres :

1. le Compte-Service Equilibre : propose une autorisation de découvert *forfaitaire*.
2. le Compte-Service Confort : propose une autorisation découvert *personnalisé*.
3. le Compte Service Privilège : propose une autorisation de découvert *personnalisé* avec une *franchise d'agios* (en cas d'imprévus !), ainsi qu'un relevé trimestriel complet de la situation des comptes du client.

Et enfin, à chaque compte est associé une carte bancaire : la carte Mozaïc, la carte Eurocard Mastercard et la carte Maestro.

Le première étape dans la vie d'un produit (bancaire) comme le compte-service est sa conception (pour répondre à quel besoin, de quelle communauté, par quels moyens, dans quel cadre contractuel, à quel prix, etc). Ce travail est l'œuvre de spécialistes internes ou externes aux établissements. Elle requiert des études systématiques sur le plan financier, marketing, etc.

2.2 Le modèle objet classique

La fiche descriptive ci-dessus distingue trois types de Compte-Service : le Compte-Service Equilibre, le Compte-Service Confort et le Compte-Service Privilège.

La Figure 98 est un diagramme de classe UML qui permet de visualiser une certaine modélisation objet de ces trois types de compte. Celui-ci est entièrement revêtu de couleur jaune. Cela signifie qu'il est dans sa globalité implanté par des programmeurs. C'est la démarche classique de développement d'applications objets. Et, c'est pourquoi nuos appelons ce diagramme le modèle objet classique du produit Compte-Service.

Le but de notre présentation de ce modèle est de rassembler dans une même section les éléments présentés de façon plutôt dispersés dans l'introduction.

[143] Notons que les établissements bancaires distinguent en général plusieurs catégories de clients. Ce sont, à titre d'exemple, clients privilégiés, clients fiables, clients classiques, clients à risque et client à découvert.

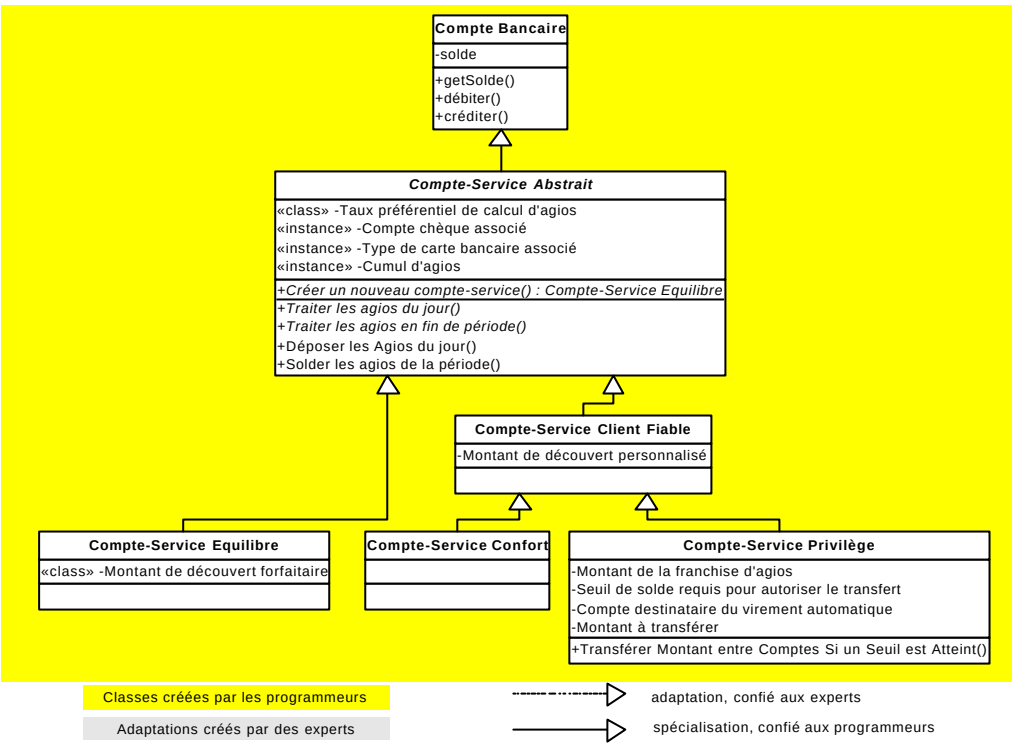


Figure 98 : Exemple de modèle objet classique d'un système pour la gestion de Comptes-Service.

Le diagramme de la Figure 98 permet également de visualiser la structure et le comportement de chacun de ces trois types de compte, tel que nous allons décrire ci-dessous.

2.3 Structuration des trois types de Compte-Service

Une partie de la définition de la structure des Comptes-Service est commune aux trois types de compte et une autre partie spécifique à chacun d'eux. De plus on distingue ici la définition de la structure des *types* eux-même ainsi que la définition de la structure des *instances* des types.

2.3.1 Structure commune des trois *types* de Compte-Service

Les *types* de comptes, *eux-mêmes*, partagent un attribut qui est le '`Taux préférentiel de calcul d'agios`'[144]. Cet attribut sert au calcul du montant journalier des agios sur le compte-chèque associé.

2.3.2 Structure commune des *instances* des trois types de Compte-Service

Les *instances* de ces trois types de comptes partagent les éléments suivants :

1. l'attribut '`Compte chèque associé`' : désigne le compte chèque associé au compte-service.
2. l'attribut '`Type de carte de bancaire associé`' : désigne le type de carte bancaire choisi, parmi les trois choix possibles : Mozaïc, Eurocard Mastercard et Maestro.
3. l'attribut '`Cumul d'agios`' : désigne le montant cumulé des agios sur une période donnée, en général d'un mois. A la fin de cette période le montant des agios est affecté sur le solde

[144] Tout au long de ce document les attributs seront en police `courrier` et entouré par des simple quotte (').

du compte, s'il n'y a pas de franchise ou si la franchise a été dépassée. La valeur courante de cet attribut est alors remis à zero.

2.3.3 Structure spécifique au Compte-Service Equilibre

Le montant de découvert forfaitaire est une valeur globale à toutes les instances du compte-service Equilibre. Aussi, ce *type* de service doit disposer lui-même d'un attribut '`Montant de découvert forfaitaire`'.

2.3.4 Structure spécifique au Compte-Service Confort

A chaque *instance* du type de compte-service Confort est associée un montant de découvert personnalisé. On prévoit alors un attribut '`Montant de découvert personnalisé`' pour stocker cette valeur.

2.3.5 Structure spécifique au Compte-Service Privilège

1. tout comme le type de compte-service Confort, à chaque *instance* du type de compte-service Privilège est associée un montant de découvert personnalisé. On prévoit alors un attribut '`Montant de découvert personnalisé`' pour stocker cette valeur.
2. chaque instance du type de compte-service Privilège doit disposer d'un attribut '`Montant de la franchise d'agios`' pour stocker la valeur concernée. Le client est dispensé du paiement des agios sur une période donnée (en général d'un mois) si le montant du cumul des agios ne dépasse pas le montant de la franchise n'est pas dépassé. En cas de dépassent il paye seulement la différence de ces deux montants.
3. trois autres attributs sont nécessaire pour permettre le stockage des informations nécessaires au transfert automatique mensuel d'une certaine somme du compte-chèque vers un compte d'épargne[145]. Il s'agit de l'attribut '`Montant à transférer`', '`Seuil de solde requis pour autoriser le transfert`' ainsi que le '`Compte destinataire du virement automatique`'.

2.4 Le comportement des trois types de compte-service

On distingue ici trois procédés communs (création d'un compte-service, traitement journalier des agios et le traitement des agios en fin de période) ainsi qu'un procédé spécifique au compte-service Privilège. Les procédés commun ont, toutefois, souvent des implantations différentes.

2.4.1 Procédé commun de création d'un compte-service

La création de chaque compte-service comporte en premier lieu deux étapes de 1) création d'une nouvelle instance du compte concerné et 2) son initialisation. Ensuite, ce procédé déclenche une suite de transactions :

1. pour enregistrer l'émission de la carte de crédit attribué.
2. pour enregistrer le contrat d'assurance associé à la clause de protection des moyens de paiement,
3. pour enregistrer une autorisation d'accès au service en ligne "Crédit agricole en ligne",
4. pour enregistrer un ordre d'abonnement gratuits de trois mois à la revue "Dossier Familial",
5. pour enregistrer une exonération de services (chèques de banque,...).

[145] Cf. le paragraphe sur la "formule dynamique et performante pour épargner l'argent qui dort sur votre compte " dans l'annexe A.

Cet ensemble de transaction peut-être assimilé au *procédé* de création d'un compte service. Dans le cas du compte-service Privilège, ce procédé comporte une transaction supplémentaire pour enregistrer l'envoi trimestriel d'un relevé d'emprunts et d'éparges.

2.4.2 Procédé de traitement journalier des agios

Chaque compte-service doit être en mesure de *calculer chaque jour les agios* dus par le clients par rapport au montant du découvert du compte-chèque associé. La fonction de calcul est toutefois *spécifique à chaque type* de compte.

2.4.2.1 Traitement journalier des agios, cas de Compte-Service Equilibre

Le traitement des agios du type de compte-service Equilibre est basé sur les éléments suivants :

1. la valeur courante de l'attribut '`Taux préférentiel de calcul d'agios`'.
2. la valeur courante de l'attribut '`Montant de découvert forfaitaire`'.
3. le solde du compte chèque associé.
4. la fonction '`Calcul des agios journaliers (solde, découvert, taux)`'. Cette fonction prend en entrée les trois arguments et retourne une valeur numérique. Cette valeur peut être zero.
5. la fonction '`Déposer les Agios du Jour (montant)`'. Celle-ci sert à reporter le montant des agios sur le compte chèque associé.

2.4.2.2 Traitement journalier des agios, cas de Compte-Service Confort

Le traitement des agios du type de compte-service Confort est basé sur les mêmes éléments que le compte-service Equilibre. La seule différence est que la valeur de l'attribut '`Montant de découvert forfaitaire`' (item n° 2, §2.4.2.1) est remplacée par celle de l'attribut '`Montant de découvert personnalisé`'.

A noter, toutefois, que l'algorithme mis en œuvre par la fonction '`Calcul des agios journaliers`' peut ici être différent.

2.4.2.3 Traitement journalier des agios, cas de Compte-Service Privilège

Le traitement des agios du type de compte-service Privilège dépend exactement des mêmes attributs que celui du compte-service Confort (§2.4.2.2). Toutefois, l'algorithme mis en œuvre par la fonction '`Calcul des agios journaliers`' peut ici être différent.

2.4.3 Procédé de traitement des agios en fin de période

Chaque compte-service doit être en mesure de solder à l'échéance prévue le montant cumulé des agios sur la période de référence. Nous proposons de confier cette tâche à la fonction '`Solder les agios de la période`'. Le calcul par défaut de cette fonction consiste à déclencher une transaction sur le compte-chèque associé qui enregistre un mouvement en débit dont le montant est égale à la valeur courante de l'attribut '`Cumul d'agios `'. Elle remet alors à zero la valeur courante de cet attribut.

Ce traitement est toutefois légèrement différent dans le cas du compte-service Privilège. En effet, ici la transaction en débit n'est pas déclenché si le cumul des agios est inférieur à la valeur courante de l'attribut '`Montant du franchise d'agios`'. En cas de dépassement, le montant de la transaction sera égale à la différence entre ces deux valeurs.

2.4.4 Procédé du virement automatique associé au Compte-Service Privilège

Le compte-service Privilège doit disposer d'un procédé pour permettre le transfert à une échéance donnée d'un certain montant du compte chèque associé vers un compte désigné si le solde du compte chèque associé a atteint un certain seuil. Nous proposons d'appeler ce procédé '`Transférer Montant entre Comptes Si un Seuil est Atteint`'. Celui-ci utilise dans son calcul les ressources suivantes :

1. la valeur courante de l'attribut '`Montant à transférer`'.
2. la valeur courante de l'attribut '`Compte destinataire du virement automatique`'.
3. la valeur courante de l'attribut '`Seuil de solde requis pour autoriser le transfert`'.
4. le solde du compte-chèque associé.
5. la fonction primitive '`Si Supérieur Faire()`'. Cette fonction reçoit en entrée trios arguments. Si la valeur du premier argument est supérieur à la valeur du deuxième argument, elle évalue alors le troisième argument.

2.5 Synthèse sur le modèle objet classique

Voici la description synthétique de la structure et du comportement de chacune des abstractions du modèle de la Figure 98, extraite de l'exposé ci-dessus.

Compte Bancaire

La première abstraction de ce modèle correspond à la notion de Compte Bancaire. Un compte bancaire quelconque est représenté par une variable d'instances `solde` et trois méthodes : `getSolde()`, `débiter()` et `créditer()`.

Compte Service Abstrait

L'abstraction suivante décrit les comptes-services en général. Elle est appelée Compte Service Abstrait. Elle comporte la structure et le comportement commun à tous les comptes-service.

Structure :

1. `Taux préférentiel de calcul d'agios`[146],
2. `Compte chèque associé`,
3. `Type de carte bancaire associé` (Mozaïc, Eurocard Mastercard et Maestro),
4. `Cumul d'agios`.

Comportement :

1. `Créer un nouveau compte-service()`,
2. `Traiter les agios du jour()`,
3. `Traiter les agios en fin de période()`,
4. `Déposer les agios du jour()`,
5. `Solder les agios de la période()`.

A noter que l'attribut `Taux préférentiel de calcul d'agios` est une valeur globale, valable pour toute instance d'un certain type de compte-service. Nous proposons de l'appeler *l'attribut de type* et montrons dans ce mémoire (cf. chapitre V) une modélisation dans le cadre de langages d'experts pour en assurer la définition et la gestion lors de l'exécution.

[146] Tout au long de ce document le nom des attributs et les traitements sera en police `courrier`. Pour permettre de distinguer les attributs des traitements, le nom de ces derniers est suivi par ().

Compte-Service Equilibre

Le compte-service Equilibre dispose d'un attribut de *type* spécifique : `Montant de découvert forfaitaire`.

Compte-Service Client Fiable

Le Compte-Service Fiable modélise les caractéristiques commune des deux compte-service Confort et Privilège. Cela comprend l'attribut de type `Montant de découvert personnalisé`, ainsi que le procédé `Traiter les agios du jour()`.

Compte-Service Confort

Le compte-service Confort n'a, pour l'instant, aucun attribut ni de traitements propre à lui. Elle représente simplement le concept de compte-service confort. Elle peut servir pour différencier le calcul du procédé `Traiter les agios du jour()` par rapport au compte-service Privilège.

Compte-Service Privilège

Un compte-service Privilège comporte les attributs et traitements suivants :

Structure :

1. `Montant de la franchise d'agios`,
2. `Seuil de solde requis pour autoriser le transfert`,
3. `Montant à transférer`,
4. `Compte destinataire du virement automatique`.

Comportement :

1. `Créer un nouveau compte-service()`,
2. `Traiter les agios en fin de période()`,
3. `Transférer montant entre comptes si un seuil est atteint()`.

Annexe III : Application à la génération semi-automatique d'adaptations

1 Objectifs

Afin de tester le fonctionnement de notre outillage et aussi de montrer son utilité dans le cadre de la génération automatique d'adaptations, nous avons créé un langage d'experts extrêmement simplifié qui permet de définir de nouveaux types de lignes brisée et de les instancier.

Comme tout langage d'experts, il s'agit donc d'un système à *deux niveaux d'usage* : dans un premier temps l'expert (e.g. ici un enseignant) défini des types de ligne brisée. Ensuite, l'utilisateur (e.g. un élève) expérimente la création des lignes brisées suivant les types précédemment définis par l'expert et cherche en particulier à deviner les fonctions associées à chaque type et de les faire exécuter sur l'instance de la ligne brisée qu'il aura dessiné.

2 Conception

En s'appuyant sur le système de classes DYCRA et le framework DYCTALK, le modèle objet de ce langage d'experts comprend essentiellement deux classes `Polyline` et `PolylineType`. La première spécialise la classe `Component` et la seconde la classe `FlowIndependentComponentType` (cf. ; chapitre III).

A ceci s'ajoutent deux éditeurs graphiques, du type des éditeurs du framework HOTDRAW[147].

Comme son nom l'indique, le premier, *l'éditeur de types de ligne brisée* permet de définir de nouveaux types de ligne brisée et cela en traçant à l'écran une ligne comportant autant de sommet que nécessaire. Chaque sommet décrit un attribut du type d'objets concerné.

[147] Cf. *http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html*.

Cet éditeur permet également de fournir le nom du nouveau type d'objets ainsi défini et de dessiner également une icône qui permet à l'utilisateur de l'identifier graphiquement lors de l'instanciation.

La fonction principale de cet éditeur consiste, toutefois, à permettre à l'expert d'associer, toujours de façon graphique, des procédures à chaque nouveau type de ligne brisée. Le procédé proposé à l'expert consiste à choisir un sous-ensemble de sommets de la ligne brisée. Ce choix se réalise en cliquant à l'aide de la souris sur chacun des sommets concernés et cela dans l'ordre souhaité. Le choix se termine lorsque l'éditeur détecte un double clique sur un des sommets. Chaque sommet ne peut, par ailleurs, être sélectionné qu'une seule fois.

Dès la fin de la sélection, si cet éditeur constate qu'il est possible de faire passer une ligne par l'ensemble des sommets choisis (il faut au moins deux points), il génère automatiquement une micro-composition à la DART[148]. Celui-ci consiste à une séquence (instance de la classe `SequenceProcedure`) qui comporte, pour un segment composé de N sommets, N instances de la classe `PrimitiveProcedure`. N-1 de ces instances portent sur le descriptif de service `Calculer distance entre deux points ()`. La dernière porte sur le descriptif de service `Calculer la somme de N nombres ()`.

Le second, l'éditeur d'instanciation de types de ligne brisée, permet à un utilisateur d'instancier les types de ligne brisé disponibles. Cette création se réalise sous contrainte des choix de l'expert, c'est à dire que chaque instance doit avoir autant de sommets que ceux de son type.

Cet éditeur sert également à deviner les segments passant par des sommets de la ligne brisée, définis également par l'expert. A chaque fois que l'utilisateur "découvre" un tel segment, cet éditeur trace une ligne qui relie le premier sommet du segment au dernier. Il procède également à l'exécution de la procédure de calcul concernée (auto-générée par l'éditeur de types) et affiche le résultat à l'écran.

3 Exemple

Dans cette sous-section nous montrons le fonctionnement de ces deux éditeurs à travers l'exemple de la création du type de ligne brisée à sept sommets et son instanciation.

3.1 Définition par l'expert d'un nouveau type de ligne brisée

Comme permet de l'illustrer la Figure 99, la première étape consiste à fournir un nom pour le nouveau type de ligne brisée. Ici l'expert saisie la chaîne de caractères "Ligne brisée avec 7 sommets" et valide la boîte de dialogue.

Figure 99 : Exemple de création d'un nouveau type d'objet (ici ligne brisée).

[148] Ce langage d'experts a été réalisés dans les phases préliminaires de cette recherche et bien avant que le système DART sous sa forme actuelle existe. Les micro-compositions générées ici se situent donc entre les micro-procédés à la Micro-worfkow et les dernières versions des micro-compositions. En effet, la notion de descriptif de service existe à ce niveau mais pas encore celle d'instance de descriptif de service. C'est pourquoi ce sont les classes du Micro-worfklow qui sont instanciées pour représenter la définition de la procédure. Nous avons expérimenté très récemment la génération automatique de la nouvelle génération de micro-compositions dans le cas du projet MOBIDYC (Cf. l'annexe VII). Une autre caractéristique importante de cette version est qu'elle met en œuvre un modèle d'activation de procédures suivant notre modèle DARC (Type Cube, cf. Figure 32, page 108). Celui a l'avantage de mieux développer cette dimension par rapport à Micro-worfklow qui s'arrête à une seule abstraction, celle de `ProcedureActivation`.

L'étape suivante consiste à définir les attributs de ce nouveau type d'objets et cela en traçant la ligne brisée elle même à l'écran. La Figure 100 ci-dessous illustre la création d'un type de ligne brisée possédant sept sommets, nommés par des lettres de A à G.

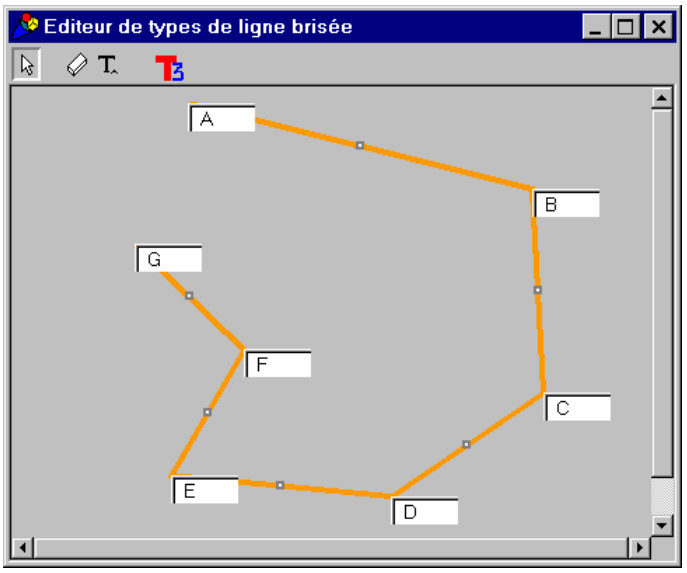


Figure 100 : Exemple d'édition d'un type de ligne brisée par l'expert.

A présent l'expert peut procéder à la définition de procédures pour ce type d'objets. Supposons ici que l'expert désigne le segment qui passe respectivement par les points A, C et F. Cet 'éditeur génère automatiquement une procédure et annonce son action à l'expert via une boîte de dialogue, telle que celle illustrée par la Figure 101 ci-dessous.

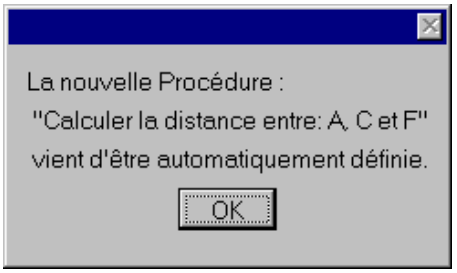


Figure 101 : Annonce de la génération automatique d'une procédure par l'éditeur de types.

3.2 Instanciation par un utilisateur du nouveau type de ligne brisée

Comme permet de l'illustrer la Figure 102, l'instanciation des types de ligne brisée commence par le choix du type concerné. Supposons qu'ici le choix de l'utilisateur porte sur le type appelé « Ligne brisée avec 7 sommets ».

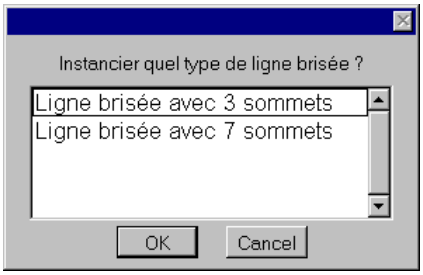


Figure 102 : Choix par l'utilisateur d'un type de ligne brisée à instancier.

La Figure 103 ci-dessous illustre l'éditeur d'instanciation, ouvert sur ce type de ligne brisée. Ce nouveau type s'identifie par l'icône  , également choisie par l'expert et qui figure en haut de cet éditeur.

L'utilisateur peut dessiner sur un tel éditeur autant d'instances d'un type de ligne brisée qu'il souhaite. Toutefois, ce dessin se réalise sous contrainte de la définition réalisée par l'expert. Par exemple ici chaque ligne brisée aura exactement sept sommets.

Cette figure visualise uniquement le nom des sommets A, C et F, ainsi que leur position à l'écran. Cette visualisation peut être configurée par l'utilisateur, qui peut également demander la visualisation de la longueur de chaque sagement.

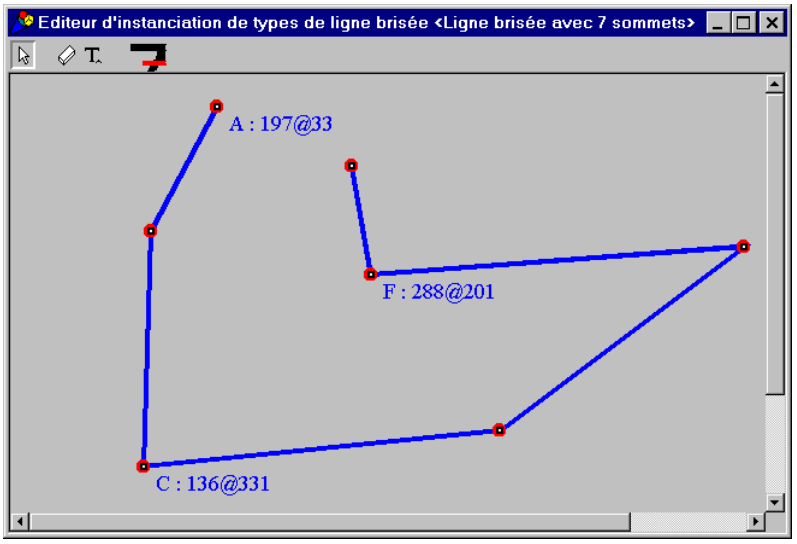


Figure 103 : Exemple d'instanciation sous contrainte d'un type de ligne brisée.

Comme permet de l'illustrer la Figure 104 ci-dessous, l'étape suivante consiste à deviner les procédures définies par l'expert et à les faire exécuter sur l'instance ainsi créée. Ici l'utilisateur a bien cliqué dans l'ordre sur les sommets A, C et F. Aussi cet éditeur a bien détecté l'existence d'une procédure de calcule dédiée au calcul de la longueur de ce segment, a procédé à son activation, a tracé une ligne qui relie le sommet A au sommet C et a affiché la longueur 551,165, en nombre de pixels (px), du segment ACF.

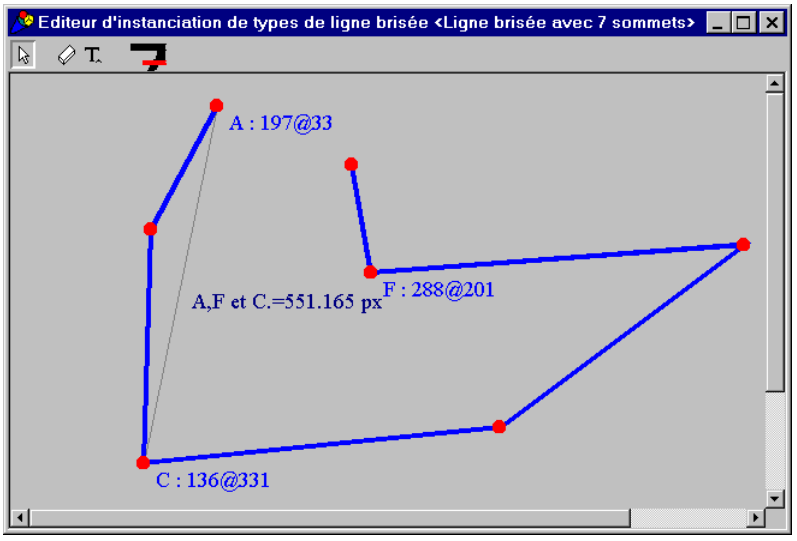


Figure 104 : Exemple de recherche et d'exécution d'un algorithme généré automatiquement.

Lorsque l'utilisateur change la valeur des attributs d'une ligne brisée en déplaçant, à l'aide de la souris, ses sommets, l'éditeur recalcule en temps réel et de façon automatique la nouvelle valeur en re-activant la procédure concernée. Ce recalcule automatique est illustré par la Figure 105 ci-dessous.

Ici le sommet F a été déplacé, ce qui conduit au calcul de la nouvelle valeur 684,76 px pour ce segment.

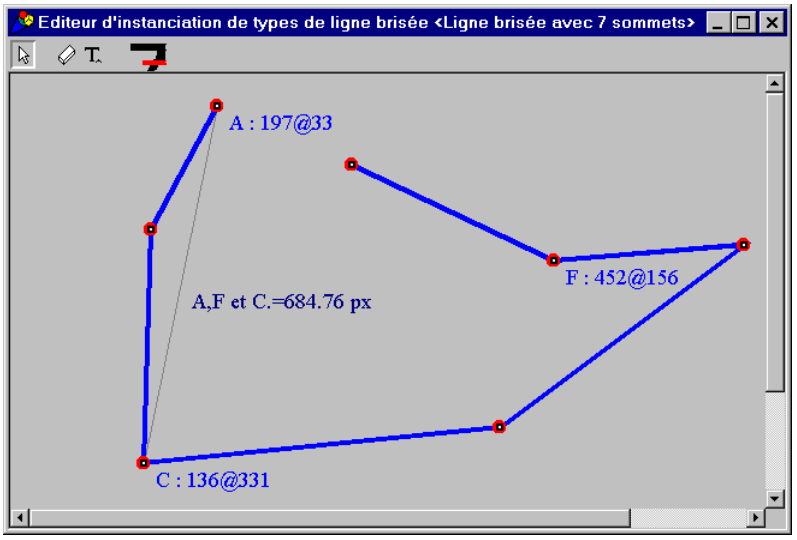


Figure 105 : Exemple de recalcule automatique de la valeur d'un segment.

Il convient de préciser que si pour une instance donnée d'un type de ligne brisée il n'y a pas de procédure associée au calcul de la longueur d'un segment passant par une suite de sommets cliqués à l'écran par l'expert, cet éditeur affiche un message comme celui illustré par la Figure 106 ci-dessous, afin de faire par à l'utilisateur de ce fait.

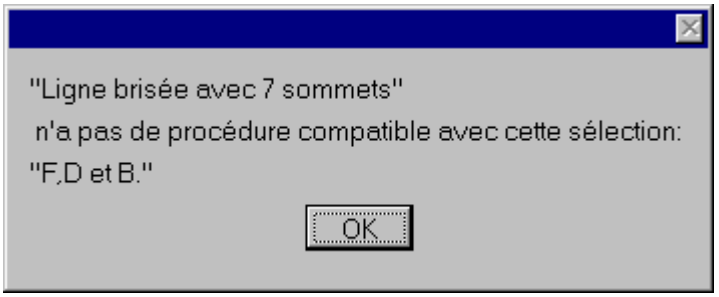


Figure 106 : Non existence d'une procédure permettent le calcul de la longueur d'un segment choisi.

4 Implantation

Outre l'implantation des deux éditeurs graphiques dont la description ne représente pas ici un intérêt particulier, la création de ce langage d'experts s'appuie entièrement sur les mécanismes mis en œuvre par le système de classes DYCRA et le frameworks DYCTALK présentés dans ce mémoire.

Nous tenons donc ici à fournir uniquement plus de détails sur les deux algorithmes de génération et de détection de procédures, utilisés respectivement par les deux éditeurs de types et d'instances.

4.1 Algorithme de génération de procédures

La génération de procédures suit l'algorithme suivant[149] :

1. si le nombre de sommets sélectionnés est inférieur à deux, l'algorithme affiche un message d'erreur et s'arrête.
2. calculer une collection ordonnée composée de nom de chacun des sommes.
3. calculer un nom pour la procédure générée, sur la base de la liste des noms calculée en (2).
4. s'il existe au sein du type de ligne brisée concerné une procédure qui pote déjà ce nom, demander confirmation pour le replacement.
5. pour chaque sagement passant par deux des sommets sélectionnés, créer une instance de la classe `PrimitiveProcedure` portant sur le descriptif de service `Calculer distance entre deux points ()`.
6. ajouter à la fin de cet ensemble une dernière instance de la classe `PrimitiveProcedure` qui porte sur le descriptif de service `Calculer la somme de N nombres ()`.Le rôle de cette instance est de calculer la somme des valeurs obtenues lors de l'exécution par l'activation de chacune des instances de `PrimitiveProcedure` lors de l'étape (5).
7. créer une instance de la classe `SequenceProcedure` qui comporte l'ensemble des instances de `PrimitiveProcedure` créées lors des étapes (5) et (6).
8. ajouter la procédure résultante de l'étape (7) dans la collection des procédures du type de ligne brisée en cours de définition.

4.2 Algorithme de détection de procédures

La détection de procédures lors d'instanciation suit l'algorithme suivant[150] :

1. si le type de ligne brisée instancicé n'a aucune procédure, on affiche un message d'erreur et l'algorithme s'arrête.
2. vérifier l'existence d'une procédure au sein du type de ligne brisée concerné, telle que les sommets choisis soient *nécessaires et suffisants* pour son calcul.
3. si une telle procédure n'existe pas, alors afficher un message et l'algorithme s'arrête.
4. sinon activer la procédure sur la valeur courante (la position à l'écran) de chacun des sommets.

[149] Cf. la méthode `tryToDefineAProcedureWith:` au niveau du code source.

[150] Cf. la méthode `handleRequestForProcedureGuessWith:with:` au niveau du code source.

5 Conclusion

Comme permet de le constater la Figure 107 ci-dessous, outre l'implantation basé sur le système de classes DYCRA-I et le framework DYCTALK, tel que nous venons de le décrire, nous avons également procéder à une implantation de ce langage d'experts suivant le système de classes DYCRA-II et le framework MIDYCTALK.

Cette second implantation nous a permis de valider une certaine équivalence entre ces deux modèles en ce qui concerne leur fonctionnalité commune, c'est à dire la spécialisation selon les compléments de classes (cf. chapitre II).

Nous avons en effet, mis en œuvre les algorithmes qui permettent de transformer une adaptation selon DYCRA-I en une représentation équivalente, sur le plan fonctionnel, selon DYCRA-II et *vis versa*. Il en est de même en ce qui concerne les instances des adaptations et compléments de classes.

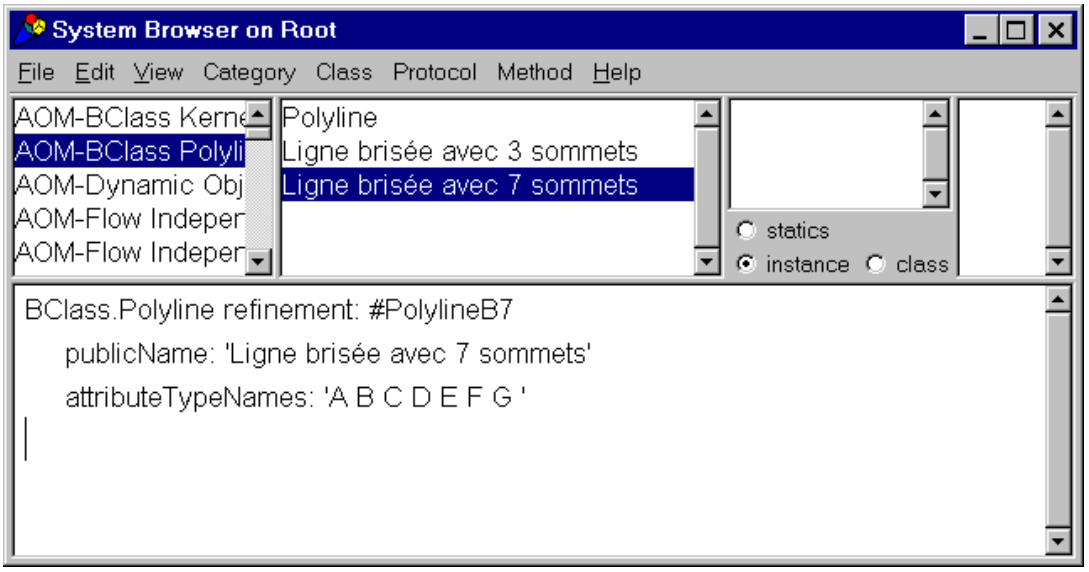


Figure 107 : Adaptations de la classe `Polyline`, vue à travers le flâneur du système VISUALWORKS.

Sur la base de ces expériences nous pouvons conclure sur le potentiel des langages d'experts à assurer l'évolution automatique de logiciels ou de leur composants, e.g. un agent au sens de [Fer95].

Une telle génération peut s'appuyer sur différentes techniques, comme de simples algorithmes décrits ci-dessus ou encore des mécanismes beaucoup plus élaborés comme des bases de règles à la NéOpus.

Il convient, toutefois, d'étudier de plus près en particulier la question de performances de l'outillage que nous proposons ici suivant le cas d'usage.

Annexe IV : Résumé du vocabulaire

1. langage d'experts (cf. §1.1.2, page 16 de l'introduction)
2. outillage (cf. §1.3.1, page 21 de l'introduction)
3. expert (cf. §1.2.2, page 20 de l'introduction)
4. outilleur (programmeur expérimenté)
5. compétemment de classe (cf. 2.2.2, page 92, du chapitre II)
6. adaptation (cf. §1.1.2, page 16 de l'introduction)
7. type d'adaptation (cf. §1.4, page 150 du chapitre V)
8. descriptif d'attribut (cf. §2.4.2, page 44 de l'introduction)
9. descriptif de service (cf. §2.3.4, page 33 de l'introduction)
10. référentiel de descriptifs de service (cf. §2.3.4.3, page 35 de l'introduction)
11. instance de descriptif de service et composition de procédures (cf. §2.1.1, page 57 du chapitre I)
12. définition d'appel de service (cf. §2.1.1, page 57 du chapitre I)
13. stratégie d'activation (cf. le paragraphe 2.2, page 59, ainsi que le paragraphe 3.3, page 67)
14. procédure, micro-composition, micro-procédé et macro-procédure (cf. §2, page 57 du chapitre I)
15. programmation par spécialisation (cf. §1.5, page 151 du chapitre IV)
16. choix local du type d'adaptation (cf. §2.3.8, page 40 de l'introduction, ainsi que §3, page 189 du chapitre V)
17. classe autonome (cf. §2.1 , page 207 des perspectives.)

Annexe V : Schémas de conception

Voici une description succincte des schémas utilisés dans la conception des systèmes de classes présentés dans ce mémoire :

1. **DOM** : utilise *Property List*, *Type Object*, *Strategy* et *Interpreter*.
2. **Micro-workflow** : utilise dans l'ensemble 13 schémas [Man00, pages 183-184]. Il s'agit de *Composite*, *Decorator*, *Execute Around Method*, *Facade*, *Manager*, *NullObject*, *Observer*, *Proxy*, *Property*, *Singleton*, *Strategy*, *Type Object* et *Variable Access Direct and Indirect*.
3. **DARC** : aucun schéma supplémentaire par rapport à ses composants DOM et le Micro-worklfow.
4. **DART** : utilise *Composite*, *Strategy*, *Type Object*, *Bridge*, *Observer* et *Value Holder*.
5. **DYCRA** : ajoute à ceux de ses composants DARC et DART les deux schémas *Mediator* et *Template Method*.

Tout n'est sûrement pas dit sur les schémas de conception, vaste sujet en lui même avec la prolifération actuelle de ce type de schémas. Mais il y a ici l'essentiel de ce qui est à ce jour mis en évidence dans ces systèmes de classes.

Annexe VI: Outillage du "typage métier"

Notre expérience industrielle montre qu'une certaine gestion du type des objets utilisés lors de la composition permet de faciliter sa mise en œuvre par les experts. Il s'agit de s'appuyer sur des informations fournies par les descriptifs de service afin de filtrer les objets possibles lors du choix d'un argument ou contrôler le type du résultat retourné par un calcul.

Ce qui suit correspond à la modélisation de cette gestion comme un composant des descriptifs de service (cf. la Figure 108 ci-dessous). Il s'agit donc d'une extension du système DART et plus particulièrement de la modélisation des descriptifs de service (troisième partie) que nous exposons ici brièvement.

Rappelons que la première et la second partie de cette modélisation sont présentées respectivement dans les pages 70et 72.

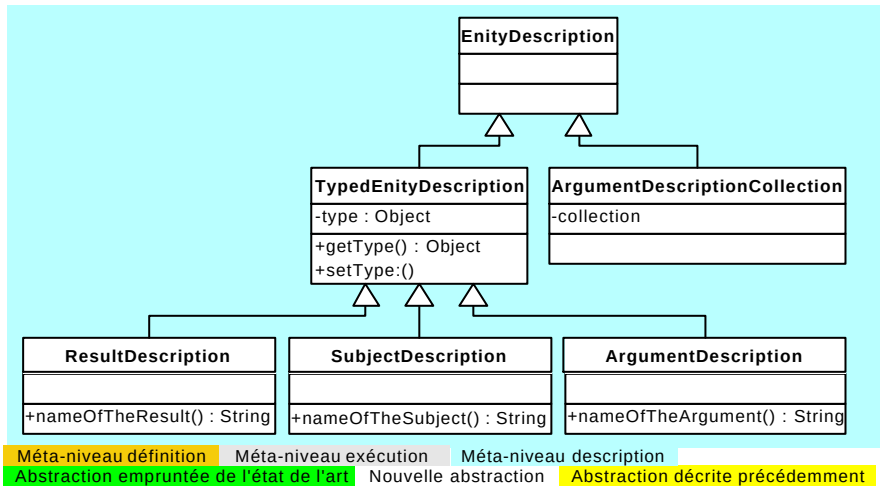


Figure 108 : Modèle de conception de la gestion des types dans DART.

La gestion de types métier est basée sur l'idée que chaque descriptif de service comporte des informations de type sur les différents éléments qui le composent.

Nous utilisons alors une nouvelle spécialisation de la notion de descriptif d'entité (la classe `EnityDescription`) pour définir le descriptif des entités typés (la classe `TypedEnityDescription`).

De cette notion nous dérivons ensuite trois autres : le descriptif du résultat (la classe `ResultDescription`), le descriptif du sujet (la classe `SubjectDescription`) et le descriptif d'argument (la classe `ArgumentDescription`).

Dans la mesure où les descriptif de service peuvent avoir besoin d'une liste d'argument, nous ajoutons également la notion de liste de descriptif d'arguments (la classe `ArgumentDescriptionColelction`).

Nous allons à présent détailler la conception de chacune de ces classes.

Nous tenons à préciser que ce modèle reste, toutefois, assez abstrait et son application effective nécessite des adaptations suivant le cas d'usage. Le but principal de notre exposé et de fournir les point d'encrage d'un tel mécanisme au sein de l'outillage que nous avons présenté dans ce mémoire.

Il est également important de noter que l'ajout de cette gestion de types métier a des conséquences sur l'implantation descriptifs de service ainsi que les stratégies d'exécution. Nous avons évoqué brièvement ces sujet lors de notre exposé de chacun d ces composants. Le code source de nos frameworks accessible via l'URL *http://www-poleia.lip6.fr/~razavi/Dyctalk/* comprend déjà une implantation de cette gestion.

La classe `TypedEnityDescription`

Le typage métier s'appui sur le type présumé de chaque composant d'un service, c'est à dire ses arguments et son résultat. Chacun de ces composants est décrit à l'aide d'une instance d'une sous-classe appropriée de la classe `TypedEnityDescription` qui représente les descriptifs d'entités typés.

Voici l'implantation de cette classe dans le langage SMALTALK-80/VISUALWORKS [Cin01]. Rappelons que la chaîne de caractères AM.EUP correspond au nom de l'espace de nommage (*naming space*) dans lequel ces classes sont implantées.

Définition

```
AM.EUP defineClass: #TypedEnityDescription
      superclass: #{AM.EUP.EnityDescription}
      indexedType: #none
      private: false
      instanceVariableNames: 'type '
      classInstanceVariableNames: ''
      imports: ''
      category: 'AM-10.5-End User Programming'
```

Structure

La variable d'instance *type* porte l'information sur le type autorisé pour l'entité décrite.

Protocoles

a. instance creation

```
TypedEnityDescription  >> name: aString type: aClass
      | aDescription |
      aDescription := super named: aString.
      aDescription setType: aClass.
      ^aDescription
```

b. class accessing

```
TypedEnityDescription  class >> forType: aClass
      "We use a default name."

      ^self forType: aClass named: aClass fullName
```

```
TypedEnityDescription  class >> forType: aClass named: aString
      ^self
            getEntityDescriptionNamed: aString
            ifAbsent: [self at: aString put: (self name: aString type: aClass)]
```

c. accessing

Ce protocole propose des messages d'accès en lecture et en écriture au type (messages `getType` et `setType:`).

La classe `ResultDescription`

Les informations de typage sur le résultat d'un service sont portées par une instance de la classe `ResultDescription`. Par exemple, l'exécution de l'expression `ResultDescription forType: Core.Number` conduit à création d'un descriptif pour un résultat du type nombre.

Définition

```
AM.EUP defineClass: #ResultDescription
      superclass: #{AM.EUP.TypedEnityDescription}
      indexedType: #none
      private: false
      instanceVariableNames: ''
      classInstanceVariableNames: ''
      imports: ''
      category: 'AM-10.5-End User Programming'
```

Structure

Sans objet.

Protocoles

d. private-process execution

```
ResultDescription  >> nameOfTheResult
      ^self getName
```

La classe `SubjectDescription`

Les informations de typage sur le sujet d'un service sont portées par une instance de la classe `SubjectDescription`. Par exemple, l'exécution de l'expression `SubjectDescription name: #myAccount type: Account` crée un descriptif de sujet dont le nom est #myAccount et dont le type est représenté par la classe `Account`.

Définition

```
AM.EUP defineClass: #SubjectDescription
      superclass: #{AM.EUP.TypedEnityDescription}
      indexedType: #none
      private: false
      instanceVariableNames: ''
      classInstanceVariableNames: ''
      imports: ''
      category: 'AM-10.5-End User Programming'
```

Structure

Sans objet.

Protocoles

e. private-process execution

```
SubjectDescription  >> nameOfTheSubject
      ^self getName
```

La classe `ArgumentDescription`

Les informations de typage sur les arguments d'un service sont portées par une instance de la classe `ArgumentDescription`. Par exemple, l'exécution de l'expression `ArgumentDescription name: 'balance' type: Number` crée un descriptif d'argument dont le nom est `balance` et dont le type est représenté par la classe `Account`.

Définition

```
AM.EUP defineClass: #ArgumentDescription
      superclass: #{AM.EUP.TypedEnityDescription}
      indexedType: #none
      private: false
      instanceVariableNames: ''
      classInstanceVariableNames: ''
      imports: ''
      category: 'AM-10.5-End User Programming'
```

Structure

Sans objet.

Protocoles

f. private-process execution

```
ArgumentDescription  >> nameOfTheArgument
      ^self getName
```

g. type checking

```
ArgumentDescription  >> checkArgTypeValidity: anObject
      (anObject isKindOf: self getType)
            ifFalse: [self notifyError: #'Incompatible Arg Type !']
```

LA classe `ArgumentDescriptionCollection`

Une instance de la classe `ArgumentDescriptionCollection` sert à stocker la liste des descriptifs d'arguments d'un descriptif de service. Par exemple, l'exécution de l'expression ArgumentDescriptionCollection `with: AM.EUP.ArgumentDescription balance with: AM.EUP.ArgumentDescription interestRate` conduit à la création d'une descriptif de liste d'arguments composé de deux arguments `balance` et `interestRate`.

Définition

```
AM.EUP defineClass: #ArgumentDescriptionCollection
      superclass: #{AM.EUP.EnityDescription}
      indexedType: #none
      private: false
      instanceVariableNames: 'collection '
      classInstanceVariableNames: ''
      imports: ''
      category: 'AM-10.5-End User Programming'
```

Structure

La variable d'instance `collection` contient une liste composée d'instances de la classe `ArgumentDescription`.

Protocoles

h. type checking

```
ArgumentDescriptionCollection >> checkTypeValidity: anObject forArgIndex: anInteger
      (self argAt: anInteger) checkArgTypeValidity: anObject
```

Annexe VII: Application au système Mobidyc

1 Contexte

MOBIDYC est un outil de création de MOdèles Basés sur les Individus pour la DYnamique des Communautés [GLP98]. Il a été élaboré dans le cadre du groupe de travail "dynamique des peuplements et systèmes multi-agents" associant l'INRA-Thonon, l'IRD (HEA et LIA), le CIRAD -Tera/Ere et l'Université Paris-VI (LIP6). Il a été financé par le programme national CNRS "Environnement Vie et Sociétés" dans le cadre de l'action thématique "biodiversité" du GIP HydrOsystèmes.

A ce jours, huit personnes ont activement participé au projet, totalisant plus de 6 années ingénieur et près de 1,5 millions de caractères de code répartis en 4500 méthodes et 250 classes[151].

2 Mode d'emploi

Les experts utilisateurs de MOBIDYC sont des chercheurs en dynamique des populations (biologistes). Ils utilisent ce système à deux niveaux : définition et ensuite simulation de modèles.

La définition de modèles s'appuie sur la notion d'agent [Fer95]. Le rôle de l'expert consiste à définir les trois types d'agents qui peuvent intervenir lors du déroulement d'une simulation (agents de l'espace, animats et agents non situés).

Définir un type d'agent consiste, *en gros*, à créer une instance de la classe `Moule` qui spécialise la classe `Agent`. Pour se faire, l'expert associe à cette instance des définitions d'attributs et de procédures. Chaque procédure est définie par la composition des primitives plus élémentaires, e.g. Moi, Mon Voisinage, Vivre, Tuer, etc..

[151] Source le site Web de Mobidyc : *http://www.thonon.inra.fr/mobidyc/*.

3 Motivations

Tout comme d'autres systèmes de ce genre dont nous avons brièvement parlé dans ce mémoire, e.g. CALIBRES, ARGOS, OBJECTIVA et UDP, en raison de l'absence d'un outillage standard pour la création de ce type de logiciels, les créateurs de MOBIDYC ont été amenés à concevoir et programmer leur propre solution.

Cette solution s'intéresse aussi bien à la spécialisation dynamique et la facilité d'apprentissage par des experts que la dimension lien causal. Elle met également en évidence la nécessité du travail collaboratif en déployant une technique de génération de code qui transforme certaines procédures composées par des experts en méthodes du langage SMALLTALK.

Afin d'assurer la spécialisation dynamique, ici la spécialisation de la classe `Agent` par des instances de la classe `Moule`, MOBIDYC met en œuvre une technique très semblable à celle de DYCRA-I (sur le plan conceptuel). En effet, l'ajout d'un nouveau type d'objets se fait sous forme d'un complément de classe à la DOM [RTJ00]. Les procédures y sont ajoutées sous forme de compositions d'instances de primitives. Un agent est donc une instance de la classe `Agent`, mais son comportement est apporté par une instance de la classe `Moule`.

Le but immédiat de ce projet a été d'intégrer le framwork DYCTALK au sein de MOBIDYC afin qu'en s'appuyant sur le système DART (cf. le chapitre I), les experts disposent de plus de souplesse lors de la composition de procédures.

En effet, à l'heure actuelle la notion de variable locale dans l'éditeur de composition de procédures de MOBIDYC n'existe pas. De ce fait, les experts sont invités à considérer que ce type de variables font partie des attributs caractéristiques de l'agent modélisé et donc de l'ajouter comme un descriptif d'attribut au sein du moule de cet agent.

Cette technique permet ensuite de spécifier l'affectation du résultat d'un certain calcul à cet attribut afin qu'un calcul ultérieur puisse s'en service à son tour.

De plus, l'expert ne peut pas spécifier explicitement les arguments nécessaires à l'appel d'une primitive. Chaque primitive ne peut, au maximum avoir qu'un seul argument en entrée et celui-ci est désigné par le système comme étant toujours le résultat du calcul précédent.

A plus long terme ce projet devait conduire au remplacement d'autres mécanismes actuel afin de permettre, à titre d'exemple, l'usage des types de service autre que l'envoi de message (type méthode), le seul à être assuré actuellement par MOBIDYC (tout comme le Micro-workflow).

Ce projet d'intégration devait, toutefois, se réaliser sous la contrainte de conserver le système MOBIDYC ainsi que ses mécanismes actuels opérationnels. Le *Refactoring* [Opd92, Rob99] joue donc ici un rôle primordial.

4 Mise en œuvre

La mise en œuvre de l'intégration préliminaire du framework DYCTALK au sein de MOBIDYC s'est déroulé en trois étapes.

4.1 Extraction automatique de descriptifs de service

MOBIDYC implante chaque service eu sens de DYCTALK sous forme d'une classe, sous-classe de la classe `Primitive`. Chacune de ses classes dispose d'un nom au claire ainsi qu'une description de l'unique argument en entrée et du résultat en sortie (le cas échéant)[152].

[152] MOBIDYC dispose des éditeurs graphiques qui permettent également à l'expert de créer relativement facilement de nouveau types de primitives (tâches générées).

Notre première action a alors consisté à extraire cette information de chaque primitive pour créer le référentiel de descriptifs de service initial de MOBIDYC.

Pour ce faire, nous avons été amené à spécialiser certaines des classes de DYCTALK :

1. la classe `ArgumentDescription` est spécialisée par la classe `MobidycArgumentDescription` afin de gérer l'attribut dimension associé aux services du type MOBIDYC.
2. la classe `ResultDescription` est également spécialisée par la classe `MobidycResultDescription` afin de gérer l'attribut dimension associé aux services du type MOBIDYC.
3. la classe `ServiceDescription` est spécialisée par la classe `MobidycServiceDescription` afin de gérer les variables d'instance `primitiveClassName`, `interfaceClassName`, `interfaceWinSpecName` et `agentType` qui caractérisent les services du type MOBIDYC.
4. la méta-classe `Refinement class` est spécialisée par la méta-classe `MobidycBridge` afin de gérer le référentiel des descriptifs de service de MOBIDYC.

Un fait remarquable est que certaines primitives de MOBIDYC représentent de fait une *classe de primitives*, car certaines informations peuvent être modifiées au niveau de chaque instance. Cette caractéristique se répercute au niveau des descriptifs de service associés.

En effet, à chaque primitive implantée sous forme d'une classe peuvent aussi être associés une multitude de descriptifs de service. Cela met à défaut notre moulinette d'extraction automatique et conduit à la nécessité d'une étude au cas par car qui est actuellement en cours de réalisation. Le but de cette étude est de déterminer l'ensemble (fini) des descriptifs de services associé à chaque classe de primitive.

4.2 Rendre DYCTALK adapté à la composition de services du type MOBIDYC

La seconde étape à consisté à la spécialisation d'une autre série des classes de DYCTALK afin de l'adapter à la composition de services du type MOBIDYC :

1. la classe `ObserverServiceCallDefinition` est spécialisée par la classe `MobidycObserverServiceCallDefinition` afin de gérer l'instance de la classe MOBIDYC qui implante le service concerné.
2. la classe `MessageSendServiceActivationStrategy` est spécialisée par la classe `MobidycMessageSendServiceActivationStrategy` afin de modifier la sratégie d'activation qui consiste ici à envoyer le message executer ou executer: à l'instance de la classe MOBIDYC qui implante le service concerné et qui est stocké par la définition d'appel de service correspondant (cf. 1).
3. la classe `MicroCompositionComponent` est spécialisée par la classe `MobidycComposition` afin de gérer l'ajout dans la matrice de composition des services du type MOBIDYC (cf. la sous-section précédente).

Ce travail se poursuit au fur et à mesure que notre projet avance.

4.3 Création d'un éditeur de composition

La dernière étape a consisté à créer un éditeur de composition de procédures et de macro-procédures (cf. Figure 109, page suivante), par la réutilisation des éditeurs existants de MOBIDYC. Celui-ci affiche à gauche de la fenêtre la liste des services disponibles. Celle-ci est calculée suivant le type de l'agent en cours de définition.

Ensuite, l'expert choisit dans cette liste le service à instancier. A ce moment là, c'est 'abord la primitive associée au service qui est instanciée et ensuite, le cas échéant, l'éditeur calcule et affiche, pour chaque argument en entrée, la liste des instances de descriptifs de service déjà présents dans la

composition et qui sont d'un type compatible avec le type de l'argument concerné (cf. Figure 109, page suivante). L'expert est alors invité à choisir dans cette liste l'instance de descriptif de service qui convient.

Une zone de saisie (en haut de la fenêtre) permet à l'expert de fournir le nom de la procédure. Un bouton permet également à l'expert de demander la transformation de la procédure composée sous forme d'un nouveau descriptif de service (macro-procédure).

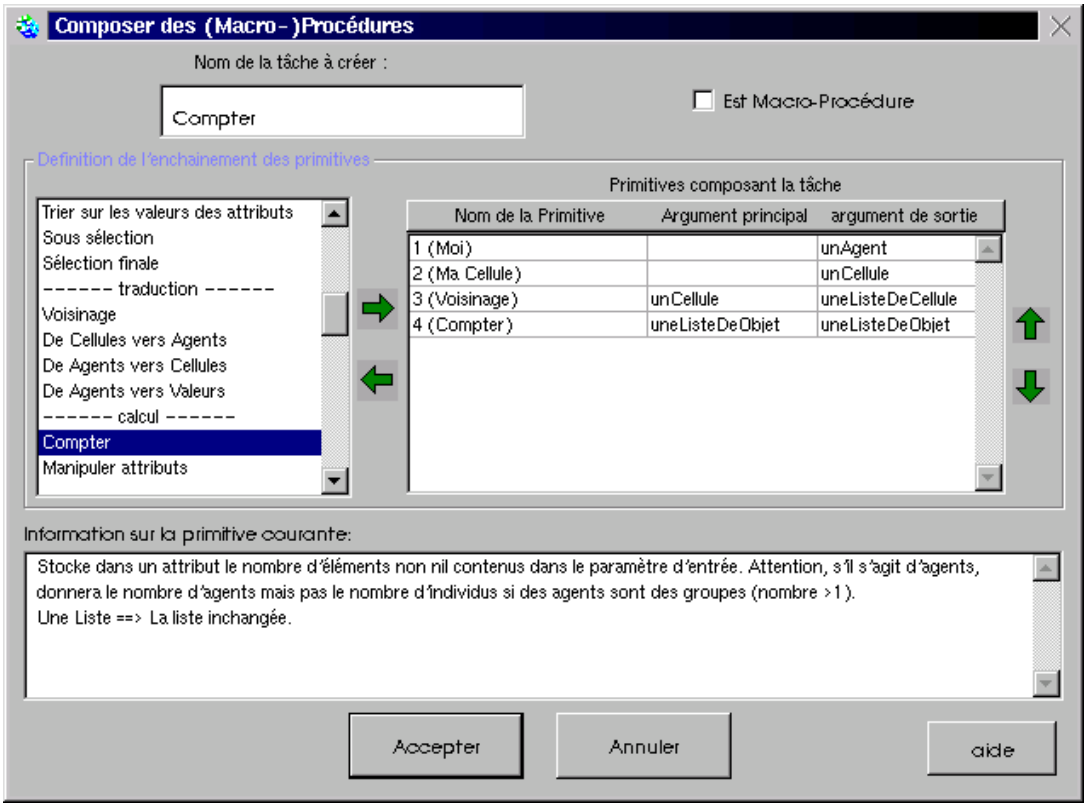


Figure 109 : L'éditeur de composition de procédures à la DYCTALK de MOBIDYC (en cours) .

Lorsque l'expert valide l'édition d'une procédure, celle-ci est ajouté dans la liste des sous-actions de l'action en cours d'édition de l'agent courant.

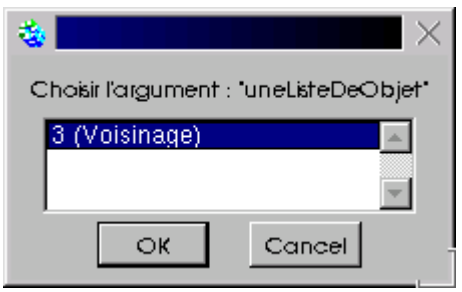


Figure 110 : Filtrage des arguments, ici lors de l'instanciation du service Compter.

5 Conclusions

Ce projet n'est pas encore terminé. Toutefois, les premiers résultats obtenus sont encourageants. Grâce notamment aux points d'extension prévus au sein de DYTALK, mais aussi à l'application soigneuse des techniques de *Refactoring* [Opd92, Rob99], nous avons pu atteindre une première série de nos objectifs.

Relativement peu d'efforts ont ainsi, dors et déjà, permis d'enrichire MOBIDYC par un éditeur de micro-compositions à la DART. Ce jeune éditeur dois, toutefois, être raffiné et complété notamment par la fonction d'édition des structures de contrôle, conditionnelles et itérations, ainsi que par l'édition des appels successifs de sous-procédures.

Cette expérience a, par ailleurs, conduit à l'enrichissement de DYCTALK. En effet, la technique de l'utilisation de classes pour implanter les primitives, déployée par MOBIDYC, a été intégrée au sein de DYCTALK comme un nouveau type de service.

Nous pouvons donc conclure non seulement sur l'effectivité de DYCTALK comme outil pour faciliter la création de langages d'experts, mais également sur sa réutilisabilité et extensibilité, qualités indispensables à tout système s'estimant comme un framework orienté-objets.

Annexe VIII : A propos de nos activités industrielles

Contexte

Entre les années 1993 à 1998 nous avons participé à des projets industriels, notamment pour le développement de deux familles de logiciels de *Métrologie*. Le premier projet, *CALIBRES*, s'intéressait à l'étalonnage des instruments de mesure (calibres) et le second, *PRELUDE INSPECTION*, à la vérification de l'aptitude d'une pièce usinée à remplir les fonctions spécifiées par son cahier des charges (*Contrôle 3D).*

Ces projets étaient initiés par des industriels français. Leur mise en œuvre avait été confiée aux experts en *Métrologie* ainsi qu'à des professionnels en *Informatique*. Ces deux projets s'inscrivaient, par ailleurs, dans la continuité du projet *MARLENE*.

Le projet *MARLENE* été également initié par des industriels français et réalisé, entre 1992-93, par des experts en *Métrologie* et en *Informatique* ainsi que des scientifiques spécialisés notamment dans la *programmation par objets* et le *génie logiciel*. Il s'agit plus précisément de la Sté ACKIA.

MARLENE, de part cette notoriété et ses aspects novateurs (un seul environnement pour le contrôle surfacique et mécanique, gestion automatique de l'ordonnancement des gammes de mesure, gestion des gammes multiples, environnement Windows, etc.), a alors été considéré par certains milieux comme le successeur légitime de *PERCEVAL*.

PERCEVAL est le logiciel de Contrôle 3D homologué notamment par la société *RENAULT SA*. Lui, ainsi que d'autres logiciels dans ce domaine (e.g. MESUVOL, LIMA) faisaient partie de la « palmarès » de certains intervenants sur le projet *MARLENE*.

L'architecture de *MARLENE* était composée d'un noyau écrit en *SMALLTALK*-80 et de modules externes. Le noyau comprenait environ 600 classes d'objets. Il gérait toutes les fonctions de conception, de mise au point et d'exécution des gammes de mesure. Mais, il déléguait aux modules externes spécialisés les tâches de communication avec les machines à mesurer 3D, les visualisation 3D ainsi que la génération de rapports de contrôle.

MARLENE représentait, approximativement, un effort de huit années/hommes et un budget de 3 MF. A titre indicatif, le prix de chaque licence de *MARLENE* se situait entre 40 à 180 KF.

Curriculum d'une *start-up* française

La société *JULIA SA* est née en Mai 1992 dans le but d'assurer la création et la commercialisation de *MARLENE*. Parmi les fondateurs et principaux intervenants de *JULIA SA* figuraient des personnages connus dans le milieux du *Contrôle (3D)*. Cette notoriété était, tout particulièrement, dûe au développement réussi du logiciel *Perceval*.

Dans cette tâche *JULIA SA* était épaulée par d'autres sociétés notamment *METROLEC*, une filiale de la société *MFO* (actuellement *CMA*). *METROLEC* était pour ce projet partenaire de *JULIA SA* dans le développement des drivers de machines à mesurer 3D ainsi que la commercialisation de *MARLENE*. La société *METROLEC* est aujourd'hui partenaire de *MATRA DATAVISION* dans la commercialisation de *PRELUDE INSPECTION* (le successeur de *MARLENE*).

JULIA SA était en activité entre 1992 et 1998. Durant cette période elle a conçu et réalisé plusieurs projets de développement de logiciels, notamment celui de *MARLENE*, *MARLENE ENQUETE*, *PRELUDE VIEW & MARKUP* et *MINI MEAS*.

JULIA SA disposait également d'un département de services en *Contrôle 3D*. Ce département était composé d'experts dans le domaine du *Contrôle 3D* qui ont participé très activement dans les tests et la validation des logiciels développés en son sein.

Notre rôle

Par rapport aux projets auxquels nous avons participé, à savoir *MARLENE*, *CALIBRES* et *PRELUDE INSPECTION*, les activités de *JULIA SA* peuvent, idéalement, être classées en trois phases.

La première phase s'est déroulée entre 1992 à 1993 et a donné lieu à la création du logiciel *MARLENE*. La seconde phase, entre les années 1994 et 1995, a permis la création de la ligne de produits *CALIBRES*. Et, la troisième phase, entre 1996 et 1998, a donné lieu à la ligne de produits *PRELUDE INSPECTION*.

Notre collaboration avec la société *JULIA SA* a débuté en mai 1993 et a duré jusqu'à la fin de ses activités en début de l'année 1998.

Elle a consisté essentiellement à mettre au point un framework orienté-objets pour la création de certaines applications de métrologie dimensionnelle. Celui-ci est créé sur la base du logiciel *MARLENE*. De cette plate-forme, nous avons dérivé deux applications dédiées à la programmation par des experts.

Le premier, ATELIER DE SPECIALISATION METIER [ASM99], est intégré au logiciel *PRELUDE* INSPECTION, actuellement commercialisé au niveau mondial par la société *MATRA DATAVISION*.

Le second, *CALIBRES* [Raz00a], permet depuis 1996 aux experts en métrologie du LABORATOIRE JEAN GOUY à NIHERNE de produire eux-mêmes des logiciels d'étalonnage pour différents types de calibres (BAGUE, TAMPON, RAPPORTEUR, etc.).

Nos activités industrielles, depuis notre DESS en génie logiciel obtenu en 1993, étaient donc centrées sur la mise en œuvre pratique des techniques avancées de modélisation par objets. Il s'agissait, en termes plus abstraits, de créer des logiciels dédiés à la définition et la simulation de modèles par des experts non-informaticiens.

Résumé

L'utilisation de certaines applications nécessite deux niveaux d'intervention : la mise en oeuvre courante et l'adaptation à de nouveaux besoins. L'idée est que la spécification du service rendu par le logiciel peut varier au cours du temps. Il y a donc d'une part les utilisateurs habituels, qui utilisent le logiciel pour obtenir le service en question. Mais il y a également des utilisateurs privilégiés qui peuvent apporter au système, alors même qu'il est en fonctionnement, des adaptations qui viendront modifier le service obtenu par l'utilisateur final.

Nous appellerons "experts" ces utilisateurs privilégiés et "langage d'experts" ce type d'applications. Dans la suite, nous supposons qu'elles sont créées à l'aide de langages à objets.

Sur la base de notre expérience industrielle en création de langage d'experts, nous classifions en deux catégories les propriétés que nous estimons souhaitables pour des langages d'experts. La première catégorie comporte les aspects techniques, c'est-à-dire ceux qui rendent l'adaptation possible, opérationnelle et adaptée à son objet. Elle comprend quatre volets : la spécialisation dynamique, le workflow, l'édition des adaptations par les programmeurs et le "refactoring" (restructuration), ainsi que le choix local du type d'adaptation. La seconde catégorie regroupe les aspects plutôt d'ordre cognitif. Ce sont ceux qui rendent l'adaptation facile à réaliser et bien intégrée dans un processus de développement de logiciels. Elle comprend deux volets : l'apprentissage par des experts et le lien causal.

Notre thèse est qu'il est possible de créer et documenter, notamment à l'aide des schémas de conception, un nouveau framework orienté-objets qui "outille" la création de langages d'experts remplissant pleinement le cahier des charges ci-dessus.

En effet, à ce jour les environnements de programmation ne supportent pas ce type de développement de façon standard. Par ailleurs, Ralph Johnson et son école à UIUC ont initié depuis 1998 des travaux de recherche dans un but similaire sous le thème appelé "Adaptive Object Models" (AOMs). L'un des problèmes majeurs actuellement posé par ces travaux consiste à outiller la création de logiciels objets qui assurent la spécialisation lors de l'exécution, c'est-à-dire la définition et l'interprétation de procédures qui opèrent sur des structures qui sont elles-mêmes définies dynamiquement.

Nous partons des recherches de Ralph Johnson et de son école sur les AOMs et sur l'outillage de la création de systèmes de gestion de Workflow. Nous nous appuyons également sur les travaux de l'équipe de Pierre Cointe sur les méta-classes explicites, ainsi que ceux de Bonnie A. Nardi sur la programmation par des experts. En partant de ces bases, nous documentons et implantons trois frameworks (DYCTALK, MIDYCTALK et MXDYCTALK) qui se complètent afin de nous conduire vers une solution satisfaisante au problème posé.

Celle-ci ébauche une *compatibilité* entre les structures de représentation de la définition et de l'interprétation de programmes/modèles des langages à objets et celles des langages d'experts.

Nous introduisons enfin les notions d'"adaptation prototypique" et de "classe autonome" comme une technique qui permet de rendre les instances terminales adaptables et d'assurer ainsi la *continuité* du processus d'adaptation.

Ces travaux sont en ce moment testés sur le langage d'experts MOBIDYC de l'INRA (définition et simulation d'agents).

Mots-clés

Modèles Objets Adaptatifs, Frameworks, Schémas de Conception, SMALLTALK-80, Réflexion, Meta-classes et Architectures à méta-niveaux, Programmation par des experts, Workflow.

Keywords

Adaptive Object-Models, Object-oriented Frameworks, Design Patterns, SMALLTALK-80, Reflection, Meta-classes and Meta-level Architectures, End-user Programming, Workflow.