Définition et réalisation d'une boîte à outils générique dédiée à la Programmation sur Exemple

Loé Sanou

École Nationale Supérieure de Mécanique et d'Aérotechnique
École Doctorale des Sciences Pour l'Ingénieur & Aéronautique Université de Poitiers

THESE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National — Arrêté du 7 Aout 2006)

Secteur de Recherche : INFORMATIQUE et APPLICATION

Présentée par

Loé SANOU

****

Définition et réalisation d'une boîte à outils générique dédiée à la Programmation sur Exemple

****

Directeurs de Thèse : ***Patrick GIRARD*** *et* ***Laurent GUITTET***

****

Soutenue le 17 Décembre 2008
devant la Commission d'Examen

****

JURY

Président :	Dominique SCAPIN	Directeur de Recherche, INRIA, Rocquencourt, Paris
Rapporteurs :	Christophe KOLSKI	Professeur, LAMIH - Université de Valenciennes, Valenciennes
	Franck POIRIER	Professeur, Valoria - Université de Bretagne Sud, Vannes
Examinateurs :	Yamine AÏT-AMEUR	Professeur, LISI - ENSMA, Poitiers
	Mesmin DANDJINOU	Maître Assistant, ESI-UPB, Université Polytechnique de Bobo-Dioulasso, Burkina Faso
	Patrick GIRARD	Professeur, LISI-ENSMA, Université de Poitiers, Poitiers
	Laurent GUITTET	Maître de Conférence, LISI-ENSMA, Futuroscope, Poitiers

À mon Père,
mes parents …

À ma famille,
mon épouse Marie-Cécile,
et ma fille Maëlle Yé Françoise.

Remerciements

La « Page » de Remerciements est quelque chose de difficile pour moi. Trouver le bon ton, en dire un peu mais pas trop, l'élan du cœur quand on reconnaît et qu'on vit la beauté de la vie avec son entourage, dire au revoir pour ne pas dire qu'on s'en va. Pourquoi s'expliquer avant de remercier et pas simplement dire quelques choses pouvant être qualifiée de merci ?

Par ces quelques mots, je voudrais remercier chacun, avec une pensée particulière. Mais il est des moments de la vie où toutes les choses sont si difficiles qu'il est même gênant pour ceux qui sont remerciés de recevoir ces remerciements par quelqu'un dont la situation ne leur laisse indifférents.

Dans ces quelques années, j'ai trouvé des parents, des amis, de véritables amis, autant que la vie peut en donner en quelques années, et c'est une grande chance. Certains le sont devenus résolument. D'autres hésitent, tout comme moi peut-être un peu frileux.

Il n'y a pas que ceux qui m'ont offert leur amitié et que je ne pourrais jamais assez remercier, mais aussi tous ceux qui ont simplement été présents, honnêtes avec eux-mêmes et avec les autres, faisant de leur mieux, chacun selon ses affinités et capacités.

À tous ceux-là, je voudrais dire merci, car c'est grâce à eux que j'ai pu avancer et finalement aboutir.

Permettez-moi de singulariser quelques personnes. Alors moi, j'ai à remercier :

Patrick GIRARD, un père adoptif, dont la confiance, le soutien et les conseils m'ont encouragé tout au long de ces années, tout en me laissant libre d'explorer. Ma famille adoptive, la famille GIRARD, une vraie famille, comment dire merci ! GRAND MERCI maman Françoise, mamie, Sylvie, Yannick, Claire.

Yamine AÏT-AMEUR, (ah ! qu'il est génial), toujours disponible et prêt à m'apporter son aide ; merci à sa famille ; Guy PIERRA, toujours sage dans ses décisions et conseils ;

Laurent GUITTET, avec toujours de bonnes idées, de larges ouvertures du travail, et surtout toujours disponible, et apporte toujours son aide ;

Sybille CAFFIAU, ma sœur d'équipe, collègue de bureau, amie, que puis-je dire, le mot "merci" n'est pas suffisant ainsi qu'à toute sa famille ; Ahmed RAHNI, d'après certains, mon alcolite de tous les temps, reste toujours « The Big » ;

Je voudrais aussi que tous mes collègues et anciens collègues sachent que j'ai une pensée sincère pour chacun d'entre eux. Je voudrais remercier plus particulièrement certains d'entre eux qui comptent aujourd'hui parmi mes meilleurs amis, Mickaël BARON, Stéphane JEAN, Nicolas GUIBERT, Hondjack DEHAINSALA, Sylvain CHARNEAU, Jeannette C. FANKAM, Antoine BERGEY, Patrice DENIS, Romain FLOHIC, d'autres d'ailleurs Moustapha BANDE, Moussa NOMBRE, Moussa DAVOU, A. Charles COULIBALY, Docteur ZOMBRE C., Issa TRAORE (Seba), Amadou TRAORE (GSK), Rasmata TOURE, Fatoumata TRAORE/DRABO. Ils m'ont accompagnée dans mes moments si difficiles que je ne peux en parler plus ici. La chaleur de leur amitié m'a aidé à ne pas plier.

Et puis il y a Jean-Claude, Dago, Guillaume, Eric, et tous les chefs, collègues et amis : Dominique G., Annie G., David M., Ladjel B., Michel K., Mikaël R., Pascal R., Claudine R., Frédéric C., Manu G., Karim T., Sadouanouan M., Kamel B., Chedlia C., Idir A., Nabil B., Gayo D., Christian F., Valérie T., François D., Dung N., Hung, Hieu, les burkinabè du Poitou-Charante.

Enfin il y a ma famille, l'ancienne, originelle, la grande, la belle, auxquelles je dois tout ce que je suis ; celle que j'ai construite, réduite à ma tendre et douce épouse Marie-Cécile, et notre enfant Maëlle; et celle de demain, dont chacun, je crois, m'a fait le présent de m'accepter.

Merci à

*... **Patrick GIRARD** pour m'avoir accueilli au sein de l'équipe IHM et pour la formation acquise au long de ces quatre années de thèse. Merci aussi pour les nombreuses conférences auxquelles j'ai eu le plaisir d'assister. Merci pour tout le soutien, la compréhension et la confiance à mon égard.*

*... **Christophe KOLSKI** et **Franck POIRIER** de m'avoir fait l'honneur d'être les rapporteurs de ce travail.*

*... **Dominique SCAPIN** d'avoir accepté de présider ce jury, et aussi pour son intérêt à mon travail.*

*... **Laurent GUITTET** pour l'encadrement du travail et surtout pour les débats d'ouverture et de compréhension des travaux durant cette thèse. Merci aussi pour les conseils et les aides.*

*... **Mesmin DANDJINOU** pour son consentement à faire partie du jury.*

*... **Yamine AIT-AMEUR** (directeur du LISI) et **Guy PIERRA** (ancien directeur), pour l'accueil au sein du LISI, et surtout de leur soutien personnel respectif. Merci aussi à Yamine pour sa participation à mon jury.*

*... à **Claudine RAULT**, assistante de direction du LISI, pour toutes les tâches administratives accomplies pour moi, pour sa bonne humeur (surtout matinale) donnant toujours du baume au cœur d'être au LISI, et aussi pour son amitié.*

*... à **Fréderic CARREAU**, administrateur système du LISI, pour les nombreux services techniques et ses interventions multiples pour le bon fonctionnement de mon matériel. Merci pour m'avoir obligé à respecter les droits et les licences.*

*... à toute la fine **équipe IHM** : ceux de passage et ceux qui y étaient, y sont et y seront encore quelques temps.*

*... à tous ceux qui ont toujours su avoir un petit mot agréable aux détours des couloirs du bâtiment du **LISI**.*

*... au **Service de Coopération et d'Action Culturelle** de l'Ambassade de France au Burkina, pour le financement de cette thèse.*

*... aux autorités de l'**Université Polytechnique de Bobo-Dioulasso** (UPB), plus particulièrement la direction de l'**Ecole Supérieure d'Informatique** (ESI).*

Résumé

L'implémentation d'un système intégrant la Programmation sur Exemple (PsE) demande au développeur de mettre à disposition de l'utilisateur final des outils d'assistance lors de la réalisation des tâches. Pour le développeur, cela passe par la mise à disposition des différents services à partir de l'interface utilisateur de l'application. Le système doit donc fournir des interfaces particulières, car non seulement l'objectif fonctionnel de l'application ne doit pas changer, mais surtout parce que les techniques de la PsE doivent être naturellement intégrées. Un système de PsE est difficile à implanter, et pourtant, la plupart possèdent des éléments en commun parmi lesquels on trouve une représentation des actions utilisateur, un historique des actions, et parfois un algorithme d'apprentissage symbolique opérant sur l'historique. Nous favorisons la création d'un tel système en fournissant les outils nécessaires sous forme d'une boîte à outils par extension de Swing. Les développeurs peuvent bâtir, avec un minimum d'effort, des applications mettant en œuvre les techniques de la PsE.

Les principaux services de base ont été identifiés et définis : enregistrement des actions utilisateur, rejeu des actions et des techniques utilisables pour la mise en œuvre d'applications types. Ils ont été prototypés à travers l'outil *PbDToolkit,* ouvrant la voie vers la simplification de la mise en œuvre des applications de PsE. En utilisant *PbDToolkit*, il n'est pas nécessaire d'implémenter les fonctionnalités de base car toutes les opérations y sont déjà implémentées avec la liberté d'usage et d'exploitation offerte aux développeurs.

PbDToolkit est instrumenté pour permettre de vérifier la conformité d'une IHM à son modèle de tâches. Le concept établit un lien entre les tâches élémentaires du modèle de tâches et les actions de l'IHM. Ainsi, à l'exécution, un scénario est généré suivant le format de scénario de l'environnement K-MADe, outil de modélisation utilisé.

Abstract

The implementation of a system integrating Programming by Demonstration (PbD) requires the developer to provide for the end user, easy to use automation tools to help in task realization. For the developer, it passes through the provision of different services from the application user interface. System must provide special interfaces because not only the functional purpose of application must not change, but also the PbD techniques must be naturally integrated. A PbD system is difficult to implement, yet most have elements in common among them are user actions representation, historic action, and sometimes a symbolic learning algorithm operating on the historic. We favor creation of such systems by providing the necessary tools in a toolbox by extension of Swing. Developers can build, with minimal effort, applications implementing the PbD techniques.

The main basic services have been identified and defined: user actions recording and replay, and technology used for implementing standard applications. They have been through the prototype tool *PbDToolkit*, paving the way towards simplifying the PbD applications implementation. Using *PbDToolkit*, it is not necessary to implement the basic features as all operations are currently implemented with the freedom of use and exploitation offered to developers.

PbDToolkit is instrumented to verify the compliance of a GUI to its task model. The concept establishes a link between the basic tasks of the task model and the GUI actions. Thus, at execution, a scenario is generated following the format of the K-made environment scenario, the modeling tool used.

Table des matières

Liste des figures

Liste des tableaux

Liste des exemples de code

Liste des abréviations et des sigles

AWT : Abstract Windows Toolkit

DTD : Document Type Defintion

IHM : Interaction Homme-Machine ou Interface Homme-Machine (suivant le contexte)

K-MADe : Kernel of Model for Activity Descritption environment

PbD : Programming by Demonstration

PsE : Programmation sur Exemple

XML : eXtensible Markup Language

Introduction générale

L'avènement de l'ordinateur personnel et la croissance du nombre d'utilisateurs ont rendu caduque l'ère des logiciels sur mesure. Actuellement, un logiciel est construit pour être utilisé dans plusieurs types d'usages, par un public nombreux composé en majorité de non-programmeurs. De ce fait, les produits sont de plus en plus impersonnels.

Dans la plupart des logiciels existants notamment dans les interfaces à manipulation directe, il n'est pas rare que l'utilisateur ait à exécuter des tâches répétitives ou veuille intégrer ou faire une correspondance avec un autre système dans l'objectif de résoudre un problème spécifique ou d'automatiser une tâche particulière. Malheureusement, une personne ne sachant programmer n'est pas capable de faire comprendre à un système ce qu'elle aimerait lui voir accomplir, si cette tâche ou action n'a pas été prévue par le concepteur ou le développeur de l'application. Comment faire en sorte qu'un utilisateur final, ignorant tout des principes de la programmation, puisse personnaliser les logiciels qu'il utilise ? S'il est capable d'exécuter des actions dans un environnement donné, pourquoi ne pas mettre à sa disposition les moyens nécessaires afin qu'il puisse créer ses propres actions, tout en restant dans son environnement familier ? Ces propos peuvent êtres appuyés par la citation suivante de Cypher :

> *« Si l'utilisateur sait effectuer une tâche avec l'ordinateur, cela devrait être suffisant pour créer un programme accomplissant cette tâche. Il ne devrait pas être nécessaire d'apprendre un langage de programmation comme C ou Basic. Au lieu de cela, l'utilisateur devrait être à même d'instruire l'ordinateur de "faire ce qu'il fait", et l'ordinateur devrait créer le programme qui correspond aux actions de l'utilisateur. »*
>
> *Allen Cypher (Cypher, 1993c)*

Dans ce chapitre introductif, nous allons présenter les différentes solutions qui ont été proposées en suivant leurs évolutions. Nous nous attarderons sur la programmation sur exemple, l'une des solutions considérées comme la plus satisfaisante dans le domaine de la programmation utilisateur. À la suite de la présentation de cette évolution, nous parlerons des tentatives de résolution générale de la programmation utilisateur. Nous terminons ce chapitre par notre motivation pour le thème du travail proposé.

1 Définitions de la PsE

La PsE, acronyme de « Programmation sur Exemple », est une traduction du terme anglais « *Programming by Demonstration* ». Cette traduction a été donnée par P. Girard dans (Girard, 2000). En effet, le mot anglais « *demonstration* » ne correspond pas exactement au mot français « démonstration ». Ainsi, « *Programming by Demonstration* » ne se traduirait pas en français par « *Programmation par Démonstration* ». Dans les sections à venir nous parlerons en détail de la naissance du concept jusqu'au terme utilisé tout au long de cette rédaction.

La PsE est un concept ayant donné un sens beaucoup plus intuitif à la programmation (Myers et al., 2000). L'introduction des systèmes de programmation sur exemple a été une très grande évolution dans le cadre de la programmation utilisateur. Halbert a recensé et comparé les approches permettant à l'utilisateur final (non-programmeur) de créer des programmes (Halbert, 1984). L'idée générale est qu'il est plus facile de comprendre le programme que l'on cherche à écrire si celui-ci s'exécute parallèlement à sa construction.

Le concept de la PsE trouve son origine dans le système Pygmalion (Cypher, 1993c) réalisé par David Smith en 1975. Il a été approfondi par Halbert en 1984 lors de sa thèse (Halbert, 1984). En 1990, Brad Myers formalise le concept et lui donne le nom de *Example-based Programming*. Plusieurs travaux de recherche se sont succédés sur le sujet ; nous y reviendrons au début de la section 2 de ce chapitre.

Des écrits et des pratiques, il se dégage de toute évidence que l'étape la plus intéressante dans l'évolution des solutions permettant à l'utilisateur final de modifier, programmer son application ou automatiser une tâche répétitive a été celle des familles de la programmation à partir d'exemples. Mais, en réalité, qu'est-ce que la programmation à partir d'exemple ? Afin de mieux appréhender cette notion, nous allons faire un tour d'horizon de ce qui est vite devenu un thème de recherche.

Le précurseur dans les années 80 des techniques nouvelles de programmation par les non-programmeurs semble être Shneiderman qui effectue en 1983 une première étude de la « manipulation directe » (Shneiderman, 1983), après de nombreux travaux consacrés à l'usage du graphique et de l'interaction graphique dans diverses tâches (programmation, accès aux bases de données, etc…). Il pose ainsi les bases de ce que certains appellent déjà la programmation visuelle (« *Visual Programming* », (MacDonald, 1982)). C'est à partir de là que les travaux de (Halbert, 1984) et (Myers, 1986; 1988; 1990) précisent la codification pour ce qui concerne la notion d'exemple d'exécution en tant qu'outil d'aide à la définition d'un programme.

1.1 PsE selon Halbert et Myers (Halbert 84, Myers 86)

Les notions de (Shneiderman, 1983) ne sont guère utilisables pour évaluer l'aide réelle que peut apporter un système de la programmation visuelle dans l'activité de programmation par un non-programmeur. Dans sa classification, le fait que l'utilisateur doive manipuler graphiquement ou non des variables (ce qui constitue la première difficulté de la programmation), ou que celui-ci puisse se contenter de manipuler des valeurs, n'apparaît pas. (Halbert, 1984) et (Myers, 1986) analysent cette importante distinction.

L'idée générale des travaux de (Halbert, 1984) est qu'il est plus facile de comprendre le programme que l'on cherche à écrire si celui-ci s'exécute parallèlement à sa construction. C'est dans le cadre de la programmation basée sur exemple (« Example-Based Programming ») que (Halbert, 1984) définit un critère secondaire sous le terme d'*inférence*. Un système est dit « *avec inférence* » s'il est capable de « déduire » un programme des actions de l'utilisateur, sans que ce dernier ait explicitement dit comment le faire.

En marge de ses travaux sur le système PERIDOT (Myers, 1993a) sur la programmation d'interface utilisateur, Myers analyse le domaine de la programmation visuelle à la suite de Halbert. S'appuyant sur la définition de (Halbert, 1984) pour la partie « exemple », il propose une taxonomie qu'il fait évoluer au fil des publications (Myers, 1986; 1988; 1990). Son idée générale est de classer toutes les approches de la programmation selon des critères orthogonaux. Nous reviendrons sur cette taxonomie dans la sous-section 3.1.1 du chapitre 1.

Le paradigme de la PsE est formalisé par Halbert et Myers par la définition : « *un système est dit de la programmation sur exemple si une instance d'exécution se déroule parallèlement à la conception du programme* ».

1.2 PsE selon Cypher (Cypher 93)

À la suite d'Halbert et de Myers, Cypher formalise la démarche de la PsE comme suit : « le fait qu'un utilisateur soit capable d'exécuter une tâche dans un environnement donné devrait être suffisant pour que le système soit en mesure de créer un programme qui exécute cette tâche » (Cypher, 1993b). Les systèmes de programmation sur exemple ont donc été conçus dans l'objectif de permettre à des utilisateurs non-informaticiens d'accomplir des tâches de programmation, sans avoir à apprendre les concepts associés en informatique. On parle de la programmation par l'utilisateur final (End User Development).

La PsE est une extension du concept des enregistreurs de macros (Cypher, 1993b). Ces derniers sont à même d'enregistrer les actions de l'utilisateur et de les rejouer à la demande. Cependant l'exécution des macros ainsi réalisées ne prend pas en compte le contexte de création ou d'exécution. L'enregistrement est au niveau des commandes et des clics souris. Dans le cas de la PsE, le système produit une généralisation des actions enregistrées. Durant la conception d'un programme, le système analyse les entrées de l'utilisateur, et construit le programme à même de générer l'exemple.

La PsE est donc une approche permettant de créer des programmes en s'appuyant sur le comportement ou les effets d'exemples, en d'autres termes sur les actions réalisées par l'utilisateur lors de la construction d'exemples. Le système enregistre et généralise le comportement ou les effets de l'exemple. L'utilisateur final est le programmeur qui ne programme que par l'intermédiaire d'une interface. La PsE résout des problèmes d'interprétation par l'intermédiaire de procédures de dialogues sophistiquées ou par l'utilisation de règles (notion d'inférence, *voir chapitre 1 § 2.2.4*).

1.3 PsE selon Lieberman (Lieberman 01)

La première idée de la programmation qu'a eue Lieberman portait sur la façon ou la manière de programmer. Il s'agit selon lui (Lieberman, 2001) d'apprendre ou d'enseigner à quelqu'un à réaliser une tâche. Après tout, n'est-ce pas l'objectif de la programmation d'obtenir de l'ordinateur un nouveau comportement ? Quel est le meilleur moyen

d'apprendre que par l'exemple ? Pour Lieberman, la PsE consisterait à montrer à l'ordinateur un exemple de ce que l'utilisateur désire réaliser étape par étape, l'ordinateur se rappellant toutes ces étapes et essayant de les appliquer dans de nouveaux exemples.

Selon Lieberman, la PsE, ou parfois «Programmation par Démonstration» (Lieberman, 2001) est un concept dans lequel l'utilisateur démontre des exemples à l'ordinateur. Un système enregistre les interactions entre l'utilisateur et l'interface classique du système et écrit un programme qui correspond aux actions effectuées par l'utilisateur. Le système peut alors généraliser le programme afin que celui-ci puisse s'exécuter sur d'autres exemples similaires mais pas nécessairement les mêmes que ceux sur lesquels il a été construit. C'est cette capacité de généralisation qui fait de la PsE des macros «programmables». La généralisation est le problème central et complexe de la PsE, qui devrait selon Lieberman, permettre de remplacer totalement la programmation conventionnelle.

1.4 PsE selon Girard (Girard 00)

Selon (Girard, 2000), la PsE n'est pas seulement de la programmation visuelle, c'est-à-dire un système de programmation dans lequel la définition du programme peut être effectuée par interaction graphique dans un espace à deux dimensions ; ce qui fait sa spécificité c'est qu'elle est « *basée sur exemple* ». Un système de programmation est dit basé sur exemple si l'utilisateur peut utiliser les valeurs d'un exemple d'exécution pour définir les objets sur lesquels portent le programme à construire (Girard, 1992).

La *programmation sur exemple* est à l'opposé de la *programmation avec exemple*. Un système de programmation est dit avec exemple si la visualisation des programmes associés à des exemples permet seulement une représentation plus concrète de ce que l'on a écrit et non une description concrète de ce que l'on veut (Girard, 1992).

1.5 Synthèse

De cette section sur la notion et la définition de la PsE, nous pouvons dire que la programmation a beaucoup évolué de nos jours. Dans les années 80, de nombreux projets ont jeté les bases de la programmation visuelle. L'idée principale de cette approche est de remplacer les textes (chaînes de caractères) par des images. La programmation visuelle utilise des images qui donnent un sens beaucoup plus intuitif à la programmation. Cette méthode a permis d'introduire des images pour remplacer, par exemple, les variables et les fonctions des programmes. Cependant, son application s'est cantonnée à la représentation de programmes statiques. Elle ne permet pas de visualiser le programme au fur et à mesure de son exécution. L'utilisation d'exemples pour concevoir des programmes constitue le second pas vers les environnements de programmation pour utilisateur final. La programmation a évolué du remplacement des chaînes de caractères par les images (programmation visuelle) à la programmation par l'utilisateur final (End-User Programming) en passant par la programmation basée sur exemple (Example-based Programming). La notion de conception de programmes à partir d'exemples est née. L'utilisateur conçoit lui-même ses fonctions sur des valeurs réelles, utilisées comme variantes par le programme. L'ensemble de cette évolution est aujourd'hui rassemblé sous l'expression « End User Development »

2 Historique de la programmation sur exemple

Les applications proposent souvent des fonctions génériques qu'il conviendrait d'adapter à chaque utilisateur. Ceci fut reconnu lors d'une étude dont le résultat figure dans le rapport du groupe de travail constitué sur le thème « *End-User Computing »* en 1991 au cours de la conférence SIGCHI'91, à la Nouvelle-Orléans (Myers, 1992). La réponse naturelle à ce besoin consiste à permettre à l'utilisateur de modifier la programmation de son application. Mais, la programmation est-elle accessible à un utilisateur final quelconque ? À cette époque, la réponse était clairement non. La programmation textuelle classique, en dépit de l'évolution très grande des langages utilisés, n'est pas à la portée de tous les utilisateurs, sans un effort d'apprentissage souvent disproportionné.

L'émergence des environnements de développement utilisables par un utilisateur final non-informaticien a consisté en l'utilisation d'exemples d'exécution pour la conception des programmes : « *Example-Based Programming »*. Au lieu de sélectionner des fonctions et définir des variables sur lesquelles les fonctions s'appliqueront, l'utilisateur utilise des fonctions sur des valeurs, qui se comportent comme des variables d'un programme. L'une des idées principales est d'éviter le niveau d'abstraction des variables en autorisant l'utilisateur à manipuler des exemples.

Dans ce cadre, Halbert définit une notion importante : l'inférence (« *Inferencing »*) (Halbert, 1984). Un système est dit avec inférence s'il est capable de déduire un programme des actions de l'utilisateur, sans que ce dernier ait effectivement dit comment le faire. En 1988, Myers cherche à préciser ce principe (Myers, 1988). Il parle alors d'inférence plausible (« *Plausible Inferencing* » ou abstraction) pour caractériser les systèmes qui génèrent des explications ou des généralisations basées sur des informations limitées. Ces explications peuvent se révéler incorrectes, et ces systèmes prennent souvent l'initiative d'un dialogue avec l'utilisateur pour lever l'ambiguïté. Ceci le conduit à distinguer ce qu'il appelle « *Programming With Example* », sans inférence plausible, et « *Programming By Example* », avec inférence plausible. Halbert les caractérise respectivement par les formules « *Do What I Did* » (« Fais ce que j'ai fait ») et « *Do What I Mean* » (« Fais ce que je pense ») (Halbert, 1984).

Chez *Apple Computing* s'est tenu, en 1992, un *Workshop* où les pionniers du domaine de « *Example-Based Programming* » ont effectué nombre de démonstrations de leurs systèmes. Les participants ont affiné les définitions et proposé une nouvelle expression, « *Programming by Demonstration* », qui aujourd'hui est largement acceptée. C'est ce que nous avons nommé la Programmation sur Exemple. Les résultats de leurs travaux sont recueillis dans un ouvrage intitulé « Whatch What I Do » faisant référence (Cypher, 1993b). Un recueil plus récent à été publié par Lieberman en 2001 intitulé « Your wish is my command » (Lieberman, 2001) et le tout dernier en 2006 « End-User Development » (Lieberman et al., 2006).

Au terme des différentes visions et des travaux du domaine, l'objectif final recherché est de permettre à l'utilisateur final lui-même d'adapter une application à ses propres besoins. Pour adapter l'application à ses besoins, l'utilisateur doit modifier la logique mise en place par le programmeur de l'application. Afin de répondre à cela, différentes approches ont été développées. Des années 75 à nos jours, plusieurs techniques ont vu le jour, des éditeurs de préférences à la programmation sur exemple en passant par les langages de scripts, les enregistreurs de macros et même la programmation graphique. Nous y reviendrons dans la section 2 du chapitre 2.

3 Contexte et motivation de la thèse

Depuis l'avènement des interfaces graphiques, les logiciels ont globalement évolué vers une augmentation des fonctionnalités. Cette course en avant aboutit à imposer à l'utilisateur des interfaces de plus en plus lourdes, où les menus s'allongent, et où le nombre d'interactions purement articulatoires (nombre d'actions élémentaires pour atteindre une valeur ou un résultat) augmente. Il est vrai que l'émergence des techniques post-WIMP[1] a engendré très récemment une salutaire remise en cause de cette tendance (Beaudouin-Lafon, 2004).

On peut considérer que la recherche de solutions à ce problème, dans les années 75, constitue le point de départ des travaux sur la programmation par l'utilisateur final. Des solutions ou réponses adaptatives ont été proposées, parmi lesquelles certaines ont été utilisées avec plus ou moins de bonheur.

Toutes ces techniques présentent néanmoins l'inconvénient de manquer singulièrement de puissance d'expression. On ne peut en effet ni créer une nouvelle commande par assemblage de commandes existantes, ni même automatiser des enchaînements de commandes. Afin d'augmenter les possibilités d'adaptation, il était temps de revenir aux principes simples et universels de la programmation. Des langages de scripts ainsi conçus permettent d'obtenir le degré de puissance souhaité mais nécessitent la maîtrise des principes de la programmation. Ils s'avèrent donc inutilisables par l'utilisateur final. L'utilisation accrue de l'image a conduit à jeter les bases de la programmation dite visuelle (« *Visual Programming* » (Glinert, 1990)). Dans le domaine de la programmation, de nombreux systèmes ont ainsi été développés. Cependant, une compréhension des principes de la programmation est là encore nécessaire. Au-delà de la simple utilisation de l'image, c'est sur celle de l'exemple que se sont concentrés les travaux. L'idée générale est que tout raisonnement autour de l'exemple est plus intuitif qu'un raisonnement abstrait. C'est ainsi qu'ont vu le jour les systèmes « avec exemple » (*Programming with example*) qui permettent l'exécution immédiate de l'exemple en cours de la construction, et surtout les systèmes « sur exemple » (*Programming by Example ou Programming by Demonstration)* (Girard, 2000).

Au vu de cette trame (historique) d'évolution de la programmation utilisateur, la programmation sur exemple s'avère difficile à réaliser (Myers, 1993b). Il n'est pas aisé d'incorporer des systèmes au processus de création des applications interactives ou de créer des systèmes indépendants intégrant les techniques de la programmation sur exemple. Dans le premier cas, on peut soit créer une application dans l'objectif de valider la conception d'une technique de la programmation sur exemple, soit fournir au concepteur une boîte à outils permettant d'intégrer directement des moyens de mise en œuvre de système de programmation sur exemple dans son application. Dans le second cas, les systèmes doivent êtres capables de s'adapter à n'importe quel type d'application existante. Ces deux principes constituent les seules voies de résolution par la programmation sur exemple. La combinaison des deux principes en un seul permet la généralisation sans dépendance des systèmes hôtes incluant déjà les techniques de cette programmation.

Le besoin majeur de la PsE est l'intégration de ses principes dans les systèmes interactifs. Cette solution doit être adaptable à toute application en s'appuyant ou non sur l'interface même de l'application. La solution consistant en la généralisation des actions

[1] Windows Icons Menus and a Pointing device

utilisateurs de façon déconnectée des fonctionnalités du noyau fonctionnel est résolue partiellement par des systèmes existants comme AIDE (Piernot and Yvon, 1993) et PbDScript (Depaulis et al., 2003). Cette résolution passe soit par une approche interne, soit par une approche externe. La PsE interne consiste à intégrer la PsE au cœur de l'application en utilisant un développement pur et simple, et pas toujours réutilisable du système. La PsE externe utilise la PsE hors du code de l'application. Les outils de PsE externes ont une très grande difficulté à agir sur l'application. Tous ces outils font le choix entre l'établissement d'hypothèses préalables sur les noyaux fonctionnels (ce qui réduit les champs des applications) et la seule considération des interactions de bas niveau (de manière à étendre le choix des applications). L'inconvénient du deuxième choix réside dans le niveau sémantique très bas des commandes espionnées. Il est donc jusqu'à présent difficile de satisfaire aux besoins de l'utilisateur à l'aide de la programmation sur exemple. Nous pouvons continuer à nous poser toujours les mêmes questions. Comment faire en sorte qu'un utilisateur final, ignorant tout des principes de la programmation, puisse personnaliser les logiciels qu'il utilise ? S'il est capable d'exécuter des actions dans un environnement donné, pourquoi ne pas mettre à sa disposition les moyens nécessaires pour qu'il puisse créer ses propres actions, tout en restant dans cet environnement familier ?

Cette tâche incombe directement au concepteur des applications interactives. Il faut trouver des solutions pour aider ce dernier afin de lui faciliter la tâche de programmation. Il faut donc fournir des outils ou services que le concepteur en programmation sur exemple utilisera facilement pour améliorer la conception de ses applications, puis par la suite au développeur d'applications. Cela rendrait plus facile l'accès au domaine de la PsE et permettrait de tester rapidement de nouvelles idées sans avoir à réinventer une nouvelle technique à chaque fois. De tels outils et services aideraient au développement d'une grande variété de systèmes de PsE en fournissant des opérations *génériques* et *réutilisables*. Une seconde caractéristique que doivent posséder ces outils et services est leur *extensibilité* de sorte que le développeur puisse les adapter à sa situation particulière, incorporer de nouvelles techniques, ou encore proposer les améliorations qu'il leur a apportées à l'ensemble des développeurs.

La finalité de notre travail est l'obtention de modules logiciels pour la création d'outils ou d'objets permettant l'intégration de la PsE. Ces modules logiciels passent forcément par la définition de fonctionnalités permettant l'enregistrement du dialogue d'une application interactive. Rejouer ce dialogue constitue la seconde fonctionnalité obligatoire du module même si cela ne saurait servir à la définition de programme qui peut avoir lieu sans la notion de généralisation. Il est nécessaire de dépasser le simple niveau événementiel des actions pour obtenir la possibilité d'automatiser une action répétitive sans dépendre de la sémantique de l'application.

Le besoin majeur de la programmation sur exemple est l'intégration de système d'enregistrement, de généralisation et, de rejeu des actions de l'utilisateur dans l'application interactive graphique. Les deux voies de résolutions possibles sont soit l'usage d'une voie interne, soit l'usage de la voie externe. Cela nous conduit à définir la possibilité d'implémentation du module logiciel suivant l'architecture de la Figure 1 ci-dessous montrant le schéma d'une application interactive avec le module logiciel de la PsE.

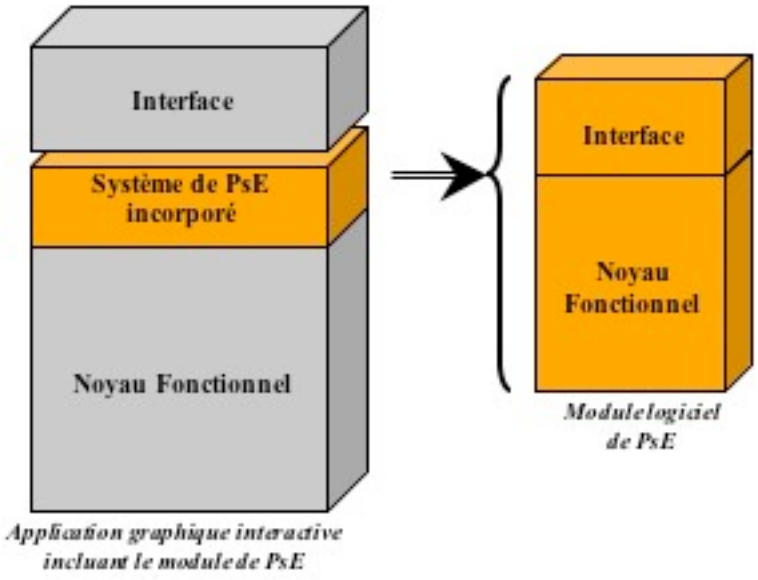

Figure 1 : Vision de l'architecture d'implémentation du module logiciel de la PsE

4 Organisation du mémoire

Ce mémoire a commencé par cette introduction générale présentant un état de l'art des différentes définitions proposées dans le cadre de la programmation par l'utilisateur final. À la suite de cet état de l'art, nous avons fait un bref historique de la PsE. Dans cette introduction générale, nous avons précisé le contexte et la motivation de notre travail. Le reste du mémoire est constitué de quatre chapitres.

Le chapitre 1 porte sur la programmation sur exemple. Nous présentons quelques systèmes existants que nous considérons comme des exemples remarquables de systèmes de la programmation sur exemple. Cette présentation conduit à l'établissement des principes de la programmation sur exemple d'une manière approfondie. À l'issue de cette étude de quelques systèmes exemples et de l'établissement des principes de la PsE s'en suit une classification des systèmes de la PsE suivant des critères que nous avons définis. Cette classification permet de fixer un domaine sur lequel nous intervenons au vu du large domaine d'application que couvre la PsE.

Le chapitre 2 permet de définir les besoins et services d'un utilisateur (ou le programmeur) et les outils existants en guise de solution dans la programmation des systèmes de PsE. Dans ce chapitre, nous parlons des composants que le programmeur utilise et les composants à mettre à sa disposition. Les boîtes à outils d'interaction sont étudiées sur le plan théorique afin de pouvoir définir facilement l'intégration des nouveaux services (ou composantes) en leur sein. Nous terminons ce chapitre en présentant une étude approfondie des solutions existantes ainsi que leurs limites.

Le chapitre 3 décrit les services à mettre en place dans une boîte à outils permettant l'intégration de la PsE dans une application. Ces services sont les principaux besoins ou techniques de la programmation sur exemple. Il s'agit essentiellement de l'enregistrement des actions utilisateur, du rejeu de ces actions, de la capacité de généralisation des systèmes et des mécanismes utilisateur à prendre en compte. Nous expliquons au début de

ce chapitre notre orientation, et nous exposons un système exemple qui présente au mieux notre vision du travail. Nous présentons des outils d'enregistrement et de rejeu, supports intéressants de tentative de proposition de solution partielle. Nous décrivons de manière détaillée la technique et la réalisation d'un système incluant la PsE à partir des services décrits. Nous présentons la technique sous deux angles à savoir celle de l'utilisateur final de l'application et celle du programmeur de ladite application. Les composantes implémentées sont présentées ainsi que les différents objets et leurs utilisations possibles. L'avant dernière section de ce chapitre porte sur une autre vision de la réalisation de l'enregistrement et/ou de rejeu des actions utilisateur : la programmation par aspect. Une comparaison des systèmes est faite. Un exemple d'étude de cas a été réalisé sous forme d'application de la technique d'implémentation. Il s'agit principalement d'une application de conversion de devises (par exemple de la monnaie « Euro » vers la monnaie « Franc » vice-versa), et d'un éditeur de texte. Nous décrivons les différentes composantes de chacune des applications et faisons une comparaison avec l'implémentation faite avec les autres solutions.

Le chapitre 4 est la présentation de l'application du principe d'enregistrement et de rejeu dans les tests des interfaces graphiques interactives. L'application en test a pour but de prouver la conformité de l'interface utilisateur par rapport à son modèle de tâches prescrit. Nous faisons une liaison entre le modèle de tâches de conception et l'interface utilisateur lors de la programmation. C'est le programmeur qui établit le lien et permet ainsi la génération de scénarii qui seront par la suite exécutés par le simulateur de l'éditeur du modèle de tâches. L'outil de modélisation utilisé est K-MAD dans l'environnement K-MADe.

À la suite de ces quatre chapitres, notre mémoire se termine par une conclusion prenant la forme d'un bilan des travaux, puis par les perspectives offertes pour l'orientation de la recherche ou des travaux dans le domaine de la Programmation sur Exemple.

Chapitre 1

Programmation sur exemple : principes et classification

SOMMAIRE

Résumé. *Ce chapitre présente des illustrations d'applications typiques de la programmation sur exemple. À travers cette illustration, les principes et les techniques de fonctionnement des systèmes sont déduits. Cette étude a permis de faire une classification des systèmes en fonction de leur usage. Quatre catégories essentielles en ressortent : l'assistance, les outils de conception, l'apprentissage de la programmation et les outils de la PsE.*

1 Introduction

La Programmation sur Exemple fournit une réponse immédiate à l'utilisateur. Dans la PsE, les opérations abstraites sont accomplies de façon concrète : pour écrire un programme, l'utilisateur montre le comportement désiré à l'aide d'un ou de plusieurs exemples réels (Cypher et al., 1993). Il n'y a pas besoin de mettre en correspondance un langage textuel avec les objets présents à l'écran : pour faire référence à une action, l'utilisateur effectue simplement cette action au lieu d'avoir recours à une instruction textuelle. Dans la terminologie de Cypher, *« l'utilisateur programme dans l'interface du système* » (Cypher, 1993b). Les systèmes de la PsE mettent la programmation à la portée du simple utilisateur.

La PsE est une méthode qu'utilise l'utilisateur final pour créer, optimiser et étendre des programmes à partir d'exemples. Au vu des écrits précédents, on peut dire que dans la PsE, l'ordinateur espionne l'utilisateur et enregistre ses actions, ou bien il interprète les objectifs de l'utilisateur et les généralise. Le système analyse les entrées de l'utilisateur lors de la réalisation de l'exemple et conçoit le programme capable de générer l'exemple, ou bien déduit un programme.

À travers ce chapitre, nous allons étudier des systèmes exemples de la PsE afin de mettre en évidence leurs principes et leurs techniques de fonctionnement. À partir de cette étude, nous déduirons une classification de l'ensemble des systèmes du domaine de la programmation sur exemple.

2 Étude des principes de la PsE

2.1 Systèmes exemples de la PsE

Dans cette section, nous analysons quelques systèmes importants de la Programmation sur Exemple afin d'en dégager les principes de fonctionnement et connaître en profondeur les tendances techniques employées.

2.1.1 Pygmalion

Pygmalion (Smith, 1977) de David Smith est un environnement graphique de programmation. C'est la première tentative de programmation visuelle, basée sur la métaphore du « tableau noir » : la zone de travail est vue comme un tableau où le programmeur peut « dessiner » ses idées. Pygmalion est à l'origine du concept d'icône disposant à la fois d'une image, d'un contenu (du texte pour une icône de document), et d'un comportement (déplacement d'un fichier par *drag and drop*) (Cypher, 1993c).

Pygmalion fournit un menu permettant d'accéder aux icônes et aux opérations portant sur ces icônes. Il dispose d'un champ de texte appelé « mouse value » dans laquelle toute valeur tapée est rattachée à la souris et peut être déposée dans n'importe quelle icône. Lorsque Pygmalion est en mode enregistrement, une zone d'historique (« remembered ») affiche les deux dernières actions exécutées. Le système dispose aussi d'une zone « SmallTalk » où le programmeur peut taper et évaluer des expressions écrites dans ce

langage. Enfin, il existe une zone « mouse » où l'on peut définir le sens d'un clic avec l'un des boutons.

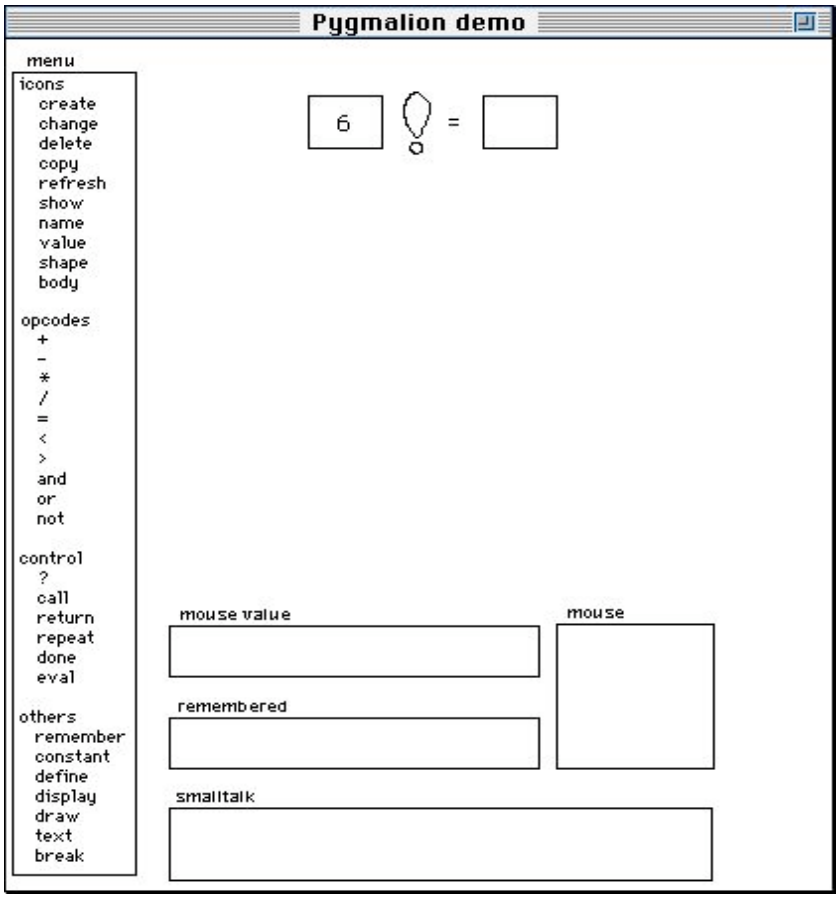

Figure 2 : Environnement Pygmalion

Pygmalion fournit les représentations graphiques standard d'arithmétique, de relationnelle et des opérateurs booléens. Pour réaliser la fonction « factorielle » à l'aide de Pygmalion, on commence par lui allouer une icône. Ensuite on lui associe deux sous-icônes pour représenter l'argument et la valeur de la fonction, et l'on dessine un symbole (Figure 2) pour la représenter. On rend la bordure invisible et l'on associe un nom avec l'icône par la commande « define ». Le système effectue alors une capture d'écran : la zone de travail sera restaurée à cet état chaque fois que la fonction sera appelée. L'appel de la fonction « define » fait basculer le système en mode enregistrement. On peut alors spécifier le comportement de la fonction, à l'aide d'un exemple (Figure 3).

Pygmalion peut être considéré comme la première application basée sur exemple, dans la mesure où la construction d'un programme se fait de façon concomitante à la définition d'un exemple. Les structures du programme sont définies explicitement par l'utilisateur, mais l'alternance entre phases d'édition et d'exécution rend le programme tout de suite très concret. La notion de procédure est définie, ainsi que celle de sous-programme, la récursivité étant possible. L'utilisateur ne travaille pas véritablement « sur » l'exemple, mais peut en permanence juger de la justesse de ses constructions.

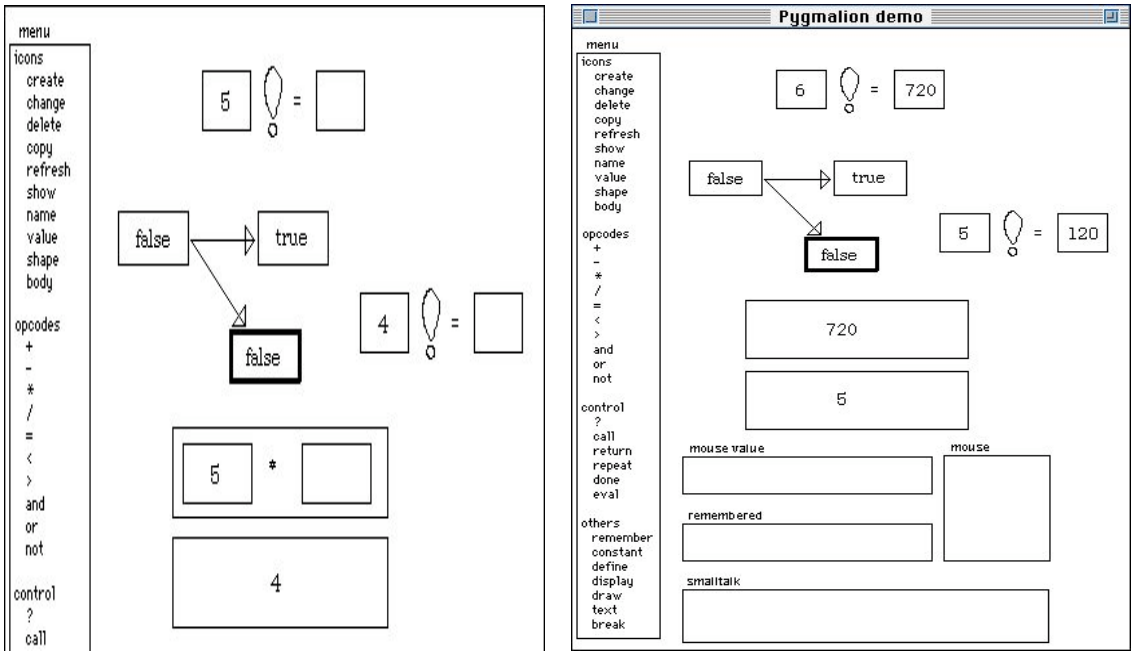

Figure 3 : Réalisation de la fonction « factorielle » à l'aide de Pygmalion

2.1.2 StageCast Creator

Environnement graphique de construction de jeux animés, StageCast Creator (Smith and Cypher, 1995) est destiné aux enfants. C'est un système de programmation par l'utilisateur final dans le domaine d'application de la simulation ou de jeux. L'utilisateur peut créer un certain nombre d'agents avec des comportements concurrents et indépendants.

Figure 4 : Environnement de StageCast Creator

L'environnement de StageCast Creator peut être décomposé en quatre parties (Figure 4). Une partie (1) que l'on peut nommer micro-monde dans lequel les agents ou personnages définis par le programmeur s'exécutent. Cet espace est basé sur un tableau dont les cases peuvent être masquées ou rendues visibles par l'utilisateur. Chaque case du tableau peut contenir un agent ou/et une image de fond. Une barre d'outils (2) permet de créer un agent ou un personnage, de modifier son comportement et de modifier son apparence. L'environnement de StageCast Creator contient un lecteur (3) servant à simuler le comportement des agents ou des personnages. La dernière partie est un panneau latéral (4) offrant au programmeur une vue statique du programme qu'il développe.

Pour programmer le comportement d'un agent, on associe une suite d'actions à partir d'un exemple d'un événement particulier. Le programmeur sélectionne la zone de l'écran correspondant à l'événement puis peut redessiner, agrandir, déplacer, … les agents concernés. StageCast Creator propose une représentation graphique du code, et cette juxtaposition du flot de contrôle est représentée explicitement dans l'environnement de développement. StageCast Creator est limité à la création de simulations graphiques ou de jeux. Ce mode de programmation est en fait très particulier, et très bien adapté à la simulation : les situations décrites graphiquement, qui peuvent être considérées comme des exemples de ce que l'on veut réaliser au niveau de chaque objet, correspondent à l'évolution de la simulation pour un temps donné. Il a été utilisé avec succès dans de nombreuses écoles américaines.

2.1.3 EBP

EBP (*Example Based Programming*) (Girard et al., 1996) est un environnement de développement de programmes paramétrés susceptibles de générer des comportements standard (Figure 5). Il permet en particulier de mettre au point par manipulation directe et visuelle les programmes générés.

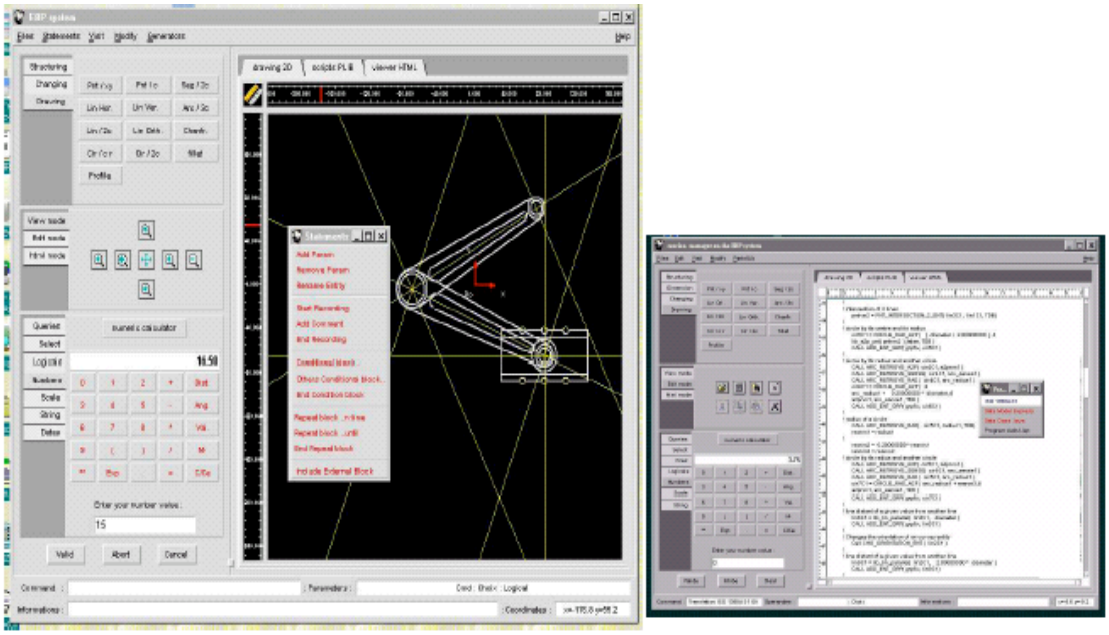

Figure 5: Environnement EBP

EBP permet la création de familles de composants standard portables par des utilisateurs de systèmes de conception assistée par ordinateur (CAO). Basé sur une interprétation orientée vers la sémantique des interactions utilisateurs, EBP utilise comme

représentation interne un arbre de syntaxe abstraite. Il utilise une double capture : au niveau lexicographique (pour filtrer les commandes et les désignations autorisées) et au niveau sémantique.

Le système EBP manipule des entités géométriques simples (points, droites infinies, segments, cercles, arcs, etc.) et des entités structurées (courbes, surfaces planes et ensembles structurés). La plupart des contraintes résultant des règles de dessin technique sont supportées. EBP fournit une calculatrice puissante permettant de combiner nombres et entrées graphiques dans des expressions grapho-numériques.

EBP permet de spécifier interactivement (de façon graphique) un programme contenant des structures de contrôle usuelles et offre la possibilité de générer du code. EBP permet également à l'utilisateur d'enrichir et de personnaliser l'interface de dialogue du système interactif qui l'héberge, en lui permettant d'associer l'exécution d'un programme construit par démonstration à une commande du système.

La programmation sur exemple est réalisée dans EBP via un mode d'enregistrement des appels de fonctions de système. Les programmes au sein d'EBP (où ils sont nommés « instances ») sont séparés des exemples. EBP espionne l'utilisateur et construit une instance après l'activation du mode enregistrement. Le passage en mode utilisation permet à l'utilisateur d'exploiter cette instance.

Une session typique d'EBP se présente comme suit : après l'analyse de la pièce, l'utilisateur commence l'enregistrement. Il définit les paramètres, puis dessine un exemple en utilisant les paramètres au lieu de valeurs directes. Une commande (« DEFINE ») ouvre une fenêtre, où l'on fournit un nom de paramètre (par exemple, longueur), lequel est affiché à chaque exécution du programme. Une autre commande (« ENTER ») permet l'introduction de la valeur des paramètres pour l'exemple. Ces valeurs sont saisies via l'interface du système de CAO, et leur type définit le type des paramètres. Des commandes permettent la lecture et l'enregistrement de valeurs à partir d'un fichier, lequel sera associé à l'instance. Cette fonction est utilisée pour enregistrer les valeurs autorisées de paramètres pour des familles de pièces. Dès que les paramètres sont définis, ils sont affichés au sein d'un menu où l'utilisateur peut les désigner, par exemple pour définir une expression à l'aide de la calculatrice grapho-numérique (exemple : « longueur x 2 »). Lorsque l'exemple est complet, l'utilisateur peut enregistrer l'instance résultante, changer quelques valeurs de paramètres et commander une nouvelle exécution.

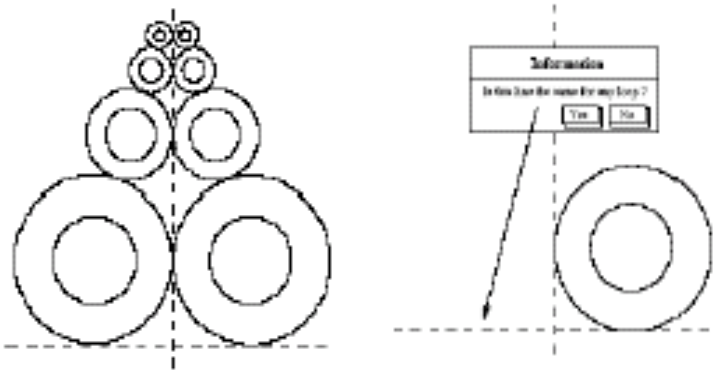

Figure 6 : Exemples de construction complexe (boucle/répétition)

Enregistrer sur fichier des valeurs de paramètres est aisé, permettant des procédures de test rapides. Les instances enregistrées sont incluses dans un menu, et sont utilisables avec un minimum d'effort.

Lors de l'exécution de l'instance en mode enregistrement, la définition de la valeur du paramètre effectif est constituée d'une expression quelconque mettant en jeu des entités ou des paramètres du programme appelant. Il s'agit d'un réel passage de paramètres comme il en existe dans les langages classiques.

EBP comporte des structures de contrôle complètes. Les sous-programmes, les alternatives et les répétitions sont totalement disponibles (Figure 6). Les alternatives et les répétitions supposent la définition d'expressions booléennes. Plusieurs schémas d'itération sont fournis. L'itération d'ensembles est disponible pour des objets sélectionnés à l'aide d'un rectangle élastique, ou pour les transformations géométriques multiples.

EBP pousse à son paroxysme le concept de programmation sur exemple, puisqu'il permet de construire des programmes en utilisant majoritairement les commandes du système CAO support. Il n'utilise pas d'inférence, ni d'heuristiques. A l'inverse, il suppose une intervention explicite de l'utilisateur pour introduire des variables, des structures de contrôle et des expressions. En ce sens, il oblige l'utilisateur à comprendre certains éléments de la programmation, même si nombre d'entre eux sont présentés à l'utilisateur dans son langage (par exemple, la notion de tables de valeurs pour des pièces paramétrées est naturelle en CAO).

2.1.4 Gamut

Gamut (McDaniel and Myers, 1999) est un outil de Programmation sur Exemple destiné à la construction d'applications interactives, comme les jeux vidéo, les simulations et les logiciels éducatifs.

Cet outil permet de fabriquer les applications en montrant à l'ordinateur comment elles doivent fonctionner. Concrètement, le concepteur commence par représenter visuellement le jeu. Il définit les éléments visibles et invisibles pendant la phase d'exécution. Il place les objets graphiques et les composants fenêtres (widgets) sur une zone d'édition.

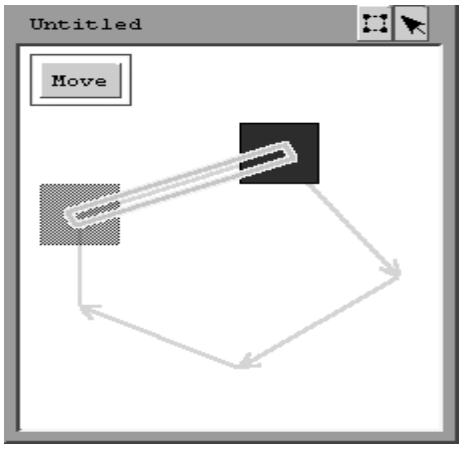

Figure 7 : Représentation graphique d'un jeu

Par exemple sur la Figure 7, on a représenté un chemin fermé défini par des flèches. On y a placé un objet rectangle dont le rôle est de se déplacer sur l'extrémité des flèches quand l'utilisateur appuie sur le bouton « move ».

Pour le mécanisme de programmation, l'outil Gamut dispose de deux boutons « Do Something » et « Stop that ». Le développeur utilise « Do Something » pour ajouter un nouveau comportement. Le concepteur provoque un événement, par exemple l'appui sur le

bouton « move ». Quand un objet ne répond pas, le développeur sélectionne ce bouton et presse le bouton « Do Something ». Cela indique au système Gamut que le développeur est prêt à lui montrer une nouvelle manipulation sur les objets du jeu.

Sur la Figure 7 le concepteur déplace l'objet rectangle vers l'extrémité de la flèche suivante. En mettant en surbrillance la flèche de liaison, il précise aussi le chemin à suivre. Et finalement il valide la programmation par démonstration par l'appui sur un bouton.

À ce moment, Gamut a traduit le fait que si l'utilisateur clique sur le bouton « move », le rectangle suit le chemin démontré pendant la programmation. Une chose importante est que le déplacement du rectangle ne se limite pas seulement à la portion d'une flèche. Gamut va déduire le déplacement total que pourra prendre le rectangle. Sur notre exemple, le rectangle se positionnera sur toutes les extrémités des flèches. Le système Gamut essaie de déduire des actions du « programmeur » une généralisation permettant de construire l'application. Par contre le bouton « Stop that » stipulera les actions à ne pas réaliser sur le jeu.

Gamut permet à l'utilisateur de modifier le comportement d'un objet à l'aide d'un menu interactif. Cette modification peut être l'ajout ou l'annulation de comportements. Le système déduit une généralisation des actions de l'utilisateur, permettant de construire l'application. Le comportement d'un programme peut être corrigé de manière interactive sur une exécution. Le mécanisme de généralisation consiste en la création automatique de structures de contrôle d'un programme modifié après la première exécution.

La démarche suivie par Gamut est à l'opposé de celle proposée par EBP. Aucune structure de programmation explicite n'est utilisée, le système cherchant à interpréter les actions de l'utilisateur pour les transformer en programme. Les auteurs décrivent de nombreux mécanismes permettant d'interpréter ces actions.

2.1.5 ToonTalk

ToonTalk (Kahn, 2001) est un environnement de programmation à l'aide d'exemples utilisant la métaphore du jeu de construction. C'est une illustration de l'importance de la métaphore et de son adaptation au public visé. ToonTalk est un système de programmation iconique, à base ou à partir d'exemples et s'adresse aux enfants, à partir de six ans.

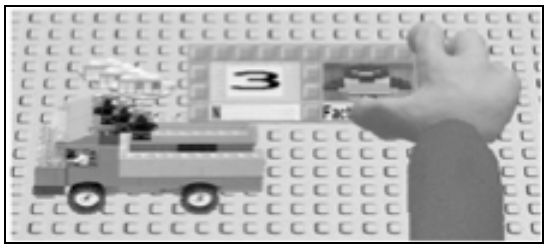

Figure 8 : La « main de dieu » dans ToonTalk

ToonTalk se base sur une approche particulière : le programmeur est un acteur dans un monde virtuel, où chaque objet abstrait est représenté grâce à un élément de la métaphore (Figure 9). Cette approche le place en marge des systèmes de PsE classiques. S'adressant à des enfants, ToonTalk tend à faire l'analogie entre la programmation et un jeu de construction. Dans la métaphore utilisée par ToonTalk, chaque objet du monde de la programmation est associé à son équivalent dans le monde de ToonTalk. Un programme complet se représente sous la forme d'une ville, les sous-programmes et les processus sont

représentés par des maisons, que l'on peut faire communiquer à l'aide de pigeons voyageurs. Ceux-ci peuvent transporter des objets depuis leur maison jusqu'à des nids. Chaque maison peut contenir des robots ; ce sont eux qui accomplissent les tâches à l'intérieur des sous-programmes. On peut affecter à ces tâches des pré-conditions qui apparaissent sous forme de bulles.

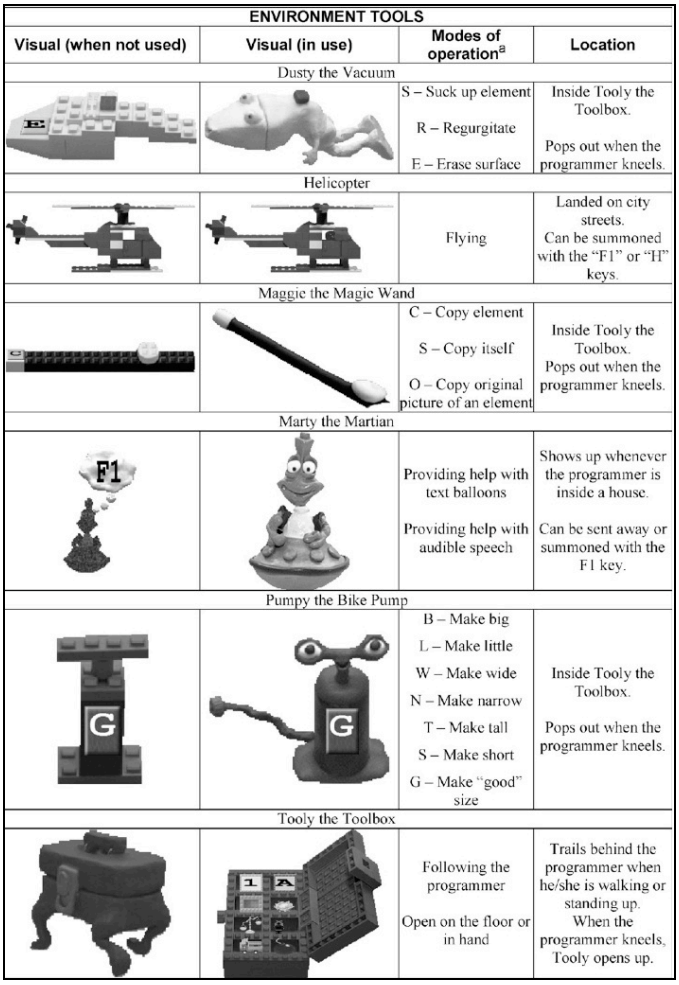

Figure 9 : La métaphore utilisée par ToonTalk

On peut apprendre à un robot à effectuer sa tâche par l'exemple. Cette tâche peut être :

- l'envoi d'un message : en donnant une boîte à un pigeon ;
- la création d'un nouveau processus (une nouvelle maison) : en chargeant une boîte et un ou plusieurs robots dans un camion ;
- copier un élément dans une boîte grâce à la « main de dieu » (Figure 8) ;
- modifier une structure de données par copier coller de boîte à boîte ;
- tuer un processus en envoyant une bombe sur la maison ;
- etc.

ToonTalk, quoique souvent qualifié de système basé sur exemple, se contente de représenter graphiquement les programmes, selon une métaphore très originale. L'exécution de ces programmes fait intervenir des animations (comme le vol d'un pigeon pour illustrer l'envoi d'un message). En revanche, la construction du programme, certes visuelle, ne correspond pas vraiment à la manipulation de l'exemple.

2.1.6 MELBA

MELBA (Guibert, 2006) pour *Metaphors-based Environment to Learn the Basics of Algorithmic* est un environnement d'initiation à la programmation et est basé sur l'exemple (Figure 10). C'est un système d'apprentissage composé de trois panels : l'espace du programme (1) qui permet de représenter et d'éditer celui-ci, une zone sémantique (2) représentant l'exécutant et une zone pragmatique (3). Chaque zone est interactive et la cohérence de l'ensemble est assurée par des mécanismes de programmation sur ou avec exemple.

MELBA présente plusieurs modes de fonctionnement. Il permet de construire un programme en manipulant les éléments syntaxiques (zone de gauche). Le mode est alors dit « avec exemple », puisque le programme s'exécute (partie droite) en parallèle. Le système permet d'animer le programme, afin d'en comprendre le fonctionnement. Enfin, MELBA permet d'agir directement sur la zone « pragmatique », créant en parallèle le programme correspondant.

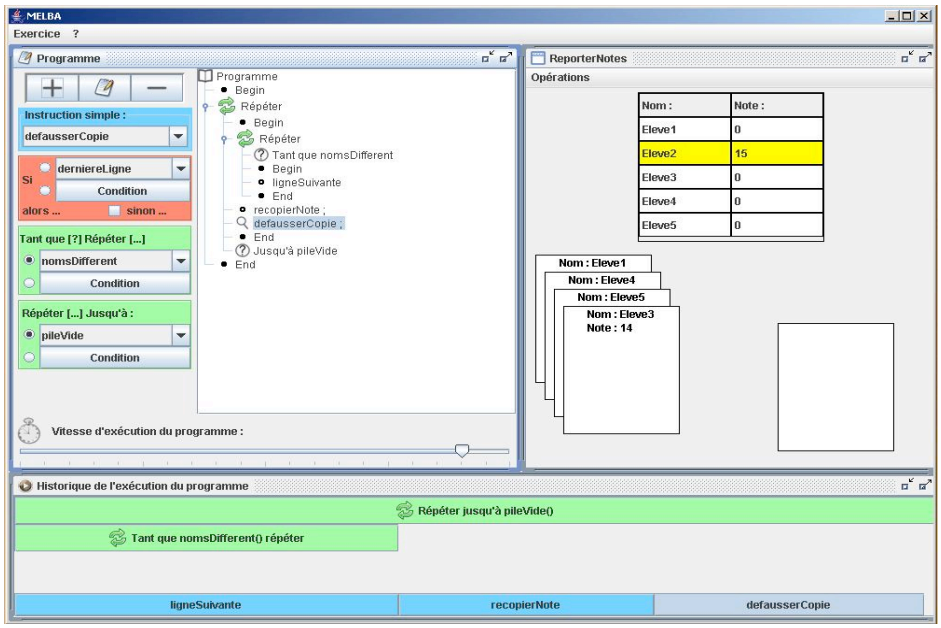

Figure 10 : Environnement MELBA

2.1.7 Synthèse des systèmes exemples

Les systèmes que nous venons de voir sont des applications de la programmation utilisateur permettant d'automatiser des tâches répétitives ou d'intégrer des applications pour développer une solution sur mesure ou encore d'adapter une application aux besoins spécifiques de l'utilisateur. La plupart des systèmes de PsE sont liés à des domaines d'application très restreints. Ils sont généralement développés dans des langages propriétaires et par conséquent, ils ne sont exploitables que sur une seule plate-forme. Ils exigent souvent la connaissance du vocabulaire et de la syntaxe des langages de programmation pour la modification de séquences et souvent même les concepts de programmation comme les structures de contrôle, les structures de données. Dans ces systèmes l'intérêt de la PsE est très limité. D'autres sont basés sur des moteurs d'inférence et ne requièrent aucune connaissance particulière de l'utilisateur.

Implanter un système de PsE est une tâche complexe car elle met souvent en jeu des techniques d'intelligence artificielle (apprentissage symbolique, représentation des

connaissances, inférences, etc.) en plus des techniques nécessaires à la conception et à l'implémentation, et des techniques d'interaction homme-machine (création, visualisation et invocation d'un programme, correction d'erreur, etc.). Notre objectif est, en nous appuyant sur ces exemples, de dégager les principes qui sous-tendent la PsE.

2.2 Principes de la PsE

À partir des exemples étudiés, nous avons pu analyser les principes de la programmation sur exemple. Comme nous l'avons montré au cours des sections précédentes de ce chapitre, la programmation sur exemple s'appuie sur une manipulation concrète de l'exemple (Sanou et al., 2006b). Dès lors que l'utilisateur est invité à démontrer les tâches à programmer, le système doit être capable d'analyser et d'interpréter ses interactions. La technique de base consiste donc à implanter dans un système interactif un système d'enregistrement/rejeu, cher aux systèmes de macros sur exemple (Kurlander and Feiner, 1992) ; (Hartmann et al., 2007).

2.2.1 Enregistrement et rejeu

Le principe de base de la programmation sur exemple consiste à enregistrer les actions de l'utilisateur de façon consciente (enregistrement de macros, par exemple) ou de façon masquée (par « espionnage »), à généraliser cet enregistrement, et à le rejouer. Le cœur de tout système de PsE s'appuie donc sur des mécanismes d'enregistrement – rejeu (Sanou et al., 2005).

Dans l'utilisation de la PsE, il convient dès lors de distinguer deux modes qui s'ajoutent ou se superposent à l'utilisation interactive classique du système.

Le premier mode est le « *mode enregistrement* » pendant lequel le système enregistre les actions de l'utilisateur pour construire une trace. Cet enregistrement peut être explicite, comme dans les systèmes de macros, ou implicite comme dans Eager (Cypher, 1993a). Dans le premier cas, l'application dispose généralement d'un « panneau de contrôle de système enregistreur », avec les fonctions enregistrer, jouer, arrêter, ou encore pause (Figure 11). Dans le second cas, implicite, l'utilisateur n'a rien à faire. Souvent même, l'utilisateur ne se rend pas compte de l'automatisation ou de l'enregistrement de ses actions. Mais une confirmation ou une validation est faite par ce dernier avant la ré-exécution des actions ou leur sauvegarde. L'utilisateur garde toujours le contrôle de la gestion de son système.

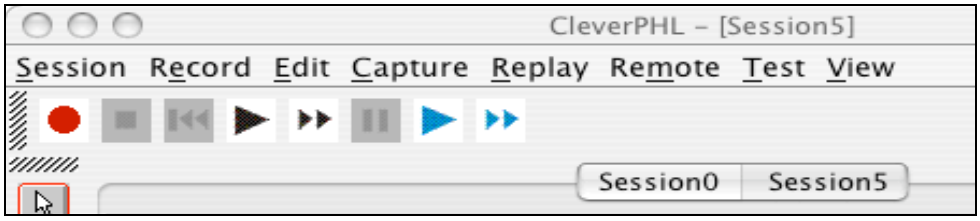

Figure 11 : Panneau de contrôle d'enregistrement – rejeu , ici dans l'application Jacareto[2]

Ce principe est poussé à son paroxysme dans les systèmes dits paramétriques, qui modélisent leurs données comme un historique d'actions constructives, enregistré au cours de la construction interactive de l'objet.

[2] http://jacareto.sourceforge.net

Le deuxième mode peut être considéré comme le « mode espionnage ». C'est toujours l'enregistrement des actions de l'utilisateur sans aucune intervention de la part de l'utilisateur lui-même. Ce mode correspond au second cas du premier mode, c'est-à-dire l'enregistrement implicite des actions de l'utilisateur.

L'enregistrement peut se faire à plusieurs niveaux qui sont équivalents aux différents niveaux des actions ou de la réaction du système aux actions utilisateur. Nous présentons en détail ces différents niveaux d'enregistrement dans la section suivante.

Le rejeu peut exprimer différents besoins. On peut faire un rejeu en mettant en évidence les interactions de l'utilisateur comme on peut faire fi de ces interactions et ne rechercher que le résultat final de la tâche. Il est possible que lors du rejeu, seules quelques interactions soient nécessaires à présenter ou que la tâche doive être rejouée partiellement. Cela nécessite alors l'intervention de l'utilisateur qui sera amené à confirmer certaines actions lors du rejeu. On peut donc déduire deux modes de fonctionnement selon le besoin du rejeu.

Le deuxième problème lié au rejeu est celui du contexte. En effet, les actions rejouées sont dépendantes de l'état du système au moment de leur exécution. Si l'exécution s'effectue dans un contexte différent du contexte initial, le résultat de l'action peut différer.

2.2.2 Niveaux d'enregistrement

Le niveau d'enregistrement peut être différent selon les systèmes. Les travaux menés sur les systèmes, surtout de conception technique (Girard and Pierra, 1995; Potier, 1995; Texier, 2000) (Sanou et al., 2006b) ont permis de caractériser cinq niveaux d'enregistrement – rejeu.

Le premier niveau correspond à *l'enregistrement des actions les plus élémentaires* de l'utilisateur. Les premiers systèmes d'enregistrement de macros fonctionnaient selon ce principe avec comme corollaire que la situation initiale joue un rôle prépondérant sur le bon déroulement de la phase de rejeu : les positions graphiques étant enregistrées en absolu, toute différence dans la configuration initiale de l'écran aboutissait à une exécution erronée de la macro. Dans Jacareto[3], l'enregistrement se passe au niveau de la boîte à outils d'interaction, Swing en l'occurrence. Ce sont ainsi les événements de bas niveau (MouseEvent, MouseMoveEvent, KeyEvent, etc.) qui sont enregistrés et rejoués, permettant un rejeu à l'identique de toutes les actions de l'utilisateur.

Le deuxième niveau concerne *l'aspect lexical de l'interaction*. Les éléments sont enregistrés au niveau de l'action élémentaire, du point de vue de l'utilisateur, et non plus de la boîte à outils. Eager (Cypher, 1991) en est la parfaite illustration. Les actions telles que le changement de focus sur une application, ou la sélection d'un item de menu, sont ainsi rejouées pour permettre à l'utilisateur de confirmer les propositions du système.

Le troisième niveau correspond à *l'aspect syntaxique de l'interaction*. Le système Like (Girard, 1992) en est un bon exemple. Dans ce système de conception technique, le dialogue entre l'utilisateur et le système est décrit par le moyen d'automates évolués (appelés Augmented Transition Networks, ou ATN (Woods, 1970)), manipulant des jetons de dialogue (commandes, paramètres). L'enregistrement de ces jetons constitue la base de l'enregistrement au niveau syntaxique. Nous verrons que ce niveau d'enregistrement présente un intérêt certain pour l'interprétation des actions de l'utilisateur. Notons que,

[3] http://jacareto.sourceforge.net/wiki/index.php/Main_Page

lorsque le style de dialogue est la manipulation directe, comme c'est le cas par exemple pour Peridot (Myers, 1993a), il est très difficile de distinguer le niveau lexical du niveau syntaxique : chaque interaction élémentaire est à la fois un élément lexical et un élément syntaxique, puisque la réaction du système est supposée continue.

Le quatrième niveau est tout naturellement le *niveau sémantique*, en référence à la théorie de la compilation. Il s'agit là d'enregistrer la fonction du système invoquée par l'utilisateur, pour être à même de la rejouer directement en phase d'exécution. Dans les systèmes de conception technique, cette approche est représentée par EBP (Potier, 1995), dont le but premier consiste à créer de véritables programmes paramétrés, échangeables d'un système de CAO à un autre. Grammex (Lieberman et al., 1998) et StageCast Creator sont deux autres exemples de ce niveau d'enregistrement.

Le dernier niveau concerne le *niveau pragmatique*. Ici, on ne considère que les actions du système qui ont un rôle dans la résolution du but final de l'utilisateur. Les actions articulatoires, ainsi que les opérations non constructives ne sont pas enregistrées. Les systèmes paramétriques en conception technique en sont la parfaite illustration. Comme expliqué dans (Texier, 2000), ce sont les données mêmes de l'application qui enregistrent leur historique de construction, ce qui permet de les rejouer après d'éventuelles modifications de paramètres.

Le choix du niveau de l'enregistrement est très important en fonction du but recherché par l'utilisation de la PsE. En effet, plus l'enregistrement s'effectue à un niveau élémentaire dans le dialogue entre l'utilisateur et l'application, et plus il sera dépendant des changements intervenant dans l'interface de l'application. Si l'objectif de la PsE consiste à construire des programmes réutilisables (comme des programmes paramétrés en conception technique, par exemple), tout changement dans l'interface invalidera ces programmes si l'enregistrement se fait à un niveau inférieur au niveau sémantique.

2.2.3 Généralisation

Le simple enregistrement – rejeu ne permet pas de faire autre chose que de la répétition à l'identique d'une séquence, qu'elle se situe au niveau de l'interaction ou du système lui-même. Ce qui distingue une simple trace d'exécution d'un vrai programme relève de la généralisation (Lieberman, 2001).

Beaucoup de systèmes de PsE descendent des enregistreurs de macros et donc enregistrent à la demande de l'utilisateur les commandes effectuées, puis les généralisent pour les rejouer ultérieurement. D'autres systèmes observent en permanence les commandes de l'utilisateur afin de prédire ses futures actions et terminer la tâche courante. Enfin, une dernière catégorie de systèmes adopte un comportement intermédiaire en enregistrant les commandes de l'utilisateur en permanence, et en permettant à ce dernier de *sélectionner* les commandes qu'il souhaite pour créer un programme.

En plus de créer le programme en montrant les commandes à effectuer, l'utilisateur doit souvent guider le processus de généralisation. Dans les systèmes sans inférence, c'est-à-dire non intelligents, c'est l'utilisateur qui doit effectuer toutes les généralisations à la main. D'autres systèmes (intelligents) infèrent les intentions de l'utilisateur pour déterminer les structures de contrôles et de données. Dans ce cas, la phase de création du programme est parfois accompagnée d'un retour utilisateur afin de montrer les inférences effectuées par le système et permettre à l'utilisateur de les corriger.

Les analyses ont montré que la généralisation se passe en général selon deux niveaux.

Le premier niveau de généralisation consiste à considérer les objets manipulés. En effet, un programme est censé pouvoir s'exécuter sur des « variables », c'est-à-dire avec des valeurs différentes d'une exécution à l'autre. Lorsque l'on travaille sur un exemple, on dispose de valeurs concrètes ; la généralisation consiste à distinguer les valeurs qui vont changer d'une exécution à l'autre de celles qui demeureront constantes. SmallStar est le premier système à se focaliser sur les caractéristiques des objets manipulés par les systèmes de PsE. Il définit une notion de descripteur, qui permet de transformer la référence à un objet unique en une définition sémantique. La notion de variable, bien connue en programmation classique, est évidemment définie. Elle est étendue par des notions plus spécifiques, par exemple la notion de collection : une variable peut être généralisée de manière à répondre à une définition générique, ce qui entraîne implicitement une itération de collection. Le système appartient à la catégorie des systèmes iconiques, et la généralisation se traduit par exemple par l'utilisation de joker, comme « *.txt » pour représenter des noms comportant obligatoirement un suffixe particulier. Notons que, dans la majorité des cas, la sémantique de la variable n'a pas besoin d'être connue par le système d'espionnage, qui se contente d'être capable de référencer l'objet pendant la phase de rejeu. Ceci n'est plus vrai dès lors que l'on ajoute des structures de contrôle au programme.

La deuxième phase de généralisation consiste à ajouter une sémantique de contrôle au programme. Très naturellement, les techniques peuvent relever d'approches différentes. Lorsque l'on s'appuie sur une sémantique de programmation structurée, il convient d'ajouter ou de repérer des alternatives et des répétitions. C'est le cas de SmallStar, par exemple. Les approches fonctionnelles (Pygmalion par exemple) ou logiques (StageCast Creator) ont également été explorées. Dans tous les cas, afin de contrôler les structures utilisées, il devient nécessaire de définir parmi les objets une sémantique booléenne. En effet, lors du rejeu, c'est une valeur booléenne qui permettra de choisir quelle alternative exécuter, ou qui dirigera la règle à appliquer. Des expressions booléennes plus complexes pourront s'avérer nécessaires (comme le montre EBP). Enfin, pour permettre à l'utilisateur de comprendre plus facilement cette sémantique, des expressions numériques seront également introduites.

2.2.4 Inférence et règles

La notion d'inférence est définie par Myers comme « *l'aptitude du système à générer de nouveaux faits à partir d'autres informations* ». Cette définition semble incomplète comme le démontre (Girard, 1992).

Certains systèmes enregistrent les commandes de l'utilisateur et laissent à ce dernier le soin d'ajouter les structures de contrôles comme les conditions, les boucles et les variables. Une seconde possibilité consiste à laisser le système « deviner » les intentions de l'utilisateur et effectuer les généralisations appropriées. Dans ce cas, le système doit posséder un moteur d'inférence contenant plusieurs heuristiques afin de généraliser les commandes enregistrées.

La mise en œuvre de la généralisation peut s'effectuer de manière très diverse. Nombreux sont les systèmes qui demandent explicitement à l'utilisateur de faire cette généralisation. SmallStar (Halbert, 1993) oblige l'édition du programme enregistré, pour effectuer manuellement la généralisation des objets, et impose l'introduction textuelle des structures de contrôles. Le caractère sur exemple devient moins marqué. D'autres systèmes (Pygmalion, EBP, StageCast Creator) exigent que les structures de contrôles soient prévues par le programmeur, et introduites par des commandes spécifiques.

Pour éviter ce côté explicite, analysé comme contraire à la logique de l'utilisateur non-programmeur, plusieurs auteurs ont choisi de proposer des méthodes utilisant l'exemple de façon plus marquée (Tuchinda et al., 2008). Ainsi, PBE (Bauer, 1979) se sert-il de deux exécutions différentes pour déduire quelles sont les variables et quelles sont les constantes. Eager demande à l'utilisateur de confirmer les actions qu'il a détectées, et analyse les déviations pour construire le programme correct.

Allant plus loin sur la voie de l'inférence, c'est-à-dire l'induction de faits nouveaux à partir de faits connus, de nombreux auteurs ont couplé des moteurs d'inférence aux systèmes de PsE. Eager ou Mondrian (Lieberman, 1993a) reconnaissent ainsi des situations complexes. Des résultats probants ont été obtenus dans le domaine de la simulation. La définition de nouvelles règles par l'utilisateur est considérée comme l'étape suivante : Grammex (Lieberman et al., 1998) permet ainsi de définir explicitement, sur exemple, de nouvelles règles qui s'appliqueront ensuite comme des règles préexistantes dans le système.

Ces approches, quoique séduisantes, se trouvent le plus souvent limitées par leur pouvoir d'expression : elles sont spécifiques du domaine d'application, et peuvent difficilement être généralisées. Elles se heurtent à un deuxième obstacle : lorsque l'on construit de nouvelles règles, il est difficile d'être sûr qu'elles s'appliqueront sans erreur dans tous les cas de figures.

3 Classification des systèmes de la PsE

Les premières classifications ont porté sur le domaine de la « *Visual Programming* » qui a littéralement explosé au milieu des années 1980. Les deux recueils de (Glinert, 1990) constituent un formidable éventail de systèmes très divers. Mais c'est en 1986 que les définitions précises et les premières classifications ont été proposées. Nous allons parcourir les différentes définitions existantes avant de proposer une nouvelle classification conforme aux nouveaux usages de la PsE.

3.1 Classifications existantes et leurs limites

3.1.1 Taxonomie de Myers

Myers s'appuie non seulement sur les définitions d'Halbert pour la partie exemple dans la programmation, mais il part surtout de la définition classique d'un programme : « *un programme est un ensemble d'instructions qui peuvent être transmises à un ordinateur et utilisées pour commander le comportement de ce système* » (Myers, 1990). Il ajoute à cette définition la nécessité de pouvoir comporter des variables, des conditions et des itérations, même de façon implicite. Ses trois critères principaux sont basés sur les définitions de :

- *Compilé / interprété* : Un système compilé se caractérise par un temps d'attente important avant que les instructions puissent s'exécuter, en raison de leur traduction en représentations de bas-niveau, tandis qu'un système interprété permet une exécution des instructions au fur et à mesure de leur création.
- « *Programmation Visuelle* » : Les systèmes qualifiés de programmation visuelle selon Myers sont les systèmes permettant à l'utilisateur de

définir des programmes dans une interface à deux dimensions ou plus. Cette définition est plus restrictive que celle évoquée précédemment.
- « *Programmation basée sur exemple* » : Ce sont les systèmes qui permettent à l'utilisateur d'utiliser un exemple de données d'exécution pendant la phase de programmation.

Cette classification apparaît insuffisante dans la mesure où elle ne porte que sur la phase d'exécution des programmes et la représentation visuelle de ceux-ci. C'est dans le cadre de cette classification, principalement sur le critère de la « *programmation basée sur exemple* » que Halbert a défini le terme d'inférence. Halbert redéfinit ainsi une sous-classification de « *Example-Based Programming* » (Halbert, 1984), (Programmation basée sur exemple).

3.1.2 Grille de classification

Cypher (Cypher, 1993c) a effectué un récapitulatif des systèmes basés sur exemple en 1993. Afin de permettre une bonne comparaison des systèmes entre eux, une grille d'évaluation a été définie. Cette grille s'organise autour de quatre points :

- *Domaine et utilisateurs* : Le domaine est l'entité logique pour laquelle a été conçue l'application. Autrement dit, l'application apporte une solution à un problème de l'entité nommée, c'est le domaine d'application. Les systèmes de PsE couvrent une large variété d'applications avec des spécialisations dans des domaines plus ou moins larges comme le traitement de texte, la création d'interfaces graphiques, l'animation graphique, etc. D'autres systèmes sont indépendants de tout domaine. Il est également important de considérer la catégorie d'utilisateurs ciblée car ce paramètre influe sur la conception du système. Les systèmes de PsE s'adressent à une grande variété d'utilisateurs, sachant ou non programmer, membres de diverses professions, utilisateurs d'une application particulière, etc.
- *Moyens d'interaction* : L'activité de programmation de façon générale, est constituée de plusieurs phases incluant la création, l'invocation, l'exécution, la correction, la visualisation et la modification d'un programme. Comme c'est le cas pour les environnements de programmation traditionnels, les systèmes de PsE enchaînent ces phases de différentes façons. Dans certains systèmes, l'utilisateur enregistre le programme pour ensuite l'exécuter. Dans d'autres systèmes, les différentes phases sont enchevêtrées.
- *L'inférence* : C'est la capacité du système à générer de nouveaux faits à partir d'autres informations comme présenté dans la section 2.2.4.
- *Les connaissances du domaine* : C'est l'implantation du système en fonction du domaine d'application ou de l'application elle même. Les langages utilisés pour l'implantation des systèmes de PsE sont variés allant de langages de bas niveau à des langages de haut niveau. Beaucoup de systèmes de PsE ont une connaissance du domaine et sont souvent trop imbriqués tandis que d'autres n'ont aucune connaissance du domaine.

Cette grille ne permet que de montrer la validité des outils ou du moins de certains outils. Elle ne permet pas en réalité une classification complète prenant en compte la représentation des programmes ni même leur exécution.

3.1.3 Classification de Girard

Girard (Girard, 2000) a analysé certaines insuffisances des critères de classification proposés par Myers pour les différents systèmes de programmation. Cette analyse l'a amené à revoir cette classification. Il propose ainsi de classifier les systèmes de programmation selon quatre critères principaux orthogonaux :

- *Compilé / Interprété* : La définition des langages compilés / Interprétés est relativement floue, comme l'indique lui-même Myers (Myers, 1990), car il n'existe pas de réelle discontinuité. Girard distingue sous le vocable de systèmes à *compilation* ceux qui effectuent la nécessaire traduction (prélude à l'exécution) de façon globale, des systèmes qu'il qualifie *d'interprétés* car effectuant cette même traduction de façon partielle, alors que le processus de conception du programme n'est pas terminé.
- *Programmation graphique ou non* : Un système de programmation est dit *graphique* si la définition du programme peut être effectuée par interaction graphique dans un espace à deux dimensions au moins.
- *Programmation sur exemple ou non* : Un système de programmation est dit *sur exemple* si l'utilisateur peut utiliser les valeurs d'un exemple d'exécution pour définir les objets sur lesquels porte le programme à construire.
- *Déclaratif ou impératif* : Un système de programmation est dit *impératif* si la structure de contrôle de l'exécution de l'algorithme résultant du programme est décrite par le programmeur. Il est dit *déclaratif* si la structure de contrôle d'exécution est définie par le système à partir de relations entre objets définies par le programmeur.

L'illustration de la classification faite dans (Girard, 2000) ne prétend aucunement au classement de tous les systèmes et environnements de programmation. Le Tableau 1, qu'il propose, se veut seulement une illustration de la taxonomie élaborée et présentée.

Tableau 1 : Illustration de la taxinomie de Girard

		SUR EXEMPLE		SANS EXEMPLE	
		Textuel	Graphique	Textuel	Graphique
IMPERATIF	Compilé	Programming by example		Ada, C, FORTRAN	Problox
	Interprété	Tinker	SmallStar	APL, Basic, LISP	LABVIEW
DECLARATIF	Compilé	Infering Lisp Program by example		PROLOG III	ThinglaB
	Interprété	Editing by examples	Peridot	PROLOG II	Juno

3.2 Nouveaux critères de classification

Si les études présentées dans le § 3.1 sont intéressantes pour tenter de classifier les systèmes par rapport à l'aspect visuel ou par rapport à l'activité de construction ou de programmation, elles ne permettent guère de les situer en fonction des utilisateurs. Ces classifications ne caractérisent non plus aucunement les systèmes en fonction de l'utilisation même de la PsE. Non seulement, l'utilisateur n'est pas pris en compte, mais surtout l'usage fait des systèmes ne ressort pas.

Beaucoup de systèmes de PsE n'ont eu pour but que de démontrer la validité du concept même de la PsE (Cypher et al., 1993). Les systèmes de PsE présentent des objectifs très divers, de l'apprentissage de la programmation à la simple automatisation de tâches répétitives (Girard, 2000). Ils couvrent une large variété d'applications (Chen and Weld, 2008). Afin de caractériser les besoins en termes de PsE et, de canaliser les efforts de programmation et d'études des systèmes de programmation sur exemple, nous avons classé les systèmes en fonction de l'utilisation de la PsE elle-même, c'est-à-dire de l'usage du système.

Dans cette sous-section, nous présentons d'abord la démarche d'étude de la classification suivie des résultats obtenus.

3.2.1 Démarche

La démarche que nous avons utilisée a consisté essentiellement en une analyse approfondie des systèmes de PsE existants. Nous avons ainsi appréhendé le principe de fonctionnement général de chacun de ses systèmes et analysé leur but, tel que défini par les auteurs. Les résultats obtenus par la mise en œuvre d'exemples de chacun de ses systèmes ont été étudiés. Enfin, l'étude des solutions techniques mises en œuvre pour la réalisation des systèmes a permis de mieux approfondir nos travaux.

Nous avons non seulement défini les domaines d'application des systèmes de programmation sur exemple, mais aussi les modes de programmation utilisés par ces systèmes. L'ensemble de ces résultats nous a permis de catégoriser les systèmes.

3.2.2 Résultats

L'analyse des systèmes incluant la programmation sur exemple montre de façon naturelle une classification à trois niveaux suivant le critère logique des différentes étapes des langages. De ce point de vue, on distingue les systèmes de *PsE syntaxiques (1)*, les systèmes de *PsE sémantiques (2)* et les systèmes de *PsE pragmatiques (3)*. Dans les systèmes de PsE syntaxiques, on interagit graphiquement avec les instructions, c'est-à-dire le programme. Pygmalion est un système exemple de la PsE syntaxique. La PsE sémantique permet une interaction directe avec les objets du contexte par le biais d'une métaphore. ToonTalk en est un parfait exemple. Enfin la PsE pragmatique permet une programmation, du côté utilisateur, dans l'interface finale (Prabaker et al., 2006). EBP ou Gamut illustrent bien ce type de PsE.

Note objectif n'étant pas de déduire une classification triviale, nous avons approfondi les objectifs visés et les techniques utilisées par les systèmes de PsE afin d'établir une classification plus approfondie et prenant en compte non seulement les objectifs du système lui-même, mais aussi ceux de l'utilisateur ainsi que leurs utilisations.

3.2.2.1 Domaines d'application des systèmes de PsE

Les domaines d'application auxquels s'appliquent les systèmes de PsE se résument à « *Tout système 'graphique' interactif* ». Il s'agit en général, de l'ensemble des Applications Graphiques Interactives (AGI). Ce concept regroupe à la fois les aspects communs aux interfaces graphiques, et le domaine des applications utilisant la représentation graphique.

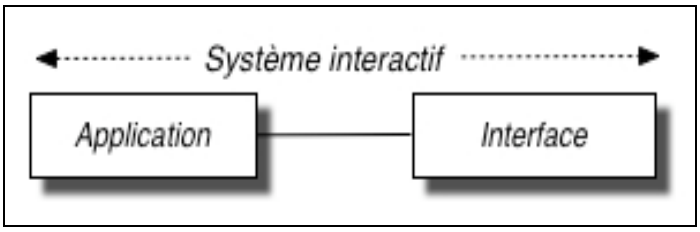

Figure 12 : Système interactif

Un système interactif peut être défini comme l'assemblage de deux composants logiciels : l'application et l'interface (Figure 12). L'application regroupe, pour un domaine donné, l'ensemble des concepts qui correspondent aux variables fonctionnelles de l'utilisateur (Faaborg and Lieberman, 2006). L'interface présente graphiquement ces concepts et assure la communication d'informations entre l'application et l'utilisateur. Cette vision suggère de répartir le modèle conceptuel du système et l'état de l'interaction entre l'application et l'interface. Le modèle conceptuel dans l'application est l'ensemble des abstractions spécifiques au domaine qui satisfont aux besoins des tâches de l'utilisateur. Cet ensemble correspond aux entités et aux opérations conceptuelles. Il détermine la sémantique statique du système. La sémantique dynamique s'exprime par l'enchaînement des états de l'interaction. Pour l'application, l'interface est la représentation du monde externe. C'est avec elle qu'elle communique. C'est d'elle qu'elle reçoit les demandes d'accès aux abstractions (par exemple sous forme d'appels de fonction). C'est à elle qu'elle exprime ses changements d'état ou ses besoins en données. Dans l'interface, le modèle conceptuel et l'état de l'interaction sont exprimés à l'aide d'abstractions spécialisées dans la communication homme machine. Ces abstractions correspondent aux entités syntaxiques et aux éléments d'interaction. Elles jouent le rôle de tampon entre les abstractions conceptuelles de l'application et les actions de l'utilisateur. Chacune participe à la réalisation de l'interface. Chacune est une mini-interface qui représente une part du modèle conceptuel et une part de l'état de l'interaction. Au niveau macroscopique, cette interface sert de traducteur entre les formalismes de l'application et la forme des expressions d'entrée et de sortie. La gestion des ressources de l'interaction est assurée par les systèmes de fenêtrage dont les concepts visibles aux actions clients incluent les événements, les surfaces d'affichages, les fenêtres, etc. La gestion du dialogue s'effectue au sein d'un gestionnaire qui contrôle un ensemble d'objets de présentation au moyen d'un modèle de l'interaction.

De ce parcours des systèmes interactifs, nous voyons que les techniques mises en place pour réaliser ces systèmes constituent le cœur de la programmation sur exemple. Les systèmes Tinker, Tels, Peridot, Pygmalion, tous dans (Cypher, 1993c), possèdent des environnements semi-graphiques permettant à l'utilisateur de définir ses objectifs tandis que StageCast Creator, ToonTalk (Kahn, 2001) et Mondrian offrent des environnements graphiques complets. Le système EBP (Pierra et al., 1996) utilise même la programmation visuelle et permet à l'utilisateur d'interagir avec l'outil qu'il crée par des actions purement graphiques.

Néanmoins, certains domaines d'application sont plus particulièrement étudiés :

- *Le domaine de l'apprentissage de la programmation* : il est utilisé pour permettre à l'utilisateur de réaliser ou d'apprendre à programmer en utilisant les techniques de programmation comme la définition de nom de procédure, de paramètres et de valeurs effectives. Le système Peridot illustre fort bien ce domaine. Dans Peridot, l'utilisateur crée une interface graphique en démontrant les représentations graphiques de

celle-ci et en précisant les actions en réponse. Les systèmes comme Tinker (Lieberman, 1993b), ToonTalk et StageCast Creator permettent à l'utilisateur de réaliser ces mêmes objectifs mais de façon différente et avec des techniques particulières.

- *Le domaine des applications de conception technique (pièces paramétrées)* : on retrouve dans ce domaine les systèmes comme EBP, GIPSE (Patry and Girard, 1999) et Mondrian. EBP permet la création d'outils paramétrés tandis que Mondrian permet à l'utilisateur de se doter de commandes graphiques pour la réalisation de tâches graphiques. Les systèmes permettant la création de programmes peuvent être aussi considérés ici car les programmes créés sont des outils exécutés pour obtenir des résolutions d'un problème (correspondant à une tâche utilisateur).
- *Le domaine des constructions d'interfaces (dessins rectangulaires – application de dessin)* : Peridot qui permet l'apprentissage de la programmation a pour objectif de pouvoir fournir des interfaces graphiques définies par l'utilisateur. Les systèmes comme Pavlov (Wolber, 1996) aident l'utilisateur à obtenir une interface personnalisée par la définition de ses besoins.
- *Le domaine de la simulation ou les jeux* : ce domaine est le plus investi par les systèmes pour créer des applications à partir d'exemples de jeux. Non seulement ils fournissent des programmes de jeux, mais aussi ils sont utilisés pour apprendre à programmer. StageCast Creator et ToonTalk utilisent ce domaine pour mieux favoriser l'appréhension facile des concepts de programmation. Les environnements utilisés sont propres à ces systèmes et ont trait à des simulations de jeux ou de la vie miniaturisée. Gamut (McDaniel and Myers, 1999) peut être cité dans cette catégorie de la création de simulation. Les systèmes appartenant à ce domaine combinent en général la notion de métaphore et les concepts de programmation.
- *Et les animations* : elles sont utilisées pour illustrer en général des interactions d'objets comme dans StageCast Creator et Mondrian. Mais dans Mondrian, l'animation est définie par l'utilisateur lors des manipulations d'objets.

Ces domaines se dégagent comme pertinents du point de vue de la PsE. On peut les résumer d'une autre manière selon les objectifs d'utilisation en définissant les champs d'application.

3.2.2.2 Champs d'application de la PsE

Les champs d'application correspondent à l'usage effectif dans un domaine particulier du système de la PsE. Pour parler de l'usage effectif, il est nécessaire de faire un rappel de ce pour quoi les systèmes de PsE ont été réalisés et à quoi ils sont utilisés, en résumant les domaines particuliers et pertinents.

L'un des premiers systèmes relevant de la PsE, comme nous l'avons déjà présenté est Pygmalion dont l'objectif est de permettre la construction d'un programme en s'appuyant sur un exemple. Le domaine des macros est un domaine privilégié de la programmation par l'utilisateur final. SmallStar est une des premières tentatives pour adapter une technique graphique de PsE à cette problématique. C'est le premier système de PsE destiné au grand public dans le but d'automatiser les tâches répétitives. Le domaine de

conception technique (incluant en particulier la CAO) a pleinement assimilé les techniques de la PsE, au point qu'aujourd'hui, les systèmes dits paramétriques, les plus utilisés au plan industriel, fondent leur modélisation sur les principes de la PsE. EBP est un exemple d'environnement de développement de programmes paramétrés, utilisant des techniques de PsE permettant la création de composants standard portables par des utilisateurs de systèmes de CAO. Un autre domaine est celui portant sur l'enseignement, l'éducation et la simulation. Fournir une assistance, par l'exemple, à la compréhension des mécanismes de programmation est un thème important dans la PsE. Un environnement d'initiation à la programmation et basé sur l'exemple est MELBA (Guibert et al., 2005) dont la cohérence est assurée par des mécanismes de PsE. Enfin, l'assistance à l'utilisateur est un autre domaine emblématique de la PsE, bien représenté par Eager qui permet d'automatiser les tâches répétitives. Il observe en permanence le comportement de l'utilisateur pour essayer de prédire ses prochaines commandes. À ces trois domaines, se dégageant comme particulièrement pertinents du point de vue de la PsE, on peut ajouter celui de l'automatisation des tests d'interfaces par le biais de la technique d'enregistrement – rejeu de la PsE. L'utilisation de cette technique s'avère extrêmement prometteuse pour la vérification et la validation des applications interactives. Ce champ est une possibilité d'application de la PsE. En résumé, nous pouvons dire qu'il existe quatre champs d'application pertinents de la PsE.

3.2.2.2.1 Fonction d'assistance

La possibilité de personnaliser son application est un besoin considéré comme majeur par les utilisateurs. De nombreux éditeurs de logiciels ont ainsi développé des solutions s'appuyant peu ou prou sur les techniques de la PsE pour enrichir les possibilités d'adaptation. Les techniques de généralisation *a priori* sont utilisées dans l'outil Automator qui permet d'automatiser des tâches dans Mac OS X. Les techniques d'inférence et d'espionnage trouvent leur utilité dans des agents d'aide, ou autres assistants. Dans MS-Excel©, par exemple, un espionnage systématique de l'utilisateur permet de détecter les situations qui s'apparentent à une gestion de liste (Figure 13), comme le fait Eager.

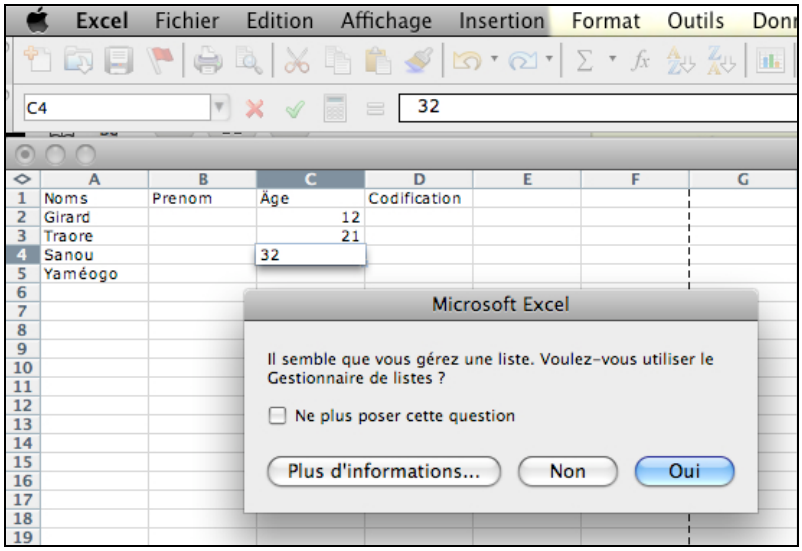

Figure 13 : Exemple d'utilisation de la PsE dans MS Excel

Les techniques de la PsE, utilisées de façon ponctuelle dans certaines situations d'interaction, sont de nature à apporter de nouvelles fonctions d'assistance à l'utilisateur.

Selon les besoins, des techniques explicites de généralisation seront préférables à l'utilisation de l'inférence, principalement lorsqu'on souhaitera que l'utilisateur puisse contrôler finement son automatisation. Elles sont la base de l'adaptabilité des applications. L'usage de l'inférence permet en plus d'atteindre des objectifs d'adaptativité.

3.2.2.2.2 Pédagogie

Les applications pédagogiques et de simulation constituent un champ d'investigation important pour la PsE. Les objectifs des systèmes qui l'utilisent sont très variés. Ils peuvent concerner l'apprentissage de la programmation (MELBA, ToonTalk), ou plus généralement de la stratégie de résolution de problème (StageCast Creator, Agentsheets[4] (Repenning and Perrone, 2001)). L'effort principal se porte dans ces systèmes sur l'aide à l'explicitation des principes de généralisation. De ce fait, un effort tout particulier est porté sur la visualisation des programmes, qui cherche à être la plus explicite possible. Des projets extrêmement avancés sont actuellement en cours (LegoSheets[5]).

3.2.2.2.3 Conception

Le domaine des outils de conception est certainement celui où la PsE est intégrée le plus profondément aux applications. Les outils graphiques modernes utilisent abondamment l'analyse temps-réel des interactions pour fournir une assistance au placement des composants (alignement, contraintes, etc.). On trouve ces fonctionnalités dans des outils de dessin (OmniGraffle[6]), de présentation (Keynote) ou encore les GUI-Builders (Matisse dans NetBeans). Comme nous l'avons décrit, le domaine de la CAO est également très concerné.

3.2.2.2.4 Tests d'interfaces

L'utilisation des techniques d'enregistrement-rejeu de la PsE s'avère extrêmement prometteuse pour la vérification et la validation des applications interactives (Sanou et al., 2008c). Alors que les techniques de tests unitaires font de plus en plus d'émules, et que leurs outils sont de plus en plus utilisés, tester les interfaces s'avère encore un processus difficile. Aujourd'hui, ce sont les approches intégrant la PsE qui semblent émerger. Ainsi, Jacareto permet d'enregistrer les interactions de bas-niveau et autorise l'automatisation de tests d'interfaces. Pour rendre ces outils plus puissants, un couplage avec les approches à base d'analyse de tâches, comme dans (Tarby, 2006), est une piste de travail très intéressante.

3.2.2.3 Modes de programmation utilisés

Il s'agit des techniques ou méthodes utilisées afin d'instruire les systèmes de PsE lors de la construction de l'exemple (Modugno and Myers, 1993). Nous pouvons regrouper les modes de programmation utilisés au niveau des systèmes de PsE en deux groupes : la programmation textuelle et la programmation iconique.

La programmation textuelle est l'instruction du système à l'aide d'instructions textuelles représentant des commandes. Ces instructions textuelles ne sont qu'une succession de chaînes de caractères respectant une syntaxe bien précise suivant le langage

[4] http://www.agentsheets.com/index.html
[5] http://l3d.cs.colorado.edu/systems/legosheets/
[6] http://www.omnigroup.com/applications/OmniGraffle/

de programmation adopté. Par exemple, dans Tinker le programmeur construit des expressions LISP en assemblant des fonctions LISP. Peridot, Mondrian et Pavlov utilisent l'usage de texte mais pas de façon complète ou générale. Cette technique est utilisée afin de mieux cibler ou définir de façon particulière une fonction ou un programme. La programmation textuelle introduit l'utilisation de primitives simples ou composées pour définir une instruction. Ces primitives demandent des arguments et des valeurs. En fonction des systèmes, les primitives diffèrent. Cette technique demande beaucoup de connaissances et de maîtrise du langage de programmation. Il est à noter que l'on s'approche beaucoup plus du langage naturel dans le domaine de la programmation textuelle (Automator[7]). Cela rend la tâche plus facile à l'utilisateur.

La programmation iconique consiste à utiliser des icônes ou des dessins rectangulaires pour instruire le système. Une icône introduit, par rapport aux instructions textuelles élémentaires, la possibilité de construire des instructions graphiques. Le mécanisme d'abstraction proposé s'apparente à la technique des macro-instructions qui encapsule un ensemble de primitives en une entité manipulable comme un tout. Le niveau d'abstraction de ces primitives est celui des instructions élémentaires. Pygmalion illustre bien ce concept. Pygmalion est une tentative de programmation visuelle en se basant sur une métaphore. Gamut dépasse le cadre de Pygmalion en réalisant du graphisme interactif. Dans cette technique, chaque icône est équivalente à un objet du système. Dans StageCast Creator ou ToonTalk, ou encore Mondrian, les éléments du système sont représentés par des objets en figure ou image permettant ainsi à l'utilisateur de réaliser ses actions par la mise en place d'une scène. On peut aussi utiliser des boîtes de dialogues dans la programmation iconique. SmallStar et Metamouse (Maulsby et al., 1989) offrent cette possibilité à l'utilisateur.

Il est à rappeler qu'un aspect très particulier de la programmation iconique est la programmation visuelle (graphique). Dans ce mode de programmation, on utilise du graphisme pur. Un système de programmation est dit graphique (« *Visual Programming* » en anglais) si la définition du programme peut être effectuée par interaction graphique dans un espace à deux dimensions au moins (Myers, 1990). Les éléments de base sont le point, le segment, la ligne, le cercle, etc. Parmi les systèmes de PsE, EBP ou GAMUT utilisent ce mode. La programmation graphique introduit, par rapport à la programmation iconique simple, des possibilités de structuration et d'expression de contraintes et d'interactions. Une icône peut inclure d'autres icônes internes. Les unités graphiques, simples ou composées, peuvent être liées par des relations géométriques.

3.2.2.4 Regroupement en catégories

Les résultats des études précédentes sur les domaines d'application des systèmes de la PsE et le mode de programmation utilisé peuvent se résumer sous forme de comparaison entre les systèmes présentés. Le Tableau 2 présente le récapitulatif de cette comparaison. On trouvera dans ce tableau un résumé du positionnement de quelques uns des systèmes principaux de la PsE au regard des résultats déduits.

Malgré l'expansion des champs d'application de la PsE, on retrouve principalement trois domaines de fonctionnalités pertinentes du point de vue de la PsE. La fonction d'assistance (1), offrant la possibilité de personnaliser son application, est un besoin considéré comme majeur par les utilisateurs. Les applications pédagogiques et de simulations (2) constituent un champ d'investigation important pour la PsE. Les objectifs

[7] http://www.apple.com/fr/macosx/features/300.html#automator

des systèmes qui les utilisent sont très variés. Ils peuvent concerner l'apprentissage de la programmation ou plus généralement la stratégie de résolution de problèmes. Le domaine des outils de conception et de validation (3) est certainement celui où la PsE est intégrée le plus profondément aux applications.

Tableau 2 : Comparaison des systèmes présentés suivant les résultats de l'étude

Système	***Domaine***	***Mode de programmation***	***Utilisateur***
Pygmalion	Programmation – Edition de programme	Visuelle (plus iconique)	Moyen
Small Star	Système de gestion de fichiers (SE graphique interactif)	Iconique	Simple débutant
StageCast Creator	Apprentissage de la programmation – Construction d'interface	Iconique (peu graphique)	Enfant (novice)
EBP	Conception technique	Visuelle	Moyen
Gamut	Construction (génération) d'application graphique	Graphique (peu iconique)	Débutant
ToonTalk	Apprentissage de la programmation – Construction de jeu	Iconique (peu graphique)	Enfant (novice)
MELBA	Apprentissage de la programmation	Dans l'interface (plus qu'iconique)	Étudiant (débutant)

L'appartenance à des domaines d'application ne permet pas ou ne signifie guère la répartition suivant une classe. En effet, le domaine d'application ne peut être considéré comme une catégorisation des systèmes. Nous allons donc définir un regroupement des systèmes suivant leur objectif et leur utilisation.

3.3 Nouvelles catégories de classification des systèmes de PsE

Les mécanismes de PsE incorporés aux systèmes ou permettant de créer d'autres systèmes, en partant de leur objectif et de leur assignation révèlent des fonctionnalités particulières. En situant les systèmes en fonction de l'utilisateur et en les caractérisant en fonction de l'utilisation même de la PsE, nous proposons une classification des systèmes de PsE, c'est-à-dire selon les besoins de la PsE elle-même et de l'utilisateur. En effet, certains systèmes (Eager, SmallStar, Aide Project, etc.) permettent d'automatiser les tâches répétitives de l'utilisateur sans nécessairement faire appel à l'utilisateur. D'autres favorisent l'apprentissage de la programmation elle-même (Tinker, Peridot, …). Il existe des systèmes (Pygmalion, EBP, ToonTalk, Pavlov, etc.) permettant l'édition de programme. Aussi, des systèmes (Metamouse, Gamut, StageCast Creator, etc.) génèrent des applications. Enfin, il y a les systèmes (EBP, Mondrian, GIPSE, etc.) qui créent des outils ou des objets de conception technique.

Ces vues divergentes de la finalité des différents systèmes de PsE peuvent se subdiviser en quatre catégories essentielles : l'assistance, les outils de conception, l'apprentissage de la programmation et les outils de PsE.

3.3.1 Assistance

L'assistance est la catégorie regroupant tous les systèmes permettant d'aider l'utilisateur à automatiser ses tâches. Ils offrent une aide contextuelle à l'utilisateur afin de

faciliter ou de réduire la difficulté d'exécution de certaines de ses tâches. C'est le but des approches actuelles de l'EUD (End User Programming) (Lieberman et al., 2006).

Les assistants n'utilisent pas tous la séquence « démarrer enregistrement – arrêter enregistrement ». Certains enregistrent en permanence ce que fait l'utilisateur et lui proposent de terminer sa tâche lorsqu'ils détectent une répétition. D'autres permettent à l'utilisateur de consulter et de sélectionner des commandes passées pour les ré-exécuter, pour la plupart dans le même contexte.

Dans cette classe le but global du système est éloigné de la PsE. On peut citer comme exemple Eager, SmallStar, qui cherchent à résoudre les problèmes d'expressivité des langages iconiques en permettant de créer des macros puissantes, ou encore Tels qui est spécialisé vers les tâches d'édition de texte.

3.3.2 Outils de conception

Les outils de conception rassemblent les applications dont le but est de produire un résultat qui découle directement de l'utilisation de la PsE. Ces outils peuvent êtres classés sous la rubrique générale de la ***conception assistée par ordinateur***. Qu'il s'agisse de pièces mécaniques comme pour EBP, d'applications à destination des enfants pour StageCast Creator, ou encore d'un GUI-Builder amélioré comme Peridot, ils ont en commun de s'appuyer sur la PsE pour atteindre leur objectif.

Ils regroupent l'ensemble des constructeurs et concepteurs d'outils documentaires (Whitley et al., 2006) ou d'outils de mise au point. Les systèmes de cette classe mettent à la disposition de l'utilisateur des outils de conception ou de construction pour la réalisation d'objets, d'outils ou de pièces (complexes). La finalité est l'obtention d'un « *programme* ». Nous pouvons aussi citer les systèmes comme Mondrian, GIPSE, Metamouse, Gamut, etc.

3.3.3 Apprentissage de la programmation

Dans cette classe, l'utilisateur apprend à programmer tout en programmant (Sanou et al., 2008a). En effet, ces systèmes donnent à l'utilisateur la possibilité de réaliser des programmes sans pour autant connaître la programmation. Ils offrent un système d'apprentissage aisé de la programmation en se basant sur le principe de l'algorithmique. Leur finalité est d'apprendre la programmation. On peut citer Tinker, MELBA, ou encore ToonTalk.

Les systèmes d'apprentissage de la programmation permettent souvent à l'utilisateur de se défaire de la notion de programmation directe, c'est-à-dire du niveau d'abstraction et des concepts de programmation telle la désignation d'objets, les constantes ou les variables sur lesquelles portent ses actions.

3.3.4 Outils de PsE

Ce sont les systèmes dont l'objectif est clairement de construire des applications de PsE, c'est-à-dire développer à moindre coût des solutions incluant la PsE (Sanou et al., 2006c). Dans cette classe, il n'existe que deux solutions : AIDE (Piernot and Yvon, 1993) et PbDScript (Depaulis et al., 2003).

Les outils de PsE fournissent une couche logicielle sur laquelle des développeurs peuvent bâtir, avec un minimum d'effort, des applications mettant en œuvre des techniques de la programmation sur exemple. De façon plus spécialisée, les outils d'enregistrement/rejeu dédiés à la définition de tests d'interfaces utilisateur peuvent être

classés dans cette catégorie. Le Tableau 3 ci-dessous donne un résumé de la répartition dans les différentes catégories de quelques-uns des principaux systèmes utilisant la PsE ou incluant la PsE au regard de cette classification. On notera qu'un même système peut appartenir à deux catégories différentes, comme StageCast Creator par exemple.

Tableau 3 : Classification de quelques systèmes de PsE

Catégories / Systèmes	Assistant	Apprentissage de la Programmation	Outil de Conception	Outil de PsE
Eager	√			
SmallStar	√			
AIDE	√			√
Tinker		√		
Peridot		√	√	
Pygmalion			√	
EBP			√	
ToonTalk		√	√	
Metamouse	√		√	
Gamut			√	
StageCast Creator		√	√	
Mondrian			√	
GIPSE			√	
Tels	√			
MELBA		√		
PbDScript				√

4 Conclusion

La plupart des systèmes de PsE sont liés à des domaines d'application très restreints. Certains mêmes sont développés dans des langages propriétaires et par conséquent, ils ne sont pas exploitables sur toutes les plates-formes. Ils demandent souvent la connaissance du vocabulaire et de la syntaxe des langages de programmation pour la modification de séquences et souvent même les concepts de programmation comme les structures de contrôle, les structures de données. D'autres sont basés sur des moteurs d'inférence et ne demandent (ou n'offrent) aucune particularité à l'utilisateur.

Implanter un système de PsE est une tâche complexe car elle met souvent en jeu des techniques d'intelligence artificielle (apprentissage symbolique, représentation des connaissances, inférences, etc.) en plus des techniques nécessaires à la conception et à l'implémentation, et des techniques d'IHM (création, visualisation et invocation d'un programme, correction d'erreur, etc.).

Nous avons examiné les applications de la Programmation sur Exemple et les avons classifiées en quatre catégories. Des systèmes assistent l'utilisateur lors de l'exécution de sa tâche et automatisent ses actions répétitives pour gagner du temps et diminuer les risques d'erreurs. D'autres intègrent des applications pour développer une solution sur mesure ou encore adapter une application à ses besoins spécifiques. Aussi, des systèmes permettent à l'utilisateur de s'adapter de façon aisée aux différentes notions de la programmation par le biais d'environnements faciles à manipuler en faisant tout de même

de la programmation. Enfin, certains systèmes permettent de construire des applications de Programmation sur Exemple.

Fournir des services pour utiliser la PsE au sein des applications est fortement dépendant de la classe de système (Wong and Hong, 2007). Les applications dont le but est l'apprentissage de la programmation ont des besoins extrêmement spécifiques ; la programmation sur exemple est au cœur même du système interactif. De même, les outils de conception demandent en règle générale une très forte imbrication du système de PsE dans le noyau fonctionnel même du système global. En revanche, les applications de la première catégorie n'utilisent la PsE que pour ajouter des fonctionnalités ou aider l'utilisateur dans sa tâche. Cependant, rares sont les travaux qui ont essayé d'apporter une solution générale permettant de développer à moindre coût des solutions incluant la PsE.

Chapitre 2

Intégration de la programmation sur exemple dans une application interactive

SOMMAIRE

Résumé. *Ce chapitre a pour but d'établir le cahier des charges du concept que nous souhaitons réaliser. L'apport est dans l'identification précise des besoins liés à la technique de programmation sur exemple. Les besoins sont les services à fournir qui se résument aux opérations de base de la PsE : l'enregistrement et le rejeu des interactions utilisateur. En plus des opérations de base, le mécanisme utilisateur est un facteur important devant être pris en compte dans les techniques d'implémentation et d'utilisation de la solution.*

1 Introduction

Le but premier de notre travail consiste à concevoir un outil à destination des programmeurs permettant d'introduire de façon aisée les techniques de PsE dans les applications interactives. S'il est relativement simple, comme nous l'avons fait dans le chapitre précédent, d'isoler les principes de base de la PsE (enregistrement, généralisation, rejeu), il s'avère beaucoup plus compliqué de décliner ces mêmes principes dans un outil qui doit demeurer très général. La plupart des études se sont en fait concentrées sur la notion de généralisation, délaissant bien souvent la problématique de l'enregistrement et du rejeu. Dans une perspective d'application « ad hoc », cela est naturel. Nous verrons qu'il n'en est pas de même si l'on cherche à définir un système généraliste.

Nous avons établi au chapitre 1 une nouvelle classification en fonction de l'usage de PsE au sein des applications. Il nous a semblé judicieux de nous appuyer sur cette classification pour restreindre le champ d'application de notre solution. En effet, il semble difficile d'imaginer que les besoins seront les mêmes qu'il s'agisse d'utiliser la PsE dans un processus de conception, dans une activité d'apprentissage, ou dans un but d'assistance à l'utilisateur. Les travaux menés en particulier dans l'équipe du LISI nous permettront d'apporter une réponse à cette question. Les classes de conception et de programmation requièrent une très forte imbrication du processus de PsE au sein des applications. De fait, vouloir construire un outil généraliste pour des besoins aussi spécifiques nous semble illusoire. La dernière classe, correspond à l'outil que nous souhaitons développer. Il apparaît donc que la classe d'applications pour laquelle nous cherchons à concevoir notre outil est la classe que nous avons appelée « Assistance », qui répond au besoin exprimé dans notre introduction générale : comment permettre à l'utilisateur d'adapter au mieux son application à sa tâche. Ceci se décline pour nous à : quel outil pouvons-nous fournir au développeur pour qu'il puisse aisément répondre au besoin d'adaptabilité de l'utilisateur ?

Seules deux études se sont réellement penchées sur cette problématique. Il s'agit du système AIDE (Piernot and Yvon, 1993), et du système PbDScript (Depaulis et al., 2003). Basés sur des principes opposés, ils proposent une répartition des rôles très différente entre le programmeur et l'utilisateur. Nous verrons que ce problème est crucial dans la définition d'un outil général de PsE.

Ce chapitre est organisé en trois parties. Dans une première partie, nous analysons de façon plus approfondie le domaine de l'adaptabilité des applications, à travers les solutions proposées par les concepteurs actuels. Nous verrons en quoi la PsE apporte un complément intéressant. Dans la deuxième partie, nous examinons les solutions proposées pour répondre à notre besoin précis : fournir un outil général de PsE. Elle détaille les deux solutions proposées par les auteurs, et en montre les limites. Enfin, la troisième partie décrit les services à fournir par un tel outil pour faciliter l'intégration de la PsE dans les applications.

2 Adaptabilité des applications

Il s'agit en fait d'un retour aux sources de la PsE. La possibilité de personnaliser son application est aujourd'hui un besoin considéré comme majeur par les utilisateurs. De nombreux éditeurs de logiciels ont ainsi développé des solutions s'appuyant sur des techniques diverses pour enrichir leurs possibilités d'adaptation. Le rapport du groupe de travail constitué en 1991 sur le thème « *End-User Computing »* en est la reconnaissance. Il est

donc nécessaire de permettre à l'utilisateur de modifier son application, ce qui n'est pas à la portée de tous malgré l'évolution des langages de programmation utilisés.

Dans cette section, nous commençons par une description des techniques classiques permettant à l'utilisateur d'adapter son application. Dans un deuxième temps, nous verrons en quoi la PsE est en mesure d'apporter une réponse à certaines des insuffisances recensées dans ces premières approches. Nous terminerons par une synthèse de ces différentes observations.

2.1 Solutions classiques

Adapter son application n'est pas une fonctionnalité standard donnée à l'utilisateur. La révolution importante qui est intervenue lors de l'arrivée des systèmes graphiques a conduit l'utilisateur à être de plus en plus exigeant sur ses capacités à dominer le système (Macias and Castells, 2007). De fait, le programmeur a été vite contraint de lui proposer des solutions pour atteindre cet objectif, allant bien au-delà de simples modifications superficielles (réarrangement des commandes dans les menus ou les barres d'outils, disposition de l'espace de travail, etc.). Très vite, c'est le comportement même de l'application que l'utilisateur a souhaité modifier. Les préférences, les langages de scripts et les éditeurs de macros constituent les réponses classiques apportées par les développeurs.

2.1.1 Editeurs de préférences

Les préférences sont prévues par le programmeur dans une application lors de sa réalisation. Elles sont très simples à mettre en œuvre et à utiliser. Il s'agit d'un ensemble d'alternatives fixes, offertes par le concepteur, présentées en général sous forme de cases à cocher pour répondre aux différents besoins de l'utilisateur. Leur possibilité d'utilisation est restreinte aux choix prévus par le concepteur.

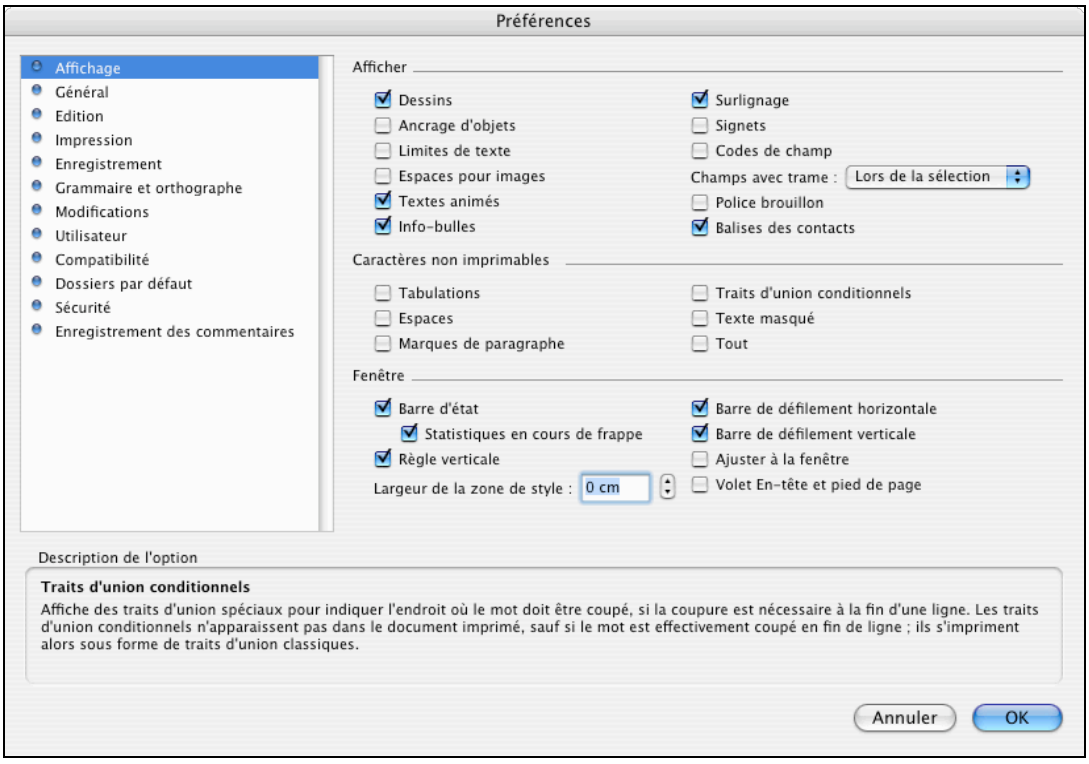

Figure 14 : Préférence d'affichage de Word™ (Office Mac 04)

Elles constituent la façon la plus élémentaire de modifier le comportement d'une application, même si elles ne mettent pas en œuvre une réelle programmation dans la mesure où l'utilisateur choisit simplement une alternative parmi d'autres. Plus les préférences offrent des possibilités, plus elles sont difficiles à manipuler, moins elles respectent les critères ergonomiques, comme la visibilité par exemple. La Figure 14 illustre ce point avec les préférences de l'application Word™.

Bien qu'utiles, les préférences présentent deux limites importantes. Du fait des besoins potentiellement variés des utilisateurs, elles ne peuvent répondre à leur totalité. Les préférences ne permettent d'accommoder qu'un nombre fini de situations prévues par le concepteur de l'application. La seconde limitation est ressentie lorsqu'il faut répondre à plus de besoins possibles. Dans ce cas, un nombre important d'options doit être proposé, rendant ainsi la configuration de l'application très difficile par l'utilisateur.

2.1.2 Scripts et langages de scripts

Les scripts sont de petits et simples programmes très populaires. Un langage de script est un langage de programmation dont la syntaxe est spécialisée à un domaine particulier. Perl et Tcl en sont deux exemples. Lorsqu'un langage de script est dédié à une application particulière, il est qualifié de *langage d'application*. Ces derniers sont des langages de programmation spécialisés de haut niveau permettant de modifier ou d'étendre les fonctionnalités d'une application, en s'appuyant directement sur les objets manipulés par l'application. Ces langages sont plus accessibles que les langages de programmation traditionnels car ils proposent à l'utilisateur un vocabulaire limité, adapté au domaine de l'application. Dans un éditeur de texte par exemple, l'utilisateur pourra insérer, mettre en italique ou effacer des caractères, des mots, des paragraphes etc. Dans un tableur, les objets manipulés seront des colonnes, des lignes ou des cellules qui pourront être sélectionnées, copiées, déplacées etc. Ces langages sont généralement puissants pour permettre à l'utilisateur confirmé de changer de façon drastique le fonctionnement d'une application.

Les limites de l'approche « langage de script » sont relativement nombreuses. Lorsque chaque application possède son propre langage de script, l'utilisateur doit fournir un effort d'apprentissage croissant avec le nombre d'applications. De façon plus générale, les langages d'application restent malgré tout des *langages de programmation* à part entière. Ils nécessitent non seulement la connaissance d'un vocabulaire et d'une syntaxe mais surtout la maîtrise des concepts de programmation tels que les structures de données et les structures de contrôle. Pour l'automatisation d'une tâche, l'utilisateur doit développer ou élaborer complètement un script. Il doit donc tout apprendre du langage, ce qui l'oblige à devenir un « vrai » programmeur. Le domaine d'application des scripts est restreint, et difficilement extensible. En général, pour la plupart des langages d'application, les scripts écrits sur une plate forme ne s'exécutent pas sur les autres.

Enfin, du point de vue du développeur de logiciels, les langages d'application sont coûteux à mettre en œuvre. Ils réclament en effet un effort de conception (choix d'une syntaxe et d'un vocabulaire appropriés) et d'implémentation (écriture d'un interpréteur ou d'un compilateur intégré à l'application) important.

L'un des langages d'application les plus prometteurs est sans doute AppleScript[8] dans la mesure où il permet de piloter de nombreuses applications de façon uniforme et peut être étendu par chacune d'elles par l'ajout de nouveaux objets et de nouvelles commandes. Ainsi

[8] http://www.apple.com/fr/macosx/features/applescript/

l'objectif est de pouvoir contrôler toute application à partir d'un langage de programmation unique, ce qui limiterait l'effort d'apprentissage requis. Un second avantage lié à cette approche est qu'elle autorise la création de programmes pilotant plusieurs applications. AppleScript possède une syntaxe propre, avec un vocabulaire restreint mais extensible et fournit un ensemble de primitives pour que le développeur n'ait pas à créer un interpréteur de toutes pièces. La Figure 15 montre un script censé compter les messages dans un mailer.

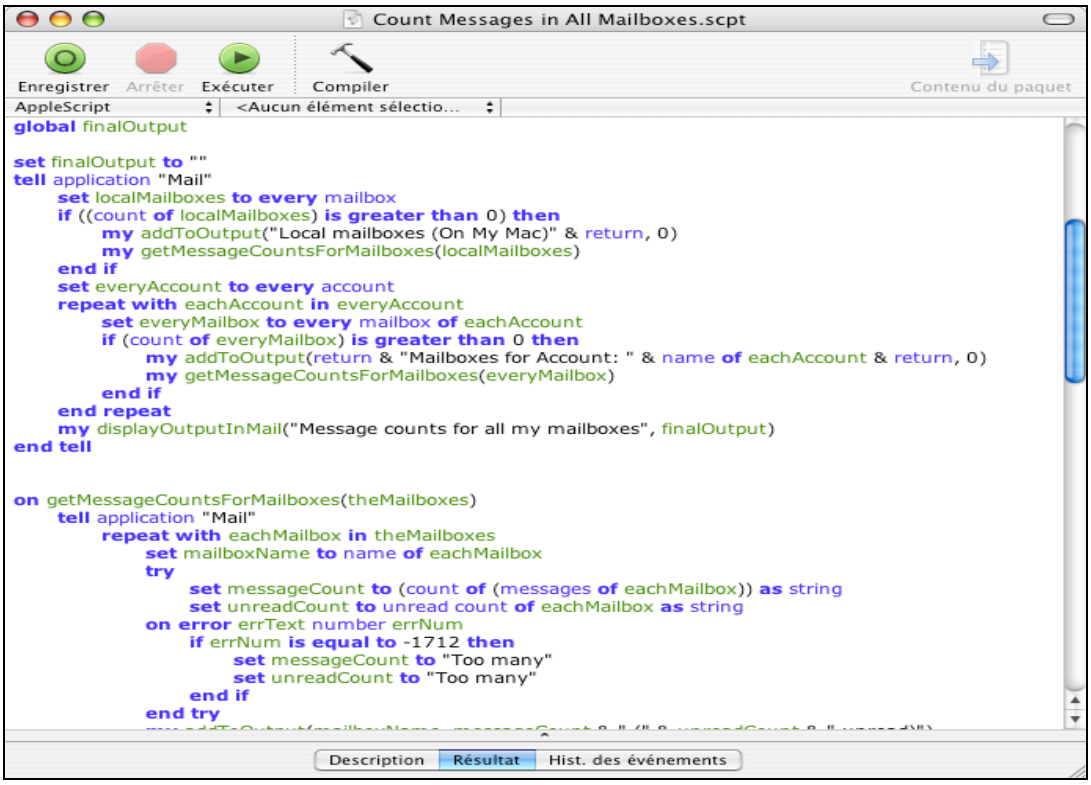

Figure 15 : Éditeur de Script sous Mac OS X

Notons que, malgré son intérêt, AppleScript ne résout pas tous les problèmes soulevés par l'approche langage d'application. En effet, si elle uniformise la démarche, et affranchit le programmeur de la définition d'un interpréteur, sa mise en œuvre dans les applications nécessite de se conformer à un mode de développement particulier. De plus, demeure le principal reproche qui consiste à imposer à l'utilisateur de maîtriser les principes de base de la programmation. Une réponse partielle à ce problème est apportée par les enregistreurs de macros.

2.1.3 Enregistreurs de macros

Les enregistreurs de macros sont aujourd'hui très fréquents dans les applications. Ils permettent aux utilisateurs de créer un programme appelé macro dans le langage d'application concerné, en enregistrant la séquence de commandes qu'ils effectuent. Ils constituent les prémisses de la PsE, puisqu'ils permettent à l'utilisateur d'enregistrer un exemple de la tâche à réaliser qu'il transformera en programme. Tout comme les préférences, les enregistreurs de macros sont aussi prévus par le concepteur de l'application. Leur utilisation très simple se résume au lancement du système par un simple clic sur bouton (en général « Enregistrer Macro »), à jouer ou exécuter les actions désirées et à arrêter l'enregistrement par un autre clic. Sur demande de l'utilisateur, la macro rejoue toutes les actions effectuées par l'utilisateur durant la période d'enregistrement. Beaucoup de logiciels proposent des enregistreurs de

macros : des traitements de texte aux logiciels de communication en passant par les tableurs. De façon analogue à ce que nous avons décrit pour les langages de scripts, des systèmes comme QuicKeys[9] ou AppleScript proposent un enregistrement de macros étendu à tout le système. Ceci présente l'avantage de proposer un mécanisme unique à l'utilisateur et de pouvoir créer des macros mettant en œuvre plusieurs applications.

Les enregistreurs de macros permettent à l'utilisateur de créer des macros pour automatiser des tâches simples sans avoir à apprendre un langage de programmation. Par exemple, dans un traitement de texte, la tâche peut être d'insérer la date courante à chaque fois qu'un nouveau document est créé. Dans un logiciel de communication, elle peut être de charger périodiquement le cours d'une action donnée et dans un tableur, de tracer le graphe mensuel d'une valeur boursière. Par exemple, la Figure 16 présente le code d'une macro permettant d'insérer un caractère spécifique, de zoomer l'affichage et de revenir à l'état précédent.

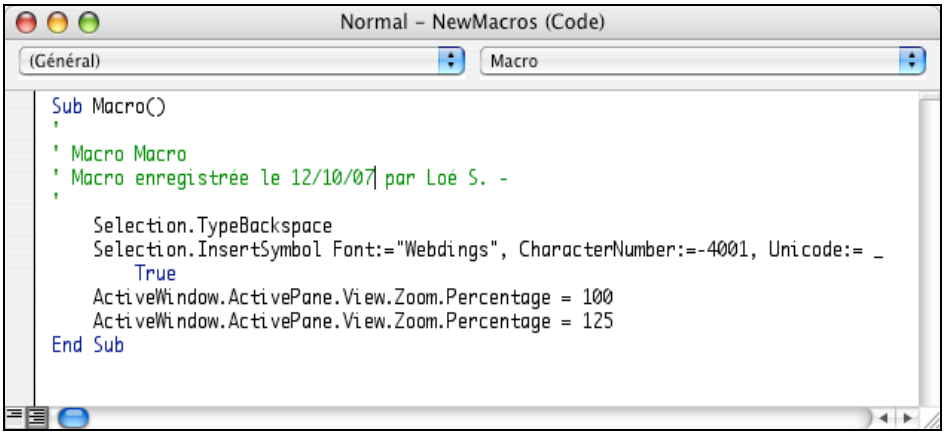

Figure 16 : Code d'une macro sous Word™ (Office Mac 04)

Le principal reproche que l'on peut faire à la technique basée sur ces enregistreurs de macros est qu'il permet de répéter une séquence de programme, mais seulement à l'identique. Son intérêt est alors très limité (Boshernitsan et al., 2007). De fait, beaucoup d'enregistreurs de macros permettent de reprendre textuellement le code généré pour l'enrichir, afin de créer de vrais programmes. Ils peuvent ainsi apporter une aide non négligeable dans la maîtrise des langages d'application, en permettant à l'utilisateur de partir d'un squelette construit à partir de l'enregistrement d'un exemple, pour y ajouter les notions de variables et des structures de contrôles qui généraliseront son programme. L'outil Automator[10] d'Apple offre un original mélange d'éditeur de macros et d'enregistreur de macros, permettant de construire de petits programmes permettant d'automatiser des tâches (Figure 17).

Certains problèmes demeurent néanmoins. Tout d'abord, ces langages sont fortement textuels. Ils demandent de comprendre une syntaxe généralement guidée par des principes de compilation. Mais surtout, ils ramènent en fait l'utilisateur non programmeur au problème de la maîtrise des principes généraux de programmation : même avec des concepts plus accessibles d'un outil Automator, il faut s'inscrire dans une démarche de programmation.

[9] http://www.cesoft.com/products/qkx.html

[10] http://www.apple.com/fr/macosx/features/300.html

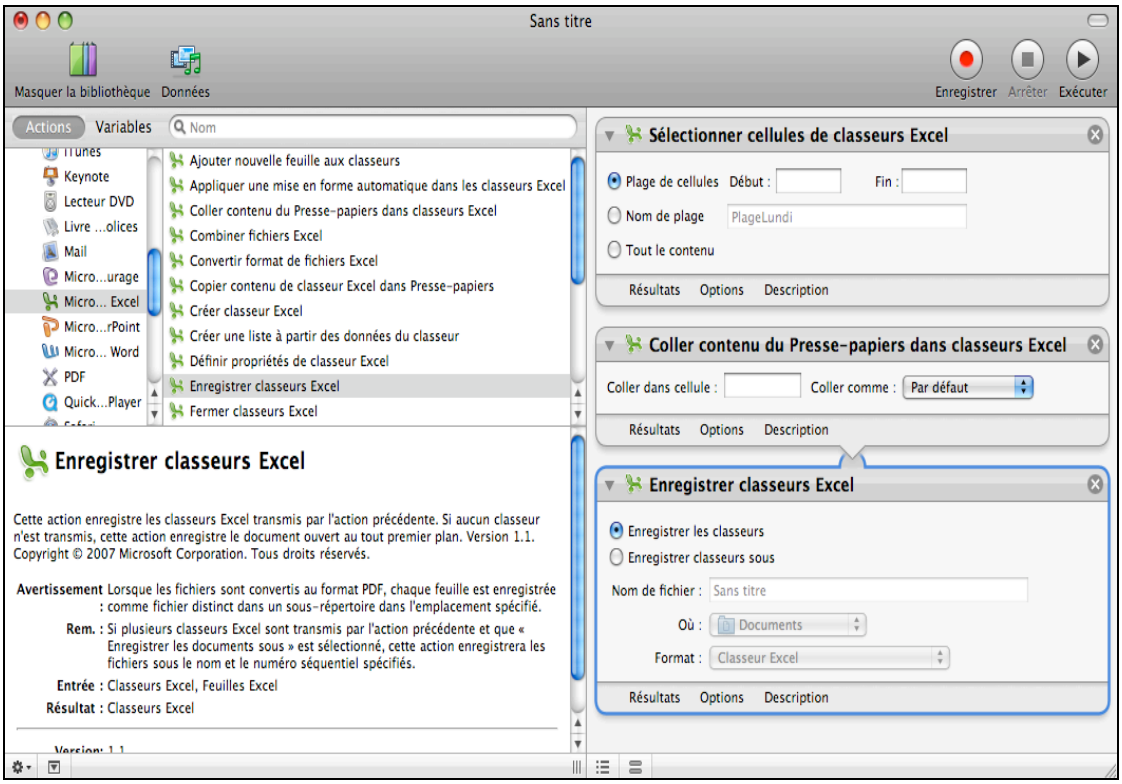

Figure 17 : Fenêtre principale de l'application « Automator »

2.1.4 Synthèse

Aucune des solutions présentées n'apportent une réponse complètement satisfaisante au problème de l'adaptation des applications aux besoins de leurs utilisateurs.

Les préférences, simples à mettre en place, ne répondent pas véritablement au problème. Elles ne peuvent répondre qu'à un nombre limité de situations prévues par le concepteur de l'application, et ne permettent aucune adaptation spécifique par l'utilisateur.

Les scripts sont réalisés à partir de langages d'application plus complexes. Ils apportent la puissance de la programmation, résolvant en cela la limitation des préférences. Mais ils nécessitent non seulement la connaissance d'un vocabulaire et d'une syntaxe, mais surtout la maîtrise des concepts de programmation telles que les structures de données (listes, tableaux, etc.) et les structures de contrôles (boucles, conditions, procédures, etc.). Même s'ils permettent d'étendre une application dans des proportions importantes, il n'en demeure pas moins que pour automatiser des tâches simples, il ne devrait pas être nécessaire de recourir à de tels langages.

Les enregistreurs de macros résolvent partiellement ce problème, mais demeurent limités par leurs principes : la séquence enregistrée est fondamentalement rejouée à l'identique. Le contexte à l'exécution n'est pas pris en compte. Pour modifier et donc étendre le fonctionnement d'une macro, il est nécessaire de passer par le langage de programmation de l'application, et donc la maîtrise par l'utilisateur des concepts de la programmation. D'un point de vue implémentation, ils possèdent également le même problème que les langages d'application puisque, là aussi il faut écrire un interpréteur et en plus un mécanisme pour enregistrer les commandes de l'utilisateur sous forme textuelle.

Comme nous l'avons vu dans le chapitre précédent, la PsE est en mesure d'apporter des réponses à ce problème de maîtrise des concepts de la programmation, en permettant à

l'utilisateur de s'appuyer sur l'exemple. Dans la section suivante, nous analysons l'apport de la PsE grâce aux deux exemples.

2.2 Exemples de solutions apportées par la PsE

Les techniques de la PsE trouvent leur utilité dans des agents d'aide ou autres assistants autour des années 1990. Utilisées de façon ponctuelle dans certaines situations d'interaction, les techniques de la PsE sont de nature à apporter de nouvelles fonctions d'assistance à l'utilisateur. Selon les besoins, des techniques explicites de généralisation seront préférables à l'utilisation de l'inférence, principalement lorsqu'on souhaitera que l'utilisateur puisse contrôler finement son automatisation. Elles sont la base de l'adaptabilité des applications. Nous décrivons dans cette partie deux outils qui peuvent servir d'exemple pour nos besoins coté utilisateur : SmallStar permettant d'éviter partiellement la programmation textuelle et Eager possédant un principe d'adaptabilité très élevé.

2.2.1 SmallStar

SmallStar (Halbert, 1993) est une simulation de l'interface graphique du système d'exploitation *Star* de Xerox, à laquelle des capacités de Programmation sur Exemple ont été rajoutées. Star (Smith et al., 1982) utilise la métaphore du bureau avec des icônes représentant les différents objets du système (programmes, imprimantes, documents, formulaires, boîtes aux lettres, tableaux etc.). Dans cet environnement, l'utilisateur peut éditer du texte et des graphiques, envoyer et recevoir des messages, remplir des formulaires, accéder à des bases de données et imprimer des documents. Star possède également un langage d'application appelé CUSP (*CUStomer Programming*) avec lequel l'utilisateur peut développer des applications sur mesure.

SmallStar est l'un des premiers systèmes de Programmation sur Exemple destiné au grand public, c'est-à-dire à des utilisateurs n'ayant pratiquement aucune connaissance de la programmation, dans le but d'automatiser les tâches répétitives souvent rencontrées lorsqu'ils utilisent le Star et ceci sans avoir recours au langage CUSP. Pour créer un programme l'utilisateur commence par ouvrir un bouton ou une icône programme puis presse le bouton « Start Recording » pour démarrer l'enregistrement (Figure 18). Il effectue ensuite les étapes nécessaires puis presse le bouton « Stop Recording » pour arrêter l'enregistrement. Une fois la phase d'enregistrement du programme terminée, l'utilisateur doit effectuer les généralisations nécessaires.

Pour chaque type d'objet présent dans le système, *SmallStar* propose un ensemble de descripteurs de données présentés à l'utilisateur dans une boîte de dialogue. *SmallStar* permet d'exécuter le programme pas à pas afin de suivre son fonctionnement. Dans le cas où une erreur se produit, *SmallStar* affiche un message d'erreur et indique l'instruction fautive dans la fenêtre représentant le programme. Au fur et à mesure que les commandes sont enregistrées, une description textuelle agrémentée d'icônes apparaît dans la fenêtre programme (partie supérieure de la Figure 18). Les icônes indiquent le type d'objet manipulé (dossier, document, sélection, tableau, etc.).

SmallStar n'effectue aucune inférence. Au lieu de cela, c'est l'utilisateur qui spécifie les structures de contrôles et les descripteurs de données. Une fois le programme enregistré, c'est l'utilisateur qui effectue les généralisations nécessaires. Une heuristique présente dans le système consiste à proposer par défaut les descripteurs de données les plus plausibles (ainsi pour un fichier, le nom est le facteur de choix par défaut). Lorsque *SmallStar* exécute un programme, il utilise les descripteurs de données pour trouver les objets dont le programme a

besoin. La première fois qu'un descripteur de données est rencontré, *SmallStar* cherche les objets correspondant à cette description. Si plus d'un objet correspond à la description, l'un d'eux est choisi arbitrairement.

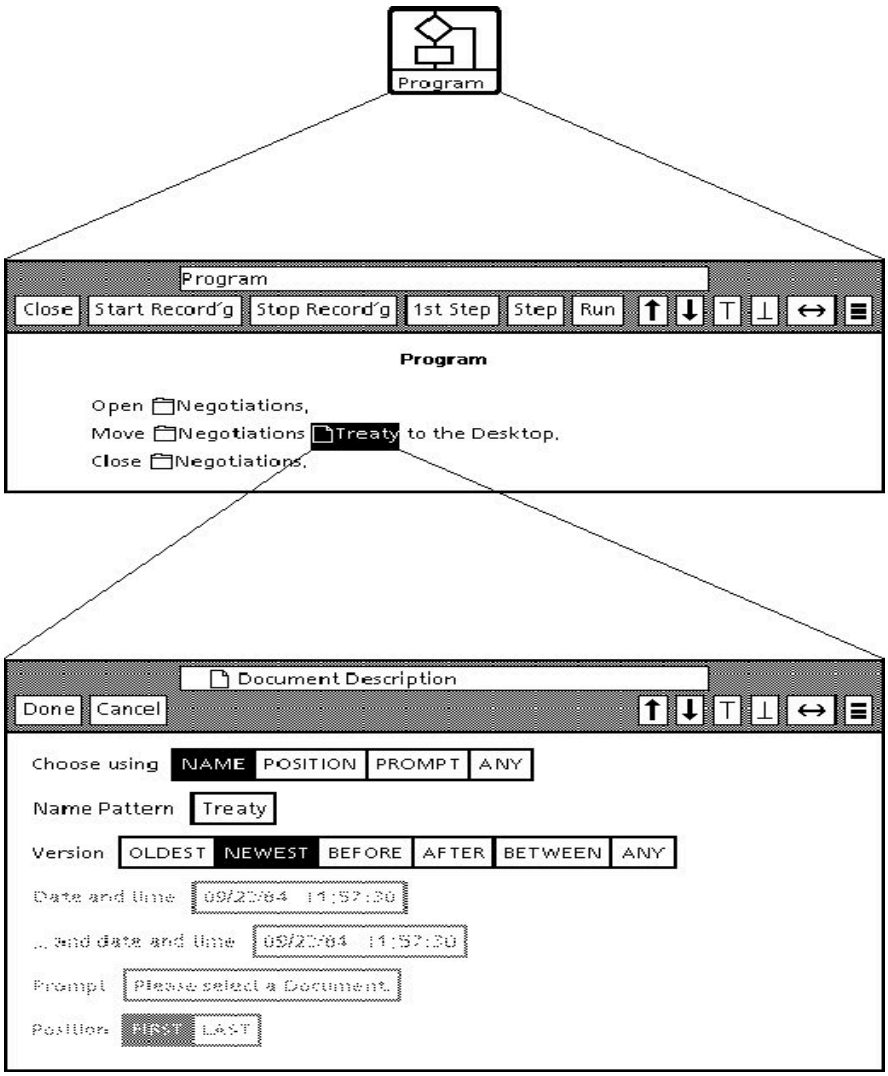

Figure 18 : SmallStar : une macro qui place la dernière version du fichier "Treaty" sur le bureau

Lorsque *SmallStar* rencontre à nouveau le même descripteur de données, il saute l'étape de recherche et utilise l'objet déjà trouvé. Ainsi, quand un même objet apparaît plus d'une fois dans un programme, toutes les références à cet objet utilisent le même descripteur de données ce qui constitue une heuristique importante. Il est cependant possible de changer manuellement les descripteurs de données pour chaque référence faite à un objet de façon à contrer cette heuristique. SmallStar enregistre les commandes sous forme de commandes haut niveau et non pas comme des opérations de la souris. Une commande possède un nom et un ensemble d'arguments. Une commande enregistrée peut impliquer plusieurs actions de la part de l'utilisateur.

2.2.2 Eager

Eager (Cypher, 1993a) automatise les tâches répétitives dans le cadre de l'environnement HyperCard sous la plate-forme Macintosh. C'est un système de PsE qui s'adresse aux utilisateurs n'ayant aucune connaissance préalable de la programmation. Il

prend en compte le fait que l'utilisateur ne réalise pas toujours immédiatement qu'il peut créer un programme pour accomplir la tâche qui l'intéresse. Très généralement, l'utilisateur commence sa tâche et au bout de quelques étapes s'aperçoit qu'il aurait pu l'automatiser. Dans la plupart des systèmes, dans une telle situation, l'utilisateur doit interrompre sa tâche, passer en mode enregistrement, effectuer les étapes nécessaires, stopper l'enregistrement, puis invoquer le programme ainsi créé. Ce changement de mode interrompt le flot de l'interaction et peut déranger le novice. Eager adopte une approche différente en observant en permanence le comportement de l'utilisateur pour essayer de prédire ses prochaines commandes. Il peut proposer de terminer une tâche sans même que l'utilisateur y ait pensé. Pour créer un programme avec *Eager*, l'utilisateur ne fait rien de particulier. C'est le système qui enregistre en permanence toutes les commandes de l'utilisateur dans l'espoir de trouver des répétitions et de les automatiser.

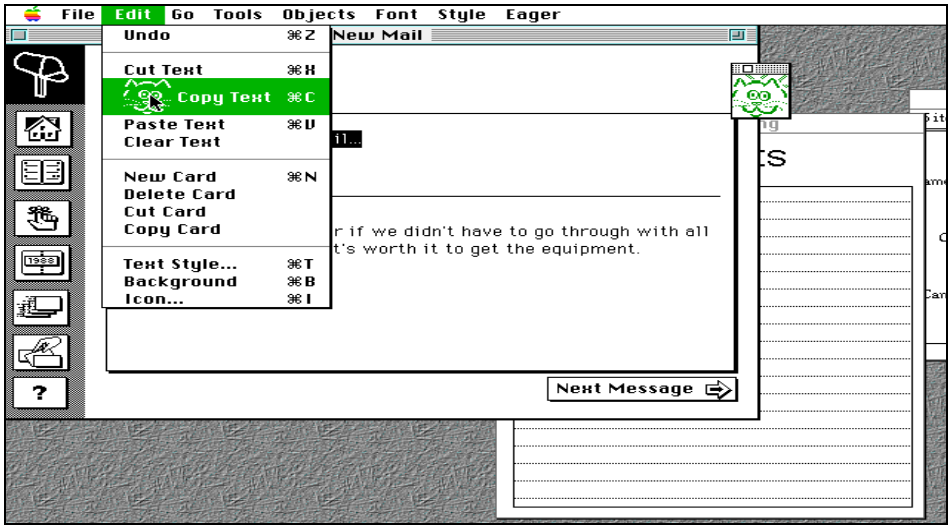

Figure 19 : Rendu spécifique à Eager

Lorsqu'il détecte une boucle, *Eager* anticipe la commande suivante de l'utilisateur sans l'exécuter. Pour cela, il sélectionne d'une couleur verte l'option dans le menu, affiche en vert le prochain texte (tel que ses mécanismes de prédiction le déterminent) et, présente une icône (tête de chat) dans l'option sélectionnée et dans la zone de texte (Figure 19).

L'utilisateur est alors invité à confirmer la proposition d'*Eager*. Si l'inférence est incorrecte, l'utilisateur doit effectuer la commande correcte, cette dernière étant alors considérée par *Eager* comme un contre-exemple et le programme est révisé en conséquence. Lorsque la séquence complète est validée, l'utilisateur est invité à confirmer la fin de la séquence d'apprentissage (Figure 20), ce qui permet à *Eager* de finaliser la tâche. Il a le choix entre l'exécution d'un seul coup, ou de faire du pas à pas. Avant toute exécution, *Eager* sauvegarde la pile HyperCard courante en prévision de cas d'erreurs pour ne pas faire perdre à l'utilisateur tout son travail. Le programme généré est un script Hypertalk que l'utilisateur a la possibilité de visionner dans un éditeur s'il le désire. Une autre forme de visualisation peut être l'anticipation des commandes utilisateur.

Écrit en LISP puis porté en C++ pour des problèmes d'efficacité, *Eager* est une application tournant en tâche de fond et reçoit des commandes de haut niveau. Sa tâche principale est la détection de boucles dans l'historique de l'utilisateur : à chaque nouvelle commande reçue, *Eager* recherche dans l'historique une commande similaire. Deux commandes sont considérées similaires si elles sont de même type (par exemple «Copier/Coller texte») et si les données sur lesquelles elles portent partagent des traits

communs (par exemple les jours de la semaine, premier mot de la ligne i et premier mot de la ligne i+1). La détection de traits communs se fait par la recherche de constantes, d'entiers consécutifs et de suites arithmétiques avec tolérance pour les données numériques. La décomposition de chaînes de caractères en sous-chaînes en se basant sur les séquences rencontrées constitue la technique de détection de trait commun pour les données textuelles. Il recherche les constantes, l'ordre numérique ou alphabétique, le passage de majuscule en minuscule, les séquences connues comme les jours de la semaine, les mois et les chiffres romains.

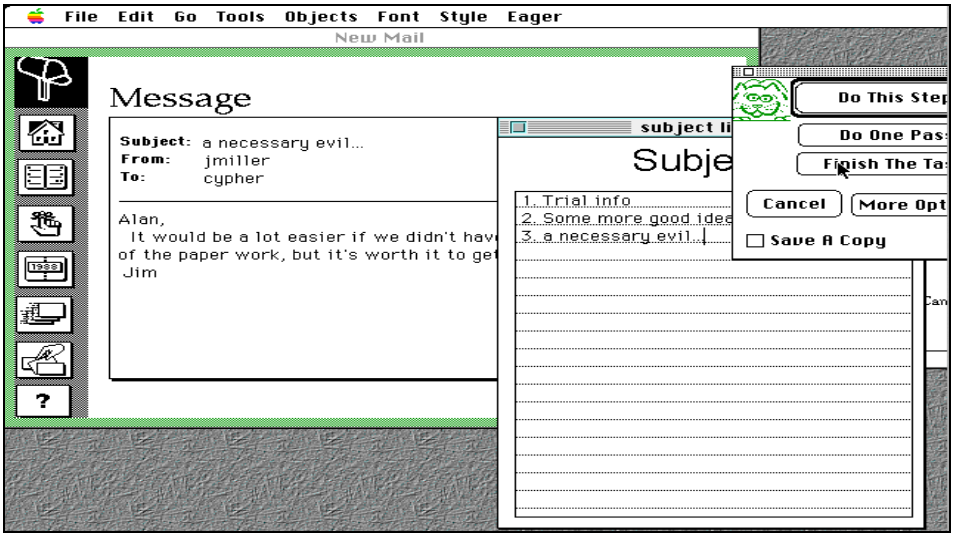

Figure 20 : Confirmation de fin de tâche dans le menu Eager

Pour l'élaboration de l'historique, des nombres apparaissent dans les numéros de cartes, les coordonnées des objets, la position du mot dans une ligne, etc. On rencontre des informations textuelles dans les passages de texte sélectionnés, les chemins d'accès aux fichiers, le nom des cartes, des boutons, des champs. Il y a une mise en correspondance du nouveau texte avec l'ancien lors de l'édition du texte, et aussi avec le nouveau texte de l'itération précédente. Lorsque deux commandes sont jugées similaires et qu'*Eager* n'a pas encore trouvé de boucle, il suppose qu'une boucle commence. Si les commandes suivantes de l'utilisateur confirment bien qu'il y a une boucle, c'est-à-dire qu'au moins deux itérations ont été effectuées, *Eager* anticipe ainsi les prochaines commandes en se basant sur le programme qu'il a généralisé. Lorsqu'un seuil de confiance est dépassé, au moins trois itérations ont été effectuées, Eager affiche l'icône particulière que l'utilisateur peut sélectionner afin de laisser le soin à *Eager* de finir sa tâche.

Eager possède un seuil de confiance qui est égal au moins à trois itérations. S'il peut détecter des répétitions dans la trace de l'utilisateur, c'est parce qu'il manipule des commandes de haut niveau et possède de plus une connaissance du domaine abordé. En effet, les cartes font partie des piles. Aussi, l'abscisse d'un objet est de type 'coordonnées d'écran'. Une tolérance de 4 pixels est accordée lors de la recherche de similarité pour tenir compte de l'inexactitude de l'utilisateur. De plus, chaque commande est enregistrée avec de l'information contextuelle. Les séquences de commandes de l'utilisateur sont regroupées pour former des commandes de haut niveau : ce sont ces commandes qui sont enregistrées puis généralisées. Cette méthode permet de tenir compte des fluctuations dans les actions de l'utilisateur. À une commande donnée peut en effet correspondre plusieurs séquences d'actions. D'autre part, le fait qu'*Eager* anticipe les commandes encourage l'utilisateur à effectuer les étapes d'une boucle de manière consistante.

Eager n'infère pas de conditions, ni de boucles imbriquées, ni de répétitions temporelles. Il ne permet pas la sauvegarde du programme créé, ce qui exclut la possibilité d'appel de procédure. L'utilisateur ne peut pas créer de programme si celui-ci ne contient pas de boucles.

2.2.3 Apport de la PsE à l'adaptabilité des applications

SmallStar et *Eager* apportent des réponses partielles au problème de l'adaptabilité des applications. Ces deux approches permettent de réaliser ce que Myers nomme « Just in time programming ». Au moment où l'utilisateur en a besoin, le système lui offre la possibilité de transformer un exemple de tâche en une exécution automatique. Dans les deux cas, on s'appuie sur les actions de l'utilisateur pour interpréter sa volonté et lui permettre d'automatiser sa tâche.

Les mécanismes sous-jacents ne relèvent pas seulement d'un espionnage des interactions élémentaires de l'utilisateur. Ils permettent aussi d'identifier les actions sémantiques sur l'application, afin de les interpréter et de les reproduire en appliquant certains schémas de généralisation.

L'aide de l'utilisateur est demandée, soit sous une forme passive (cas de *SmallStar* qui requiert l'édition du script par l'utilisateur), soit sous une forme active (cas d'*Eager*, qui intervient d'autorité dans le dialogue pour proposer son aide). Dans les deux cas, le système doit être en mesure de contrôler le flux d'exécution de l'application interactive, en alternant les phases de rejeu et les phases sous contrôle de l'utilisateur.

Les mécanismes proposés s'attachent autant que possible à éviter que l'utilisateur ne soit obligé de maîtriser les concepts de la programmation. Ainsi, l'itération de collection proposée par *SmallStar* est-elle un concept naturel pour l'utilisateur dont la volonté sera le plus souvent de répéter une même séquence d'actions sur un ensemble (la collection) d'objets. De même, l'intervention d'*Eager* pour proposer d'aider l'utilisateur à terminer une tâche répétitive est-elle tout aussi naturelle.

2.3 Synthèse

Notre objectif consiste à fournir aux développeurs d'applications un outil permettant d'implémenter simplement des fonctionnalités d'adaptation de ses applications par l'utilisateur. Les techniques classiques de préférences ou de macros en langage d'application ne sont ni satisfaisantes, ni généralisables. Les enregistreurs de macros, en raison de leur appui sur les actions de l'utilisateur, sont un premier pas intéressant.

Les techniques de PsE ont montré leur adéquation à ce problème, en permettant d'offrir à l'utilisateur des mécanismes pertinents pour généraliser les programmes produits sans avoir besoin de maîtriser les concepts et techniques de la programmation.

Malheureusement, à ce jour, aucun système ou bibliothèque ne permet d'aider un développeur à implémenter ces techniques au sein de ses applications. Un tel outil devrait bien sûr permettre de mettre en place facilement une technique d'enregistrement/rejeu des actions de l'utilisateur. Au-delà, il devrait laisser beaucoup de latitude au programmeur pour décider quelles fonctionnalités il souhaite réellement implémenter dans son application. Deux approches ont tenté d'apporter une réponse à ces questions. Nous les présentons dans la section suivante.

3 Outils pour la PsE

La PsE consiste fondamentalement en l'enregistrement d'un exemple pour permettre de le généraliser, puis de le rejouer. La quasi-totalité des systèmes présentés au chapitre 1 ne s'attardent pas sur les aspects enregistrement/rejeu, et se concentrent sur les aspects liés à la généralisation. En effet, l'interaction de l'utilisateur, gage de ses objectifs dans la réalisation d'une tâche semble ne pas être mise en avant. Afin de pouvoir généraliser ou même simplement analyser les actions de l'utilisateur, la connaissance de ces dernières est obligatoire. Il s'agit de l'enregistrement. De même, la généralisation porte sur l'action initiale afin de l'adapter à une nouvelle exécution. Cette exécution de l'action a pour origine l'action espionnée. Ainsi, le rejeu ne peut avoir lieu sans enregistrement. La généralisation, mise en avant par les systèmes précédemment présentés, ne peut se faire que si l'enregistrement (ou l'espionnage) est effectué. L'enregistrement/rejeu des actions de l'utilisateur, de par son côté évident, est souvent oublié dans la description des systèmes, alors que tout en dépend.

Dans une application interactive, il convient donc de s'intéresser aux interactions entre l'utilisateur et l'application. L'utilisateur agit sur l'application à travers l'interface. C'est son seul point d'accès aux fonctions du noyau fonctionnel. Nous décrivons plus en détail l'interface utilisateur dans la sous-section 2.2.1 du chapitre 3. L'ensemble des interactions de l'utilisateur passe par l'interface, qui en retour réagit en fonction des fonctionnalités de ses composants. Suivre les interactions de l'utilisateur sans perturber le fonctionnement du système revient à écouter les réactions de l'interface (parmi d'autres possibilités). C'est ainsi que l'on espionne l'interface utilisateur afin d'appréhender les différentes interactions de l'utilisateur. Cependant, il n'est pas aisé d'accéder au noyau fonctionnel de l'application pour appréhender les activités de ce dernier par rapport aux interactions de l'utilisateur. Le niveau sémantique est difficile d'accès et constitue l'une des principales difficultés de la PsE

Bon nombre d'outils ont tenté d'apporter une aide à la programmation utilisateur, mais rares sont ceux qui ont tenté d'apporter une aide à l'intégration de ces techniques dans une application, c'est-à-dire des outils permettant au programmeur d'incorporer des techniques de la programmation dans les applications lors de leur réalisation. Ces *outils de la PsE* ont pour objectif de faciliter la création d'application de PsE ou incluant la PsE. Ils mettent à la disposition de l'utilisateur les moyens nécessaires pour développer à moindre coût ce type d'applications. Deux systèmes seulement ont tenté d'apporter une réponse à ce problème, les systèmes PbDScript et AIDE. Ces deux systèmes abordent techniquement le problème sous deux angles différents. Le premier utilise une approche externe tandis que le second se fonde sur une approche interne. Nous développons ces deux outils dans les sections suivantes.

3.1 PbDScript

PbDScript (Depaulis et al., 2003) est une application autonome permettant à tout utilisateur de construire des programmes, sous forme de scripts, pour une application cible n'ayant pas été conçue pour cela. Elle se base sur le principe de l'espionnage des événements produits sur les *widgets* de l'application. Développée en Java, elle exploite les possibilités d'introspection des boîtes à outil AWT (Abstract Window Toolkit) et Swing. PbDScript se présente sous la forme d'une application indépendante écrite en Java, qu'il convient de lancer en parallèle de l'application que l'on souhaite programmer. L'utilisation de PbDScript se décompose en deux étapes : l'apprentissage de l'application cible et la création/utilisation de scripts.

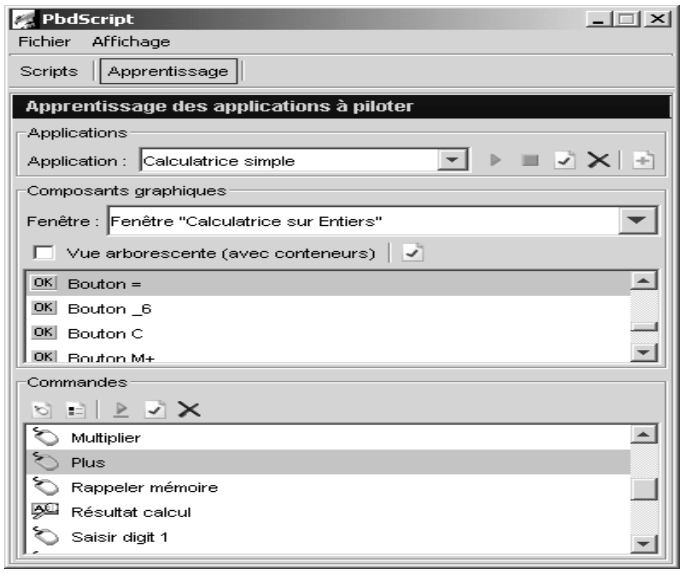

Figure 21 : Apprentissage des applications cibles dans PbDScript

La phase d'apprentissage de l'application cible (Figure 21) consiste en la définition du lien sémantique entre les actions sur l'interface et les fonctions réelles du logiciel. Cette étape, indispensable à une programmation efficace, doit être effectuée par l'utilisateur. La phase de création de scripts peut se faire par enregistrement d'un exemple « joué » sur l'application cible, ou la création d'un script textuel défini dans l'interface de PbDScript. Sous sa première forme, elle s'apparente à celle de Topaz (Cypher, 1993c). Topaz est un système qui permet de créer interactivement des programmes paramétrés, sur un ensemble d'applications graphiques interactives. Ce système est capable d'espionner les actions de l'utilisateur sur des applications cibles programmées avec la boîte à outils Amulet (Myers et al., 1997). Dans Topaz, après une phase d'enregistrement de macros, l'utilisateur généralise l'exemple en éditant un code simplifié. L'apport de Topaz se situe au niveau de la sélection des objets graphiques entrant en jeu dans un programme. Outre son caractère beaucoup plus général, le grand avantage de PbDScript réside dans sa capacité à décrire les scripts dans un langage naturel proche de l'utilisateur et à offrir une large palette de mécanismes de généralisation. PbDScript dispose d'un éditeur graphique de scripts (Figure 22).

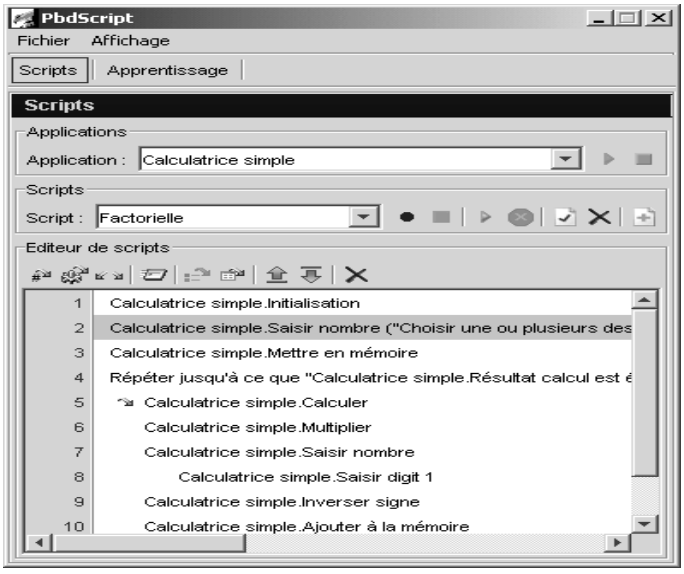

Figure 22 : Édition des scripts sous PbDScript

La mise en œuvre de ces deux étapes se fait par l'introduction de trois phases :

- identifier l'ensemble des composants graphiques (widgets) de l'application cible ;
- donner la possibilité à l'utilisateur de nommer les interactions de base ;
- permettre à l'utilisateur de généraliser des scripts en insérant des paramètres et structures de contrôle.

PbDScript part du fait que toute application Java est représentée par des widgets conteneurs qui peuvent eux-mêmes contenir d'autres widgets. Il récupère l'arbre des widgets de présentation de l'application par un mécanisme d'introspection puis l'affiche (Figure 23). PbDScript ne s'applique qu'aux applications Java reposant sur la boîte à outils Swing et AWT, mais ses principes peuvent être facilement appliqués à d'autres boîtes à outils.

La représentation des *widgets* dans l'application et leur représentation dans PbDScript sont liées dynamiquement de sorte que si l'utilisateur agit sur le *widget* dans l'application, celui-ci apparaît en surbrillance dans PbDScript. Inversement lorsqu'un élément de la hiérarchie est désigné dans PbDScript, le *widget* correspondant est mis en évidence dans l'application.

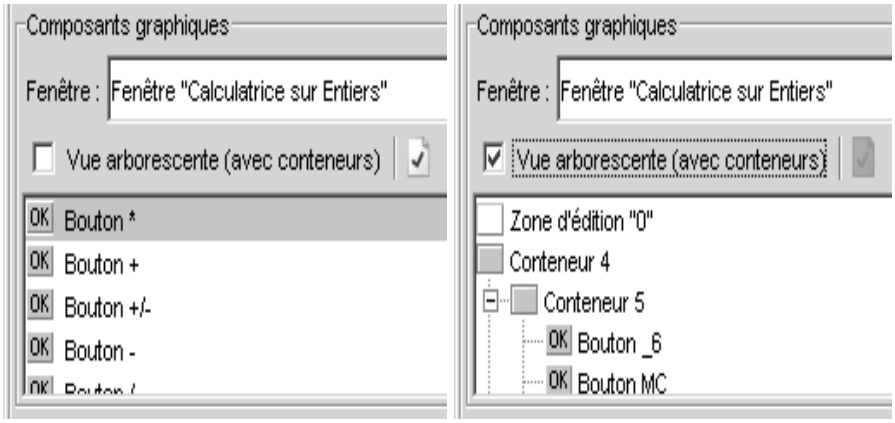

Figure 23 : Affichage de la hiérarchie des widgets dans PbDScript

PbDScript s'appuie sur certains principes qui gouvernent généralement les applications interactives classiques, de type WIMP[11] :

- une série d'interactions correspond à une tâche utilisateur ;
- une tâche utilisateur non-articulatoire correspond à une action du noyau fonctionnel ;
- le noyau fonctionnel est réfléchi dans les *widgets*.

PbDScript demande à l'utilisateur d'ajouter de l'information aux *widgets* (partie statique) et aux événements provenant de ces *widgets* (partie dynamique). L'apport de la sémantique, basé sur les trois principes précédents consiste à proposer à l'utilisateur d'associer une phrase à un couple (*widget*, action). L'utilisateur crée ainsi un dictionnaire de commandes associé à son application. Le terme action représente ici aussi bien une interaction de l'utilisateur sur le *widget* (clic, frappe au clavier, etc.) dans l'application cible qu'une fonction de récupération d'un attribut de présentation. L'utilisateur doit traduire ainsi la correspondance entre l'attribut de présentation et l'état du noyau fonctionnel. Cette nomination faite par l'utilisateur lui permet l'écriture du script en manipulant l'application, et

[11] Windows, Icons, Menus and Pointers

de le visualiser dans ses propres termes. Le formalisme utilisé repose alors sur un langage d'interaction plus que sur un langage de programmation.

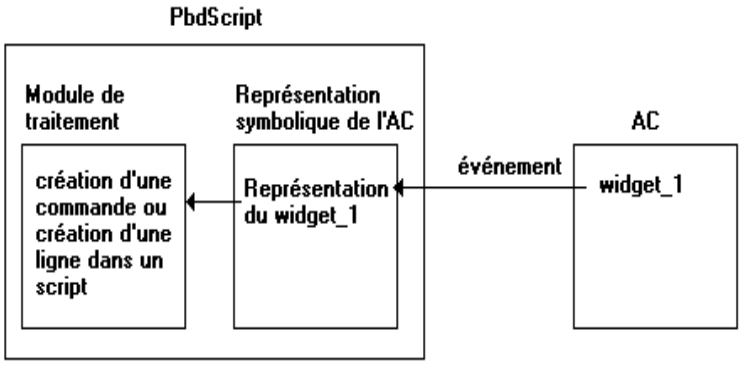

Figure 24 : Gestion d'événements émis par l'application cible (AC)

Le principe d'espionnage et de pilotage des applications cibles repose sur la gestion des événements. En mode enregistrement, PbDScript intercepte les événements émis par l'application cible (Figure 24).

Un événement est émis au sein de l'application cible par un *widget* ; cet événement est intercepté par l'objet représentant le *widget* dans PbDScript qui le transmet au module de traitement. Cet événement se traduit soit par la création d'une nouvelle commande, soit par la création d'une ligne dans un script (en mode d'enregistrement d'un script). En mode rejeu de script, PbDScript émet des événements en direction de l'application cible. La ligne de script en cours d'exécution appelle la commande créée lors de la phase d'apprentissage qui elle-même appelle une méthode associée à l'objet représentant le *widget* dans PbDScript qui finalement envoie au *widget* de l'application cible l'événement correspondant.

PbDScript fait un pas vers les structures de contrôle même si dans sa représentation de langage d'interaction, les paramètres sont implicites. L'apport de la sémantique utilisateur est cependant considérable. C'est l'utilisateur qui non seulement enseigne le système par ses actions de façon volontaire, mais c'est également lui qui définit l'ensemble des interactions en établissant une correspondance entre les événements enregistrés et les actions à effectuer. PbDScript ne permet pas la réutilisation d'un script défini par l'utilisateur par un autre script. Il ne permet donc pas les appels de procédures simples ou récursives. L'apport de la sémantique par l'utilisateur constitue un point majeur dans la solution PbDScript.

3.2 AIDE

Le système AIDE (Piernot and Yvon, 1993) vise à faciliter la création de systèmes de PsE en fournissant une couche logicielle écrite en SmallTalk sur laquelle des développeurs peuvent bâtir, avec le minimum d'effort, des applications mettant en œuvre des techniques démonstrationnelles. AIDE est un squelette d'application dont les principes de conception sont l'extensibilité du squelette, la représentation de l'historique et des commandes, et un mécanisme de récupération d'erreur. À partir de ces principes, l'architecture d'AIDE s'articule autour des commandes de haut niveau, souvent obtenues à partir de commandes de bas niveau, mettant en œuvre la notion d'agrégats (Kosbie and Myers, 1994) et d'un gestionnaire de commandes gérant l'historique et la création de macros (Figure 25).

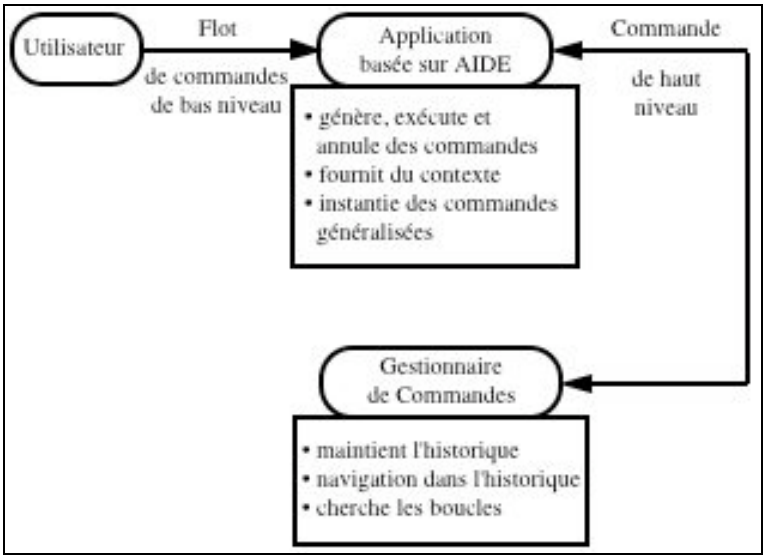

Figure 25 : Architecture de AIDE

La notion d'agrégation de commandes proposée par Kosbie (une commande est composée d'autres commandes) conduit à une représentation arborescente de l'historique (Figure 26). Cette technique a été utilisée (en l'améliorant par l'ajout d'une construction incrémentale d'arbre) pour construire l'historique des commandes.

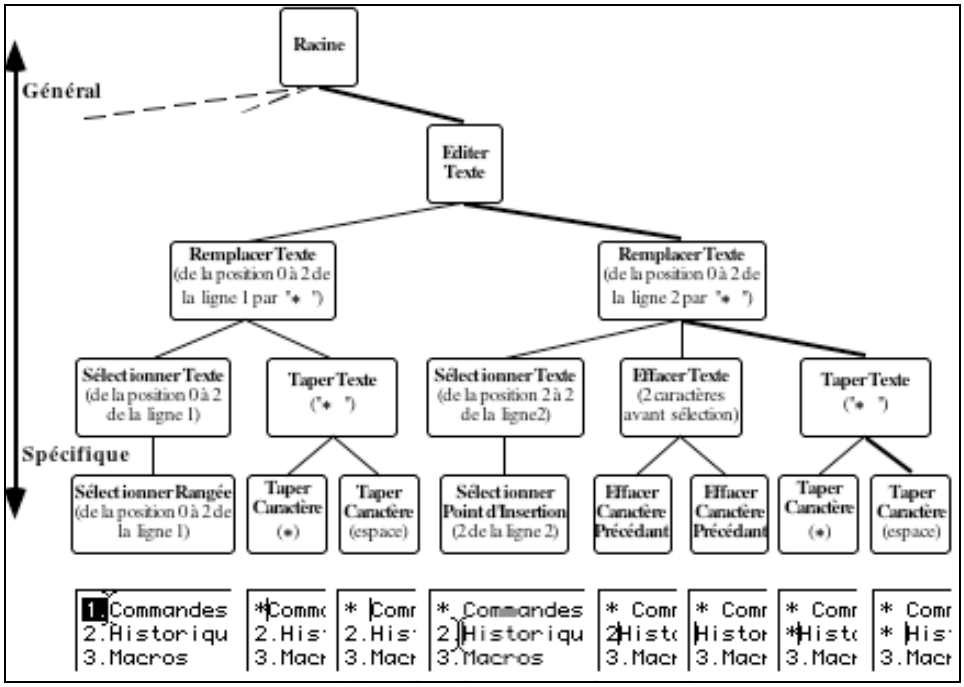

Figure 26 : Arbre de commandes pour une tâche de « Remplacer les numéros par des astérisques » dans une liste

La gestion de commandes d'AIDE est similaire au gestionnaire d'AppleEvent duquel il a tiré ses bases de construction et ses principes de fonctionnement. Son mécanisme de récupération d'erreur est couplé à celui de l'historique (comme dans AppleEvent). Dans AIDE, le gestionnaire de commandes est chargé de la création d'un programme à partir de l'historique des commandes utilisateur alors que dans le modèle AppleEvent, il a pour

fonction d'assister les applications à résoudre les spécificateurs d'objets complexes (similaires aux descripteurs de données de SmallStar (Halbert, 1993)).

Le dialogue entre une application et le gestionnaire de commande se fait par échange de commandes et diffère selon le mode enregistrement ou le mode exécution (Figure 27 et Figure 28). Il est possible de naviguer dans l'historique, d'exécuter et d'annuler des commandes entre applications grâce à une interface fournie par le gestionnaire de commandes d'AIDE.

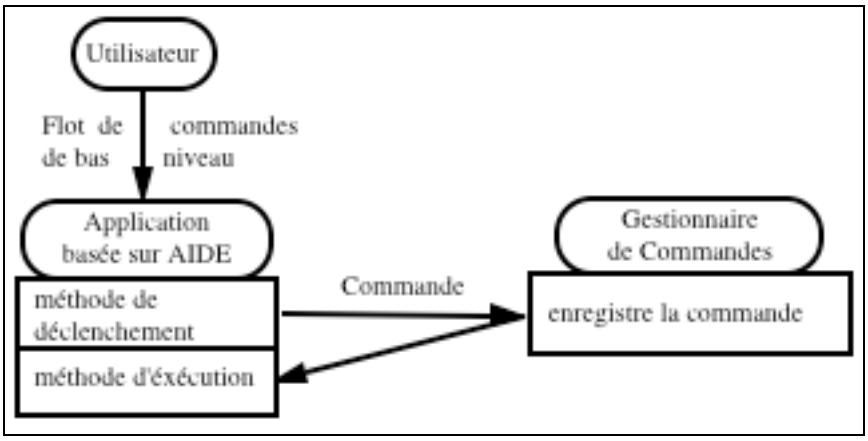

Figure 27 : Dialogue entre l'Application et AIDE lors de l'exécution d'une commande

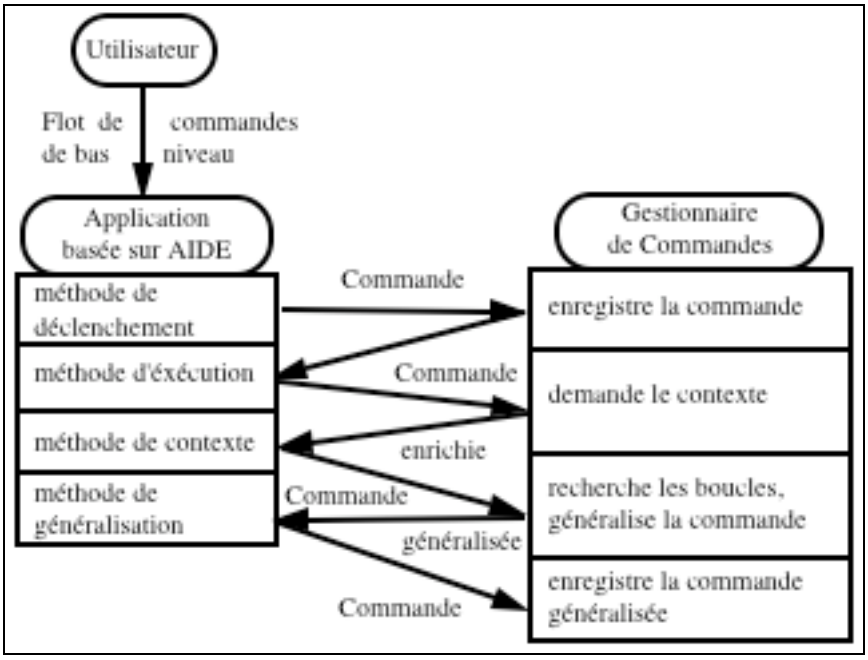

Figure 28 : Dialogue entre l'application et AIDE en mode enregistrement de macros

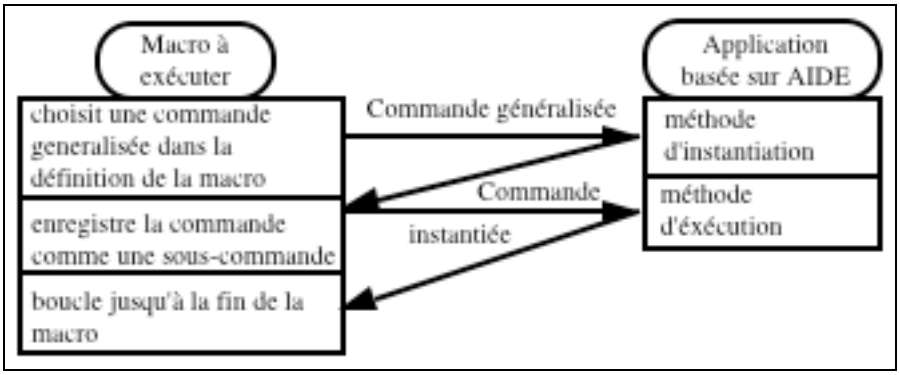

Figure 29 : Dialogue entre l'application et AIDE en mode exécution de macros

Le processus de généralisation s'effectue par la détection de boucles non imbriquées (par correspondance entre commandes comme Eager), à l'abstraction des arguments (liée à la détection de séquences) et la détection de séquences (reconnaissance de séquence numérique, de chaîne de caractères, des séquences de points et de classes tout comme *Eager*) (Figure 28 et Figure 29). Bien qu'AIDE soit d'un apport considérable, il présente quelques difficultés. En effet, pour arriver à un effet de détection de boucles imbriquées, l'utilisateur doit d'abord créer une macro accomplissant la première boucle et ensuite une seconde macro appelant la première macro à l'intérieur d'une boucle.

Les macros sont des procédures définies par l'utilisateur opérant sur une collection ordonnée d'arguments et éventuellement retournant une valeur. L'utilisateur les définit au moyen d'un nom et d'une liste d'arguments. En fait les macros sont implantées par une classe Macro qui est elle-même une sous-classe de la classe Commande. Pour enseigner une macro au système, l'utilisateur sélectionne les objets qui formeront ses arguments, choisit ensuite *« Enregistrer Nouvelle Macro »* dans le menu « *Macros* », entre le nom de la macro, démontre les étapes de la macro en effectuant les commandes appropriées et enfin sélectionne l'option « *Stopper Enregistrement* » dans le menu « *Macros* ». Les objets sélectionnés à la fin de la macro constituent la valeur retournée par la macro. Une fois la macro définie, elle apparaît dans le menu « *Macros* » et peut être invoquée par l'utilisateur à tout moment comme n'importe quelle autre commande de l'application.

Il faut noter que le concepteur d'une application basée sur AIDE doit décrire l'ensemble des commandes du noyau fonctionnel. Les commandes (de haut niveau) forment le cœur du système AIDE car elles fournissent une description sémantiquement riche des actions effectuées par l'utilisateur. Cette description des commandes demande un lourd investissement au niveau du développement de l'application. Ainsi, pour chaque commande, pour définir la grammaire des commandes, le développeur décrit sous quelles conditions cette dernière doit être générée. De telles conditions incluent une commande déclenchante, une commande intermédiaire et une commande terminale (Kosbie and Myers, 1994). C'est au développeur de structurer la hiérarchie des classes représentant les commandes.

AIDE présente aux développeurs des structurations d'applications mettant en œuvre la PsE. Mais, il n'est pas un outil permettant d'ajouter facilement des capacités de PsE à une application. Le modèle du système repose sur l'utilisation des AppleEvents qui n'existent que dans l'environnement des Macintosh rendant ainsi AIDE non exploitable sur les autres plates-formes. Enfin, SmallTalk est loin d'être largement utilisé.

3.3 Limites des outils existants

En plus des difficultés et limites citées lors de l'analyse, ces deux systèmes présentent d'autres limites. Dans le cas de PBDScript, il s'agit d'un outil très basique, déléguant toute la sémantique à l'utilisateur. Dans le cas de AIDE, on fournit à l'utilisateur un environnement prédéfini (genre AppleScript) qui demande un travail important au programmeur pour un résultat trop limité.

De plus, AIDE et PbDScript utilisent des connaissances implicites ayant trait au domaine d'application. Ils utilisent une représentation graphique des commandes même s'il est difficile de représenter la généralisation car cette représentation est étroitement liée à l'interface du système ainsi qu'à la nature de la commande. Que ce soit AIDE ou PbDScript, l'apport de l'utilisateur est nécessaire. La tâche de définition du noyau fonctionnel revient à l'utilisateur, ce qui est une très lourde contribution de sa part. PbDScript offre une large palette de généralisations possibles qui non seulement ne satisfont pas à tous les besoins utilisateur, mais rendent difficile la configuration du système.

AIDE impose la définition de commandes, qui seules peuvent être enregistrées. Il se révèle insuffisant dans son approche, car le rejeu est loin de satisfaire tous les besoins d'assistance ou d'automatisation de tâches d'un utilisateur. PbDScript ne permet pas à un concepteur d'introduire des fonctionnalités de PsE dans l'application qu'il réalise. L'utilisation de l'abonnement, qui a servi de base à la réalisation de PbDScript, est faite pour satisfaire le besoin d'espionnage des commandes explicites.

La solution des hiérarchies d'action préconisée par AIDE impose une architecture d'application particulière. Cette architecture ne semble pas appropriée à plusieurs niveaux de la réalisation d'une application simple de programmation sur exemple.

Enfin, dans les deux systèmes, le travail du programmeur d'application est enorme lors de la gestion des actions implicites. Il n'est pas non plus facile au concepteur de contrôler finement le rendu des widgets pour ajouter les primitives nécessaires au rejeu.

4 Cahier des charges pour un outil

Le besoin majeur de la programmation sur exemple est l'intégration de système d'enregistrement, de généralisation, et de rejeu des actions de l'utilisateur (Sanou et al., 2006c). L'un de nos objectifs est de rendre adaptable ce système à toute application en s'appuyant ou non sur l'interface même de l'application. En fonction des différents besoins liés à la programmation sur exemple, il faudra fournir des composants ou outils permettant de faciliter la conception de programmes surtout en diminuant le niveau d'abstraction exigé au programmeur.

Les deux exemples que nous avons détaillés permettent de mettre en avant de nombreux points importants pour construire le cahier des charges de l'outil que nous souhaitons définir.

AIDE donne une bonne indication de ce qui doit être fourni en tant que fonctionnalités de haut-niveau : gestion de l'historique, récupération du contexte, généralisation. Mais à l'inverse, il faut pouvoir fournir ces services pour l'utilisateur (paramétrage externe) comme pour le programmeur (intégration de ces fonctionnalités à l'intérieur de l'application).

PbDScript démontre que beaucoup de choses peuvent être faites en l'absence de l'intervention du programmeur de l'application support. Ceci est très utile dans notre

contexte, où l'on veut avoir un minimum d'efforts à faire de la part du programmeur, surtout pour éviter oublis et erreurs de programmation. PbDScript intègre une représentation spécifique des *widgets* des systèmes. Il établit une connexion avec l'application cible par le biais de la gestion des événements. Cela se fait par un apport sémantique important de la part de l'utilisateur pour l'identification des actions du noyau fonctionnel. C'est la phase d'apprentissage où les correspondances seront les commandes dans PbDScript. Il offre différentes possibilités pour la création des commandes.

4.1 Répartition des rôles entre le programmeur et l'utilisateur

Notre travail s'oriente principalement vers l'adaptation des applications par l'utilisateur, ce qui constitue le but principal de beaucoup de systèmes de PsE. L'objectif est d'assister l'utilisateur dans sa tâche courante. Il consiste généralement à automatiser une tâche quelconque faisant intervenir l'utilisateur à plusieurs reprises avec les mêmes actions considérées comme répétitives. Bien que l'utilisateur ne soit généralement pas capable de programmer avec la même aisance qu'un programmeur (Smith et al., 2001), il peut employer des solutions qui rendent la programmation plus accessible. La mise en place de ces solutions relève du rôle du programmeur.

Dans l'application de la PsE, le rôle de l'utilisateur peut se résumer en trois parties :

- automatiser des tâches répétitives pour gagner en temps et diminuer les risques d'erreurs (la vocation première de la programmation est l'automatisation de tâches répétitives) ;
- intégrer des applications pour développer une solution sur mesure (un programme peut accomplir des tâches qui mettent en œuvre plusieurs applications, fournissant ainsi le moyen de créer des solutions sur mesure) ;
- adapter une application à ses besoins spécifiques (un programme peut également être utilisé pour modifier ou ajouter des fonctions à une application existante).

La programmation est une tâche difficile, car non seulement il faut maîtriser une quantité de détails sur les langages de programmation, mais aussi être capable de créer des plans abstraits (structuration algorithmique par exemple). L'art de la programmation consiste à réduire la masse de détails à connaître et l'effort mental à mobiliser pour créer un programme.

4.2 Services à fournir

Nous avons identifié les besoins directement liés à la PsE par l'analyse des systèmes en guise de systèmes exemples. Nous avons aussi orienté le travail vers l'assistance au travers d'analyses de systèmes types de la PsE. L'enregistrement et le rejeu se sont avérés comme les opérations de base des techniques de la PsE. Bien que l'application Eager soit un exemple typique de l'assistance, elle présente l'enregistrement sous une autre forme : l'espionnage. L'espionnage est considéré, du côté utilisateur, comme une forme passive de l'enregistrement des actions utilisateur par le système. C'est aussi un résultat des analyses faites sur les systèmes de PsE vus dans les chapitres précédents. En plus des opérations de base, les mécanismes utilisateur de la PsE sont théoriquement illustrés ainsi que la capacité de généralisation des interactions. Nous avons pris en compte les objectifs visés par les différents systèmes et les techniques utilisées pour leur conception. Mais l'interaction de l'utilisateur reste un problème non négligeable dans la réalisation ou du moins la mise en œuvre de la PsE. Que devons nous réaliser pour faciliter l'utilisation d'une application interactive incluant la PsE ?

4.2.1 Enregistrement

L'enregistrement des actions utilisateur est la récupération des événements matériels (tels que la frappe d'une touche ou le déplacement d'une souris) et des événements systèmes (comme la création d'une fenêtre, son apparition à l'écran, etc.). L'enregistrement s'appuie sur la définition *a priori* de commandes. En réalité, il s'agit d'espionner les événements du type du clic sur un bouton ou de touche clavier. Les événements de type clic sur un bouton doivent être explicitement programmés par le développeur de l'application. Mais dans les autres cas, on doit tenir compte de deux possibilités : l'utilisateur peut utiliser les menus, ou bien les raccourcis claviers qui sont généralement implantés par défaut dans les composants d'interface utilisateur évolués, sans intervention explicite du programmeur.

Se restreindre aux actions de type « commande » n'est cependant pas suffisant pour permettre d'espionner en totalité l'utilisateur. L'utilisateur peut effectuer des actions de frappe au clavier à l'intérieur d'un éditeur de texte ou d'une zone de saisie. Lorsque l'on utilise des composants d'interface utilisateur évolués, ce type d'interaction n'a pas besoin d'être programmé. Le composant d'interface le gère tout seul. À l'inverse, n'espionner que les opérations de très bas niveau ne permet pas de s'abstraire des tâches articulatoires qui constituent une part importante du dialogue homme-machine.

L'enregistrement définit une intervention à des niveaux divers de la boîte à outils ou son niveau est fonction du résultat attendu lors du rejeu. En effet si l'on souhaite simplement espionner les actions effectives du système, un enregistrement des actions de haut-niveau du type « clic » sur un bouton est suffisant. Mais s'il s'avère nécessaire de reproduire le feedback précis de l'interaction, l'enregistrement des événements de bas-niveau permet de produire ce résultat.

L'application *Eager* montre que la capacité principale à fournir est l'espionnage. Il consiste à l'observation en permanence des actions de l'utilisateur, au repérage des séquences dans la suite des actions et au renseignement d'une liste de « jetons » ou d'occurrences de commande (Cypher, 1991).

Le premier service à fournir est donc l'enregistrement des interactions de l'utilisateur avec le système suivant les possibilités des entrées des actions à plusieurs niveaux (niveau d'enregistrement des événements).

4.2.2 Rejeu

Le rejeu consiste en la récupération des séquences d'événements et à leur ré-exécution. Pour effectuer le rejeu, deux angles sont envisageables : on peut réactiver les interactions de l'utilisateur sans son intervention explicite au démarrage ou on peut lui fournir la possibilité de déclencher la réactivation. En d'autres termes, il s'agit d'effectuer le rejeu avec ou sans interface. Le rejeu avec interface donnera la possibilité à l'utilisateur de lancer et suivre ces traces tandis que le rejeu sans interface s'activera au gré de l'utilisateur et mettra en évidence ces traces.

Après analyse, il ne devrait pas être toujours nécessaire d'attendre l'intervention de l'utilisateur au niveau du rejeu. Lorsque l'on souhaite faire valider l'analyse d'une tâche exemple, on doit réactiver les interactions de l'utilisateur en les mettant en évidence. Par exemple, lorsque l'utilisateur est invité à confirmer une commande, on peut modifier la présentation de celle-ci. À l'inverse, lorsque la tâche doit être terminée automatiquement, il n'est pas forcément nécessaire de rejouer exactement les interactions de l'utilisateur. Deux modes de fonctionnement doivent donc être pris en compte en fonction des besoins du rejeu. Dans *Eager*, on veut ajouter une vérification, avec une mise en évidence de l'interaction à

effectuer. C'est aussi le cas dans PbDScript où lorsqu'un élément de la hiérarchie est désigné dans PbDScript, le *widget* correspondant est mis en évidence dans l'application. Dans d'autres cas, on voudra réexécuter les événements avec temporisation, ou sans temporisation. Dans ces cas, la nécessité de la temporisation est souvent liée à la généralisation où l'on demande de renseigner certaines variables du système lors du rejeu.

4.2.3 Capacité de généralisation

La généralisation est le mécanisme le plus complexe dans la mise en place d'un système de PsE. En effet, non seulement la phase d'enregistrement des actions est capitale, mais la technique et le contexte de rejeu doivent être pris en compte. La généralisation porte sur trois éléments. La première généralisation est celle des commandes. En effet, toute action de l'utilisateur est une commande au niveau du système. L'interprétation de l'action, qui est la commande dans le système, dépend du principe et de la structure de l'enregistrement. Cela aussi a une implication sur le rejeu. Pour être plus précis, dans une interaction, qu'est-ce qui doit être considéré comme une commande ? Les événements souris (appui sur le bouton, relâchement du bouton, …), les événements clavier (appui sur une touche, relâchement d'une touche, ..) ou des actions sémantiques de plus haut-niveau, comme le choix d'un item dans une liste par exemple ?

La seconde généralisation est celle des objets contenus dans les commandes. Des attributs sont affectés à ces objets que la généralisation devra prendre en compte. Les attributs peuvent aussi provenir de ces mêmes objets. Le système devra identifier les variables et les collections d'objets. La généralisation doit permettre, à partir des objets, de construire un vrai programme qui accepte des paramètres.

La troisième généralisation est celle des structures de contrôle de l'application. Ces structures disposent des mécanismes qui doivent être implémentés. Mais, en plus de cette implémentation, on doit pouvoir ajouter un mécanisme de détection des structures de contrôle. Les contextes associés aux objets lors de la construction de séquence permettent la détection des boucles, des attributs et l'identification des variables.

La généralisation, source de réutilisation et d'extensibilité, est plus facile à obtenir avec l'intervention et l'interaction explicite de l'utilisateur, mais il est judicieux de l'obtenir de manière générique et de fournir une possibilité d'intervention *a posteriori* de l'utilisateur.

4.2.4 Mécanismes utilisateur

Les boîtes à outils possèdent aujourd'hui des composants d'interface utilisateur évolués qui permettent de s'affranchir de beaucoup de programmation. Du côté programmeur, il s'agit de recenser et minimiser les actions de programmation explicite nécessaires pour construire les fonctionnalités de la PsE. Mais ce n'est pas ce côté qui nous intéresse le plus pour cette section. C'est plus du côté utilisateur de l'application que nous devons définir ces mécanismes.

En effet, comme il a été souligné, l'apport de l'utilisateur en termes d'interaction est très important dans les applications interactives. Les services offerts par la PsE dans ces applications peuvent avoir besoin de certaines interactions afin de pouvoir réaliser des fonctionnalités prédéfinies. On peut par exemple définir une barre d'outils (avec les actions "Enregistrer"/"Fin"/"Rejeu") comme dans certains systèmes. Mais devons nous aller plus loin que cela comme dans PbDScript, avec la possibilité pour l'utilisateur de définir sa sémantique ? Pour minimiser l'apport de l'utilisateur final, il est souhaitable de mettre en place une application prenant en compte, de manière automatique, l'exécution des différentes

fonctionnalités, avec la possibilité de contrôle offerte à l'utilisateur. C'est donc au programmeur qu'incombera la tâche de définir le mécanisme simplifié d'intervention de l'utilisateur.

Ainsi, pour l'enregistrement, trois mécanismes peuvent êtres envisagés. (1) L'utilisateur peut à l'aide d'une commande lancer le début de l'enregistrement des actions et arrêter l'enregistrement à la fin de la tâche dont il souhaite l'automatisation. C'est le processus classique des enregistreurs de macros. (2) Il est aussi possible de déclencher de façon implicite sans l'intervention de l'utilisateur l'enregistrement des événements et permettre à l'utilisateur de les sauvegarder à la fin de sa tâche. Il doit être aussi en mesure de restaurer, c'est-à-dire rejouer ces événements. (3) Enfin, une possible combinaison des deux modes (enregistrement explicite et enregistrement implicite) peut être pratiquée. Que l'utilisateur ait déclenché ou pas l'enregistrement, celui-ci est automatiquement lancé sur toutes les actions dès le début des interactions, mais l'utilisateur a la possibilité de le contrôler en fonction de ses besoins. Il pourra choisir certaines actions à prendre en compte et pouvoir en soustraire d'autres.

Pour le rejeu des actions, seulement deux possibilités s'offrent. (1) L'utilisateur peut déclencher le rejeu systématique de toutes les actions enregistrées à l'aide d'une commande (un bouton par exemple). (2) Il doit être aussi possible de rejouer pas à pas les actions sous le contrôle de l'utilisateur. Dans ce contexte, l'utilisateur pourra soit exécuter entièrement toutes les actions jusqu'à la fin de la tâche, soit s'arrêter à une étape voulue volontairement.

5 Conclusion

Notre objectif est de définir un outil pour introduire des techniques de PsE dans une application interactive, principalement dans le but d'assister l'utilisateur dans sa tâche. L'outil doit pouvoir supporter un fonctionnement de type adaptatif (système automatique) ou adaptable (construction explicite de macros). Il doit également permettre d'intervenir à tous les niveaux de l'enregistrement/rejeu, selon le besoin ressenti. À ce jour, seuls deux outils (PbDScript et AIDE) existent pour répondre partiellement à ces besoins. Malgré leurs apports considérables, ces outils restent limités par rapport aux besoins réels de l'application de la PsE.

L'analyse de ces deux systèmes a permis de conforter les services à fournir pour faciliter l'intégration des techniques de la PsE dans les applications. Le rôle de l'utilisateur dans l'application de la PsE est l'automatisation de tâches répétitives, l'intégration des applications pour créer une solution sur mesure ou l'adaptation d'une application à des besoins spécifiques. Le rôle du programmeur est de permettre la réalisation des objectifs de l'utilisateur en extrayant ce dernier des concepts de la programmation et des qualités d'abstraction de développement.

Les services essentiels à fournir sont l'enregistrement, le rejeu et la capacité de généralisation. La prise en compte d'un mécanisme utilisateur convenable est également nécessaire pour le bon usage de l'application.

Chapitre 3

Une boîte à outils pour la programmation sur exemple : principes et mise en œuvre

SOMMAIRE

Résumé. *Ce chapitre présente les solutions pour implémenter un outil général de programmation sur exemple. Les techniques d'implémentation possibles sont discutées. Le travail est orienté vers l'assistance à l'utilisateur et l'extension de la boîte à outils Swing est mise en place par l'enrichissement des widgets. Chaque nouveau composant est doté d'une capacité d'enregistrement et de rejeu. L'outil générique implémenté est PbDToolkit. Il facilite le développement d'application intégrant la PsE en offrant des modules permettant de gérer les techniques et principes de la PsE.*

1 Introduction

L'implémentation d'un système intégrant la PsE demande au développeur, dans une vision simpliste, de mettre à la disposition de l'utilisateur final, des outils d'assistance pour l'automatisation de tâches. Nous avons détaillé ces besoins dans le chapitre précédent. Pour le développeur, cela passe par la mise à la disposition des différents services (décrits dans la section 4.2 du chapitre 2), à partir de l'interface utilisateur de l'application qu'il construit. Il doit donc fournir des interfaces particulières car non seulement l'objectif fonctionnel de l'application ne doit pas changer, mais surtout les techniques de la PsE doivent être naturellement intégrées dans l'application. Le développeur a besoin de composants d'interface particuliers pour réaliser des interfaces particulières. À partir des fonctionnalités des widgets, il est facile au développeur de fournir des applications intégrant la programmation sur exemple. En effet, un système de PsE est un système difficile à implanter comme le montrent les besoins et les systèmes existants. Pourtant la plupart des systèmes de PsE possèdent des éléments en commun parmi lesquels on trouve une représentation des commandes ou actions utilisateur et souvent du contexte associé, un historique des actions, et parfois un algorithme d'apprentissage symbolique opérant sur l'historique. Nous favorisons la création d'un tel système en fournissant les outils nécessaires sous forme d'une boîte à outils par l'extension de la boîte à outils Swing. Les développeurs peuvent bâtir, avec un minimum d'effort, des applications mettant en œuvre les techniques démonstrationnelles de la PsE.

Nous avons identifié et défini les principaux services de base. Prototyper ces principaux services (d'une manière générique) à travers un outil ouvre la voie vers la simplification de la mise en œuvre des applications de PsE. Ces principaux services de base sont l'enregistrement des actions utilisateur, le rejeu de ces actions et des techniques utilisables pour la mise en œuvre des applications types. Lors du développement de l'application, il est impératif de tenir compte de la mise en place d'un historique de commande. En utilisant l'outil générique, il ne sera pas nécessaire car toutes les opérations y sont déjà implémentées avec la liberté d'usage et d'exploitation par le développeur. Quoi qu'il en soit, il ne devra plus être question que ces tâches de codage incombent plus à un développeur, c'est-à-dire écrire le code correspondant aux services d'espionnage ou de scrutation du système et l'historique des interactions.

Dans ce chapitre, il est question du développement de la solution. Nous parlons des techniques d'implémentation possibles ainsi que des différents objets implémentés et leur utilisation. Dans une première partie, l'orientation de la solution est présentée et argumentée, à partir d'une présentation des outils de construction d'interfaces. Nous avons opté pour l'extension d'une boîte à outils. Nous montrons les besoins en widgets particuliers et les différentes techniques d'ajouts de nouveaux composants dans une boîte à outils, en précisant au passage la description approfondie d'une boîte à outils d'interaction. La deuxième partie expose l'outil générique, PbDToolKit. Le mécanisme d'enregistrement et de rejeu est décrit. De son architecture générale à ses différentes composantes, nous décrivons l'ensemble des techniques utilisées dans PbDToolKit. La dernière partie du chapitre présente un exemple d'utilisation ainsi que l'effort de programmation à fournir par un développeur.

2 Vers une solution « boîte à outils »

Nous souhaitons intégrer les techniques de la PsE dans les applications interactives, et cela d'une manière simple, facile et efficace. Cette intégration passe par l'ajout de modules logiciels qui, elle-même, passe par la définition d'une architecture d'intégration. Pour la mise en place des techniques de la PsE, que doivent contenir ces modules logiciels ? Nous avons montré dans le chapitre 2 qu'il s'agit d'incorporer des techniques d'enregistrement et de rejeu, qui sont les fonctions basiques. Mais, dans un tel outil, quels sont les besoins du développeur ?

Il existe des solutions d'implémentation de certaines techniques allant plus ou moins dans le sens de l'objectif de nos travaux, mais souvent non définies dans ce cadre à l'origine. Une technique particulière d'implémentation émerge aujourd'hui, la programmation par aspect. Elle mérite d'être analysée, et confrontée à nos objectifs. Souvent présentée dans une optique de recueil de « traces », peut-elle permettre de réaliser des applications intégrant la PsE ou permettant l'enregistrement et le rejeu des interactions de l'utilisateur ? Nous verrons cela dans la dernière partie de cette section. Mais avant, dans un premier temps, nous présentons les outils de construction d'IHM et nous justifions le choix de travailler sur les boîtes à outils. La réponse aux besoins du développeur d'applications interactives constitue la deuxième partie de la section. Les solutions d'extension des boîtes à outils dans le contexte objet font l'objet de la troisième partie.

2.1 Outils de construction d'interfaces

L'augmentation de la complexité des IHM a conduit à accroître la quantité de code nécessaire à son développement. Pour des raisons d'homogénéité de l'interface, de complexité (rendre accessible la programmation à tous) et d'économie (temps de réalisation), un grand nombre d'outils d'aide au développement ont été créés (Shneiderman, 1998) (Patry, 1999). Pour certains de ces outils, le noyau est la programmation de la couche présentation jusqu'à la description du dialogue de l'application. D'autres sont des outils de description pour générer tout ou partie de la présentation et du contrôleur de dialogue. Notre intérêt se situe principalement au niveau de la première catégorie d'outils. Leur objectif est de fournir des outils pour construire la couche présentation au moyen d'éléments de présentation tels que des boutons, des fenêtres, des menus, etc. Ensuite, le développeur relie les éléments d'entrée (réaction aux événements) et de sortie (attribut d'un élément) de la présentation avec des fonctions du noyau fonctionnel.

Ainsi, quand l'utilisateur interagit avec la couche présentation, des éléments du noyau fonctionnel sont modifiés par des fonctions (*modifieurs*) et l'état du noyau fonctionnel est retourné à la présentation par d'autres fonctions (*accesseurs*). Les éléments de la présentation appelés communément *composants graphiques* ou *widgets* sont regroupés dans les boîtes à outils.

D'autres systèmes se sont basés sur les boîtes à outils pour aider à concevoir des interfaces. Les squelettes d'applications réalisent les fonctions usuelles de l'interface sous la forme d'un logiciel réutilisable et extensible (Coutaz, 1990). Il incombe au développeur d'ajouter sur le squelette les outils spécifiques à l'application finale. Le squelette est construit au dessus d'une boîte à outils et suivant une certaine architecture logicielle. Enfin, les générateurs d'interfaces permettent de construire l'interface de l'application de manière graphique tout en se basant sur un squelette d'application. La Figure 30, tirée de (Coutaz, 1990) illustre cette représentation en couche.

Dans cette section, nous présentons ces outils de construction d'interfaces, puis justifions notre choix de la boîte à outils.

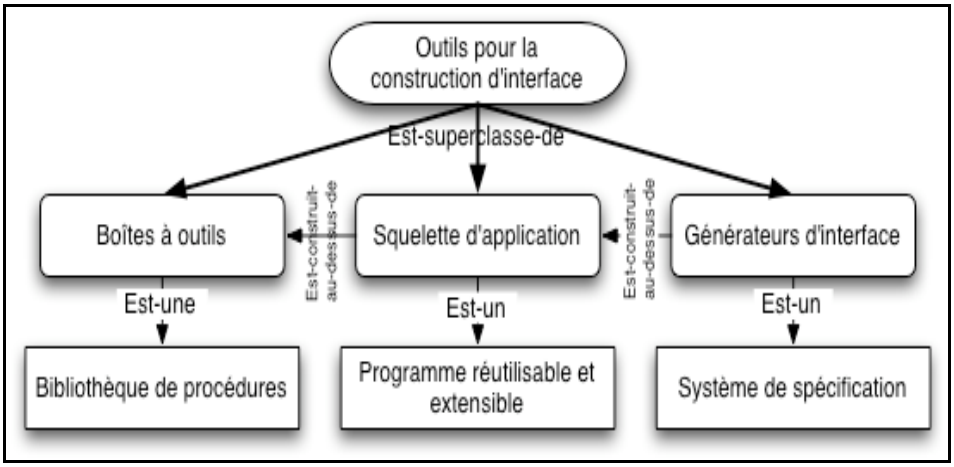

Figure 30 : Classification des outils de construction d'interfaces

2.1.1 Boîtes à outils

Les *boîtes à outils* sont des bibliothèques de procédures adaptées à la programmation d'interfaces homme-machine (Coutaz, 1990). Elles offrent de nombreux services tels que la gestion des événements souris et clavier, l'affichage des fenêtres et de leur contenu, la gestion des objets d'interface comme les menus, les boutons et les champs de textes, etc. Ces boîtes à outils se situent généralement à un niveau proche de la machine et dans les systèmes d'exploitation modernes elles peuvent contenir des milliers de fonctions.

Pour réutiliser une boîte à outils, le développeur doit réimplanter des fonctionnalités similaires d'une application à l'autre (telles que les séquences d'initialisation, le rafraîchissement des fenêtres, la gestion des événements, etc.) car les boîtes à outils ne fournissent pas la charpente d'une application mais seulement les briques de base (Coutaz, 1990). Les boîtes à outils fournissent une interface de programmation (API[12]) orientée objet qui encapsule les données. L'interface de l'application est créée en combinant un ensemble de composants graphiques d'interface utilisateur ou *widgets*, tels que les fenêtres, les boutons, les menus, etc. Les composants graphiques, les objets d'interaction, réagissent aux interactions de l'utilisateur. Les widgets encapsulent des comportements facilitant ainsi leur utilisation. Ils définissent des étiquettes pour leurs données membres et leurs méthodes afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non. Les boîtes à outils utilisent le mécanisme d'événement pour gérer le dialogue entre l'utilisateur et la présentation. Une boîte à outils permet de donner un style à l'interface et permet d'obtenir une homogénéité visuelle et comportementale entre différentes applications.

2.1.2 Squelettes d'application

Les *squelettes d'application* sont des assemblages logiciels réutilisables et extensibles qui réalisent une large part des fonctions de l'interface d'une application (Coutaz, 1990). Ils remédient à certains des problèmes liés aux boîtes à outils, en particulier la duplication d'efforts et à un degré moindre l'apprentissage. Ainsi, la tâche du développeur consiste à

[12] Application Program Interface : Interface de Programmation d'Applications.

remplir les trous du squelette et à redéfinir les parties prédéfinies du système, inadaptées aux besoins particuliers de son application. La réutilisation, l'extensibilité et la surcharge de logiciel se traduisent directement dans les termes de la programmation par objets. La plupart des squelettes d'application utilisent cette technique et sont construits au-dessus d'une boîte à outils.

L'avantage des squelettes d'application est qu'ils fournissent une architecture logicielle saine : le développeur n'a plus à déterminer l'assemblage correct des briques logicielles. De plus, le niveau de service procuré est plus proche des besoins du développeur qui doivent être les mêmes que ceux pour lesquels le squelette a été construit. L'inconvénient majeur tient essentiellement aux difficultés de la réutilisation logicielle : pour réutiliser de façon optimale un squelette d'application il faut bien comprendre son fonctionnement et il faut aussi que le type d'application à développer rentre dans le moule. Enfin, lorsqu'il doit étendre ou surcharger certaines parties du squelette, le programmeur retombe inévitablement sur les services de la boîte à outils.

2.1.3 Générateurs d'interfaces

Les *générateurs d'interfaces* ou encore GUI-Builder (*Graphic User Interface Builder*) créent l'interface d'une application à partir d'une spécification graphique (Coutaz, 1990). Cette interface est reliée à un noyau d'exécution qui s'apparente à un squelette d'application. Cet outil, malgré sa grande évolution de nos jours (Blanch, 2005) ne permet pas de créer des interfaces avec le même niveau de généralité que les boîtes à outils ou les squelettes d'application. Les générateurs d'interface emploient généralement des techniques de programmation visuelle (manipulation directe des objets graphiques au moyen de la souris, visualisation graphique des informations, …). Ils sont faciles à utiliser et permettent d'obtenir rapidement des maquettes. Il n'est pas nécessaire d'être un spécialiste en programmation pour construire la couche présentation. Néanmoins des connaissances approfondies en programmation sont nécessaires afin d'ajouter la dynamique du dialogue ou pour déterminer le comportement, et les appels aux fonctions du noyau fonctionnel pour construire l'application.

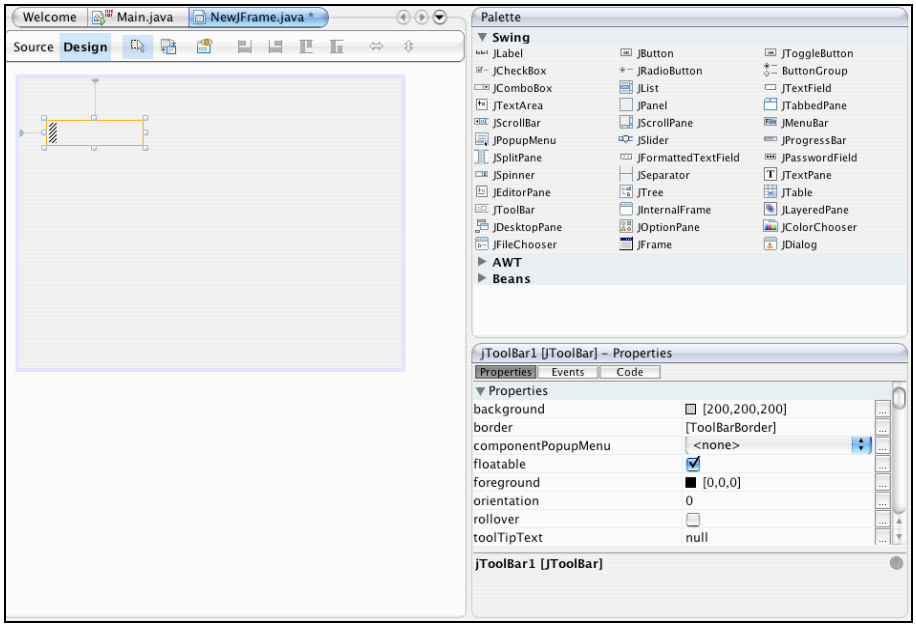

Figure 31 : Générateur d'interface sous NetBeans 5.5

Les générateurs d'interfaces intègrent de nombreux squelettes d'applications qui facilitent le développement de l'application interactive, par exemple l'utilisation d'assistants (*Wizard* en anglais). Cette catégorie ne définit pas la succession des objets à l'écran, c'est-à-dire qu'elle ne définit généralement pas la dynamique du dialogue. Mais certains outils tels que BeanBuilder[13] associé à la technologie JAVA/beans[14] apportent une contribution dans ce domaine. En effet ce dernier autorise la construction de la couche présentation et une ébauche d'un pseudo dialogue au travers des objets Beans.

2.1.4 Boîte à outils, un choix logique

Comme nous l'avons établi, la PsE requiert des techniques de programmation spécifiques et l'intégration de ses mécanismes nécessite, le plus souvent, une recomposition de l'application cible afin de pouvoir capturer les actions utilisateur à un niveau variable. La catégorie la plus susceptible d'être d'un grand apport pour le développeur est celle des boîtes à outils. Nous fournissons donc un concept de bibliothèque de procédures. Il n'existe pas de plate-forme robuste, c'est-à-dire un système de création ou de gestion de programmes, pour la construction de système de PsE. On ne dispose pas non plus d'outils spécifiques permettant l'implantation de ces systèmes. Fournir un outil, afin de faciliter l'incorporation de la PsE dans les applications se veut une grande étape vers la réalisation d'une telle plate-forme, susceptible de mettre les techniques de la PsE à la portée de tout programmeur.

Il est fréquent qu'une application soit composée de classes issues d'une ou de plusieurs bibliothèques de classes prédéfinies, c'est-à-dire des boîtes à outils. Une boîte à outils est un ensemble de classes apparentées et réutilisables conçues pour offrir des fonctionnalités d'intérêt général. Les boîtes à outils n'imposent à une application aucune technique particulière de conception ; elles fournissent simplement des fonctionnalités qui facilitent la tâche d'implémentation des applications. Elles dispensent le développeur d'avoir à reprendre le codage des fonctionnalités d'usage courant. Les boîtes à outils sont les faire valoir de la réutilisation de code.

Plutôt que d'inventer un système propriétaire, nous nous appuyons sur l'extensibilité naturelle des boîtes à outils (sous-section 2.3) pour construire un système généralisable. Enrichir une boîte à outils existante, élimine l'augmentation des facteurs de difficultés et d'utilisation liées à la boîte à outils d'une part, et d'autre part favorise l'utilisation du système par un grand nombre de développeurs.

Le choix de la boîte à outils est donc justifié d'une part par la nécessité d'intégrer de nouveaux mécanismes aux composants existants d'une manière simple et efficace, et d'autre part par la nécessité d'une intervention dans le cadre du principe de l'enregistrement à un niveau proche du noyau fonctionnel de l'application.

2.2 Besoins du développeur

Les développeurs d'IHM suivent en général, lors de la réalisation d'un logiciel, un cycle de développement itératif (Winston, 1970) (Boehm, 1988). Les cycles de développement sont particulièrement adaptés au processus de développement des IHM. Des modèles de base, parmi lesquels les modèles d'architecture, interviennent dans ces cycles de développement des systèmes interactifs (Bass et al., 1988).

[13] Sun Microsystems : http://www.sun.com
[14] http://java.sun.com/poducts/javabeans/index.jsp

Les modèles d'architecture sont nés du constat que la conception des IHM est un processus complexe et itératif. Par conséquent la possibilité d'apporter des modifications au logiciel d'IHM est un élément important du cahier des charges. Dans ce sens, la décomposition de l'application interactive en différents éléments directeurs est rendue nécessaire. Les modèles d'architecture définis pour l'IHM reposent sur un principe commun : la séparation entre le modèle sémantique de l'application (noyau fonctionnel) et l'interface fournie à l'utilisateur pour interagir avec le modèle. Les modèles d'architecture décrivent également la manière dont ces deux composants communiquent entre eux. Ainsi, ils offrent une structure générique destinée au concepteur de l'application.

Les apports d'une telle organisation modulaire sont importants. D'une part, cette organisation permet une conception plus simple dans la mesure où chaque module peut être réalisé de manière plus ou moins indépendante. D'autre part, les modifications apportées aux modules s'en trouvent amoindries et leur fiabilité accrue.

De nombreux modèles d'architectures[15] ont été élaborés au fur et à mesure des avancées techniques. Nous nous sommes particulièrement intéressés au modèle d'architecture ARCH (Bass et al., 1991). Ce modèle s'intéresse à la séparation de la partie interface du reste de l'application, afin de rendre cette dernière indépendante de tout élément spécifique à la présentation. En revanche, il ne définit ni la structure interne des composants, ni les interfaces d'échange entre ces composants.

ARCH divise une application interactive entière en cinq composants : boîte à outils, présentation, dialogue, adaptateur de domaine et domaine (Figure 32).

- Le *domaine* représente les objets et les fonctions propres à l'application qui demeurent indépendantes de l'interface utilisateur.
- L*'adaptateur de domaine* joue le rôle d'intermédiaire entre le domaine et le dialogue en garantissant une totale indépendance entre eux. Il permet d'implémenter notamment les tâches utilisateurs relatives au domaine.
- Le *dialogue* est chargé du contrôle du dialogue. Il est chargé de maintenir la cohérence entre les différentes vues d'un même objet.
- La *présentation* agit en intermédiaire entre le dialogue et la boîte à outils. Elle est composée de représentations abstraites d'objets d'interactions.
- Enfin, le *composant boîte à outils* implémente l'interaction physique avec l'utilisateur.

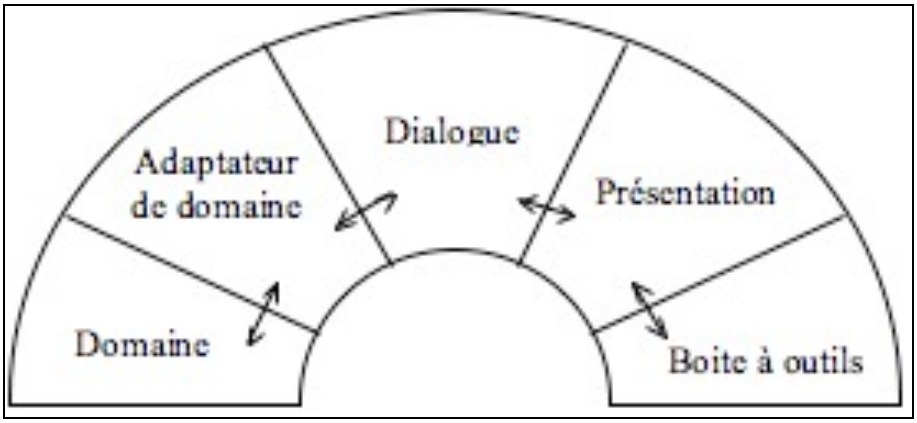

Figure 32 : Le modèle ARCH

[15] Exemples de modèles d'architectures : Seeheim, MVC (Model Vue Control), PAC (Presentation Abstraction Control), H[4]

D’une manière générale, les besoins du développeur sont satisfaits par les objets d’interaction. Chaque boîte à outils dispose de sa stratégie propre pour répondre aux besoins du développeur, en fonction du paradigme utilisé (programmation fonctionnelle, programmation objet, etc.).

Les systèmes informatiques ainsi implémentés proposent aux utilisateurs la possibilité d’interagir avec l’application. Ces possibilités sont offertes grâce aux interfaces utilisateur. Les interfaces utilisateur sont implémentées à l’aide de technologies permettant de restituer et de formater les données destinées aux utilisateurs, ainsi que d’acquérir et de valider les données fournies par ces derniers. L’interface utilisateur contient les composants nécessaires à l’interaction de l’utilisateur. Les interfaces utilisateur les plus simples contiennent des widgets comme des boutons, des zones de saisie ou d’affichage de texte. Pour obtenir des interactions utilisateur plus complexes, on peut concevoir des composants de processus utilisateur afin d'orchestrer les éléments d'interface utilisateur et de contrôler l'interaction utilisateur. Ces composants sont particulièrement utiles lorsque l'interaction utilisateur suit une succession prévisible d'étapes, comme lorsqu'un assistant est utilisé pour accomplir une tâche.

Nous pouvons dire que la satisfaction première des besoins du développeur est la fourniture de widgets prenant en compte ses préoccupations (capacités d’enregistrement et de rejeu). En effet, les widgets peuvent être implémentés de différentes façons pour gérer l'interaction avec l'utilisateur. Ils affichent les données à l'utilisateur, acquièrent les données fournies par ce dernier et interprètent les événements que l'utilisateur déclenche pour influer sur les données de l’application, modifier l'état de l’interface ou aider la progression de l'utilisateur dans sa tâche. Quelles sont les fonctionnalités ou la puissance d’expression des widgets ? Nous abordons la question dans cette sous-section suivante.

2.2.1 Fonctionnalités des widgets

Les widgets doivent afficher les données aux utilisateurs, acquérir et valider des données à partir des entrées de l'utilisateur et interpréter les gestes de l'utilisateur indiquant que celui-ci souhaite effectuer une opération sur les données (Bass and Coutaz, 1991). De plus, l'interface utilisateur doit filtrer les actions disponibles afin de permettre aux utilisateurs d'effectuer uniquement les opérations appropriées à un moment donné.

Les widgets comportent une référence à un composant de processus utilisateur actif s'ils doivent afficher les données de ce dernier ou agir sur son état. Ils peuvent encapsuler à la fois une fonctionnalité d'affichage et un contrôleur. Lors de l'acceptation de l'entrée de l'utilisateur, ils acquièrent des données en provenance des utilisateurs et les assistent en leur fournissant des repères visuels. Les widgets restreignent les types d'entrées disponibles pour un utilisateur. Par exemple, un champ peut limiter les entrées de l'utilisateur à des valeurs numériques uniquement. Ils exécutent une validation de l'entrée des données, en limitant la plage des valeurs qui peuvent être entrées dans un champ donné, par exemple. Ils effectuent un "mapping" et des transformations simples des informations fournies par les contrôles utilisateur pour les transformer en valeurs requises par les composants sous-jacents (par exemple, le composant d'une interface utilisateur peut afficher un nom, mais transmettre une référence aux composants sous-jacents). Les widgets interprètent les gestes de l'utilisateur (comme une opération glisser-déplacer ou des clics sur un bouton) et appellent une fonction du contrôleur.

Lors de la restitution des données, les widgets acquièrent et restituent des données à partir des composants internes ou des composants de la logique d'accès aux données dans l'application. Ils effectuent le formatage des valeurs (comme le formatage des dates de façon

appropriée). Ils restituent généralement des données relativement à des entités du noyau fonctionnel. Les widgets peuvent restituer les données selon leurs attributs d'affichage et ceux des collections des composants de l'entité, si l'entité est déjà disponible.

Les widgets jouent donc un rôle important dans la satisfaction des besoins du développeur. Ces composants sont fournis par les outils de développement disponibles en l'occurrence les boîtes à outils d'interaction pour la réalisation d'application graphique interactive. Nous allons décrire ces boîtes à outils afin de mieux comprendre leur fonctionnement et de pouvoir mieux se situer dans la mise en place du module d'intégration des techniques de la PsE.

2.2.2 Boîtes à outils d'interaction

Dans cette section, nous cherchons à approfondir les principes de base des boîtes à outils d'interaction.

2.2.2.1 Généralités

Une boîte à outils est donc une bibliothèque d'objets d'interactions (Baudel, 2002) appelés composants d'interface utilisateur ou widgets. Les boîtes à outils fournissent une interface de programmation orientée objet. Elles sont relativement nombreuses, pour n'en citer que quelques unes : Swing (Java), MFC (Microsoft Foundation Class), QT, Tk ou AMULET. L'interface de l'application est créée en combinant un ensemble de composants graphiques, comme les fenêtres, les boutons, les menus, etc. Les widgets réagissent aux interactions de l'utilisateur. Dans ce but, les boîtes à outils utilisent le mécanisme d'événements pour gérer le dialogue entre l'utilisateur et la présentation. Quand l'utilisateur interagit avec un composant graphique de la présentation, son action est transformée par la boîte à outils en un ou plusieurs événements. À la réception d'un événement, le composant graphique réagit au travers de deux comportements :

- le comportement interne représente une réaction propre d'un composant graphique à un événement. Il n'est généralement pas modifiable et son rôle essentiellement esthétique n'a d'importance que dans la présentation. Par exemple l'état d'un bouton enfoncé ou non enfoncé est modifié quand l'utilisateur appuie sur le bouton de la souris et que le pointeur se trouve sur ce dernier ;
- le comportement externe est associé au résultat que le concepteur souhaite obtenir quand le widget subit une action de la part de l'utilisateur. Ce comportement doit être programmé par le développeur. Pour cela, le développeur utilise un mécanisme de traitement des événements qui diffère selon la boîte à outils employée : boucle d'événement, fonctions de rappels ou abonnements. Par exemple, dans le cas du traitement des événements par abonnement, utilisé dans Swing, la conception consiste à associer à chaque widget émetteur d'événements un ou plusieurs objets récepteurs d'événements qui se chargeront d'appeler des éléments du noyau fonctionnel. Ainsi, les boîtes à outils permettent la description de la présentation, mais aussi assurent le dialogue de l'application.

Le bout de code dans le cadre Code 1 montre un exemple qui décrit l'utilisation du traitement des événements par la technique de l'abonnement. L'objet *bouton* est créé sur la première ligne puis il est associé à un objet écouteur *ActionListener* sur la deuxième ligne.

Suivant la nature de l'événement émis, la méthode, ici *actionPerformed,* appelle la méthode du noyau fonctionnel.

```
JButton bouton = new JButton('Ok')
  bouton.addActionListener( new Actionlistener() {
     public void actionPerformed( ActionEvent e) {
        nf.convertir() ;
     }
  }) ;
```

Code 1 : Abonnement d'un composant graphique Swing à un écouteur

Nous avons montré qu'une boîte à outils est une bibliothèque de composants chargés de maintenir des structures de données et réaliser des opérations communes à l'implémentation d'applications interactives (à caractère graphique en général). En d'autres termes, il s'agit de gérer les entrées de l'utilisateur, le dessin, les contraintes de positionnement, l'insertion de l'application dans un environnement et l'établissement d'un lien entre les objets d'interaction et la sémantique de l'application. L'objectif d'une boîte à outils est de fournir un cadre de programmation adéquat pour implanter l'interface utilisateur d'une application (Lecolinet, 1999). Une boîte à outils repose sur un langage, un système d'exploitation et un environnement d'utilisation (GUI du système hôte). Une application se définit comme un ensemble d'objets de cette sorte fournissant des fonctions diverses.

2.2.2.2 Objets d'interaction

Les objets d'interaction dans une boîte à outils ne sont pas obligatoirement graphiques. Ils ont des caractéristiques aussi bien communes que distinctes. Ils possèdent une représentation déclinée éventuellement sur plusieurs modalités. Les objets d'interaction reçoivent des entrées qui leur font changer d'état et produisent des sorties. Ils constituent les principaux composants des boîtes à outils.

2.2.2.3 Organisation des objets graphiques

Les objets graphiques d'une boîte à outils sont organisés en une structure définie (arbre, tableau, etc.). La structure a pour but la présentation spatiale ou temporelle des objets ou widget. Un deuxième type de structure est le contrôle. Il s'agit là de l'organisation des objets pour permettre les transferts de données au cours du dialogue.

La structure de présentation est généralement explicite, alors que la structure de contrôle est souvent implicite. *A fortiori*, si l'on utilise des mécanismes évolués (contraintes), il est fort possible de rencontrer des problèmes de compréhension de code.

2.2.2.4 Communication entre objets

Les objets interactifs sont organisés en une structure définie et ne sont donc pas conçus de façon isolée. Ils représentent une hiérarchie de concepts signifiant l'application. Il doit être possible d'établir des liens, fonction de la sémantique de ces objets. Pour assurer le dialogue au niveau des widgets, les objets de la boîte à outils communiquent entre eux. Plusieurs techniques ou méthodes de communication existent : appel de fonction, envoi de message - méthode (*dispatch*) ou aussi la continuation, callback, notifier - observer ou encore le polling, événement et dispatcher plus listener, lien directionnel entre deux objets, propagation unidirectionnelle, résolution des contraintes généralisées, etc.

2.3 Comment intégrer de nouveaux outils aux boîtes à outils

Dans une boîte à outils, des services supplémentaires peuvent être inclus, et ils le sont souvent (Baudel, 2002), c'est l'extension des composants prédéfinis des boîtes à outils (Lecolinet, 1999). On peut dire en ce sens qu'une boîte à outils est une bibliothèque permettant l'implémentation directe au niveau des applications et permettant de gérer des dispositifs d'interactions supplémentaires. Pour permettre la prise en compte de nouveaux dispositifs d'interaction, une implémentation des liens forts avec la sémantique est nécessaire, voire obligatoire. On fournit dans de telles situations une bibliothèque logicielle de base, ensemble d'objets composables à des finalités précises et prêtes à l'emploi. Cette bibliothèque logicielle devient un service de plus pour la boîte à outils qu'elle intègre. Elle est intégrée suivant la taxonomie de la boîte à outils et la stratégie d'implémentation existante. Cela conforte plus notre choix d'extension d'une boîte à outils.

Suivant les boîtes à outils, les objets sont composables, hiérarchisables et spécialisables, avec plus ou moins de facilité. La structure des objets et le langage influent sur les possibilités d'extension et de modularisation offertes par la boîte à outils. Les boîtes à outils fournissent plusieurs types de services. Par exemple, les classes de base proposent les conteneurs, les algorithmes de base (tri, recherche, …), la gestion de la mémoire, des chaînes de caractères, des objets géométriques, etc. Malgré de nombreux progrès, il n'existe pas de couche d'utilitaires de base suffisamment standard amenant ainsi presque toutes les boîtes à outils à redéfinir leurs classes de base. Aussi, les boîtes à outils disposent d'une description des ressources disponibles à la présentation du contenu de l'application. Il s'agit en général d'un écran, avec des capacités graphiques que l'on peut supposer uniformes de plus en plus sur l'ensemble des stations de travail. Une boîte à outils permet de maintenir un processus à l'écoute des actions de l'utilisateur. Pour cela, elle utilise un modèle à événement où chaque action est échantillonnée en événement discret portant un type, des paramètres propres aux types ainsi que des paramètres sur l'état des dispositifs ou du système au moment où l'action s'est produite (état des « *modifieurs* », fenêtre située sous le pointeur, …). Cette spécificité de la boîte à outils est le support clé de la mise en place de la technique d'enregistrement des interactions utilisateur. La boîte à outils doit posséder un modèle des objets de l'application lui permettant de déterminer ceux qui doivent êtres informés de l'événement pour le transformer (directement ou non) en un changement de l'état de l'application.

Les services inclus dans une boîte à outils sont assurés par les composants prédéfinis. L'intégration de nouveaux outils correspond à l'ajout de nouveaux composants ou à l'extension de composants existants. Nous nous appuyons dans la suite de cette section sur l'étude de (Baudel, 2002) qui résume les mécanismes d'extension à cinq mécanismes essentiels : le paramétrage, la composition, la délégation, la redéfinition, la dérivation. Nous présentons chacun de ces mécanismes avant de discuter de notre option pour la boîte à outils.

2.3.1 Paramétrage

Il concerne la boîte à outils et consiste en son adaptation par l'introduction de données de réglages modifiant son fonctionnement. Il s'agit du réglage des variables dans l'outil, désignant également le fait de spécifier les options offertes par la boîte à outils. Les paramètres sont les options de l'outil permettant de choisir entre différents traitements ou modes de fonctionnement. Ils désignent une option de procédé ou de traitement. La nuance entre argument et paramètre est variable en fonction du contexte et du langage de programmation utilisé. Souvent, le paramètre peut être modifié et/ou passé à une routine et récupéré, mais pas l'argument (traité souvent aussi d' « option »). De façon ambiguë, le paramètre désigne parfois une donnée communiquée au système (afin qu'il la traite).

Le paramétrage est une technique favorisant la réutilisation des fonctionnalités. Il fait intervenir les types paramétrés, également qualifiés souvent de génériques, et de templates. Cette technique permet d'utiliser un type, sans avoir à spécifier tous les autres types qu'il utilise. Les types non spécifiés sont fournis comme paramètres lors de l'utilisation. Par exemple, une classe Liste peut être paramétrée avec le type d'éléments qu'elle contient. Les types paramétrés offrent un autre moyen pour combiner les comportements des systèmes (orientés objet).

2.3.2 Composition

Il s'agit d'une solution alternative à l'héritage d'objet (ou de classe). L'héritage permet de définir l'implémentation d'un objet (ou d'une classe) à partir de l'implémentation d'autres objets (ou classes). Avec l'héritage, le contenu des objets parents est généralement visible aux sous-objets. Dans le cas de la composition, on obtient de nouvelles fonctionnalités par assemblage d'objets pour en construire de plus complexes. La composition d'objets impose que les objets entrant dans la composition aient des interfaces bien définies. Aucun détail interne des objets n'est visible. Elle est définie dynamiquement, à l'exécution, par l'introduction dans des objets, de références à d'autres objets. Elle nécessite que les objets respectent rigoureusement les interfaces des uns et des autres, ce qui, par conséquent, impose une conception soigneuse des interfaces, qui n'interdise pas l'utilisation d'un objet avec une grande variété d'autres. Mais il y a un bénéfice, du fait que l'accès aux objets ne peut se faire que par l'intermédiaire de leur interface, il n'y a pas de rupture de l'encapsulation. Chaque objet peut être, lors de l'exécution, remplacé par un autre, pourvu que celui-ci ait la même fonction. De plus, comme le développement d'un objet est codé en termes d'interfaces d'objets, il y a beaucoup moins d'interdépendance au niveau du code. Dans la composition, les classes et hiérarchies de classes évoluent très peu en nombre et restent facilement gérables. Une conception fondée sur la composition d'objets contiendra plus d'objets (mais toutefois moins de classes), et le comportement du système dépendra de leurs relations, au lieu d'être défini dans une seule classe.

Idéalement, on ne devrait pas avoir à créer de nouveaux composants pour effectuer une réutilisation. On devrait pouvoir obtenir toutes les fonctionnalités nécessaires, par le simple assemblage de composants existants en utilisant de la composition d'objets. Mais c'est rarement le cas, car l'ensemble de composants disponibles en pratique, n'est jamais tout à fait assez riche. La réutilisation à travers l'héritage facilite la création de nouveaux composants, à partir d'anciens. De ce fait l'héritage et la composition d'objets se complètent.

2.3.3 Délégation

La délégation est un moyen qui fait de la composition un outil de réutilisation aussi puissant que l'héritage. Dans la délégation, deux objets sont impliqués dans le traitement d'une requête : un objet récepteur qui délègue les opérations à son délégué. C'est un comportement analogue à celui des sous-classes, qui défèrent les requêtes à leurs classes parentes. Dans l'héritage, une opération héritée peut toujours faire référence à l'objet reçu, par l'intermédiaire de la variable membre (*this* en Java). Pour obtenir le même effet avec la délégation, le récepteur passe sa propre référence à son délégué, afin que l'opération déléguée puisse s'adresser à lui directement.

L'avantage principal de la délégation est de permettre facilement la combinaison des comportements à l'exécution, et de modifier la façon dont ils sont combinés. L'efficacité de la délégation dépend du contexte et de l'expérience qu'on a de son utilisation. La délégation

fonctionne au mieux lorsque son utilisation est très conventionnelle, c'est-à-dire dans les modèles standard. C'est un exemple poussé à l'extrême de la composition d'objets. Elle montre que l'on peut toujours remplacer l'héritage par la composition d'objets comme mécanisme de réutilisation de code.

2.3.4 Redéfinition

La redéfinition est un mécanisme de développement par lequel on réécrira totalement un objet ou une classe initiale. On décrit de nouveau, sur la base de l'original, la structure interne, les opérations, et les relations statiques qui doivent exister. Il est parfois utile de simplement ajouter un comportement au comportement original. Il est possible, en programmation orientée objet, de faire appel au constructeur original (principal) dans le corps de l'objet ou de la classe qui le redéfinit. En y ajoutant de nouvelles instructions, l'objet original est doté de comportements supplémentaires.

Le concept de redéfinition est un concept différent de celui de l'héritage qui est un des concepts les plus importants de la programmation orientée objet car il conditionne irréversiblement la façon selon laquelle un code est écrit. L'héritage est un mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui proférant ses propriétés et ses méthodes. La redéfinition est faite en fonction d'une classe existante en modifiant ou en réécrivant ses propriétés et/ou méthodes ainsi que ses comportements. Dans l'héritage, les classes héritées forment une hiérarchie descendante, au sommet de laquelle se situe la classe de base (superclasse). Dans la redéfinition, la classe redéfinie n'est pas forcément en dessous de la classe origine, elle peut être au même niveau. Une classe redéfinie ne possède pas toujours l'ensemble des propriétés et des méthodes de la classe origine. La redéfinition peut se servir des avantages de la hiérarchie des classes permettant ainsi une réutilisation aisée des composants existants et l'ajout de comportements nouveaux. La modification de la classe origine n'implique pas automatiquement la modification des classes redéfinies.

2.3.5 Dérivation

La dérivation est un principe permettant de créer une nouvelle classe à partir d'une classe existante. Le terme « dérivation » provient du fait que la classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive). L'intérêt majeur est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles qu'elle a héritées. Par ce moyen, on crée une hiérarchie de classes de plus en plus spécialisées. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière, il est possible de construire des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir. La dérivation permet de définir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui leur sont passés. Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objets, en les masquant par une interface commune (qui est de la classe de base de laquelle dérivent les autres).

2.3.6 Mécanismes d'extension utilisés

Il y a des différences importantes entre ces techniques. La composition d'objets permet de modifier à l'exécution la combinaison des comportements, mais elle nécessite aussi

des références d'adressage supplémentaires, et peut s'avérer peu efficace. L'héritage permet de définir un comportement par défaut pour les opérations, qui peut être surchargé par des sous-classes. Le paramétrage permet de modifier les types que peuvent utiliser les classes. Mais ni l'héritage, ni le paramétrage ne peuvent êtres modifiés à l'exécution. C'est la conception et les contraintes de l'implémentation qui déterminent les meilleures approches.

Notre choix de boîte à outils a porté sur Swing du langage Java, et cela pour de multiples raisons liées à cette bibliothèque (sous-section 3.4.1). Pour nous, il n'est pas possible d'imposer une technique particulière de modification de cette boîte à outils. Le choix de la technique d'extension des widgets est fonction de l'utilisation de chacun d'eux. Le choix est guidé par les composantes principales de l'extension en elle-même. Nos composantes principales sont basées sur les principes de base de la PsE, l'enregistrement et le rejeu. Ces fonctionnalités doivent être utilisées par les développeurs de la plus simple façon que possible. Les techniques les plus courantes pour la réutilisation de fonctionnalités dans un système objet sont l'héritage de classe et la composition d'objets. La facilité d'implémentation des widgets du système a pour origine l'héritage. En plus de l'héritage, nous devons bénéficier de la réutilisation des objets Swing, propriété que Swing même a tirée d'AWT (Abstract Windows ToolKit). La redéfinition de certaines classes sera nécessaire. Enfin, pour obtenir des événements génériques et faciles à manipuler, la dérivation doit être mise en place.

Nous avons utilisé une combinaison de plusieurs techniques d'extension afin de pouvoir fournir des modules plus maniables et légers à l'exécution. Cette combinaison de techniques n'est pas un handicap, mais plutôt un plus car cela confère à notre système une large ouverture pour les développeurs et son extension n'exige aucune technique particulière.

2.4 Programmation orientée aspect

Les développeurs d'application ont généralement plusieurs préoccupations à prendre en compte dans leurs développements. Ces préoccupations peuvent être divisées en deux catégories : les préoccupations fonctionnelles, qui correspondent aux cœurs de métier de l'application, et les préoccupations non fonctionnelles ou techniques liées à l'environnement d'exécution. Le principe de séparation des préoccupations cherche à rendre indépendantes ces préoccupations afin d'améliorer la modularité des applications. La programmation orientée objet a permis d'atteindre un certain degré d'indépendance sans pour autant casser totalement les liens entre les préoccupations. Ainsi, les préoccupations fonctionnelles restent encore dépendantes des préoccupations non fonctionnelles.

La programmation orientée aspect (POA) ou Aspect-Oriented Programming (AOP), (Renaud et al., 2004) est un paradigme de programmation relativement récent, dont les fondations ont été définies au centre de recherche de Xeros à Palo Alto au milieu des années 1990. Elle a émergé suite à différents travaux de recherche dont l'objectif était d'améliorer la modularité des applications pour faciliter la réutilisation et la maintenance. La POA cherche à améliorer la séparation des préoccupations en modularisant les éléments transversaux des applications. Bon nombre de préoccupations, notamment techniques, étant transversales à une application, elles sont modularisées sous forme d'aspects. La POA ne remet pas en cause les autres paradigmes de programmation, comme l'approche procédurale ou l'approche objet, mais les étend en offrant des mécanismes complémentaires pour mieux modulariser les différentes préoccupations d'une application. Elle ne remet pas non plus en cause les applications existantes. Elle fournit des outils permettant de les étendre sans toucher à leur code.

Nous définissons la POA avant de présenter la technique d'implémentation à partir de ce paradigme.

2.4.1 Définition de la POA

La programmation orientée aspect est un nouveau paradigme informatique. Par paradigme, nous entendons un ensemble de principes qui structurent la manière de modéliser les applications informatiques et, en conséquence, la façon de les développer. La POA introduit un nouveau concept, l'aspect (Renaud et al., 2004). La POA et le concept d'aspect apportent une nouvelle façon de structurer les applications et d'écrire des programmes. Ce paradigme cherche à améliorer la séparation des préoccupations en modularisant les éléments transversaux des applications. La POA complète la programmation orientée objet en apportant des solutions à deux challenges que sont l'implémentation de fonctionnalités transversales et le phénomène de dispersion du code. Les fonctionnalités transversales traduisent les corrélations dans des ensembles (des entités cohérentes, les classes en objet) bien que ces derniers soient bien indépendants. C'est typiquement le cas des contraintes d'intégrité référentielle. Le phénomène de dispersion du code en programmation orientée objet est le fait que l'implémentation d'une méthode est localisée dans une classe, tandis que son invocation ou utilisation est dispersée.

Défini en 1996 par Gregor Kiczales et son équipe du centre de recherche Xerox PARC de Palo Alto en Californie, ce concept a été rapidement mis en œuvre dans un langage de programmation, AspectJ, dont les premières versions ont été disponibles en 1998. Depuis cette date, AspectJ est resté le langage de POA le plus utilisé. AspectJ étend le langage Java avec de nouveaux mots-clefs permettant de programmer des aspects. L'aspect est un concept général, qui, comme l'objet, peut être appliqué à différents langages. C'est une entité logicielle qui capture une fonctionnalité transversale à une application. Un aspect est une fonctionnalité à mettre en œuvre dans une application (la sécurité, la persistance, la gestion de traces, etc.), dont l'implémentation comprendra les données et les traitements relatifs à cette fonctionnalité. L'apport de la POA est de fournir un moyen de rassembler une fonctionnalité dans un aspect du code, qui, autrement, serait dispersé au sein d'une application.

En POA, une application comporte des classes et des aspects. Un aspect se différencie d'une classe par le fait qu'il implémente une fonctionnalité transversale à l'application, c'est-à-dire une fonctionnalité qui, en programmation orientée objet ou procédurale, serait dispersée dans le code de cette application. La présence de classes et d'aspects dans une même application introduit donc deux dimensions de modularité : celle des fonctionnalités implémentées par les classes et celles des fonctionnalités transversales, implémentées par les aspects. En plus de l'aspect, la POA se pose sur d'autres concepts comme les notions connexes de coupe, de greffons (en anglais advice), de points de jonction et d'introduction. La coupe définit le caractère transversal de l'aspect. Elle désigne un ensemble de points de jonction. Les greffons quant à eux fournissent le code associé à l'aspect. C'est un bloc de code définissant le comportement d'un aspect. Le point de jonction est le point d'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés. En gros, les coupes désignent des points de jonction dans l'exécution de l'application (appel de méthode, exécution, lecture d'attribut, etc.), tandis que les codes advice ajoutent du code avant ou après ces points. Le mécanisme d'introduction permet d'étendre le comportement d'une application en lui ajoutant des éléments, essentiellement des attributs ou des méthodes. Le terme introduction renvoie au fait que ces éléments sont introduits, c'est-à-dire ajoutés à l'application.

Au delà d'AspectJ, la POA rencontre depuis 1998 un engouement important dans le milieu de la recherche. Cela a donné lieu à l'émergence de nombreux autres langages et outils. La plupart d'entre eux sont construits autour du langage Java. Outre AspectJ, c'est le cas de JAC (Java Aspect Components), de JBoss AOP et d'AspectWerkz.

Cependant, rien de la POA n'est spécifique ni de la programmation orientée objet en général, ni de Java en particulier. On trouve ainsi des outils de POA plus ou moins avancés en C, C++, C# ou Smalltalk. Bien que le domaine de la programmation orientée aspect commence à être mature, des solutions sérieuses pour les étapes amont de conception ainsi qu'un processus de développement orienté aspect restent à définir.

2.4.2 Implémentation de la POA

Il existe deux approches permettant d'implémenter le concept que représente la POA. La première consiste à dire qu'il faut absolument séparer le code des aspects du code sur lequel nous voulons les appliquer (appelé code de base). Un langage tiers permet d'établir les relations exactes qui existent entre le code de base et les aspects. Flexible, cette approche est parfois jugée un peu trop « magique » : le développeur d'une classe particulière sera étonné de la voir se comporter différemment de ce qu'il a véritablement écrit, et se posera donc sans cesse la question « quel aspect s'applique à ma classe ? », d'où la nécessité d'avoir des outils efficaces de gestion de la traçabilité. Cette orientation est suivie par les projets du type AspectJ.

L'autre option serait de dire qu'il faut d'abord développer les aspects, certes, mais qu'il est ensuite nécessaire de « masquer » le code en utilisant une construction syntaxique adaptée. L'avantage que procure cette technique est d'expliciter dans le code la relation avec les aspects : du coup, le développeur n'est pas sans cesse en train de se demander « dois-je implémenter cela, ou un aspect s'en chargera-t-il ? » : il voit les aspects auxquels il fait appel (ce qui n'ôte pas le besoin de comprendre leur sémantique). Cette seconde approche est suivie par Microsoft avec l'intégration en standard des Attributes .NET[16], et par certains outils de génération de code du type XDoclet[17] en Java (cette approche semble ne plus être de la POA).

2.4.3 AspectJ et notre implémentation

La POA est partie du constat simple selon lequel quel que soit le mode de développement adopté, il est difficile de ne pas écrire de code redondant. L'objectif de la POA est donc de fournir une technique apte à factoriser ce que les autres techniques obligent à répéter à travers le code des projets. Notre système d'implémentation n'a pas pour objectif la factorisation du code, mais la possibilité de récupérer l'ensemble des interactions de l'utilisateur. On pourrait utiliser la technique de factorisation du code pour peut-être essayer une amélioration de notre implémentation. Le principe de base avec AspectJ diffère du nôtre non seulement par le fait que nous automatisons des actions de l'application et non pas que nous factorisons le code de l'application, même si AspectJ permet de tracer l'exécution de méthodes d'une application. Il ne s'agit en effet que de traçages d'exécution, ce qui nous amène d'ailleurs à nous pencher sur la POA. Le traçage d'aspect n'est que de la génération automatique de codes insérés à certains points d'exécution du système développé. Il produit

[16] Il s'agit de ces petits marqueurs que l'on place en en-tête d'une déclaration (de méthode, de classe, d'attribut, de namespace…) et qui permet de donner à cette déclaration une sémantique enrichie.

[17] Outil du monde Java permettant de générer du code et des descripteurs de déploiement relatifs aux EJB, Servelets et JSP (entre autres). http://xdoclet.sourceforge.net/xdoclet/index.html

un code qui peut être difficile à analyser (problème de génération automatiquement) lors des phases de mise au point des logiciels (déboguage, test). Cette difficulté est du même ordre que celle apportée par toute décomposition non linéaire (fonctionnelle ou objet par exemple) du système.

Dans AspectJ, l'insertion d'intermédiaires capables d'intercepter chaque invocation de méthode est plus lourde dans un système que l'ajout de quelques instructions (simples et légères) d'automatisation ou de récupération d'objets. Les objets sur lesquels il faut installer un message en POA doivent tous hériter d'une seule et même classe. Or cette classe hérite d'une autre. Ce qui signifie que dans une application distribuée, tous les objets sur lesquels on aura apposé un aspect par ce biais seront passés par référence, et non par valeur. Cela a un impact non négligeable sur l'architecture de ladite application, ainsi que sur ses performances globales. Dans notre cas, la connaissance des différentes méthodes ou classes redéfinies (presque toujours égales aux originaux de Swing, avec une extension ou une surcharge de méthodes) est le seul besoin. Il est nécessaire d'acquérir les connaissances de l'aspect de la même manière qu'on a acquise celles de l'objet. Le bon usage des aspects prend beaucoup de temps (Renaud et al., 2004).

3 Implémentation de PbDToolkit

Les systèmes de programmation sur exemple sont complexes et difficiles à construire. Nous visons à faciliter la création d'application mettant en œuvre des techniques de la PsE, en définissant PbDToolkit, une extension de la bibliothèque Swing de Java, maîtrisée par nombre de développeurs.

Dans le concept de PbDToolkit, certaines fonctionnalités découlent de Swing. En effet, PbDToolkit est entièrement réalisée en modules Swing. Les différentes fonctionnalités seront expliquées lors de la présentation de l'implémentation de PbDToolkit. Notre architecture est liée à Swing. Nous présenterons l'architecture Java pour montrer l'intégration du système par rapport au développement d'application.

L'implémentation du principe d'enregistrement/rejeu est fondée sur l'utilisation des files d'événements, composants par lesquels transitent les événements lors de l'utilisation d'une application. Nous décrirons les files d'événements dans la section sur l'implémentation du concept (confère § 3.5.3).

Dans cette section, nous décrivons le mécanisme de base sur lequel s'appuie PbDToolkit. Ensuite, nous présentons les modes de son utilisation ainsi que son architecture. Puis, arrive la description des principes généraux de notre construction. Enfin, nous présentons la technique de réalisation des modules et l'implémentation des composants.

3.1 Mécanisme d'enregistrement et de rejeu

Le mécanisme d'enregistrement et de rejeu est un ensemble de fonctionnalités qui permet à une application graphique interactive de mémoriser les événements générés suite aux actions de l'utilisateur lors d'une exécution et de les rejouer (en général, à une prochaine exécution). Ce mécanisme est le principe phare de la PsE. Il se définit par la définition de l'enregistrement et de celle du rejeu.

L'enregistrement consiste à récupérer les données affichées à l'écran (caractéristique d'une fenêtre, position de clic de la souris, etc.), les commandes entrées par l'utilisateur, et de les sauvegarder. Le rejeu consiste à refaire les commandes entrées par l'utilisateur et les

actions qui se passent à l'écran, et qui ont été mémorisées lors de l'enregistrement. Des librairies sont modifiées pour capturer les événements et des éléments sont sauvegardés pour rejouer ces mêmes événements. Stocker ou sauvegarder les événements peut se présenter selon la forme préconisée par le but recherché en fonction de l'usage futur. Les événements sont sauvegardés avec les informations essentielles et nécessaires.

Dans un système, les événements sont placés dans une file d'attente des événements que les applications scrutent en continu. Les systèmes d'exploitation gèrent les événements de manière asynchrone. Le mécanisme d'enregistrement/rejeu utilise cette gestion et surtout la file des événements pour mettre en place ses principes (Sanou et al., 2006a).

Le mécanisme d'enregistrement et de rejeu fonctionne en deux phases. Lors de la phase de capture (1), l'utilisateur lance son application et réalise une tâche. Des événements sont générés à la suite de cette utilisation. Ceux-ci sont capturés et sauvegardés selon le format défini. Pour la phase de rejeu (2), l'utilisateur lance à nouveau l'application. Les événements de l'utilisateur capturés et sauvegardés sont inhibés et reproduits. L'application est toujours celle de l'utilisateur et le contexte d'exécution est toujours (en général) préservé. Pour un clic sur un bouton par exemple, un événement est produit. Celui-ci est capturé puis, reproduit. En plus de cette reproduction, la fonction (de l'utilisateur) associée au bouton sera elle aussi reproduite.

Plusieurs travaux (Cypher, 1993c) ont défini les besoins de couplage entre les applications hôtes c'est-à-dire incluant la PsE, et les systèmes de PsE eux-mêmes, en se focalisant le plus souvent sur la généralisation. Or, dans l'optique d'être indépendant des applications et pouvoir fournir des outils utilisables de manière générale, il ne faut s'intéresser qu'aux opérations de base de la PsE, caractérisant le mécanisme d'enregistrement et de rejeu. C'est ce que nous avons proposé en réalisant des modules génériques pour l'enregistrement et le rejeu.

3.2 Différents modes d'utilisation de PbDToolkit

PbDToolkit peut être utilisé de deux façons différentes. En effet il est possible de lancer le système de façon externe à toute application, ou bien de façon interne. Dans ce dernier cas, le programmeur peut choisir d'utiliser des outils prédéfinis, ou bien d'intégrer totalement les fonctionnalités de PbDToolkit à son application.

3.2.1 Fonctionnement externe

Dans son mode externe, PbDToolkit fonctionne de façon assez semblable à l'outil Jacareto[18] que nous avons précédemment décrit. L'utilisateur d'une application a la possibilité de bénéficier des fonctionnalités d'enregistrement/rejeu de toute application. Notons que la seule contrainte est le fait que cette application soit écrite en Java/Swing.

Le principe de fonctionnement est simplifié au maximum : l'utilisateur lance son application au moyen de l'interface représentée sur la Figure 33. Cette interface permet à PbDScript de se lier à l'application demandée, et par la suite d'exercer les fonctions d'enregistrement/rejeu.

[18] http://jacareto.sourceforge.net/wiki/index.php/Whitepapers

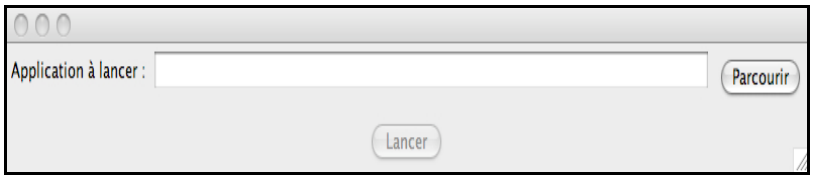

Figure 33 : Interface de chargement d'application de PbDToolkit en mode externe

Une fois l'application lancée, et à tout moment de son utilisation, il est possible d'activer l'enregistrement des interactions, de l'arrêter, d'enregistrer ou de rejouer une séquence (Figure 34).

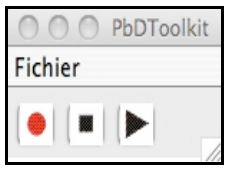

Figure 34 : Interface de contrôle de l'enregistrement/rejeu

Dans ce mode d'exécution externe, l'utilisateur ne peut qu'enregistrer des événements de bas niveau. Ces derniers sont stockés sous forme d'une liste d'événements que l'utilisateur à la possibilité de faire ré-exécuter à l'identique, après la phase d'enregistrement. À la fin de l'enregistrement, il a la possibilité de sauvegarder le fichier contenant les événements interceptés (menu « Fichier », action « Enregistrer »). Pour rejouer des événements sauvegardés, l'utilisateur charge d'abord le fichier d'événements sauvegardés (menu « Fichier », action « Charger »), puis lance l'application sur laquelle on effectue le rejeu. Si un événement ne peut être ré-exécuté sur l'application chargée, un message d'erreur s'affiche sur la console de sortie standard.

Ce mode d'enregistrement/rejeu est bien évidemment très pauvre, et sa description ici n'est faite que par souci d'exhaustivité.

3.2.2 Fonctionnement interne « clés en main »

L'intérêt principal de PbDToolkit réside dans la possibilité d'utiliser ses fonctionnalités depuis l'intérieur d'une application. Il s'agit alors d'utiliser des composants de PbDToolkit au cours du développement de l'application. Une grande liberté est offerte au **développeur** pour spécifier l'utilisation du contrôle des opérations. En effet, il a la possibilité d'afficher une interface de gestion standard (Figure 35) de PbDToolkit, qui permet d'effectuer des tâches standard d'enregistrement/rejeu, comme il peut spécifier à partir de sa propre implémentation les opérations à réaliser suivant ses besoins, ou à travers une interface de gestion personnalisée.

Tout comme pour le mode externe, lorsque le **développeur** offre la possibilité d'utiliser le module PbDToolkit dans sa version « clé en mains », **l'utilisateur** peut enregistrer les événements produits sur l'interface, et les ré-exécuter, avec ou sans sauvegarde en fichier. Un contrôle fin du rejeu peut être fourni à travers un réglage de la vitesse de rejeu (Figure 35).

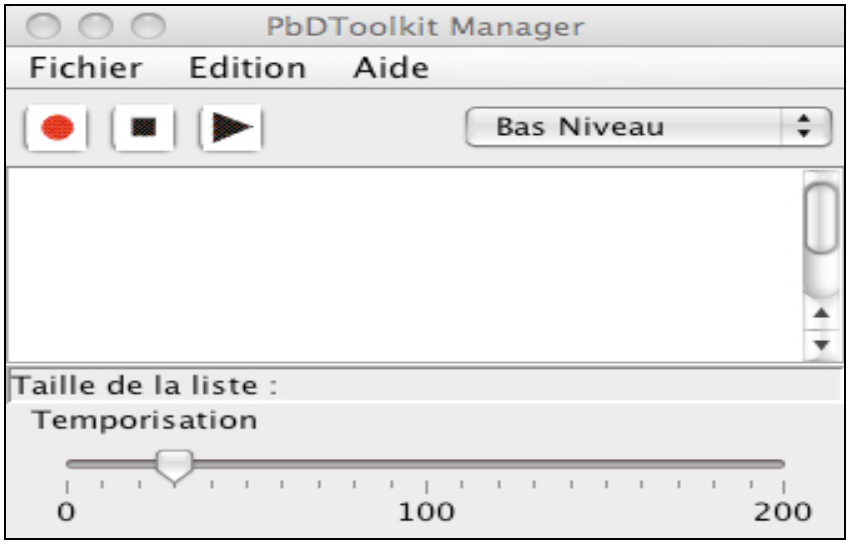

Figure 35 : Interface de gestion de PbDToolkit

Le mode programmé donne cependant accès à des fonctionnalités plus étendues, sous réserve que le programmeur le prévoit dans son développement. PbDToolkit est en effet capable d'enregistrer selon deux modes différents : l'enregistrement d'interactions de bas niveau et l'enregistrement d'interactions de haut niveau. Les événements de bas niveau sont les événements clavier, souris, ceux concernant les changements de focus, les modifications de composants, et les fenêtres. Ils correspondent à l'aspect lexical de l'interaction, et donc permettent d'effectuer de l'enregistrement/rejeu à ce même niveau, en référence à ce que nous avons présenté dans la section 2.2.2 du chapitre 1. Nous verrons dans la sous-section 3.5.3 de ce chapitre la raison de cette distinction, qui provient du mode de gestion des événements dans Swing.

L'enregistrement d'événements de haut niveau permet de s'intéresser à un autre mode d'enregistrement/rejeu. Il se réfère aux événements qui sont syntaxiquement pris en compte par les différents widgets, et qui permettent d'exprimer la sémantique de l'interaction au niveau des composants. Par exemple, on pourra travailler au niveau de l'interaction correspondant à un clic sur un bouton, à la sélection d'un item dans une liste, ou la modification d'un composant de texte. **L'utilisateur** peut ainsi préciser le niveau d'enregistrement qu'il souhaite. Il est possible à **l'utilisateur** de réaliser un mélange des niveaux d'enregistrement. Il peut commencer une exécution par un enregistrement haut niveau et la terminer à bas niveau.

L'effort de programmation nécessaire au **développeur** est minime s'il se contente de faire appel au *composant interactif de contrôle* de la Figure 35. Nous verrons dans la section 4.2 qu'il se résume à la substitution des composants Swing standard par ceux de PbDToolkit, et à la possibilité de lancer la boîte de dialogue de contrôle de l'enregistrement/rejeu.

3.2.3 Fonctionnement interne programmable

La fonctionnalité la plus originale de PbDToolkit consiste en la mise à disposition sous forme d'interface de programmation d'application (API) de toutes les fonctionnalités de contrôle de l'enregistrement/rejeu. En effet, grâce à cette API, le **développeur** d'applications peut utiliser le mécanisme de base, et en contrôler finement les différentes fonctions. Il peut ainsi fournir une interface équivalente au composant de contrôle, mais mieux intégrée à son application. Surtout, il peut contrôler la phase de rejeu pour la gérer selon ses besoins. Il peut

ainsi permettre une exécution pas à pas, programmer un comportement tel que celui présent dans Eager (Cypher, 1993a), ou encore fournir une représentation textuelle compréhensible par l'utilisateur des enregistrements produits.

L'élément clé est ici le contrôleur de PbDToolkit. Ses fonctionnalités principales sont :

- L'enregistrement des événements (fonction « *enregistrer* ») permet de lancer l'enregistrement effectif des événements lors de l'exécution. Par défaut c'est le bas niveau qui est considéré.
- La gestion du niveau d'enregistrement est assurée par la fonction « *setNiveauEnregistrement(int i)* », avec un entier i comme paramètre (prenant la valeur 1 pour le bas niveau et 2 pour le haut niveau).
- Une fonction «*stop* » permet de passer en mode d'exécution normale, sans enregistrement.
- Le rejeu s'effectue à l'aide de la fonction « *rejouer* » qui se charge de relancer les événements interceptés dans le système.
- Lors du rejeu, une fonction « *pause* » autorise un arrêt de la ré-exécution et la fonction « *reprise* » relance l'exécution à partir du point d'arrêt.
- L'obtention de la liste des événements est assurée par la fonction « *getListe* ». À partir de la liste, on dispose des fonctions de gestion standard d'une liste en Swing (ajouter un élément, supprimer un élément, parcourir la liste, etc.). Cette fonction peut permettre au programmeur, par exemple, d'ajouter une fonction de visualisation du « programme ».
- On peut enregistrer la liste des événements interceptés (contenu de la liste) dans un fichier texte à partir de la fonction « *enregistrerListe* ». Aussi, la fonction « *chargerListe* » permet d'ouvrir un fichier d'événements et transférer le contenu dans le vecteur actif d'événements afin de permettre un rejeu.
- Enfin, la fonction « *setVisibleControle(Booléan)* » permet d'autoriser ou non l'interface de gestion standard de PbDToolkit.

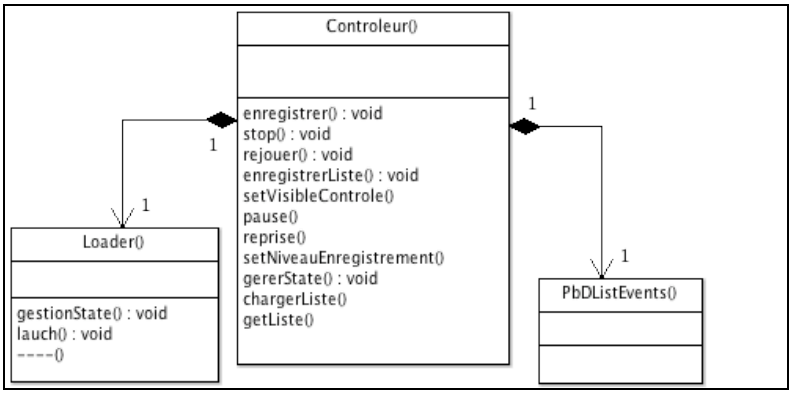

Figure 36 : Représentation UML des classes principales de gestion de PbDToolkit

3.3 Architecture de coopération

Qu'elle fonctionne en mode interne ou externe, PbDToolkit correspond à un module autonome qui communique avec l'application. Une application implémentée à l'aide de PbDToolkit fonctionne ainsi suivant l'architecture logicielle représentée par la Figure 37 ci-dessous.

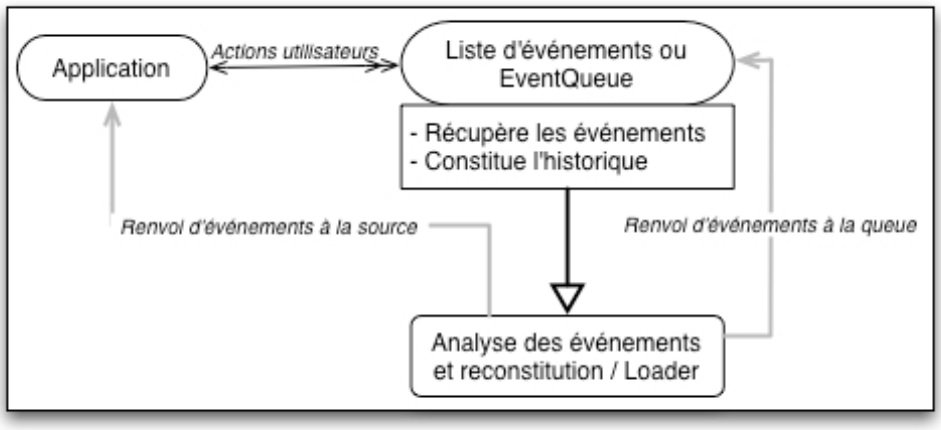

Figure 37 : Schéma d'architecture de coopération d'application incluant la PsE

D'un point de vue technique, le système repose sur l'interception des événements, et leur stockage dans une liste (partie enregistrement) et la régénération de ces événements (partie rejeu). L'interception des événements s'effectue par le biais de la redéfinition de la classe gérant la file des événements. Nous reviendrons plus en détail dans la sous-section 3.5.3 sur les techniques précises d'implémentation.

D'un point de vue plus global, on distingue trois parties essentielles, en plus des composantes fonctionnelles (sans PsE) contenues dans la partie « Application » du schéma. La première concerne la reformulation de la file des événements. Elle sert à intercepter tout événement transitant dans la file des événements et permet la mise en œuvre (réalisation, reconstitution) de tout événement régénéré pour être rejoué. Cette partie joue aussi le rôle de filtre d'événements car il n'est pas toujours nécessaire d'enregistrer la totalité des événements.

La deuxième partie de l'architecture s'occupe principalement de la partie « enregistrement ». C'est elle qui récupère les événements transitant par la première partie, extrait les différents paramètres de l'événement, et construit l'historique des événements. Elle est particulièrement importante, car c'est cette partie qui va permettre une certaine forme de généralisation.

La troisième partie porte sur la détermination ou vérification du type des événements stockés et leur reconstitution pour le rejeu. Elle permet le chargement des événements enregistrés au niveau de la file des événements, mais surtout est responsable de tout l'aspect rejeu. Pour cela, en fonction du niveau de l'enregistrement, elle renverra les événements reconstitués directement à la source (fonctionnement de haut niveau) ou à la file d'événements substituée par PbDToolkit (fonctionnement de bas niveau).

La communication est assurée par des méthodes de transmission de messages au sein du système. Cela peut être sous forme de paramètres ou encore sous forme de notification, en fonction des composantes ou des modules.

3.4 Principes généraux de construction

PbDToolkit est un concept développé à partir de l'extension de la boîte à outils Swing. Sa construction repose sur le principe objet de Java. Sa construction est donc fortement liée à l'implémentation de Swing.

3.4.1 Base de construction : la bibliothèque Swing

Java[19] a inclu, dans un premier temps, la boîte à outils AWT (*Abstract Window Toolkit*) pour programmer des interfaces graphiques. Devant les problèmes posés par la portabilité insatisfaisante de cette dernière, les concepteurs de Java ont proposé une deuxième version, dénommée Swing, qui comporte de nombreux modules spécialisés. Ces deux API peuvent être utilisées pour développer des applications. De fait, Swing s'appuie sur AWT pour une partie de ses fonctionnalités, en particulier pour la gestion des événements et surtout la couche graphique de base (Figure 38).

La mise en œuvre d'une interface graphique à l'aide de la bibliothèque Swing s'appuie sur les concepts suivants :

- des composants d'interface utilisateur ou widgets qui permettent à l'utilisateur de visualiser et/ou de saisir des informations ;
- des conteneurs, qui contiennent les composants affichés, et ont pour rôle de les structurer ; ces conteneurs sont soit des fenêtres ou des boîtes de dialogue, soit des conteneurs intermédiaires aux fonctionnalités variables, allant des simples panels aux conteneurs permettant une gestion dynamique (panels à onglets, panels gérant le défilement ou scroll, panels gérant un découpage dynamique de zones, etc.) ;
- des gestionnaires de mise en page, ou Layout, qui déterminent le positionnement logique des composants dans un conteneur ;
- un système de gestion des événements pour associer des traitements aux actions de l'utilisateur sur les composants.

Swing propose de nombreux composants dont certains possèdent des fonctions étendues, une utilisation des mécanismes de gestion des événements performants et une apparence modifiable à la volée. Les composants de Swing forment une hiérarchie parallèle à celle de AWT et Swing possède plusieurs packages. La classe de base d'une application est la classe *JFrame*. Les concepts particulièrement utile dans notre approche sont les composants d'interface utilisateur (ou widgets) et le système de gestion des événements. Ces deux concepts sont la base de prise en compte des interactions (utilisateur et système) dans l'application.

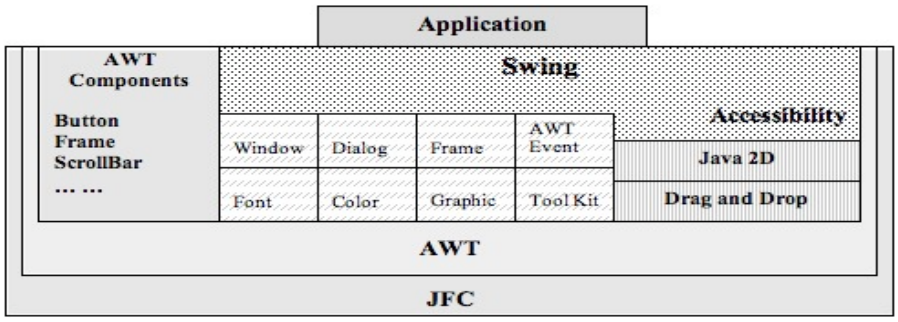

Figure 38 : Architecture de Java avec AWT et Swing (extrait de http://java.sun.com)

Les composants Swing d'une fenêtre sont entièrement dessinés à l'intérieur d'une fenêtre vide (package *java.awt* avec les classes *Frame* et *Graphics*) à l'aide des fonctionnalités graphiques offertes par AWT. Afin de garantir un *Look & Feel* constant pour toute application développée à l'aide de Swing, le package *javax.swing.plaf* est en charge de

[19] http://java.sun.com/

la définition précise de l'apparence et du comportement des widgets sur chaque système de fenêtrage.

Dans Swing, la classe « *Graphics* » contient les outils nécessaires pour dessiner. Cette classe est abstraite. Les instances nécessaires sont fournies par le système d'exploitation qui instanciera via la machine virtuelle une sous-classe de « *Graphics* » dépendante de la plate-forme utilisée. Dessiner tous les composants et reproduire leur comportement pour chaque *look and feel* a permis aux concepteurs de Swing de contrôler complètement chacun des composants mais également aux développeurs de personnaliser éventuellement ces composants par dérivation. La conception de Swing a bénéficié de la réutilisation des composants du système à partir de AWT. En privilégiant cette réutilisation des composants, nous bénéficions de la richesse des fonctionnalités proposées par le système graphique de Java.

3.4.2 Structure générale de PbDToolkit

Pour ajouter des fonctionnalités supplémentaires à des composants, deux pistes sont possibles : (1) soit implémenter ces fonctionnalités dans la bibliothèque dynamique Swing en faisant directement appel à celles du système qui existent, ou en écrivant les fonctionnalités qui manquent (par la délégation) ; (2) soit écrire ces fonctionnalités en Swing en faisant appel aux méthodes de dessin existantes d'AWT, par exemple en créant des sous-classes des classes d'AWT. La première méthode est l'utilisation des techniques de la réutilisation, de la composition ou de la dérivation. Un usage combiné (mélange des techniques) est possible. Par contre, la deuxième piste porte sur la redéfinition de composants (Swing à partir d'AWT) surtout.

Nous avons choisi de nous engager sur la première voie, à savoir l'enrichissement des packages *java.awt* et *javax.swing*, plutôt que de recréer toutes les classes de composants graphiques dans le package *javax.swing* en réutilisant AWT. En effet, recréer toutes les classes s'avère lourd et fastidieux, et nous aurions manqué à une large interopérabilité avec les classes existantes. Nous exposons ainsi les fonctionnalités de chaque composant enrichi grâce à des classes indépendantes du système d'exploitation. Constituée de classes Swing, PbDToolkit propose les mêmes concepts (mais évolués) nécessaires à la mise en œuvre d'une interface graphique que la bibliothèque Swing.

PbDToolkit offre des composants redéfinis à partir des mêmes composants types de Swing. En effet, pour réaliser une interface graphique, on utilise les composants nécessaires de PbDToolkit de la même façon que si l'on employait ceux de Swing. La redéfinition des composants n'est guère un souci puisqu'on ne fait appel qu'aux méthodes et fonctions existantes afin de satisfaire nos besoins. Seulement, cela reste très dépendant et très lié à la bibliothèque de base. Dans Swing, il existe plusieurs types d'événements. Afin de pouvoir utiliser une même structure de traitement, des événements ont été redéfinis suivant les caractéristiques importantes à l'exécution de l'action sur l'interface. Ainsi, des composants redéfinis à partir d'une base et héritant des composants de Swing permettent d'obtenir des événements particuliers redéfinis sur les événements standard de Swing. Dans la même logique la gestion de ces événements ne peut se faire que d'une façon particulière, d'où la mise en place d'un composant de gestion de liste d'événements. Si nous avons tout dérivé à partir d'un arbre d'héritage, il est nécessaire de permettre l'utilisation des événements dans le système pour le même résultat que les événements de la hiérarchie ordinaire de Swing. Ainsi, il est mis en place une file d'événements pour la transmission des événements lors de l'exécution.

Dans le système PbDToolkit, les objets dynamiques à instancier sont obtenus à l'exécution par chargement dynamique. Le modèle de base des objets est celui de Swing. Néanmoins, nous avons évité de créer une grande hiérarchie de classes des objets, qui répliquerait la hiérarchie de classes des fonctions (ou comportements).

Les opérations de base du système sont l'enregistrement et le rejeu des événements. Cela doit permettre la mise en place d'un objet essentiel, commun d'utilisation pour les composantes fonctionnelles ayant trait à ces opérations. En effet, le rejeu obtient ses ressources à partir des ressources sauvegardées lors de l'enregistrement. Il est donc nécessaire d'établir une unité commune afin de pouvoir satisfaire par la suite aux besoins de rejouer les interactions. L'enregistrement prend ses sources dans l'espionnage qui consiste à observer en permanence les interactions de l'utilisateur, à repérer des séquences dans les suites d'actions et à renseigner le système interne d'enregistrement. L'interaction utilisateur engendrant une action est un événement ou un ensemble d'événements dans le système. Une action peut être constituée de plusieurs événements.

Les événements sont donc représentés à l'aide d'un modèle simple, l'interface *PbDEvent* (Figure 39). Il permet d'encapsuler un événement comme un objet, autorisant le paramétrage des éléments par différentes actions. Il permet dans le système de construire des événements d'objets d'applications non spécifiés, en transformant l'événement lui-même en un objet. Cet objet peut être stocké et distribué comme les autres objets. Le fondement de ce modèle est une classe abstraite, qui déclare une interface pour l'exécution des opérations.

Les objets *PbDEvent* sont regroupés pour former l'historique. L'historique est constitué des événements utilisateur enregistrés. C'est une représentation des séquences d'actions (événements). Sa structure est dynamique, c'est-à-dire que l'ajout d'un nouvel événement n'entraîne pas une réinterprétation totale de l'historique et suppose que chaque événement peut avoir un lien avec le précédent. Cette technique est bien illustrée par AIDE avec sa structuration d'arbre hiérarchique des commandes. Le constructeur de l'historique est utilisé dans les classes événements comme un ensemble de méthodes et possède un point d'entrée. Pour ajouter un événement, on envoie un message à la classe mère de l'historique, avec l'événement à ajouter comme argument. Cette technique est utilisée en partie dans le système AIDE. Une application peut facilement étendre le constructeur en surchargeant une méthode d'ajout dans ses propres commandes. Ainsi, le comportement par défaut décrit dans les caractéristiques de départ peut être modifié pour répondre aux besoins particuliers d'une application.

L'espionnage (phase d'enregistrement) et la construction de l'historique vont de pair : tout ce qui est écouté est en général enregistré sauf exclusion explicite (soustraire du système d'enregistrement certains types d'événements à l'aide de condition ou de filtre). En Swing, l'information relative à un événement est encapsulée dans l'objet événement dérivant de la classe *java.util.EventObject*. De là on détermine la source ou l'émetteur de l'événement, l'écouteur ou la destination de l'événement ainsi que les propriétés de l'événement.

Une autre possibilité de mettre en place l'utilisation des objets précédemment définis est de manipuler le modèle interne des widgets. En effet, les éléments comme les boutons et les zones de texte de Swing ont été construits selon le modèle de conception MVC (Model View Controler). Cette stratégie est indépendante des éléments de la boîte à outils puisque, pour chaque élément, il faut créer un nouveau « Modèle Enregistreur ». Cela ne concerne que des éléments de très haut niveau comme les commandes du genre « Copier – Coller » à l'aide de raccourcis clavier. Dans ce modèle, la partie « Control » capte les actions de l'utilisateur et modifie le modèle interne de l'élément (partie Model) qui informe automatiquement l'aspect graphique de l'élément (partie View) de son changement. Par exemple, en appuyant sur un

bouton, le paramètre « appuyé » de la partie Model du bouton passe à vrai, puis ce même Model informe la partie View qu'elle doit prendre un aspect graphique « bouton enfoncé ». On va donc substituer la partie Model par une autre ayant les mêmes fonctionnalités avec en plus la possibilité d'enregistrer ses changements d'états. Ainsi, il suffit de modifier les paramètres du modèle interne des éléments de l'interface en parallèle avec les modifications enregistrées pour pouvoir rejouer les actions de l'utilisateur.

Dans tous les cas, rejouer une interaction de l'utilisateur s'effectue par la création d'une liste de séquences comportant les événements constituant l'interaction à ré-exécuter.

3.5 Principes d'implémentation

Après avoir décrit dans la section précédente les principes généraux de notre implémentation, nous allons dans cette section approfondir les fonctionnalités des différents constituants de PbdToolkit. Lorsque cela est nécessaire, nous expliciterons les choix d'implémentation réalisés.

3.5.1 Elément clé : l'événement

La classe « *PbDEvent* » permet de regrouper tous les événements dans une même enseigne qu'il soit de type « InputEvent » ou « EventObject » en général. Pour rester indépendant vis-à-vis du système, il est nécessaire d'avoir la main pour gérer les événements. C'est dans cette optique que cette classe a été définie afin de disposer des traces d'exécution propres. On ne peut utiliser la classe Event de Swing car seuls les événements de type *AWTEvent* sont traités à partir de cette classe, et non les autres types de *EventObjet*. Aussi, les événements lors de l'exécution dans le système sont dynamiques, voire éphémères, et ne sont donc pas conservés en l'état. Il s'agit d'une classe abstraite contenant les informations nécessaires à la reconstruction d'un événement quelconque. Ces informations portent sur l'identifiant de l'événement (*id*), la source de l'événement (*sourceClasse*), le temps de l'action de l'événement (*time*), la hiérarchie du composant source (*path*) et bien évidemment l'objet lui-même, son type (*obj*). La Figure 39 ci-dessous explicite la classe « *PbDEvent* ». Elle est la super classe pour les cellules (ou événements) qui servent à stocker les informations sur les événements. Ces événements peuvent être des événements de la classe *AWTEvent* et des autres types de la classe *EventObject*. Cette classe représente le constructeur principal pour l'enregistrement des événements.

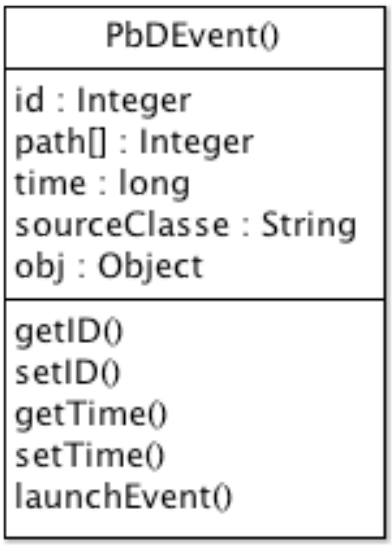

Figure 39 : La classe PbDEvent

Chacune des cellules représentant les commandes contient une caractéristique « date » et une caractéristique « identité ». Les commandes contiennent une méthode polymorphe nommée *launchEvent()* chargée de provoquer l'événement lui-même.

3.5.2 Structures de stockage

Les événements interceptés doivent êtres stockés afin de réaliser l'historique. Une liste de cellules appelées « *PbDListEvent* » est mise en place (Figure 40) dans ce but dans PbDToolkit, sous la forme d'un tableau dynamique. Les cellules contiennent les événements capturés et reconstitués. Le contenu du vecteur peut être sérialisé dans un fichier, en format XML. La structure du fichier de sauvegarde est modifiable, grâce à la description du document type.

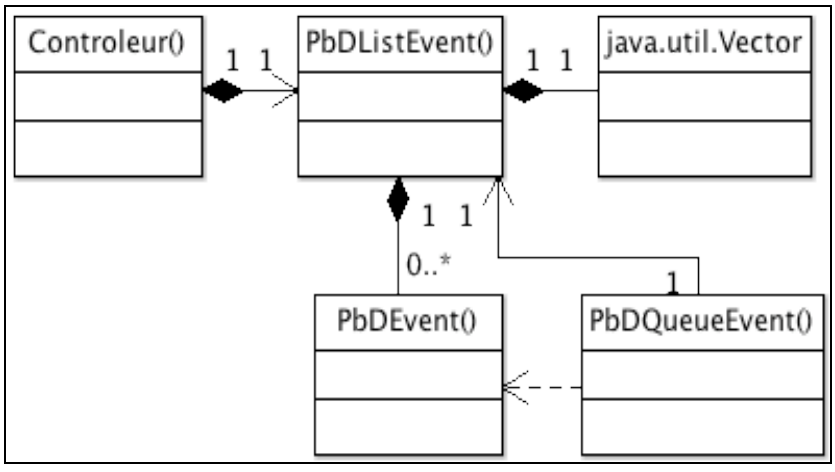

Figure 40 : Objets de coopération avec la classe PbDListEvent

3.5.3 Structures liées à l'enregistrement

Avec la technique liée à l'usage de la file des événements système au travers de la classe « *EventQueue* », une variante de cette file a été mise en place (comme justifié dans la

sous-section 3.4.2), avec en plus la capacité d'enregistrer dans la liste des commandes. Cette variante « *PbDEventQueue* » permet aussi de lancer les différents événements enregistrés. Elle est dérivée de *java.awt.EventQueue*.

« *PbDEventQueue* » capture les événements et les stocke dans la liste. Elle utilise la méthode *dispatchEvent* contenant un événement *java.awt.AWTEvent*, et la surcharge pour permettre la redéfinition des événements.

Pour intercepter les actions internes aux widgets, nous avons des méthodes définies dans certains composants, ayant comme capacité d'expédier directement un clone de l'objet à la classe *PbDListEvent*.

À la base du point de vue du développeur, les actions de l'utilisateur sont des événements matériels tels que la frappe d'une touche ou le déplacement de la souris. Les événements matériels sont des événements issus directement de l'action utilisateur à partir d'un dispositif physique (matériel). L'ensemble des événements produits à la suite des actions utilisateur sont soit des événements souris (*MouseEvent*), soit des événements clavier (*KeyEvent*). Le traitement des interactions consiste à abonner les composantes sources aux écouteurs que l'on souhaite utiliser pour faire le traitement. Cependant, la notion d'événement s'étend à des concepts ne concernant pas le matériel, par exemple la création d'une fenêtre, son apparition à l'écran, etc. Le système de fenêtrage génère des événements pour les applications en exécution. *WindowEvent* est un exemple de ce type d'événement Swing. Le concept de PbDToolkit permet de prendre en compte les *EventObject* qu'ils soient de type *AWTEvent* ou non.

Les événements sont gérés de manière asynchrone par les systèmes d'exploitation (les systèmes d'exploitation multitâches en général). Les systèmes d'exploitation placent les événements dans une file d'attente des événements que les applications scrutent en continu. Certaines applications possèdent leur propre file d'attente des événements. Mais l'ensemble des événements transite obligatoirement par la file d'attente des événements du système. Dans Java, Swing repose entièrement sur AWT. Outre le fait que Swing dessine ses composants dans un canevas AWT, il utilise également son système d'acheminement des événements. Le lancement d'une application Swing entraîne la création automatique de trois threads. Le premier est le « main de l'application » qui se charge d'exécuter la méthode *main()* de l'application. Le deuxième thread est le « *Toolkit thread* » dont le rôle est de recevoir les événements du système d'exploitation, par exemple un clic de souris, et de les transmettre au dernier thread, appelé « *event dispatching thread* ». Ce dernier est extrêmement important car il répartit les événements reçus vers les composants concernés et se charge d'invoquer les méthodes d'affichage. La Figure 41 présente l'acheminement de ces threads.

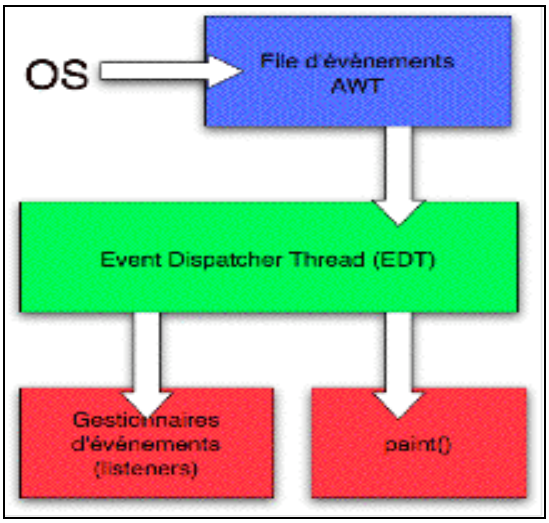

Figure 41 : Transmission des événements en Swing

Espionner les interactions de l'utilisateur revient à écouter les messages ou les événements de la file des événements. Mais comme les applications peuvent avoir leur propre file d'événements, il est aussi possible de n'écouter que cette file. Du fait que c'est le programmeur lui-même qui définit explicitement les abonnements des différents composants de l'interface graphique utilisateur, il lui est donc possible de choisir quels sont ceux qu'il souhaite soumettre au système d'interception ou d'enregistrement des événements. Cela nous offre donc la possibilité d'écoute directe soit par la file des événements, soit par la file de l'application.

3.5.3.1 File des événements du système

Nous avons vu que tous les événements transitaient par la file des événements avant d'être transmis à l'objet destinataire. En Swing (on devrait dire AWT), cette file est la classe *java.awt.EventQueue*. Elle encapsule une machine asynchrone d'expédition des événements extraits à partir de la file d'attente des événements et les achemine en appelant la méthode *dispatchEvent (AWTEvent)* avec pour argument l'événement.

Le rejeu ne peut s'effectuer que s'il existe des événements capturés (sauvegardés). Pour cela, nous redéfinissons ainsi la file des événements par la classe *PbDEventQueue*, pour permettre l'enregistrement des événements acheminés. Puisque c'est la méthode *dispacthEvent* qui se charge de l'expédition des événements, alors la modification ne porte que sur elle. Elle incorporera une fonction d'enrichissement de l'historique (*PbDListEvent*). La nouvelle file de traitement mise en place a sa structure présentée par l'extrait de Code 2 ci-dessous. L'extrait de code ne présente que la partie sur un événement de type « *ComponentEvent* » que nous dérivons en un événement « *PbDComponentEvent* ». Il est à l'image des autres types d'événements transitant par la classe.

```
public class PbDEventQueue extends EventQueue {
 private PbDListEvents historique;//la liste des événements
 //Constructeur principal de la classe de la file des événements
 public PbDEventQueue(PbDListEvents liste) {
    super(); historique = liste;
 }
 //Méthode de décomposition des événements et expédition
 protected void dispatchEvent(AWTEvent event){
    super.dispatchEvent(event);
    … … … … … … …
    //Test du type de classe de l'événement au ComponentEvent
    if(event.getClass() == (ComponentEvent.class)) {
       ComponentEvent etmp = (ComponentEvent)event;
          //Transformation du ComponentEvent en PbDComponentEvent
       PbDComponentEvent me = new PbDComponentEvent(etmp.getID(),
          etmp.getComponent(), System.currentTimeMillis());
       Point newpos = etmp.getComponent().getLocation();
       me.setPosition(newpos.x, newpos.y);
       Dimension newsize = etmp.getComponent().getSize();
       me.setSize(newsize.width, newsize.height);
       boolean newvisib = etmp.getComponent().isVisible();
       me.setVisible(newvisib);
       //Sauvegarde de l'événement dans l'historique
       … … … … … … … … … }
    else if { //Effectuer le même traitement avec l'ensemble des
              //différents types d'événements
       … … … … … … … … … …}
}
```

Code 2 : La gestion de la file d'événement : PbDEventQueue

Cette classe constitue la partie centrale du système d'enregistrement. Elle est incorporée à d'autres classes permettant la gestion complète du système avec une interface pour démarrer ou arrêter l'enregistrement, ou pouvoir lancer le rejeu des actions. De cette classe *PbDListEvent* est enrichie. La technique du rejeu des événements est implémentée à partir de la classe de sauvegarde temporaire, *PbDListEvent*. L'implémentation en thread est nécessaire pour cette classe (cf. § Code 3) afin de permettre des traitements en tâche de fond (principe des thread). La liste des événements est parcourue et à chaque élément, un appel au système standard de traitement des événements est lancé et l'événement est ainsi injecté dans la file d'exécution des événements du système.

```
public class PbDListEvents implements Runnable {
 private ArrayList<PbDEvent> histo;
 private Thread thread;  … … … … … … …
 public static PbDListEvents historique;
 //Vérification et création d'instance unique de la sauvegarde
 public static PbDListEvents getInstance(){
    if (historique == null) {
       historique = new PbDListEvents();
    } return historique; }
 //Contructeur de l'instance
 private PbDListEvents() {
    histo = new ArrayList<PbDEvent>();     … … … … … … …     }
 //Ajout d'événement de type PbDEvent dans la sauvegarde
 public void add(PbDEvent event) { … … … … … … … }
 … … … … … … …
 public void addKeyEvent(KeyEvent ke) {
    if (this.systemstate != 2)
       histo.add(new PbDKeyEvent(ke.getID(), ke.getComponent(),
ke.getKeyCode(), System.currentTimeMillis(), ke.getModifiers(),
System.currentTimeMillis()));  }
 public void addMouseEvent(MouseEvent me){… … … … … … …}
    … … … … … … …
 public PbDEvent getEventFromListEvent(int i) {
    return (PbDEvent)histo.get(i);   }
 public void record() {  if (this.systemstate != 2) {… … … … … … …
       Toolkit.getDefaultToolkit().getSystemEventQueue().push(new
PbDEventQueue(this)); }
 }
```

Code 3 : La classe gérant l'historique

3.5.3.2 File des événements de l'application

Il s'agit ici d'écouter les événements à partir des composants même de l'application. En effet, avec Swing, on écoute les événements à l'aide des écouteurs appelés « *Listener* ». Ces écouteurs contiennent chacun des méthodes à implémenter par le développeur. Ces méthodes sont appelées lors de la réception d'un événement. On place les « *Listener* » appropriés sur les composants graphiques d'interface utilisateur subissant les interactions de l'utilisateur. Ici, on est très proche des widgets pour effectuer l'enregistrement et le rejeu sur ces composants. Les événements ne sont pas récupérés à un bas niveau comme au niveau de la file des événements système, mais au niveau du modèle des composants (rappelons que les objets Swing sont à l'origine, orientés modèle MVC). En ayant accès au modèle, on intercepte les appels à ses fonctions internes en sélectionnant celles qui nous intéressent particulièrement. Ainsi s'effectue l'enregistrement des événements à un haut niveau dans PbDToolkit.

Au rejeu, on suivra le cheminement inverse des captures dans le composant à partir des fonctions de son modèle, c'est-à-dire un renvoi de l'événement à l'objet destinataire par l'objet source.

L'enregistrement des événements suit le même principe que celui de l'implémentation par la file des événements système. En effet, les événements captés à partir du modèle sont dérivés pour obtenir des *PbDEvent* qui seront enregistrés dans *PbDListEvent*. Ce sont les paramètres qui évoluent par rapport aux événements captés au bas niveau. La différence majeure se situe au niveau d'écoute, à la dérivation des événements et à la technique de réexécution utilisée. On notera que ces événements peuvent être qualifiés de type objet particulier.

Il existe, selon les types, plusieurs méthodes pour déclencher par programme un événement. Par exemple, pour l'implémentation d'un événement de type « clic » sur objet sous la hiérarchie de la classe « *JComponent* », il existe deux possibilités pour rejouer l'action. La première possibilité est l'utilisation de la méthode « *doClick()* » de la classe « *AbstractButton* » (*de javax.swing*). Son invocation se fait explicitement sur la source de l'événement. Cette méthode n'existe que pour les composants héritant de la classe « *AbstractButton* ». Cela limite son utilisation pour les autres composants non héritiers. Elle présente bien l'exécution de l'action par la mise en évidence du comportement du composant et le résultat fonctionnel y est associé. C'est la correspondance des événements bas niveau de type « *MouseEvent* » sur de tels composants.

```
((AbstractButton)ElemDeListEvents.getSource()).doClick() ;
```

Code 4 : La méthode « doClick »

La deuxième possibilité est l'utilisation de la méthode *fireActionEvent()*. Cette méthode informe l'ensemble des composants qui ont enregistré un intérêt pour le type d'événement invoqué. Sa structure implémentée dans le module de rejeu se décompose en la définition de la méthode elle-même (i), la construction de l'événement pour l'invocation (ii), et l'invocation de la méthode pour l'exécution (iii). Elle est pratiquement égale à la première, à la différence que son exécution ne se lie pas directement à un composant source pour être exécuté, mais est envoyée par message au composant destinataire. Elle ne se limite pas seulement aux composants héritant de « *AbstractButton* », mais à l'ensemble des composants héritant de la classe « *JComponent* ». Elle s'avère donc plus intéressante que la première possibilité. De plus, une méthode semblable existe pour tous les types d'événements, ce qui n'est pas le cas pour la méthode « *doClick()* ». Pour plus de possibilité d'écoute de composant à un haut niveau, nous avons donc choisi cette deuxième possibilité pour l'implémentation de l'écoute des composants. À chaque action reçue, une *ActionEvent* est envoyée en retour lors du rejeu par la méthode *fireActionEvent*.

```
    //Définition de la méthode pour l'enregistrement de l'écouteur
(i) : public void fireActionEvent(ActionEvent ae){
            if(listener != null) listener.actionPerformed(ae) ; }
 //Construction d'un événement de type ActionEvent pour la méthode
(ii) : public void actionPerformed(ActionEvent ae){
         fireActionEvent(new actionEvent(ae.getSource(), ae.getID(),
         ae.getactionCommand(), ae.getWhen(), ae.getModifiers())); }
 //Invocation de la méthode pour l'éxecution avec reconstruction
(iii) : fireActionEvent(new ActionEvent(evt.getSource(), evt.getID(),
       evt.getactionCommand(), evt.getWhen(), evt.getModifiers())) ;
```

Code 5 : La méthode « fireActionEvent »

En considérant *PbDToolkit* comme un système applicatif simple, nous représentons l'intégration de ses différents modules, pour une vision globale simple de communication à l'aide de la Figure 42. La composante « Queue d'événements » (*PbDEventQueue*) est la file définie entre autre pour la transmission des événements à l'application. Le module Enregistreur sert à l'enregistrement des différents événements lors de l'exécution de l'application. Le module enregistreur se charge de la sauvegarde des données au travers du vecteur d'événements. Le module de reconstruction se charge de la définition des différents paramètres des événements et a deux utilisations. La première consiste à récupérer les différents paramètres d'un événement afin de disposer des données nécessaires pour la reconstitution du même événement. Elle a lieu dans la composante queue d'événements. La

deuxième utilisation se fait dans le module de rejeu incorporé au vecteur d'événements. La classe abstraite *PbDEvent* est la super classe permettant de définir une structure fixe à l'ensemble des types d'événements pris en compte. La composante « *Path* » permet de définir le chemin de la hiérarchie des composants intervenant dans un événement. Les événements actions et les événements composants sont les événements des actions utilisateur. *PbDListEvents* n'est que la base des événements sauvegardés. C'est lui qui permet la relance ou le rejeu des événements après leur reconstruction.

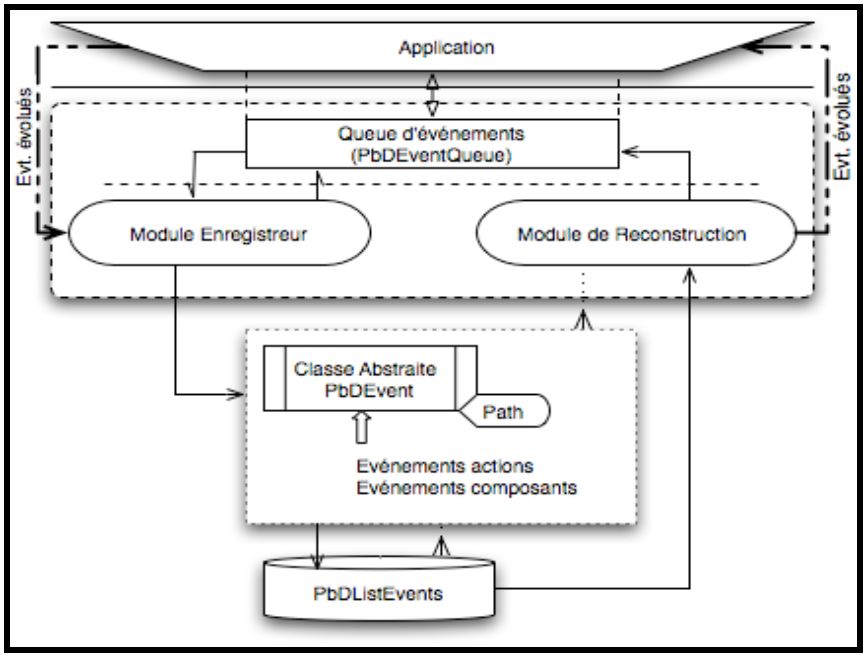

Figure 42 : Structuration des modules intégrant les composantes

3.5.4 Composants d'interface utilisateur ou Widgets

Les widgets de *PbDToolkit* ressemblent aux composants dans Swing à la seule différence que ceux de *PbDToolkit* sont dérivés par héritage des premiers. Chaque composant a été enrichi de sorte à lui adjoindre la capacité d'enregistrement des événements dont il a la charge. La hiérarchie est celle de Swing, l'héritage est lié à la classe du super composant (Figure 43).

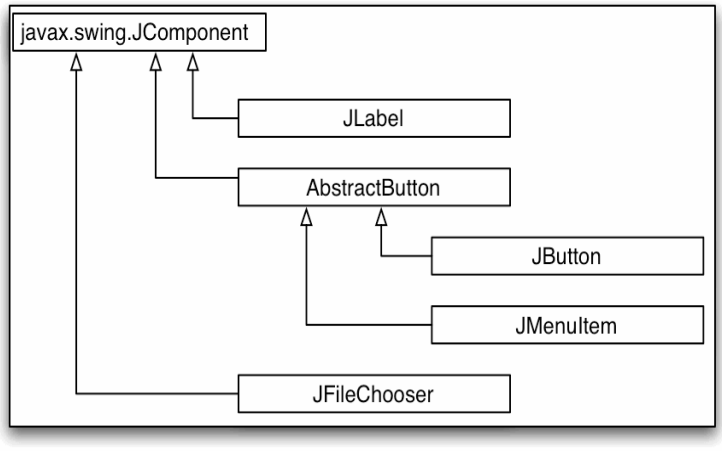

Figure 43 : Hiérarchie de classes de composants

Les composants enrichis portent le nom générique de « PbD*NomComposant* ». Par exemple, l'enrichissement du composant *JButton* de Swing donne lieu à *PbDButton*. Le composant *PbDButton* est défini à partir de la classe *JButton* et hérite ainsi de toutes les propriétés du *JButton*. La structure de sa classe est présentée par le diagramme UML de la Figure 44 ci-dessous. L'attribut « *pbdListeInstance* » fait référence à l'instance unique du système de gestion de PbDToolkit. La méthode « *getControle* » donne accès au gestionnaire de l'outil. La méthode « *addActionListener* » est redéfinie pour permettre l'interception des « *ActionEvent* » du composant directement à la source et les transmettre à la file des événements. C'est à ce niveau que repose l'enregistrement de haut-niveau.

Figure 44 : La classe PbDButton

3.5.5 Synthèse des techniques d'implémentation

L'implémentation à partir de la file des événements standard prend en compte l'ensemble des événements résultant de l'interaction à partir de dispositifs matériels d'entrée de l'utilisateur. Ce sont des événements de bas niveau, qui seront effectivement tous enregistrés. L'enregistrement et le rejeu de l'interaction, sont ainsi facilement satisfaits. En retour, il est nécessaire de traiter le fait que beaucoup d'événements inutiles (les actions articulatoires de l'utilisateur) sont également interceptés.

Il est parfois souhaitable de ne pas prendre en compte toutes les interactions de bas niveau, comme dans le cas du deuxième besoin exprimé par le système Eager (cf. chapitre 2, § 2.2.2). L'installation d'un filtre avant la construction de l'historique permet de gérer cette situation, mais la mise en place d'un tel filtre n'est pas très efficace. Il s'agit d'une méthode à intégrer à la file des événements de bas niveau afin de ne prendre en compte que les événements dont on a besoin. Tous les événements devront passer par ce filtre avant d'enrichir l'historique des événements suite aux actions de l'utilisateur. Cela alourdit l'exécution de l'application.

La technique de l'écoute par abonnement fixe dès le départ le niveau des événements à enregistrer. Les événements de haut niveau engendrés par les widgets comme les *JButton*, *JTextField*, ou encore *JTextArea* sont traités facilement et le rejeu à leur niveau donne un rendu visuel complet. Cette technique peut être vue d'un côté comme étant lourde, mais la spécialisation de composants natifs de Swing autorise de ne prendre en compte que certains événements sans utiliser la notion de filtrage d'événements. C'est une manière simple et correcte de répondre au second besoin. La difficulté de la technique consiste à choisir les bons événements à enregistrer. En effet, l'existence de widgets évolués encapsulant un comportement automatique (comme le « Couper-Coller » par exemple) peut aboutir à un comportement incohérent de l'enregistrement.

La combinaison des deux techniques est donc nécessaire. Pour plus d'indépendance du système, la mise en place d'un passage commun serait l'idéal afin de faire transiter tous les

événements quelque soit leur type, et cela après des traitements différents de dérivation et de reconstruction.

Pour le développeur, la technique d'implémentation de l'enregistrement et du rejeu se résume à une analyse simple. Il faut observer et analyser les interactions de l'utilisateur en continu. En effet, le système d'assistance doit espionner en permanence le comportement du système et les entrées de l'utilisateur. Une solution simple ou du moins naïve consisterait à simplement enregistrer les événements utilisateur, de manière à pouvoir les analyser et les rejouer. Cela étant, deux types de rejeu doivent être envisagés. Un rejeu très fin peut être produit, avec reproduction de tous les feedbacks, pour permettre à l'utilisateur de visualiser chaque étape de l'interaction. En revanche, lors de la phase de terminaison de tâche (exécution fonctionnelle), seules les actions de haut niveau doivent être rejouées. Pour satisfaire le premier besoin, un travail au niveau des événements les plus élémentaires (tel *MouseEvent* en Swing) est effectué. Dans le deuxième cas, des événements plus évolués (par exemple *ActionEvent* en Swing) sont suffisants. Il convient au développeur d'une solution de PsE de choisir le niveau sur lequel il souhaite intervenir. Notons cependant que l'utilisation d'événement de haut niveau doit être particulièrement réfléchie. Lorsque l'on conçoit une application Swing, la programmation de la réaction à un événement comme *ActionEvent* (le click sur un bouton) est explicite, ce qui n'est pas toujours le cas ; certains composants de fenêtre permettent sans programmation explicite des comportements avancés dans les applications WIMP (Curtis and Hefley, 1994). Par exemple, les composants fenêtres de texte (*TextField* ou *TextArea* dans Swing) acceptent en standard le « *Copier-Coller* », et ce par « *Drag and Drop* » ou par raccourcis clavier. Il est donc nécessaire, si l'on veut gérer toutes les possibilités d'interaction des composants de fenêtre évolués, de travailler à un niveau sémantique, et donc par exemple au niveau des modèles des composants (au sens MVC du terme).

4 Exemple d'illustration : un convertisseur de devises

Afin d'illustrer l'utilisation qu'un développeur peut faire de *PbDToolkit*, nous avons choisi de proposer l'étude d'un cas simple. Elle porte sur un convertisseur de devises intégrant un mini éditeur de texte. L'utilisateur peut convertir un montant exprimé dans une devise A en son équivalent dans la devise B à l'aide d'une zone de saisie. Pour effectuer des tâches plus complexes, enchaînant par exemple plusieurs conversions, il peut reporter la valeur de sa conversion dans l'éditeur, puis poursuivre ses conversions vers d'autres devises.

Dans cette section, nous présentons tout d'abord l'application initiale, c'est-à-dire sans prise en compte des fonctionnalités de la PsE. Nous montrons ensuite comment le programmeur peut adjoindre ces dernières, ce qui nous permettra de mettre en exergue la simplicité de la démarche. Pour finir, nous montrerons comment l'utilisateur peut, à partir de ces fonctionnalités, automatiser aisément une tâche répétitive.

4.1 Application initiale du convertisseur

Dans une première partie, nous allons décrire l'application réalisée, à travers son interface utilisateur. Puis, nous décrirons les principes généraux de programmation de cette application en Swing, en insistant notamment sur les composants utilisés et les événements associés.

4.1.1 Interface du convertisseur

Le convertisseur de devises est un logiciel utilitaire permettant de réaliser des conversions d'une valeur donnée en une première devise en son équivalent dans une seconde devise. Dans l'implémentation que nous proposons, l'utilisateur choisit la devise de départ, saisit la valeur qu'il désire convertir dans la zone de saisie correspondante, choisit la devise d'arrivée, puis déclenche la conversion par le bouton « Convertir » ou par le choix d'une nouvelle devise. Il lit alors le résultat dans la zone de texte d'affichage dans la devise de conversion souhaitée. Lorsqu'il change la devise de conversion à l'arrivée, il obtient une nouvelle valeur de conversion. Par contre, en changeant la valeur à convertir, il devra enclencher le bouton « Convertir » pour pouvoir lire le résultat. On peut représenter la dynamique de cette application par l'automate (StateChart) au moyen de la Figure 45.

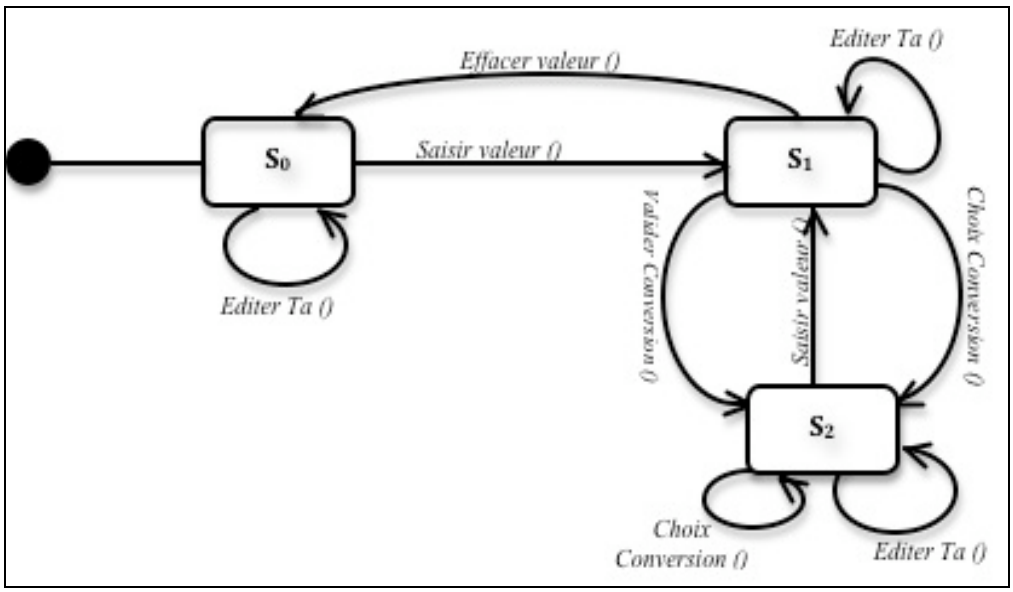

Figure 45 : Dynamique de l'application du Convertisseur

A l'état initial (S_0) de l'application, seuls l'éditeur de texte et la zone de saisie de valeur sont actifs. L'utilisateur peut saisir une valeur (*saisir valeur ()*) dans le champ texte (S_1). Il peut vider le contenu du champ en effaçant la valeur (*effacer valeur ()*) ou effectuer sa conversion (S_2) à l'aide du bouton (*Valider Conversion ()*) ou en choisissant la devise de conversion et/ou à convertir (*Choix Conversion ()*). À tout moment de l'application, il a la possibilité de faire une édition dans l'éditeur de texte (*Editer Ta ()*).

L'utilisateur interagit avec l'application via l'interface (Figure 46) construite au moyen de widgets comme une fenêtre, des menus déroulants, des boutons de choix, et des composants de gestion de textes (une zone de saisie et une zone d'affichage du résultat, une zone multi lignes pour reporter les opérations).

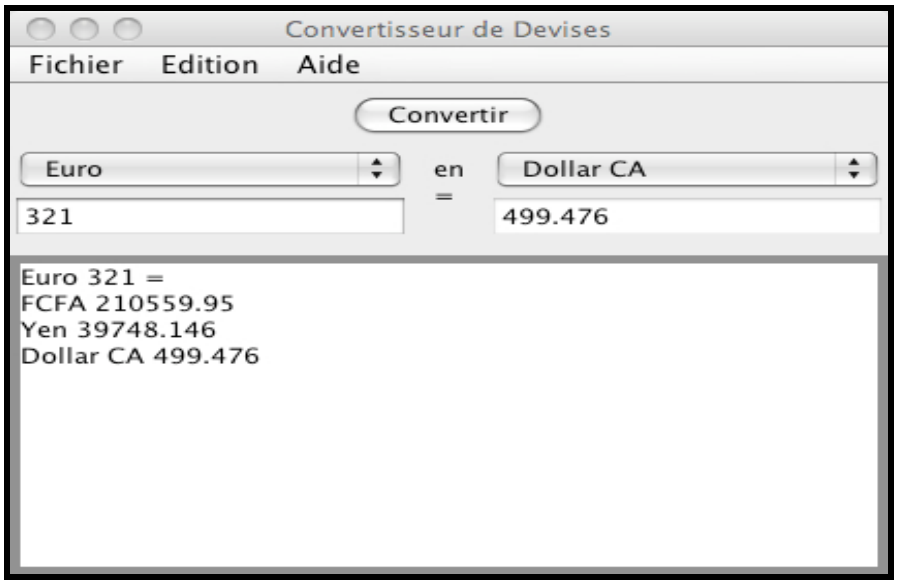

Figure 46 : Interface du Convertisseur de devises

Les widgets nécessaires à la réalisation d'une telle interface sont décrits hiérarchiquement sur la Figure 47.

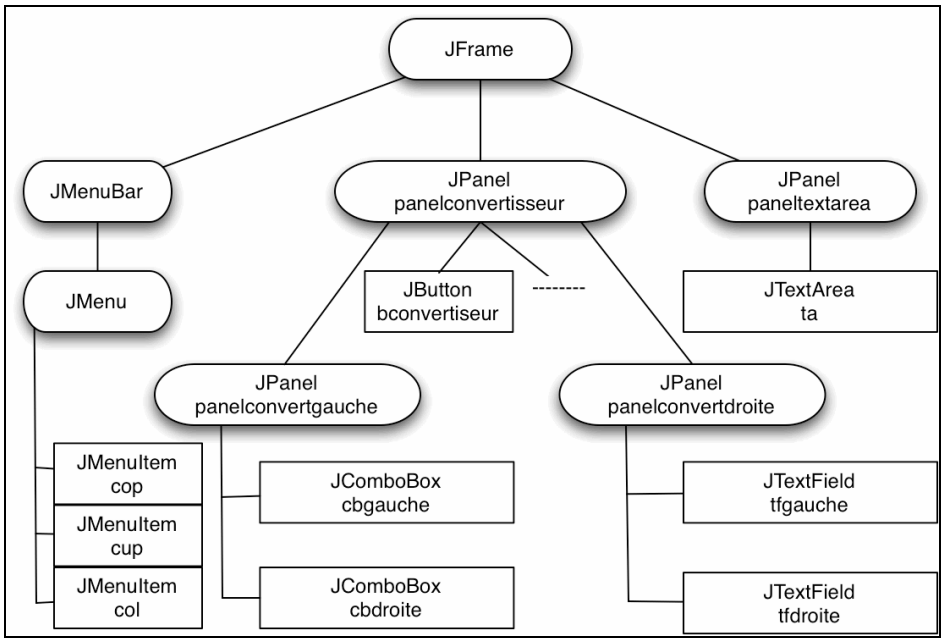

Figure 47 : Hiérarchie des composants de l'interface du convertisseur initial

Les interactions possibles à partir de l'interface sont le choix de la devise de départ, la saisie de la valeur à convertir, le choix de la devise de conversion, et le déclenchement de l'action de conversion. L'affichage du résultat de la conversion se fait dans la zone de texte en dessous de la devise de conversion, ainsi que dans la zone de texte du bas de la fenêtre. La zone de texte jouant le rôle de champ d'affichage n'est accessible à aucune interaction de l'utilisateur. Chaque interaction effectuée pour le choix d'une nouvelle devise provoque une modification de la conversion.

4.1.2 Gestion des événements « ActionEvent »

Malgré leurs apparences très diverses, trois des composants utilisés dans notre application (le bouton et les deux combox) utilisent une même gestion d'événements : l'événement *ActionEvent*. Le clic sur un bouton se traduit par le déclenchement d'un événement *ActionEvent*, tout comme l'activation d'un item de menu, ou la sélection d'un élément dans une ComboBox. La programmation de ces éléments sera donc très similaire : après leur définition (création d'une instance de la classe du composant), il conviendra d'associer un écouteur de cet événement (*ActionListener*) à chacun de ces objets (à l'aide de la méthode *addActionListener*) pour leur permettre de déclencher l'action nécessaire (*ActionEvent*). La programmation de ces différents composants est représentée par la portion de code suivante :

```
//Création d'une instance de liste de choix
JComboBox cbgauche = new JComboBox (ListeDevise);
//Abonnement à l'écoute du choix dans la liste
cbgauche.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent ae){
      convertir() ;
}  });
JComboBox cbdroite = new JComboBox (ListeDevise);
cbdroite.addActionListener(new ActionListener() {
      -------------  }); -------------
//Définition des items de menu pour le menu déroulant
JMenuItem couper = new JMenuItem("Couper", KeyEvent.VK_CUT);
JMenuItem copier = new JMenuItem("Copier", KeyEvent.VK_COPY);
JMenuItem coller = new JMenuItem("Coller", KeyEvent.VK_PASTE);
-------------
//Ajout des items dans le menu edition pré-existant
edition.add(couper);
edition.add(copier);
edition.add(coller);
-------------
```

Code 6 : Définition et abonnement de widgets

4.1.3 Composants JTextField et JTextArea

Les composants de texte posent un problème différent. Les composants *JTextField* servent à la saisie de la valeur de conversion et à l'affichage du résultat de la conversion (de la gauche vers la droite). Le composant *JTextArea* sert de zone d'édition de texte libre, par exemple pour saisir une liste de résultats de conversion d'une même devise vers différentes autres devises. Même s'ils sont susceptibles de réagir au même type d'événement (un *ActionEvent* est ainsi déclenché par l'appui de la touche « Return » lorsque le composant a le focus), la richesse de leurs comportements prédéfinis impose de prendre en compte leur programmation de façon différente. Par exemple, dans notre exemple, nous devons écouter la saisie de la valeur à convertir à partir du *JTextField*. Les valeurs sont saisies à partir du clavier. Nous écoutons les événements claviers (*KeyEvent*). Par contre le *JTextArea* ne présente pas d'intérêt à la conversion. C'est un composant pouvant être traité comme le *JTextField*. Mais, la prise en compte d'un autre type d'événement est nécessaire pour ce dernier afin de pouvoir intercepter toute modification apportée à la zone : *DocumentEvent*.

4.2 Effort de programmation

L'effort de programmation demandé au développeur pour bénéficier des fonctionnalités de la PsE est minimal dans cette application. Pour disposer des fonctionnalités standard, il lui suffit de respecter une discipline de programmation. C'est ce qui sera décrit dans la première sous-section. Pour offrir à ses utilisateurs un service plus adapté, il devra programmer quelques éléments supplémentaires.

4.2.1 Base de la prise en compte de la PsE

La prise en compte de la PsE dans l'application se fait en utilisant des composants de PbDToolkit (Figure 48). Les changements par rapport à une réalisation standard en Swing portent sur les composants *PbDButton* (à la place de *JButton*), *PbDTextArea* (à la place de *JTextArea*), *PbDComboBox* (à la place des *JComboBox*), *PbDTextField* (à la place de *JTextField*) et des *PbDMenuItem* (à la place des *JMenuItem*).

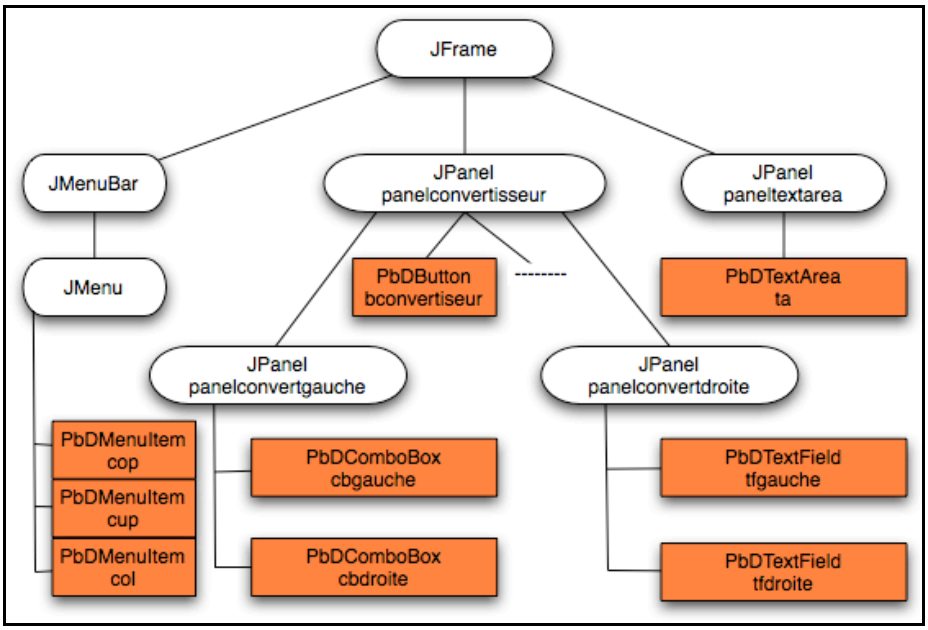

Figure 48 : Hiérarchie des composants de l'interface du convertisseur avec prise en compte de la PsE

```
//Création d'une instance de la liste de choix
PbDComboBox cbgauche = new PbDComboBox (ListeDevise);
//abonnement à l'écoute des événements de type ActionEvent
cbgauche.addActionListener(new ActionListener(){
      ------------  });
PbDComboBox cbdroite = new PbDComboBox (ListeDevise);
cbdroite.addActionListener(new ActionListener() {
      ------------  });
```

Code 7 : Définition et abonnement de widgets

L'enregistrement des événements liés aux actions sur les *ComboBox* s'effectue directement par le biais de l'appel de la classe d'interception des événements lors de la création du composant *PbDComboBox*. En effet, dans la classe *PbDComboBox*, il est mis en place l'appel à une instance du contrôleur de *PbDToolkit*. Cette instance est créée s'il n'existe pas d'instance fonctionnelle. L'enregistrement s'effectue à un haut niveau pour ces

composants. En effet dans la redéfinition de la méthode « ActionListener », nous écoutons l'événement « ActionEvent ».

L'implémentation d'une barre de menu n'est pas une programmation complexe en Swing. Il s'agit simplement de la définition d'items de menu dans un menu et de l'insertion de ce dernier dans la barre de menu. Les actions du menu sont liées aux items. Ce sont eux qui sont chargés de la réception des actions utilisateur. Les items de menu sont en quelque sorte des boutons. En effet, les items de menu *JMenuItem* sont issus de la classe *AbstractButton*. Ils sont identiques au *JButton* ou au *JComboBox* et sont au même niveau de la hiérarchie. Les *PbDMenuItem* sont donc au même niveau et nous pouvons faire la même hiérarchie avec les classes *PbDButton* et *PbDComboBox*.

Tous les widgets ont été réécris suivant le même principe. Nous avons hérité de la classe *JTextField* de *javax.swing* pour créer la classe *PbDTextField*. Aucun effort supplémentaire de programmation n'est nécessaire pour l'implémentation de ces composants.

L'application réalisée de la sorte est un simple convertisseur de devises. Le développeur ne fait appel à aucune fonctionnalité de la PsE. Or l'implémentation est réalisée à partir de composants de *PbDToolkit*. L'application incorpore donc les techniques de la PsE. C'est au développeur de les mettre en fonction.

Le développeur peut activer le système en activant tout simplement l'affichage de l'interface du contrôleur de *PbDToolkit*. Pour cela, il n'a qu'une seule instruction à écrire (Code 8) dans le constructeur de sa classe d'application, où des composants *PbDToolkit* ont été utilisés. Il peut décider de passer par le composant *PbDButton* de son interface applicative pour l'activation de l'interface de contrôle de *PbDToolkit*. L'instruction est la suivante :

```
//Création du bouton « Convertir »
PbDButton bconvertir = new PbDButton("Convertir") ;
//Affichage de l'interface du contrôleur de PbDToolkit
bconvertir.getInstanceOfPbDListEvents().setAffichePresentation(true);
…….
```

Code 8 : Affichage de l'interface du contrôleur de PbDToolkit

À l'exécution de l'application du convertisseur, l'interface de gestion principale de *PbDToolkit* sera à la disposition de l'utilisateur lui permettant ainsi d'enregistrer et de rejouer les événements issus des actions utilisateur selon ses besoins. Il pourra utiliser l'outil comme nous l'avons présenté plus haut dans la sous-section 3.2.2.

Activer l'interface de gestion de *PbDToolkit* est une possibilité d'utiliser ses fonctionnalités sans aucun effort de programmation. L'utilisation reste standard à celle décrite au § 3.1. Si le développeur cible les besoins des utilisateurs de l'application en termes de tâches, il peut implémenter, d'une façon simple, des fonctionnalités supplémentaires et les intégrer directement dans l'interface de l'application. Dans la section suivante, nous mettons cela en évidence.

4.2.2 Fournir des fonctionnalités plus intégrées à l'application

Lors de l'utilisation du convertisseur de notre étude de cas, l'utilisateur peut effectuer plusieurs conversions à la suite sur une même devise de départ par exemple. À chaque conversion, il copie la valeur convertie dans le mini éditeur de texte, puis change de devise de conversion. Il est possible qu'une telle opération soit quotidienne. Dans une telle situation, une assistance s'impose. L'assistance à fournir est de permettre une automatisation de la

recopie des résultats et des changements de devise. C'est une automatisation de tâches simples. Comment le développeur peut-il procéder ?

Il est simple de fournir des fonctions d'automatisation de tâches à l'utilisateur du moment que l'implémentation de l'application intègre *PbDToolkit*. En effet, le développeur peut mettre à la disposition de l'utilisateur la possibilité de créer des macros, les sauvegarder, et de pouvoir les ré-exécuter à partir de l'interface du convertisseur. L'idée à exploiter est la mise en place et la mise à disposition du menu permettant la création, la sauvegarde et l'exécution des macros. La macro sera une séquence d'événements enregistrés par le système d'enregistrement de *PbDToolkit*. Il s'agira de petites séquences qui seront enregistrées comme une séquence normale après l'espionnage d'une application.

Pour ce faire, le développeur ajoute dans la barre de menu, un menu « Macro » contenant les commandes « *Nouvelle* » (pour la création d'une nouvelle macro), « *Exécuter* » (pour exécuter une macro enregistrée) et « *Stop* » (pour arrêter l'enregistrement d'une macro). La nouvelle interface du convertisseur de devise est présentée à l'aide de la Figure 49 ci-dessous.

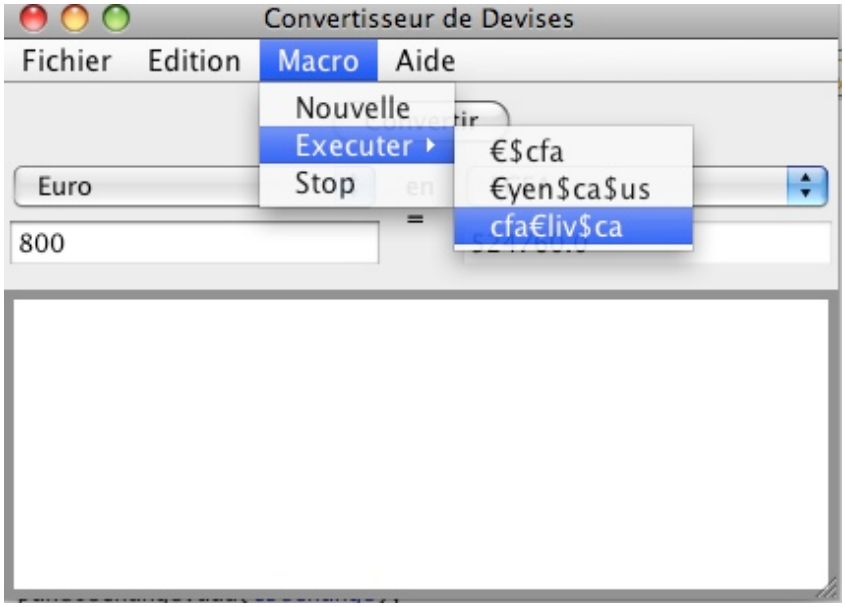

Figure 49 : Nouvelle interface utilisateur du convertisseur de devises

Le menu « *Nouvelle* » permet à l'utilisateur de donner un nom à sa macro à l'aide d'une petite fenêtre surgissant. Le nom de la macro est celui du fichier source dans lequel seront enregistrés les événements de la création de sa tâche utilisateur. À l'item « *Nouvelle* » sont liées deux actions pour le système *PbDToolkit* : la transmission du nom du fichier d'enregistrement des événements et l'activation du démarrage de l'enregistrement. N'importe quel composant *PbDToolkit* utilisé permet d'accéder à ces fonctions. Dès renseignement du nom de la macro par l'utilisateur, l'enregistrement est aussitôt activé. L'utilisateur met fin à l'enregistrement à l'aide de l'item « Stop » du menu « Macro ». Cette commande suspend l'espionnage et lance la sauvegarde du vecteur d'événements dans un fichier ayant pour nom celui renseigné par l'utilisateur lors de la création (lancement) de la macro (comme nom de la macro). Nous faisons abstraction du code d'implémentation du menu « Macro », car identique à une implémentation de menu en Swing. L'instruction dans le code de l'application convertisseur permettant d'activer l'enregistrement dans *PbDToolkit* à travers un composant est celle présentée par l'extrait du Code 9.

```
……
bconvertir.getInstanceOfPbDListEvents().enregistrement();
……
```

Code 9 : Activation de l'enregistrement dans PbDToolkit

Pour arrêter l'espionnage, un appel est fait à la méthode d'arrêt du système. Pour l'enregistrement physique de la macro (du transfert du contenu du vecteur d'événements dans un fichier), l'instruction d'appel à la fonction de création de fichier avec le nom de la macro est réalisée comme toutes les autres opérations. L'extrait de code 10 montre l'implémentation de lien faite par le développeur.

```
……
bconvertir.getInstanceOfPbDListEvents().stop();
bconvertir.getInstanceOfPbDListEvents().fichierXML("nom-macro");
……
```

Code 10 : Arrêt de l'espionnage et sauvegarde de la macro

La commande d'exécution de la macro créée s'effectue à l'aide de l'item « Exécuter » du menu « Macro ». L'action liée à cet item est l'appel à la méthode de rejeu du système PbDToolkit. L'instruction dans le code de l'application du convertisseur permettant cette liaison se présente par l'extrait de Code 11.

```
……
bconvertir.getInstanceOfPbDListEvents().rejouer("nom-macro");
……
```

Code 11 : Lancement du rejeu à partir d'un fichier

Le développeur met ainsi à la disposition de l'utilisateur, avec un minimum d'effort, la possibilité de créer des macros pour automatiser ses tâches répétitives. L'utilisateur n'a besoin d'aucune connaissance supplémentaire pour faire cette automatisation.

Une autre forme d'assistance peut être envisagée comme celle dans le système Eager, c'est-à-dire suivre l'utilisateur et lui proposer l'action suivante à exécuter. Mais, à l'état actuel du système de notre historique, nous ne pouvons pas proposer à l'utilisateur la tâche suivante à effectuer. La recherche de séquences répétitives est possible par la mise en place d'un algorithme de recherche de motif dans l'historique. Le motif sera un ensemble « fini » de commandes (actions ou événements), sous la forme d'éléments composites. Mais, il nous est possible de reprendre l'ensemble des actions successives enregistrées au cours d'une exécution. Donc, l'assistance peut être de refaire les mêmes choix de devises et de copier le résultat de chaque conversion dans le mini éditeur de texte. Pour cela, l'utilisateur est amené à modifier la valeur à convertir de façon manuelle en ouvrant le fichier de sauvegarde des actions passées. Ainsi, en rejouant ce fichier sur l'application, il obtiendra les résultats des nouvelles conversions.

La mise en place d'une interface de gestion du fichier est nécessaire pour permettre à l'utilisateur de pouvoir modifier des valeurs saisies lors d'une séquence d'actions passées. L'idéal serait de permettre à l'utilisateur la saisie directe de nouvelles valeurs dans les champs concernés lors du rejeu. Cela nécessite plus d'implication dans le contexte d'exécution et surtout sur la nomination des variables d'exécution et des objets pris en compte dans *PbDToolkit* lors de l'enregistrement des événements.

4.3 Point de vue de l'utilisateur

Comment l'utilisateur peut-il réellement employer les fonctionnalités de PsE ? Supposons que l'utilisateur ait besoin d'effectuer une tâche fortement répétitive, qui consiste à enchaîner plusieurs opérations de conversion portant sur la même valeur. Par exemple, à partir d'une valeur en euros, il a besoin de connaître les contre-valeurs en dollars et en yens. La tâche à réaliser, pour chaque valeur, consiste à (1) saisir la valeur en euros, (2) sélectionner la devise « Dollars US », (3) demander la conversion, (4) sélectionner la devise « Yens », (5) demander la conversion. Cette action simple doit ainsi être répétée de nombreuses fois.

L'utilisation des fonctionnalités de PsE commence par l'activation de l'enregistrement d'une macro, juste après la saisie de la valeur en euros. Les quatre actions suivantes doivent être exécutées à l'identique. Après la conversion en yens, la macro est sauvegardée et devient disponible dans le menu de macros. Il suffit alors de ré-invoquer cette macro à chaque saisie d'une valeur à convertir.

5 Conclusion

Nous avons présenté un concept permettant d'introduire simplement des techniques de PsE dans une application interactive, principalement pour la classe d'utilisation de l'assistance. L'outil *PbDToolkit* supporte un fonctionnement de type adaptatif et est adaptable selon les besoins du développeur. La réponse partielle des deux outils qui existaient, PbDScript et AIDE, est complétée par l'implémentation de *PbDToolkit*. Les besoins explicites des applications sont bien réalisables avec une minimisation du rôle du développeur pour éviter les erreurs.

PbDToolkit réalise le mécanisme d'enregistrement/rejeu basé sur des files d'événements, par une extension des composants de Swing. L'extension majeure porte sur les widgets qui sont les composants indispensables à la construction d'interface homme-machine. Des fonctionnalités supplémentaires y ont été rajoutées et le développement d'IHM ne change pas par rapport à l'utilisation de la boîte à outils Swing initiale.

Chapitre 4 Extension de PbDToolkit pour l'automatisation de tests d'interfaces graphiques

SOMMAIRE

***Résumé**. La technique d'enregistrement et de rejeu de la PsE est utilisée pour mettre en place un système de vérification de la conformité d'une IHM à son modèle de tâches. Il établit un lien entre les tâches élémentaires du modèle de tâches et les actions élémentaires de l'interface utilisateur. À travers un fichier de correspondance mis en place par le développeur, celui-ci rajoute simplement les noms des tâches aux actions de l'interface. À l'exécution, il est généré un fichier de scénario sous le format K-MADe qui peut être exécuté à partir du simulateur de l'environnement K-MADe.*

1 Introduction

La validation et la vérification des systèmes informatiques constituent un aspect important du cycle de développement logiciel. La validation des systèmes a connu de grandes avancées par l'utilisation de méthodes formelles, telles que les méthodes à base de preuves (Aït-Ameur et al., 2003) (méthode B par exemple) ou les méthodes appliquant la théorie du model-checking (D'Ausbourg, 1998). Malgré leur réussite certaine, ces techniques n'ont pas rendu obsolète la notion de test. En l'absence d'un cycle complet de développement formel, le test demeure la seule solution permettant de s'assurer que le système effectivement réalisé, éventuellement validé par les modèles développés pour sa conception, effectue bien le travail pour lequel il a été conçu. De nombreux travaux ont ainsi été menés sur la notion de test, comme en témoigne l'abondante littérature sur la question publiée dans les revues et conférences dans le domaine du génie logiciel. De nombreuses méthodes et davantage d'outils adressent le problème de l'automatisation du test, afin de permettre l'instrumentation des tests de non-régression, censés permettre d'éviter une régression des fonctionnalités du logiciel au cours de son évolution. Le test a pris une importance primordiale dans les méthodes agiles telles l'«eXtreme Programming » (Wells, 2006), ce qui a donné lieu à des outils particulièrement efficaces comme ceux de la série xUNIT. Cependant, les outils proposés actuellement s'intéressent principalement, voire exclusivement, à l'aspect logiciel du développement, en se concentrant sur les fonctions calculables.

La validation des systèmes interactifs est loin de ne concerner que ce strict aspect logiciel. La fiabilité du calcul réalisé ne saurait en aucun cas être une mesure suffisante de la qualité du système. Les spécificités de la participation humaine ont été prises en compte dans des travaux qui ont conduit à l'établissement de critères ergonomiques, susceptibles de représenter un aspect non fonctionnel important des systèmes interactifs. Plus généralement, la définition de la notion de conception centrée utilisateur a permis d'intégrer l'activité humaine au centre du processus de conception des applications, dans le but d'obtenir une meilleure utilisabilité. Ceci a conduit la communauté de l'interaction homme-machine à définir des modèles spécifiques, dont les modèles de tâches, qui s'appuient sur la théorie de l'action de Norman (Norman, 1986) pour modéliser l'activité de l'utilisateur.

Tout naturellement, de nombreux travaux ont cherché à transposer les notions de validation et de vérification dans le domaine de l'interaction homme-machine. Comme le prouve le nom même de la principale conférence dédiée à ce problème, DSVIS pour Design, Specification and Validation of Interactive Systems, le domaine de la validation a été le plus intensément exploré, les différents auteurs cherchant à adapter les méthodes aux spécificités de l'IHM. Comme pour les travaux en génie logiciel, les méthodes à base de preuve et les méthodes à base de model-checking ont été largement utilisées, parfois de façon complémentaire. L'approche des systèmes à base de modèles (Model-Based Systems) très en vogue aujourd'hui avec le développement de l'ingénierie dirigée par les modèles (Model-Driven Engineering), intègre une bonne partie de ces résultats, et a abouti à des outils comme PetShop (Bastide, 2000) ou TERESA (Berti et al., 2004). PetShop permet de spécifier un système interactif dans le respect de son modèle sous-jacent, lui-même basé sur des réseaux de Petri. TERESA permet de générer une application interactive à partir de son modèle (de tâches en l'occurrence).

Cependant, comme pour le développement logiciel à dominante fonctionnelle, l'utilisation de ces méthodes ne saurait éviter le besoin de recourir à des méthodes de tests. Pourtant, très peu de travaux ont été consacrés à la définition de méthodes ou d'outils de tests

d'IHM. Certes, une catégorie d'outils à base d'espionnage de l'interaction entre l'utilisateur et le système, a vu le jour, tels que *Jacareto*[20]. Ces outils permettent de traduire une série d'interactions de l'utilisateur en logs plus ou moins évolués, susceptibles d'êtres ré-exécutés sur le système. En dehors du fait que ces outils sont très sensibles à la partie lexicale de l'interaction homme-machine (tout changement d'un composant de l'interface entraîne l'obsolescence du test), le principal reproche que l'on puisse faire à ces méthodes est le très faible niveau sémantique des données qu'ils manipulent. Memon et son équipe (Memon et al., 1999; 2000) ont effectué de nombreux travaux spécifiquement dédiés au test des interfaces utilisateur graphiques. Fortement inscrits dans les travaux autour des tests de non-régression, ils proposent des méthodes statistiques (Memon and Lou, 2003) pour générer des jeux de tests présentant des taux de couverture importants. Cependant, cette approche est essentiellement centrée sur le système car se basant sur une représentation des fonctionnalités du système et non sur l'activité de l'utilisateur. Elle s'avère donc incapable de vérifier que le système répond à ses spécifications utilisateur. Les seules approches qui envisagent un lien direct entre l'application et l'activité sont à rechercher dans les travaux de (Bourguin et al., 2006; Tarby, 2006) et (Balbo et al., 2006). Notre approche intègre la PsE, plus particulièrement, la technique d'enregistrement et de rejeu.

Dans ce travail, nous avons exploré les liens possibles entre un modèle de tâches et l'application finale censée le représenter. Tournant le dos aux approches dirigées par les modèles où l'application serait dérivée des modèles, et donc en partie du modèle de tâches, nous considérons le cas où le développeur d'application souhaite donner la possibilité de vérifier que l'exécution de tests sur l'interface est bien conforme au modèle de tâches prescrit. Pour cela, nous proposons une méthode d'instrumentation du code développé, qui s'avère très légère pour le développeur. Couplée à un outil de simulation de modèles de taches, K-MADe, cette méthode permet facilement de construire des séries de tests qui peuvent être confrontées directement au modèle de tâches.

Dans un premier temps, nous allons définir la notion de test. À l'issue de cette première partie, quelques outils favorisant ces tests sont présentés. Ils sont introduits par une généralité sur les tests. Nous détaillons ensuite une technique permettant l'automatisation de tests d'interface par une liaison entre l'interface et le modèle de tâches. Ces deux composants sont décrits. Le chapitre se termine par une analyse critique de la solution proposée ainsi que les perspectives du travail.

2 Les Tests

Un test (mot dérivé de l'anglais) désigne une procédure de vérification partielle d'un système informatique. Le but est de s'assurer que le système réagit de la façon prévue par ses concepteurs ou est conforme aux attentes du client l'ayant commandé. Un test ressemble à une expérience scientifique. Il examine une « hypothèse » formulée par un triplet (données en entrée, objet à tester, observations attendues). Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions.

2.1 Définition

Un test est un ensemble de cas à tester (état de l'objet à tester avant exécution du test, actions ou données en entrée, valeurs ou observations attendues, et état de l'objet après

[20] http://jacareto.sourceforge.net/wiki/index.php/Main_Page

exécution), éventuellement accompagné d'une procédure d'exécution (séquence d'actions à exécuter). Il est lié à un objectif (*norme IEEE 829-1998*).

La définition d'un test revient donc à définir cet ensemble. Différents types de tests permettent de détecter différents types de défauts. Des méthodes de spécification de test ont été élaborées pour permettre une plus grande rigueur dans cette activité de définition. Un test vise à mettre en évidence des défauts de l'objet testé. Cependant il n'a pas pour objectif :

- de diagnostiquer la cause des erreurs,
- de les corriger,
- de prouver la correction de l'objet testé.

La définition d'un cas à tester précise les exigences s'appliquant à une spécification. Un objet ne peut être testé que si l'on peut déterminer précisément le comportement attendu en fonction des conditions auxquelles il est soumis. Si la spécification ne permet pas cette détermination, la propriété du logiciel qu'elle définit ne peut être testée. Soumettre la spécification à cette contrainte de « testabilité » permet d'en améliorer la précision puisqu'elle oblige à expliciter les caractéristiques de l'objet. Ceci permet, en retour, de trouver plus tôt les erreurs de spécification (incohérence, manque de complétude, etc.).

L'activité de test d'un logiciel utilise différents types et techniques pour vérifier que le logiciel est conforme à son cahier des charges (vérification du produit) et aux attentes du client (validation du produit). Elle est une phase du processus de développement du logiciel permettant d'assurer un niveau défini de qualité en accord avec le client. Une procédure de test peut donc être plus ou moins fine, et par conséquent l'effort de test plus ou moins important et coûteux selon le niveau de qualité requis.

2.2 Classification des tests

Il existe différentes façons de classer les tests informatiques parmi lesquelles la plus connue propose une classification selon trois perspectives :

- la nature de l'objet à tester (perspective étroitement liée au cycle de développement),
- le niveau de connaissance de la structure de l'objet,
- le type de caractéristique ou propriété.

Ces trois perspectives ne permettent cependant pas de classer les tests de non-régression. Ce type de test est défini comme cherchant à mettre en évidence, dans la partie inchangée du logiciel, des défauts mis à jour ou introduits par un changement dans le logiciel (mise à niveau, correction, etc.) ou son environnement d'exécution. Les tests de non-régression ne sont donc pas restreints à une phase particulière du cycle de développement. Ils ne sont pas non plus restreints à un type de propriété à tester.

2.2.1 Classification selon le niveau

Le niveau de test correspond à un groupe d'activités de test qui sont organisées et gérées ensemble. Un niveau de test est lié aux responsabilités dans un projet. Les niveaux de test sont : les tests de composants, les tests d'intégration, les tests système et d'acceptation.

Par ailleurs, on parle aussi de « niveaux de test amont » et « aval » pour désigner ces quatre niveaux de test (Brooks and Memon, 2007):

Unitaire + Intégration = Amont
Système + Acceptation = Aval.

Le terme « phase de test » est parfois préféré au mot « niveau de test ». On dit que, le test unitaire est un niveau de test, le test d'intégration est également un niveau de test, alors que le test de non-régression n'est pas un niveau, mais un type de test.

2.2.1.1 Tests de composants

Le test de composants ou test unitaire, est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité »). Il s'agit pour le programmeur de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles, et qu'il fonctionne correctement en toute circonstance. Cette vérification est considérée comme essentielle, en particulier dans les applications critiques. Elle s'accompagne couramment d'une vérification de la couverture de code, qui consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester. L'ensemble des tests unitaires doit être rejoué après une modification du code afin de vérifier qu'il n'y a pas de régression. Dans les applications non-critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, les méthodes agiles comme *l'Extreme Programming* ont remis les tests unitaires au centre de l'activité de programmation.

2.2.1.2 Tests d'intégration

Le test d'intégration est un test qui se déroule à la suite des tests unitaires. Il consiste, une fois que les développeurs ont chacun validé leurs développements ou leurs correctifs, à regrouper leurs modifications ensemble dans le cadre d'une livraison. Il s'agit d'établir une nouvelle version, basée soit sur une version de maintenance, soit sur une version de développement. L'intégration fait appel en général à un système de gestion de versions, et éventuellement à des programmes d'installation. L'intégration a pour but de valider le fait que toutes les parties développées indépendamment fonctionnent bien ensemble. Pour les applications utilisant les nouvelles technologies et donc des ateliers de génie logiciel (Eclipse - Visual Studio - JBuilder - JDeveloper ...), les tests d'intégration ont évolué vers de l'intégration continue. L'intégration continue est la fusion des tests unitaires et des tests d'intégration car le programmeur détient toute l'application sur son poste et peut donc faire de l'intégration tout au long de son développement.

2.2.1.3 Tests système

Un test système couvre trois aspects de l'application. Il s'agit de tester (*1*) le système du point de vue de la fonctionnalité, (*2*) le temps de réponse du système, et (*3*) les interfaces homme-machine selon une charte graphique (Blanch, 2005) ou ergonomique (Scapin, 1986). Dans le premier cas, on dérive les cas de tests des cas d'utilisations tandis que dans le second cas, on dérive les cas de tests des spécifications non fonctionnelles sur la performance jusqu'à obtenir le temps de réponse requis. Le troisième cas consiste à vérifier point par point le respect de la charte, et cela jusqu'à l'obtention de conditions acceptables d'utilisation.

2.2.1.4 Tests d'acceptation

Les tests d'acceptation consistent à tester le système dans les conditions d'utilisation de l'utilisateur. L'utilisateur reçoit les cas de tests déjà préparés pour le test système. Il doit préparer des scénarii d'utilisation habituelle du logiciel à partir de ces cas de tests. Le test d'acceptation s'achève à la satisfaction de l'utilisateur.

2.2.2 Classification selon le niveau d'accessibilité

Il existe deux types ou techniques de conception type pour le niveau d'accessibilité :

- La technique de conception de test structurel : il s'agit de la *technique de conception de test, en général structurel*, fondée sur l'analyse de la structure interne du composant ou du système.
- La technique de conception de test de type boîte noire : c'est l'opposé de la première technique, c'est-à-dire, la *technique de conception de test, fonctionnel (ou non)*, qui n'est pas fondée sur l'analyse de la structure interne du composant ou du système.

Les tests résultant d'une technique de conception de test de structure (test structurel) vérifient la structure interne de l'objet, par exemple l'exécution des branches des instructions conditionnelles. Les tests unitaires sont souvent spécifiés à l'aide de telles techniques. Pour certains types de logiciels, des normes prescrivent les techniques de conception de test de structure à utiliser. Dans les tests résultant d'une technique de conception de test de type boîte noire les données en entrée et le résultat attendu sont sélectionnés non pas en fonction de la structure interne mais de la définition de l'objet. Ces tests sont les plus fréquents parmi les tests fonctionnels d'intégration et de recette, mais rien n'empêche d'utiliser ces techniques de conception pour définir des tests unitaires.

Par extension, on appelle couramment les tests issus de ces types de technique de conception tests boîte blanche et tests boîte noire. La référence à de telles façons de faire se trouve en général dans un plan de test.

2.2.3 Classification selon la caractéristique

On ne peut pas être exhaustif dans la détermination des caractéristiques des applications. Pour cela, on se contentera de quelques exemples pour la classification selon la caractéristique :

- Test de performance : vérifie que les performances annoncées dans la spécification sont bien atteintes ;
- Test fonctionnel : vérifie que les fonctions sont bien atteintes, par rapport aux attentes ;
- Test de robustesse : vérification que le logiciel reste toujours opérationnel ;
- Test de vulnérabilité : vérification de sécurité du logiciel ;
- …

En dehors du cas très particulier de systèmes extrêmement simples, il est impossible de tester exhaustivement un logiciel, car le nombre de configurations possibles croît comme 2^n où n est le nombre de bits dans la mémoire du calculateur ; le nombre de configurations accessibles, bien qu'inférieur, reste tout de même prohibitif. La réussite des tests ne permet donc pas de conclure au bon fonctionnement du logiciel. On essaye cependant, heuristiquement, de faire en sorte que si un bogue est présent, le test le mette en évidence, notamment en exigeant une bonne couverture des tests :

- couverture en points de programme : chaque point de programme doit avoir été testé au moins une fois.
- couverture en chemins de programme : chaque séquence de points de programme possible dans une exécution doit avoir été testée au moins une fois (impossible en général).
- couverture fonctionnelle : chaque fonctionnalité métier de l'application doit être vérifiée par au moins un cas de test.

Les bibliothèques de tests *JUnit* en langage Java, facilitent l'écriture de tests unitaires par l'apport des méthodes « assert » permettant de vérifier le comportement du programme. Selon la complexité du logiciel, des séquences de vérification globale peuvent s'avérer nécessaires. Celles-ci mettent en jeu la maîtrise d'ouvrage et toutes les composantes du projet, au-delà du logiciel lui-même (processus, organisation, formation, accompagnement du changement) : réception, qualification, certification, homologation, simulation, vérification d'aptitude au bon fonctionnement, etc... les termes varient selon les contextes.

2.3 Tests de non-régression

Les tests de non-régression ont pour but de vérifier que les évolutions apportées par une nouvelle version d'un logiciel n'altèrent pas les fonctionnalités préexistantes, de manière directe ou indirecte. L'intérêt principal de ces tests est de limiter les anomalies relevées lors de la recette de l'application. Ils viennent compléter les tests unitaires et les tests d'intégration en amont des tests de recette. Les tests de non-régression permettent de vérifier que les modifications apportées n'ont pas entrainé d'effets de bord non prévus qui pourraient dégrader le comportement du logiciel antérieurement validé. Ils portent sur l'exécution de tests déjà joués afin de s'assurer que le système répond toujours aux exigences spécifiées.

3 Tests d'IHM

La vérification et la validation par des tests sont une des phases importantes du processus d'assurance qualité d'un projet informatique ((Bryce and Memon, 2007) ; (Jérôme, 2004)). L'utilisateur d'une application ne voit que l'interface graphique de celle-ci, les tests d'IHM sont donc essentiels. En outre, ces tests couvrent toute l'application puisqu'une grande majorité des fonctionnalités sont directement accessibles depuis l'interface utilisateur.

Cependant, la part des tests d'IHM est souvent réduite (D'Ausbourg and Cazin, 1999). La raison en est qu'ils sont souvent difficiles à mettre en place et chers. Chers, car très consommateurs de ressources, essentiellement humaines, et difficiles, car les outils existants ne supportent que quelques technologies d'interfaces graphiques ou utilisent des techniques de test qui les rendent trop sensibles aux changements et évolutions de l'IHM pendant les phases de développement.

Après avoir présenté les spécificités des IHM, nous montrerons des exemples d'outils permettant le test d'IHM dans le contexte d'enregistrement des interactions avant de survoler les différentes démarches de mise en œuvre de l'automatisation des tests d'interfaces.

3.1 Spécificités des IHM par rapport aux tests

L'interface homme-machine doit satisfaire à des critères de qualité, dont le principal est l'utilisabilité du système interactif, c'est-à-dire être en adéquation avec les besoins et les capacités de l'utilisateur, afin de permettre à cet utilisateur d'atteindre ses objectifs au travers de trajectoires d'interactions intuitives et sûres. Aussi, au cours des vingt dernières années, les interfaces utilisateurs sont-elles devenues de plus en plus complexes. L'apparition de nouvelles techniques d'interaction, qualifiées généralement du vocable post-WIMP (Jambon et al., 2001), ont remis sur le devant de la scène le rôle important du développeur, et plus particulièrement du spécialiste d'interface. La conséquence principale de ceci est que les outils développés autour des concepts de génération d'interfaces, qui avaient éventuellement permis de prendre en compte des aspects liés à l'utilisabilité, deviennent inadaptés dès que l'on cherche à utiliser les nouvelles techniques d'interaction.

L'utilisabilité est née de la prise de conscience des difficultés d'utilisation des logiciels même lorsque leur interface se dit « conviviale ». Pour (Brun and Beaudouin-Lafon, 1995), « *L'utilisabilité sert à poser la frontière entre l'utilité réelle et l'utilité potentielle* ». Dans cette optique, l'*utilité* concerne l'aspect fonctionnel d'un logiciel interactif c'est-à-dire la présence de fonctionnalités prévues dans le cahier des charges, alors que l'utilisabilité concerne l'aspect opérationnel c'est-à-dire la possibilité pour les utilisateurs finaux d'utiliser aisément ces fonctionnalités dans leur contexte habituel. Cette notion a pris une telle importance en IHM qu'elle a été définie dans une norme (*ISO 9241-11, 1998*) : l'utilisabilité « *est le degré selon lequel un produit peut-être utilisé par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction dans un contexte d'utilisation spécifié* ». Cette définition est plus large que la seule facilité d'utilisation et prend en compte également l'utilité et l'acceptabilité du produit. Elle met en évidence que l'utilisabilité n'est pas une qualité intrinsèque de l'artefact mais est relative à un usage dans un certain contexte. Bastien et Scapin (Bastien and Scapin, 1993) proposent l'expression « *qualité ergonomique d'un logiciel* » pour éviter de restreindre le concept à la seule manipulation de l'interface.

Certains auteurs ajoutent une troisième composante qui est qualifiée d'accessibilité universelle pour tenir compte soit des personnes à besoins spécifiques comme les handicapés, les minorités (on parle alors de conception inclusive), soit de la diversité des dispositifs d'interaction comme les stations de travail, portable, téléphone, PDA (on parle alors de plasticité des interfaces (Thévenin, 2001) ; (Calvary et al., 2002).

À cette difficulté intrinsèque se surajoutent les difficultés d'ordre méthodologique. Les nouvelles méthodes centrées utilisateur s'appuient généralement sur un cycle itératif de conception. Il est donc nécessaire d'adapter les tests et leurs outils à cette méthodologie. L'intérêt des tests de non-régression est ici évident. Mais comment gérer dans ces tests les changements profonds d'interfaces ?

3.2 Outils actuels pour le test

Plusieurs études ont été réalisées pour permettre le test d'application à partir de l'enregistrement et le rejeu des événements issus des interactions de l'utilisateur. L'enregistrement des interactions de l'utilisateur permet la mise en place de données de test sur l'interface. La possibilité de rejouer ces interactions permet une vérification des résultats obtenus par rapport à l'exécution précédente. Plusieurs outils ont été réalisés dans le cadre des tests, la plupart sont orientés vers les tests fonctionnels. Même si certains outils commercialisés comme Rational Functional Tester[21] d'IBM ou encore QFTest[22] de Kapîtec Software, semblent prendre en compte des spécificités pertinentes des IHM, leurs capacités d'adaptation aux cycles itératifs sont très limitées. Nous présentons dans cette sous-section trois exemples de systèmes permettant la construction de tests à partir de l'enregistrement des interactions de l'utilisateur.

3.2.1 Jacareto

Ce logiciel, que nous avons déjà présenté au chapitre 3, a été initialement conçu comme un outil de génie logiciel dans le but de capturer et de rejouer les interactions sur des prototypes afin de visualiser les fonctions spécifiques des systèmes applicatifs mis au point à

[21] http://www-01.ibm.com/software/awdtools/tester/functional/
[22] http://www.kapitec.com/Produits/QFS/fr/index.html

partir de ces prototypes. Il a été renforcé en un outil favorisant les tests automatiques des fonctionnalités des systèmes. Jacareto combine des outils pour l'acquisition de données et de méthodes d'analyse des données. Des techniques spécifiques permettent l'enregistrement des actions (encore appelées événements) sur les programmes (considérés comme la fonction d'enregistrement) et de stocker ces interactions dans un format analysable (représentation symbolique définie par les concepteurs du système). Le programme avec lequel interagit l'utilisateur est appelé « application cible ». L'application cible doit être écrite en Java et peut être utilisée conjointement avec des composants de Jacareto sans avoir accès à son code source. Il n'y a pas nécessité d'adapter le programme pour enregistrer les interactions. C'est une approche externe du système applicatif mais ayant des connaissances de base sur la constitution des composants du programme.

Dans Jacareto (Figure 50), les actions sont conservées chronologiquement, combinées avec des informations temporelles. La succession linéaire des actions peut être consultée dans un composant de Jacareto, et les attributs de ces actions peuvent êtres inspectés (voir Figure 50). Tous les événements produits pendant le processus d'exécution de l'application cible sont enregistrés et stockés, aucune information n'est filtrée pendant l'acquisition des données. Ce qui signifie que tous les mouvements de souris, de clavier, des gestes articulatoires, etc., sont sauvegardés.

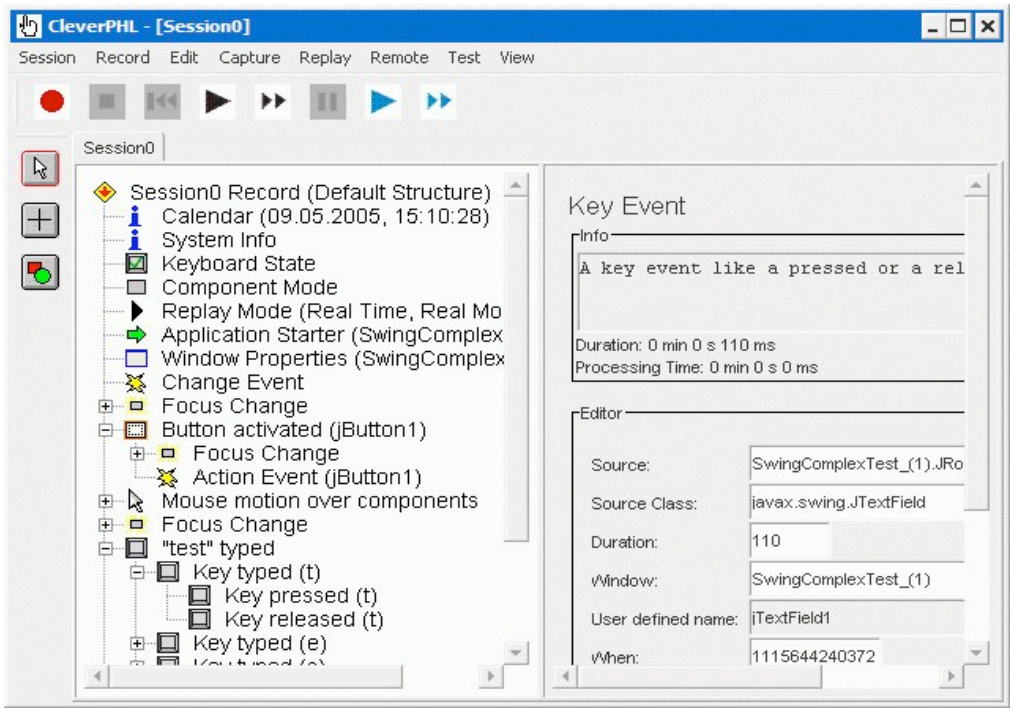

Figure 50 : Fenêtre principale de Jacareto (CleverPHL)

Les interactions de l'utilisateur sur le programme, contenues dans un dossier d'interactions, peuvent êtres rejouées. C'est le rejeu des actions ou événements. À cette fin, l'application cible est de nouveau lancée, et tous les événements souris et clavier lui sont envoyés. Par conséquent, le pointeur de la souris est automatiquement transféré sur le coin supérieur gauche de l'application cible, et les clics souris ainsi que les entrées clavier sont effectués par le système en lieu et place de l'utilisateur. L'impression visuelle du comportement est reconstituée à partir des interactions enregistrées et rejouées sur une instance réelle de l'application cible afin qu'elle puisse être observée.

Le composant *CleverPHL* de *Jacareto* dispose d'un mécanisme de classification des interactions de l'utilisateur par l'intermédiaire de « catégorie ». Un événement est considéré comme un élément de la catégorie. Les catégories peuvent êtres définies à différents niveaux, formant ainsi une hiérarchie : la catégorie des éléments d'un niveau inférieur peut faire partie d'une catégorie des éléments d'un niveau supérieur, et ainsi de suite. Sur l'exemple, on peut voir que les événements « Key typed », eux-mêmes composés d'événements « Key pressed » et « Key released », sont réunis en une action « "test" typed ». Les catégories sont définies par la programmation d'algorithmes de détection de séquences d'actions spécifiques. Ce système de catégories se compose d'un ensemble d'algorithmes de détection, et des parties de l'enregistrement linéaire des interactions sont classées à l'aide d'interaction de ces algorithmes. Le résultat est une vue hiérarchique de l'interaction de l'utilisateur sur le programme. La classification des parties de l'interaction est basée sur des méthodes dérivées de la conception du compilateur (Aho et al., 2007). Elle est similaire à la création d'un arbre de syntaxe suivant la grammaire d'un langage formel.

Dans *Jacareto*, l'enregistrement et le rejeu ne se font pas de façon automatique. Au départ, c'est à l'utilisateur de cliquer sur le bouton d'enregistrement des événements dans l'interface de *CleverPHL* après avoir chargé l'application cible à partir du menu « Session » de la même interface. L'enregistrement des actions de l'utilisateur sur le programme se fait alors de façon automatique ainsi que la mise en place de la vue hiérarchique des catégories. L'interruption de l'enregistrement se fait grâce à l'activation de la fonction « Stop » soit par l'intermédiaire du menu déroulant soit par le bouton « Stop » de l'interface. L'utilisateur a la possibilité d'enregistrer la suite des actions réalisées lors de la session et de les réutiliser à sa guise. Comme pour l'enregistrement, le rejeu est aussi actionné par l'utilisateur à travers l'interface offerte par *CleverPHL* de *Jacareto*. Toujours, deux possibilités sont offertes à l'utilisateur pour cela : soit par le bouton « Replay » de la barre d'outils, soit par le menu « Replay » dans la barre de menu. Le rejeu est global ; il ne peut se faire étape par étape. C'est l'ensemble de la suite des événements qui est lancé sur l'interface de l'application, mais l'utilisateur a la possibilité de choisir la représentation et les déplacements de la souris ou non. Il peut aussi arrêter le rejeu même si la suite des événements n'est pas terminée.

Jacareto est capable de structurer, à partir du bas niveau, les événements, allant jusqu'à un niveau d'interaction syntaxique (« Button activated », « "test"typed », etc.), Cependant, aucune sémantique propre à l'application ne peut être associée, et donc contrôlée.

3.2.2 ReplayJava

ReplayJava[23] est un système utilisé pour enregistrer et rejouer les actions de l'utilisateur lors d'une session. Il peut aussi servir à générer des actions pour une session utilisateur. Réalisant l'enregistrement et le rejeu des événements utilisateur, ReplayJava permet de tester l'état d'une application graphique légère basée sur Swing (Java). ReplayJava est basé sur la génération d'événements pour la file d'événements système. Une application avec une interface utilisateur graphique est écrite pour réagir aux événements postés dans la file des événements par le système (en général le système de fenêtrage). L'outil tient compte du fait que certains événements ne sont pas postés dans la file d'événements mais sont directement livrés à la composante graphique correspondante. Ce qui le conduit à définir un niveau un peu plus élevé pour ces actions.

ReplayJava agit de manière extérieure à l'application. Il permet de lancer ou de charger une application en ligne de commande et suit les traces à travers la file des

[23] http://jan.netcomp.monash.edu.au/java/replayJava/

événements système. C'est de là que l'outil récupère au fur et à mesure la suite des événements postés.

Un outil d'enregistrement et de rejeu des actions utilisateur doit être capable de localiser les composantes graphiques d'une interface utilisateur graphique. ReplayJava est apte à cela. Il enregistre les objets avec des coordonnées absolues et tente une nomination distincte de chaque objet enregistré. ReplayJava décompose l'interface graphique ou classe les composantes de l'interface suivant l'arborescence des composants Swing. Cela définit une hiérarchie des composants de l'interface et facilite ainsi la nomination ou l'identification des objets. Lors du rejeu des actions, pour retrouver un composant destinataire, un parcours hiérarchique est effectué dans l'arbre des composants. L'envoi de l'événement par la file d'attente des événements du système se fait en ajoutant le composant cible de l'application par encapsulation des coordonnées et des caractéristiques de l'événement.

Les événements de haut niveau sont considérés comme des événements sémantiques par l'outil ReplayJava. Des méthodes sémantiques sont définies sur la base du modèle MVC utilisé par Swing afin de mieux appréhender le comportement et l'identification de certains objets. À partir du modèle MVC, il est possible de manipuler le modèle interne des éléments graphiques. Par exemple, pour réaliser le clic sur un bouton, on se sert du modèle MVC pour redéfinir à nouveau le composant ou du moins une nouvelle instance en mettant en exergue le principe de fonctionnement des objets « bouton » de Swing.

L'automatisation des actions de l'utilisateur lors d'une session permet la mise en place de tests pour l'interface utilisateur graphique. Ces actions peuvent être enregistrées dans un fichier et être utilisées de nouveau.

3.2.3 Abbot

Abbot[24] (Figure 51) est un environnement réalisé pour permettre à l'utilisateur de tester l'interface graphique Java. C'est une extension des tests unitaires *JUnit*. Il est composé de *Abbot*, qui permet de suivre la programmation des composants d'interface utilisateur, et Costello (construit sur *Abbot*) qui permet de lancer facilement, d'étudier et de contrôler une application. Le système peut être utilisé à la fois avec des scripts ou du code compilé. L'outil *Abbot* peut être invoqué directement à partir du code Java (dans le cas des tests unitaires) ou dans un cadre plus simple et structuré, en utilisant des scripts basés sur le langage XML. Ces deux méthodes sont conçues pour être utilisées avec l'outil de test *JUnit*. L'usage de *Abbot* se résume à lancer une interface graphique, invoquer de manière arbitraire les actions de l'utilisateur sur celle-ci, et examiner son état. L'environnement *Abbot* comprend également l'éditeur de *Costello* qui facilite l'édition de scripts. L'éditeur supporte l'enregistrement des actions de l'utilisateur dans un script.

Abbot fournit un cadre pour l'examen de l'interface graphique indépendamment de l'état actuel du code. En effectuant un test sur un premier développement avec beaucoup de tests unitaires, *Abbot* peut fournir au développeur des outils nécessaires pour écrire des tests unitaires individuels. Si l'on dispose d'une base de code sans test unitaire, on peut utiliser la partie d'édition de scripts de *Abbot* pour débuter la création de tests fonctionnels échafaudés autour de l'application jusqu'à ce qu'ils soient suffisamment stables pour soutenir le « refactoring » et l'ajout de tests unitaires supplémentaires.

[24] http://abbot.sourceforge.net/doc/overview.shtml

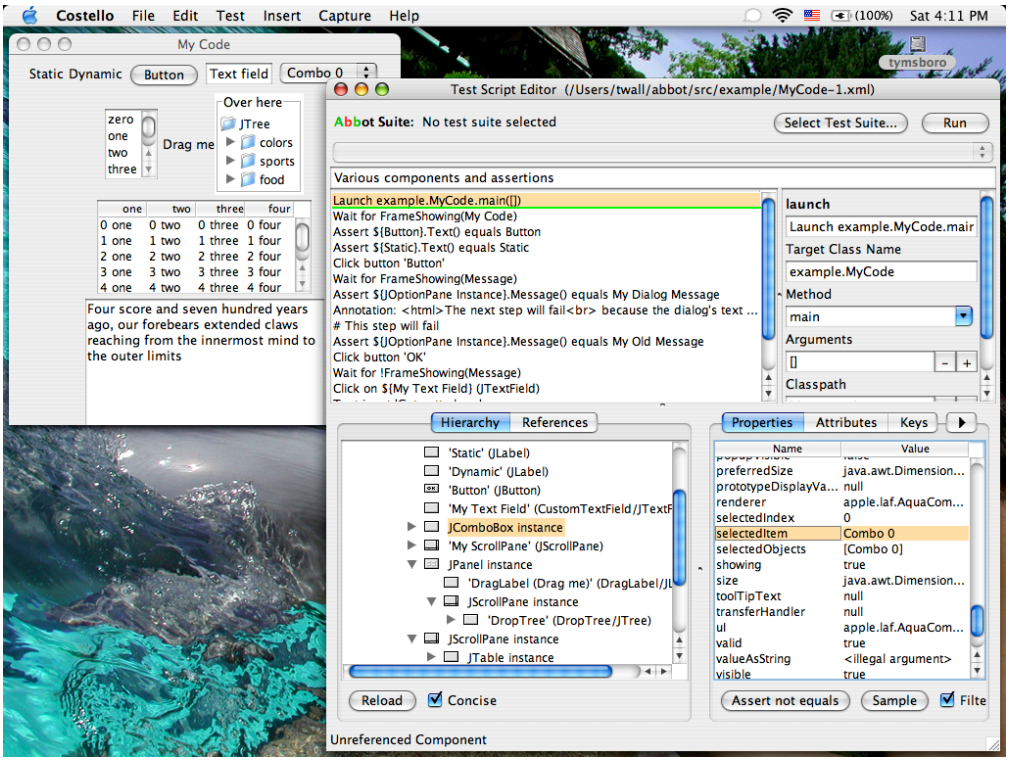

Figure 51 : Ecran de Costello

En général, les tests avec *Abbot* consistent à obtenir les références des composants d'interface graphique soit en exécutant les actions de l'utilisateur sur ces composants, soit en faisant certaines affirmations au sujet de leur état. Le système propose des outils à cet effet. Ces opérations peuvent êtres effectuées à partir soit d'un haut niveau de script, soit à partir du code Java (par exemple dans une méthode JUnit *TestCase*). Le composant fondamental de *Abbot* est « Robot », le générateur des événements utilisateur et la racine de tous les tests. Il fournit des événements au niveau des composants de base sur lesquels on peut construire plusieurs objets testeurs plus complexes. En l'absence d'un testeur spécifique d'objet pour un composant donné, les événements prévus par les classes du composant « Robot » sont suffisants pour reconstruire toutes actions de l'utilisateur. Pour chaque composant à tester, il y a un testeur de classe qui fournit les actions utilisateur et les tests spécifiques nécessaires à ce composant. La classe de base prévoit des actions par défaut pour la plupart des actions utilisateur communes, y compris le clic sur un bouton de la souris, la sélection dans un menu, et même en tapant du texte. Le système de test est extensible en fournissant d'avantage d'actions spécifiques ou de propriétés des composants personnalisés. Lorsque l'on utilise un script, toutes les informations nécessaires pour faire tourner le test sont encapsulées dans le script lui-même afin de fournir l'indépendance par rapport à n'importe quel environnement de test.

L'éditeur de scripts est fourni pour faciliter la création et la maintenance des scripts dans Abbot. L'éditeur supporte l'enregistrement direct des événements de l'utilisateur pour faciliter l'écriture de script. Il peut aussi rejouer ces événements. Un composant de Abbot, le composant « Recorders » fournit à l'éditeur de scripts la possibilité de capturer les événements de haut niveau sémantique au lieu des événements souris et clavier. « Recorders » prend en compte la plupart des composants standard de Swing, mais les événements sont correctement capturés même en l'absence d'un enregistreur personnalisé pour un élément donné. L'éditeur comporte des plug-in d'enregistrement pour supporter l'ajout de nouveaux composants personnalisés.

En dehors de l'objectif principal de la fiabilité d'automatiser les tests de composants graphiques et des composants d'interface utilisateur, Abbot poursuit d'autres objectifs. Le système Abbot permet (1) la reproduction de la saisie de l'utilisateur pour les tests. Une raison de ne pas tester un composant graphique de l'interface utilisateur est qu'il n'est pas facile pour un développeur de simuler la saisie de l'utilisateur. Il est possible de faire des tests unitaires de ces composants. Abbot autorise (2) l'écriture de scénarii de contrôle et d'inspection des actions. Plutôt que d'écrire du code qui exige une nouvelle compilation, les scripts de test sont dynamiquement interprétés, ce qui est plus approprié pour ces niveaux de test. Les schémas d'interfaces graphiques utilisateur ont tendance à changer au fil du temps. Le mécanisme systématique d'enregistrement et de rejeu des flux d'événements ne se passe pas toujours correctement si on est en exploitation du système en coordonnées absolues, les différentes plates-formes ont des dimensions standard différentes pour chaque composant. Dans ce contexte, Abbot ne se soucie pas (3) des emplacements des composants du moment où il les consulte dynamiquement à l'aide de marquage par identifiants. Abbot (4) permet de spécifier des actions sémantiques de haut niveau, mais utilise des événements système de bas niveau pour les mettre en œuvre. Il utilise la classe *java.awt.Robot* pour reproduire directement les événements utilisateur. Mais ce niveau est très basique pour être utilisé pour les tests unitaires. Abbot construit un niveau d'abstraction au-dessus de cette classe afin de faciliter la maintenance des tests unitaires. Le système Abbot se base sur le principe qu'il est plus facile pour l'utilisateur d'analyser un groupe d'événements de bas niveau dans lequel chaque élément correspond à une instruction de l'utilisation de l'application. Abbot supporte (5) l'enregistrement direct des événements sémantiques de haut niveau. A l'aide de l'éditeur Costello, Abbot enregistre automatiquement une séquence d'actions de l'utilisateur et les enregistre dans un script. L'utilisateur n'a pas à écrire du XML à la main, ou à modifier les scripts spécifiques de façon individuelle à chaque étape de l'exécution. Abbot permet enfin (6) l'extension des actions utilisateur enregistrées et leur généralisation. Les composantes Tester et Recorders objects ainsi que plusieurs autres composantes sont extensibles à l'appui des composants que l'on souhaite tester.

Abbot est sûrement l'un des outils de test d'interface les plus aboutis aujourd'hui. Malgré tout, il souffre de deux problèmes importants. Tout d'abord, sa capacité à absorber les modifications d'interface dans le cadre d'un développement itératif est très limitée. La mise en œuvre de tests de non-régression efficace est donc illusoire. D'autre part, malgré la possibilité d'ajouter la sémantique des actions, le système ne peut pas faire une vraie relation avec les buts de l'utilisateur.

3.2.4 Synthèse

Les outils présentés dans cette section permettent d'enregistrer et de rejouer les interactions de l'utilisateur soit dans le but de l'automatisation de rejeu soit dans le but de test de fonctionnement. Jacareto ne tient compte que des interactions de bas niveau et n'a aucune indication de la sémantique, ni des tâches de l'utilisateur. Il effectue son enregistrement, dans le but tout au plus de réaliser des tests de non-régression. C'est aussi le cas de ReplayJava où aucune sémantique des actions utilisateur n'est connue.

Les spécificités des IHM ne peuvent être prises en compte par ces outils. Ils ne disposent pas de connaissances sur les critères de conception et n'établissent aucun lien pour permettre cette vérification.

3.3 Principaux travaux de recherche

Sans le support d'outils, les tests d'IHM doivent êtres effectués par un testeur qui exécute manuellement des jeux d'actions prédéfinis et qui vérifie visuellement la réaction de l'application à ces actions. Cette procédure peut être automatisée par un outil de test d'IHM qui simule les actions de l'utilisateur sous forme d'événements clavier et souris. Plusieurs démarches ont été proposées dans le cadre d'assister le testeur dans ses activités. Des méthodes de test automatiques ont été proposées pour les systèmes interactifs. Ces méthodes utilisent un modèle pour générer les tests, comme la méthode proposée dans (Jambon et al., 2000) qui utilise un modèle de tâches simplifié. La méthode de (Richard and Daniel, 1997) s'appuie sur un modèle d'états finis à variables tandis qu'avec la méthode de (Memon et al., 2001) l'interface est modélisée au moyen d'un ensemble d'opérateurs hiérarchiques où, pour chaque opérateur, une pré-condition et une post-condition doivent être définies. Dans la suite de cette section nous approfondissons quelques méthodes, et relevons leurs limitations par rapport à nos besoins.

3.3.1 Utilisation de structure hiérarchique

Memon (Memon et al., 2001) préconise d'abord une représentation de l'interface utilisateur suivant un graphe afin de l'utiliser comme source dans un environnement de tests. Il part du fait que l'interface graphique est construite à partir de spécifications précises, susceptibles d'être traduites par le testeur pour tester l'interface. L'interface est développée de son côté à partir de ces spécifications dans le cahier de charges. Le testeur, utilise le même cahier de charges pour générer ses cas de tests. Memon préconise la définition de spécifications exécutables permettant la génération de l'interface, et cela à partir d'un environnement de génération automatique. Le point de départ de la démarche de Memon est la définition de l'interface graphique en termes d'opérateurs. La spécification d'une interface graphique est la même lors de la conception, de l'implémentation et du test. Cette spécification peut être généralisée en fonction des dispositifs d'implémentation.

L'approche proposée par Memon utilise un modèle hiérarchique pour guider la génération de cas de tests. Elle définit un ensemble d'opérateurs qui sont organisés de façon hiérarchique. Les opérateurs du niveau supérieur sont construits à partir d'autres plus simples. Les opérateurs simples correspondent aux actions de l'utilisateur. Chaque opérateur possède une pré-condition qui doit être vraie avant l'exécution de l'opérateur, et une post-condition, vraie après la réalisation de l'action. Le testeur spécifie un ensemble d'opérateurs, un état initial et un état final pour le planificateur (fondé sur les principes de l'intelligence artificielle). Ce dernier produit une séquence d'opérateurs qui change l'état initial en état final. Il génère les cas de tests de niveau supérieur d'abstraction. Le processus de composition conduit à des cas de tests de bas niveau qui sont des événements utilisateur. Ces cas de tests sont utilisés pour tester l'interface graphique. Ils sont soumis à un exécuteur automatique de test (composant intégré dans la plate-forme) qui exécute chaque événement du plan de test.

Cette méthode permet de générer des cas de tests, et de produire une bonne couverture de tests. Cependant, l'automatisation n'est pas instrumentée. De plus, elle impose une méthodologie et des formalismes spécifiques, qui n'ont pas prouvé leur généralité. Enfin, elle est très centrée système.

3.3.2 Utilisation de flux de données

Un exemple de modélisation du système sous la forme de flux de données pour le test correspond aux travaux d'Ausbourg (D'Ausbourg, 1998). Dans cette approche, ce modèle

peut être exprimé en utilisant les équations de flux. Cette approche s'appuie sur le concept d'interacteur. L'interface est modélisée par un réseau d'interacteurs. Un interacteur est un automate qui réagit à des actions d'entrée en modifiant son état et en générant des événements de sortie. Les actions d'entrée sont des actions de l'utilisateur (par exemple enfoncement d'une touche) et des réactions générées par l'application. Les événements de sortie sont des actions provenant de l'interacteur vers l'application et sa présentation. Ces actions correspondent à des commandes d'activation de traitements de données ou des commandes de visualisation. Un réseau d'interacteurs définit un interacteur plus complexe et se construit par l'intermédiaire d'un opérateur de composition parallèle des interacteurs. Un tel modèle d'interacteur décrit en fait un réseau de processus parallèles et communicants : les processus sont des automates et la communication entre les automates est assurée par la connexion des sorties de certains automates aux entrées d'autres automates.

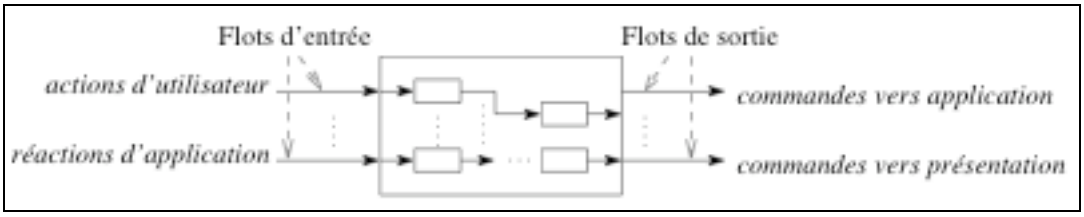

Figure 52 : Interacteur à flots de données

Cette approche de vérification d'IHM adopte le formalisme Lustre pour la modélisation et la validation des interfaces homme-machine. Le langage Lustre est fondé sur le modèle à flots de données. Il représente un système à travers un réseau d'opérateurs agissant en parallèle sur leurs entrées. Dès que les entrées nécessaires à un opérateur sont disponibles, ce dernier calcule les sorties correspondantes. Les sorties des opérateurs peuvent être connectées aux entrées d'autres opérateurs. Les suites d'événements d'entrée et de sortie des opérateurs sont appelées « flots ». Le modèle générique utilisé dans cette approche représente un interacteur comme un réseau d'opérateurs sur flots booléens (Figure 52). Les flots d'entrée des opérateurs sont associés aux actions de l'utilisateur et aux réactions générées par l'application. Les flots de sortie sont associés aux actions en direction de l'application et de la présentation. Le comportement des interacteurs en Lustre s'exprime sous la forme de dépendances entre flots de sortie et flots d'entrée. Lustre permet également la description modulaire des interacteurs (sous forme de nœuds). L'approche a été appliquée à la production de spécifications d'interfaces à fenêtres. Le modèle Lustre de l'interface est extrait automatiquement à partir de la description de l'interface dans les langages UIL et C. Ce modèle est utilisé pour la vérification de propriétés à satisfaire par l'interface au moyen de l'outil de model-checking Lesar (D'Ausbourg et al., 1996). L'analyse de propriétés sur le modèle Lustre permet d'améliorer la description de l'interface dans l'étape de conception. Les propriétés vérifiées sur le modèle Lustre sont des propriétés génériques ou spécifiques. Les propriétés génériques sont automatiquement extraites comme l'observabilité et l'honnêteté, tandis qu'un éditeur graphique est utilisé pour exprimer des propriétés spécifiques.

3.3.3 Utilisation d'objets coopératifs interactifs (PetShop)

Cette approche est fondée sur les réseaux de Petri et le concept d'objet. Elle permet la spécification et la validation formelles d'une application interactive (Bastide, 2000). La spécification formelle peut être utilisée pour la conception de l'application interactive finale. La spécification formelle est réalisée dans un formalisme appelé "Objets Coopératifs Interactifs" ou ICO (en anglais "Interactive Cooperative Objects"). C'est un langage orienté

objet où le comportement des objets est décrit par un réseau de Petri de haut niveau. Un modèle ICO est une description formelle d'un système construit de plusieurs objets communicants. Le comportement des objets et leurs communications sont décrits par des réseaux de Petri. Quand deux objets communiquent, l'un (le client) demande un service et l'autre (le serveur) exécute le service. Dans le formalisme ICO, un objet est une entité décrite par quatre composants : le comportement, les services, l'état et la présentation.

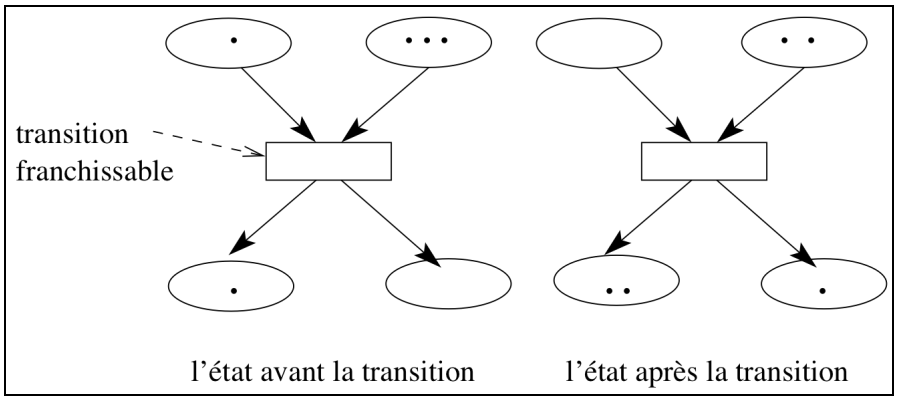

Figure 53 : Exemple de réseau de Petri

Un ICO offre un ensemble de services qui définissent l'interface de l'objet avec son environnement. L'environnement peut être les usagers ou d'autres objets de l'application. Chaque service est lié au moins à une transition et un service est disponible quand au moins une de ses transitions est franchissable. On distingue deux sortes de services : les services utilisateur (dans ce cas, l'environnement correspond aux usagers) et les services aux autres objets. La présentation d'un objet est son apparence externe. C'est un ensemble d'éléments d'interface (widgets) organisés dans un ensemble de fenêtres. L'interaction de l'utilisateur sur le système a lieu uniquement au travers de ces widgets. Chaque action de l'utilisateur sur un widget peut déclencher un service utilisateur du modèle ICO. La relation entre services utilisateur et widgets est entièrement décrite par la fonction d'activation qui associe à chaque couple (widget, action de l'utilisateur) un service utilisateur. L'interaction du système vers l'utilisateur est spécifiée par la fonction de rendu qui associe à chaque nœud (places ou transitions) un ensemble de widgets qui peuvent être utilisés pour rendre l'information à l'utilisateur. En général, le rendu est soit sur les places, soit sur les transitions.

L'application interactive est spécifiée formellement avec plusieurs objets coopératifs interactifs. Par exemple, si l'application est conçue selon l'architecture MVC (Modèle, Contrôleur, Vue), il y a trois sortes d'ICO : le modèle, le contrôleur et la vue. Dans la classe ICO Modèle, les fonctions d'activation et de rendu sont vides. Dans la classe ICO Contrôleur, la fonction de rendu est vide. Dans la classe ICO Vue, la fonction d'activation est vide. Ensuite, cette spécification formelle peut être exécutée, testée et vérifiée en utilisant les outils relatifs aux réseaux de Petri.

PetShop propose l'approche ICO dans le sens où il modélise le système interactif (domaine, dialogue et présentation) suivant des objets ICO. L'outil PetShop est un éditeur-interpréteur d'ICO. Il permet de spécifier le modèle du domaine au travers d'un ensemble de classes d'objets coopératifs (CO-classes). Le comportement d'un objet est décrit au moyen d'un réseau de Petri appelé ObCS (Object Control Structure). L'ensemble des ObCS décrit le système. Le concepteur peut évaluer chaque instance d'une CO-classe par l'intermédiaire de l'outil PetShop. Il peut éditer, exécuter et analyser son réseau de Petri sans perte de contexte.

PetShop permet d'établir une liaison entre le modèle de tâche (décrit indépendamment du système) et le modèle du système, de façon manuelle. Cette liaison est assurée par un éditeur de correspondance. Ce dernier extrait, dans un premier temps, l'ensemble des tâches interactives qui doivent être exécutées par le système, puis l'ensemble des services utilisateur de la spécification du système. À partir de cette extraction, le concepteur effectue la mise en relation d'une tâche interactive avec un service utilisateur. Cette étape peut être désignée comme la spécification abstraite de l'interface du modèle. Le niveau de spécification concrète, dans l'approche PetShop est représenté par le prototype du système. Il est obtenu en définissant le lien (fonction d'activation) entre l'ensemble des objets de la présentation et des ObCS. Ces éléments de la présentation sont obtenus par l'intermédiaire du générateur de présentation JBuilder. Toute modification apportée aux ObCS (modification d'une transition par exemple) est directement perceptible sur l'interface. PetShop permet de tester l'application en cours de développement mais ne propose pas la génération qui permettrait aux utilisateurs finaux d'utiliser directement le système (choix sur les utilisateurs).

3.3.4 Synthèse

Les contributions passées en revue ici montrent les possibilités de mise en œuvre de démarche raisonnée d'ingénierie des systèmes interactifs, basée pour la plupart, sur des techniques formelles. Dans le cas des interacteurs, les travaux s'intéressent principalement à la composition d'interacteurs pour modéliser des interfaces complexes. D'autres s'intéressent plus à la modélisation complète d'un système interactif (description du noyau fonctionnel, le dialogue et la couche présentation). Ces approches sont trop partielles et manquent d'intégration au processus complet d'ingénierie des systèmes interactifs. En effet, les propriétés vérifiées ne couvrent qu'une partie des besoins et interviennent à des étapes distinctes du développement complet d'ingénierie. D'autre part, la place de l'utilisateur dans la modélisation (tâches utilisateur par exemple) n'est pas prise en compte, ou partiellement. La spécificité des IHM n'est pas toujours validée par ces approches. De façon générale, ces approches de validation manquent de sémantique utilisateur, sont hétérogènes et souffrent des inconvénients des systèmes de preuves.

Dans tous les cas, la vérification requiert une démarche lourde de spécification que la plupart des concepteurs des applications interactives ne peuvent pas effectuer. Les méthodes formelles sont utilisées dans la phase de conception. Elles modélisent le système et prouvent les propriétés sur le modèle. Cependant, il n'y a pas une garantie de l'équivalence entre ce modèle et le système final. Pour ces raisons, il est important de bien tester et, de façon automatique, les systèmes interactifs. Il y a peu de méthodes pour tester automatiquement ces systèmes. Nous proposons l'utilisation de la PsE, plus précisément de sa technique d'enregistrement et de rejeu pour automatiser les tests de conformité de l'IHM au modèle de tâches prescrit.

4 Vers une automatisation des tests d'IHM à l'aide de la PsE

Les tests automatisés aident à réduire significativement les temps et les coûts liés à la phase de test. De plus, l'automatisation est le meilleur moyen pour s'assurer que des tests sont réalisés régulièrement et uniformément, permettant ainsi la détection précoce des erreurs, donc une qualité logicielle améliorée et une mise à disposition sur le marché dans des délais plus courts.

Les tests d'Interface Homme-Machine sont les seuls permettant de tester l'application de bout en bout en simulant (ou en exécutant) les actions de l'utilisateur sur l'interface graphique de l'application. Le test d'IHM peut être considéré comme une technique de « test à boîte noire » très utile pour les tests fonctionnels pour lesquels la connaissance interne du comportement ou de l'implémentation de l'objet testé n'est pas nécessaire. La plupart du temps, les tests IHM sont assimilés aux tests utilisateurs et l'application est validée par un ou plusieurs utilisateurs sur la base d'un cahier de tests regroupant l'ensemble des scénarii utilisateur devant être joués, puis validés ou rejetés par l'utilisateur testeur. Étant donné le temps consommé par ce type de tests, ils ne sont effectués généralement qu'aux étapes importantes de livraison d'un projet et sur des applications nécessitant une très bonne qualité.

Les outils actuels se basent uniquement sur les fonctionnalités de l'application et ne se soucient guère des modifications d'enchaînement des actions et de la disponibilité de composants d'interaction pouvant engendrer des erreurs (Sanou, 2007). Cela ne favorise pas une utilisation optimale de l'application par l'utilisateur quel que soit l'état de l'interface. La prise en compte de l'utilisateur, ou l'intégration des activités de l'utilisateur aux tests d'IHM est donc primordiale. Il nous semble d'une grande importance de mettre en place une automatisation des tests prenant en compte l'activité de l'utilisateur en fonction de ses spécifications de départ.

Dans cette partie, nous allons présenter une technologie qui permet de simplifier les tests d'IHM et de les automatiser afin de pouvoir les rejouer plus souvent et ainsi améliorer la qualité des développements d'applications graphiques interactifs. Nous nous appuyons sur la PsE en utilisant la technique d'enregistrement des interactions, et plus particulièrement la boîte à outils (PbDToolkit) que nous avons présentée au chapitre 3. Nous présentons l'interface utilisateur et le modèle de tâches en soulignant le rôle de ce dernier dans la construction de l'interface utilisateur. Dans une deuxième partie, nous montrons les liens que l'on peut définir entre les actions élémentaires de l'interface utilisateur et les tâches élémentaires du modèle de tâches. Enfin, la troisième partie présente la technique d'implémentation par la génération de scénarii à partir de l'interface utilisateur en exécution.

4.1 Interface utilisateur et modèle de tâches

Les tests d'IHM que nous avons présentés tout au long de nos sections précédentes n'intègrent pas ou partiellement l'activité de l'utilisateur, pourtant nécessaire à la validation des actions de celui-ci sur le système. Il semble donc important de considérer l'activité ou le modèle de l'utilisateur lors du test (Sanou et al., 2008b). L'activité de l'utilisateur peut être intégrée à travers le modèle de tâches. Les modèles de tâches ont été conçus dans le but de modéliser l'activité humaine. Cependant, de nombreux auteurs ont cherché à utiliser ces modèles dans la construction des interfaces, et ces derniers ont ainsi largement évolué vers la prise en compte de spécificités d'interaction. De nombreuses techniques ont été employées à différents niveaux de spécification du système interactif. Cependant, les approches existantes ne permettent pas (même si certaines s'en rapprochent) de modéliser complètement le système, ni de vérifier automatiquement le système. C'est donc dans l'optique de permettre la vérification automatique du système par rapport à des éléments de conception que nous proposons d'établir un lien entre l'interface réalisée et la modélisation de l'activité de l'utilisateur au départ de la conception.

4.1.1 Modèles de tâches

Pour exprimer les besoins des utilisateurs, souvent non-informaticiens, de nombreuses notations de description ont été proposées dans le domaine des IHM. Ce sont les modèles de tâches. Ces modèles sont souvent éloignés des implantations informatiques. Les modèles de tâches possèdent des qualités et des finalités très différentes. Différentes classifications ont été proposées pour résoudre ce problème. Un effort salutaire de simplification a été proposé au travers de leur regroupement en deux catégories dans (Baron, 2003) ; les modèles d'analyse de tâches et les modèles de description de l'interface homme-machine.

L'analyse de tâches consiste à collecter des informations sur la façon dont les utilisateurs accomplissent une activité. L'acquisition de ces informations est obtenue par les récits des utilisateurs au moyen de lectures de rapports, d'interviews, de vidéos ou de simulations. Mais cette analyse doit être indépendante de toute idée d'interface à réaliser et ne doit pas comporter de sous-entendus sur les dispositifs d'interaction. Elle doit donc rester à un haut niveau d'abstraction. Ces modèles liés à l'analyse de tâches servent essentiellement de notations support pour la collecte et l'analyse des informations. En l'espèce, ils sont rarement formalisés. Rentrent dans cette catégorie les modèles HTA (Hierarchical Task Analysis) (Shepherd, 1989) et MAD (Méthode Analytique de Description) (Scapin and Pierret-Golbreich, 1989) par exemple.

Les modèles de description de l'IHM définissent la vue que l'utilisateur aura du système interactif. Ces modèles s'inscrivent dans une logique de continuité par rapport à celle de l'analyse de la tâche. En plus, ils intègrent le retour d'information en provenance de l'interface. Ils améliorent la communication entre l'ergonome et le concepteur de logiciel. On peut citer ici des modèles comme UAN (Hartson and Hix, 1992) et CTT (Paternò et al., 1997).

Une tâche est un but que l'utilisateur vise à atteindre à l'aide d'un système interactif (Normand, 1992). Ce but est assorti d'une procédure (ou plan) qui décrit les moyens pour l'atteindre. Un modèle de tâches décrit l'interaction entre l'utilisateur et le système interactif en termes de tâches et sous-tâches. Dans ce modèle, les tâches sont représentées d'une manière hiérarchique : une tâche est composée de sous-tâches liées par des opérateurs temporels. Cela signifie que le modèle de tâches fournit des informations sur les sous-tâches qui doivent être exécutées afin d'accomplir une autre tâche plus complexe, ainsi que des informations sur le comportement du système interactif : la structure hiérarchique d'une tâche montre les sous-taches à exécuter afin de réaliser cette tâche tandis que la représentation comportementale prend en compte les conditions pour réaliser la sous-tâche.

Malgré la grande diversité de modèles de tâches proposés dans la littérature, il est possible de dégager un certain nombre de points communs (voir à ce sujet (Limbourg and Vanderdonckt, 2003)). Parmi ces modèles, nous présentons le dernier né, largement inspiré de MAD, K-MAD (Lucquiaud, 2005a) (Kernel of Model for Activity Description), que nous avons utilisé. Ce choix a été motivé par la présence de riches opérateurs qui permettent de couvrir le cadre de description d'une application interactive, et la disponibilité d'un outil, K-MADe (Baron et al., 2006). Cet outil édite et effectue des vérifications sur les modèles K-MAD. K-MADe, à l'instar de CTTE (Paternò et al., 2001), édite les modèles en respectant les contraintes du formalisme, construit des scénarii et simule l'exécution de ces modèles. Il va plus loin que CTTE puisqu'il prend en compte les valeurs des objets concernés par les conditions définies dans le modèle. De plus, une interface d'échange, basée sur une description XML, permet d'utiliser d'autres modèles au format K-MAD, ou des scénarii, ce qui est essentiel pour nos besoins.

4.1.2 K-MAD

K-MAD est un modèle hiérarchique. Il s'agit du modèle N-MDA (Noyau du Modèle de Description de l'Activité), ((Lucquiaud et al., 2002) ; (Lucquiaud, 2005b)) ou (en anglais) K-MAD (Kernel of Model for Activity Description). Il représente l'activité de l'utilisateur sous forme d'arbre de tâches, du plus général (tâche mère) au plus détaillé (actions élémentaires), en passant par des tâches intermédiaires (tâches filles).

Une tâche est définie par un nom, un numéro (automatique), un but, une durée, un retour d'information (effets observables par l'utilisateur) :

- une tâche peut donner lieu à des observations (de nature textuelle) ;
- une tâche est caractérisée par un niveau d'importance (peu, assez, très, non déterminée) ;
- une tâche est caractérisée par une fréquence (élevée, moyenne, faible) ;
- une tâche est caractérisée par un exécutant :
    - « Utilisateur » : tâche réalisée par l'utilisateur autre qu'une action sur le système ;
    - « Système » : tâche réalisée par le système ;
    - « Interactif » : tâche déclenchée par l'utilisateur et réalisée conjointement sur le système ;
    - « Abstrait » : soit la tâche n'est pas complètement décrite, soit la tâche contient des sous-tâches d'exécutants différents ;
    - « Inconnu » : tâche dont l'état est non déterminé.
- quand l'exécutant est un utilisateur (individu intervenant dans l'activité décrite), la tâche est caractérisée par des modalités (sensori-motrice, cognitive). Les différents utilisateurs associés aux tâches sont identifiés en tant qu'acteurs, caractérisés par un nom, une expérience (novice, moyenne, expert) et une compétence.

À une tâche, sont associées des *conditions d'exécution éventuelles* : des pré-conditions (condition à respecter pour que la tâche puisse être exécutable), des itérations (condition de répétition d'une tâche) ; et des *effets de bord éventuels* : des post-conditions (actions résultantes de la tâche, c'est-à-dire dynamique des objets du modèle : modification des valeurs des attributs, création ou suppression d'objets des objets), des évènements (événements pouvant être engendrés lors de l'exécution de la tâche).

Les objets du modèle caractérisent l'environnement de l'utilisateur, ce sont les objets qu'il manipule ou qui influencent le déroulement de l'activité (aléas extérieurs dont l'utilisateur n'a pas l'initiative, caractérisés par un nom et une source). Les divers types d'objets sont les suivants :

- Objets abstraits : caractéristiques relatives aux concepts manipulés par l'utilisateur ;
- Attributs abstraits : caractéristiques de l'objet abstrait ;
- Objets concrets : correspond à une instance d'un objet abstrait ;
- Groupes : regroupement d'objets concrets ;
- Attributs concrets : permet d'associer une valeur pour chacune des caractéristiques définies par les attributs abstraits de l'objet abstrait ;
- Utilisateurs : ensemble des utilisateurs qui interviennent dans l'activité décrite ;
- Événements : ensemble des événements qui peuvent être déclenchés ou provoqués par l'activité décrite.

Une tâche peut avoir plusieurs tâches-filles dont l'ordonnancement est décrit par : un événement déclencheur, une nécessité (optionnelle, obligatoire), une interruptibilité (oui, non), et des opérateurs (séquentiel, alternatif, parallèle, ordre entrelacé et élémentaire).

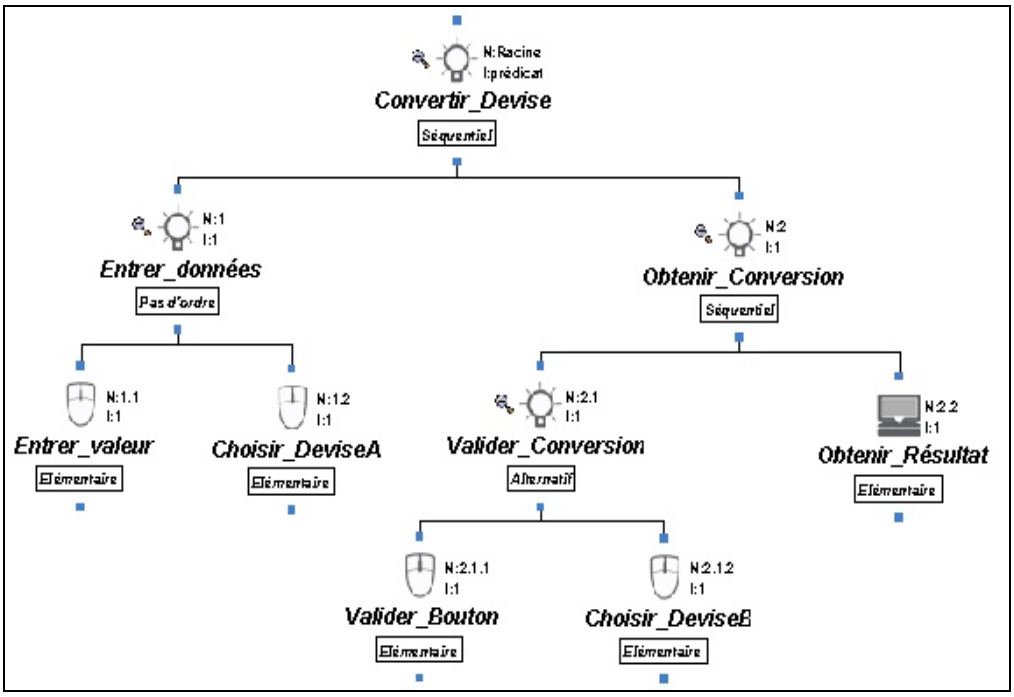

Figure 54 : Modèle de tâches K-MADe du Convertisseur de devise.

Pour illustrer nos propos, la Figure 54 présente le modèle de tâches du « Convertisseur de devises » de notre étude de cas énoncée dans la section 4 du chapitre 3. Nous limitons ici le travail à la conversion de devises sans intégrer la partie de traitement de texte. Sur la figure, « *Conversion_Devise* » modélise la tâche liée à l'application du convertisseur de devises. Elle décrit le fait que l'utilisateur saisit une valeur à convertir (tâche "*Entrer_valeur"* du modèle de tâches), choisit la devise à convertir (tâche "*Choisir_DeviseA"*). L'ordre d'exécution de ces deux tâches n'a pas d'importance. A l'issue de ces deux tâches, l'utilisateur a deux possibilités pour obtenir le résultat de la conversion. Il peut l'obtenir soit à partir de la validation par le bouton (tâche "*Valider_Bouton*"), soit en choisissant la devise de conversion (tâche "*Choisir_DeviseB*"). Une fois le choix de la validation effectué, la valeur convertie est affichée (tâche "*Obtenir_Résultat*").

L'outil K-MADe implémente l'ensemble des caractéristiques du modèle K-MAD. Il permet d'éditer, modifier et interroger les modèles de tâches utilisateur. C'est un environnement issu de la recherche en matière d'ergonomie et d'IHM, destiné à faciliter la mise en œuvre d'une perspective analytique centrée-utilisateur, avec une approche basée sur les modèles, pour la conception et l'évaluation ergonomique des logiciels interactifs. L'outil se compose principalement :

- d'un éditeur graphique de modèles de tâches K-MAD (utilisation de techniques de manipulation directe pour la construction, la manipulation et la suppression des tâches) ;
- d'un éditeur de caractéristiques de tâches ;
- d'un éditeur d'objets abstraits, d'utilisateurs, d'événements et d'objets concrets (possibilité d'ajouter, modifier et supprimer des objets ; la

modification et la suppression d'un objet entraînent obligatoirement une modification sur les objets directement liés) ;
- d'outil d'édition des expressions pour les pré-conditions, les post-conditions et les itérations ;
- d'un simulateur pour animer des modèles de tâches K-MAD ;
- de différents outils d'analyse de modèles de tâches (statistiques, cohérence sur modèle, recherche approfondie, …) ;
- et d'outil d'impression des arbres de tâches et des caractéristiques des tâches.

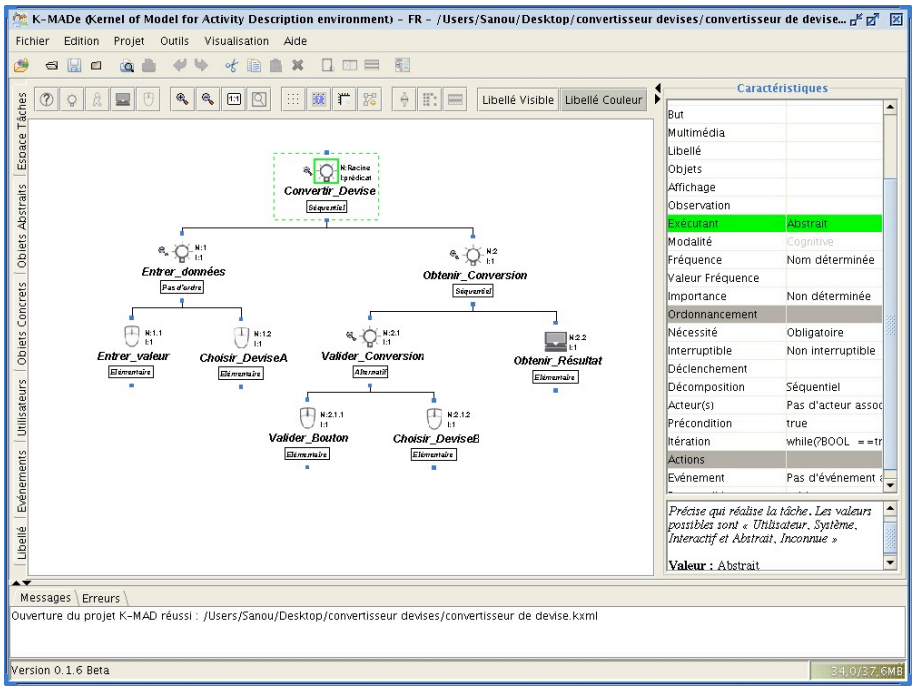

Figure 55 : Environnement K-MADe

Dans l'environnement K-MADe, l'ordre des sous-tâches d'une même tâche-mère (priorité des tâches ou contrainte de précédence) est défini par l'ordre gauche-droit sur l'espace de tâches.

4.1.3 Modèle de tâches et validation des IHM

Suivant l'origine de la naissance des modèles, et au vu des évolutions réalisées ((Palanque, 1997) ; (Jambon, 2002) ; (Baron, 2003)), les modèles de tâches constituent une bonne base de départ pour la réalisation d'une IHM. D'après (Balbo, 1994) le modèle de tâches issu du domaine de l'IHM est utile pour deux raisons principales. D'une part, il peut servir d'outil dans la conception et d'autre part, il s'adresse à l'utilisateur pour l'aider dans son travail via l'utilisation d'une aide sous forme de documentation, définie dans la phase de conception. Nous ajouterons à ces deux points que le modèle de tâches est un bon candidat pour servir de support à la validation du système (Sanou, 2007). De par sa nature, il permet de vérifier les propriétés des tâches. Une hypothèse de travail est que la modélisation de la tâche, suivant un modèle (pouvant être qualifié de processus informel), aboutit à des représentations conceptuelles correctes, claires et cohérentes.

Les modèles de tâches permettent de détecter les erreurs de conception les plus grossières ou d'effectuer des analyses comparatives et donc de guider des choix de conception

avant même la mise en œuvre, réputée coûteuse, du prototype. Leur utilisation pour valider les systèmes interactifs ne date pas d'aujourd'hui. En effet, les méthodes comme GOMS (Goal, Operator, Method, Selection) et ses dérivés, telle la théorie de la complexité cognitive, offrent une évaluation prédictive à partir du modèle de tâches selon l'objectif recherché par le concepteur.

Cependant, utiliser les modèles de tâches pour aider à concevoir les applications est paradoxalement resté relativement peu développé. Quelques approches ont tenté de générer le contrôle des applications à partir des modèles de tâches, comme TERESA par exemple. Malheureusement, la méthode utilisée ne garantit pas le respect des propriétés lors des transformations. La majorité des travaux développés jusqu'ici s'appuie sur la vérification de propriétés sur des modèles de l'application. Dans (Paternò and Santoro, 2002) et (Wilson and Jonhson, 1996) la génération de l'interface a été prouvée, et cela de façon théorique avec la mise en pratique d'un test en laboratoire, mais la conformité de l'application aux modèles censés la représenter ne peut être prouvée. Comment peut-on s'assurer de la présence de toutes les tâches et de leur ordonnancement ? Comment s'assurer que l'interface utilisateur respecte le modèle de tâches ?

Pourtant, si l'on pouvait établir un lien entre l'application réalisée et le modèle de tâches, il serait possible de vérifier des propriétés importantes comme la conformité de l'application aux tâches prescrites, ou la complétude de l'application par rapport aux tâches. C'est à ce problème que notre travail souhaite apporter un début de réponse. La modélisation de tâches pour la conception de lIHM, faite en amont, voit ses bénéfices estompés au fur et à mesure de l'avancement du développement. A la fin de la conception, il est donc difficile de vérifier que l'application réalisée est effectivement utilisée de la façon dont on l'avait prévu.

Nous sommes partis de l'idée que la vérification de la conformité d'une application à son modèle de tâches prescrites était une activité qui, si elle ne pouvait s'appuyer sur une preuve formelle, pouvait aisément relever du domaine du test (Sanou et al., 2008b). Est-il envisageable de réaliser des tests dont les résultats pourront êtres confrontés au modèle de tâches ? Si oui, ces tests peuvent-ils être enregistrés, pour éventuellement servir dans une approche de non-régression ? En nous appuyant sur les techniques de la PsE, nous avons défini les bases d'une méthode constructive de test basée sur l'enregistrement des interactions de l'utilisateur. La solution que nous présentons s'appuie donc sur la définition d'un lien direct entre l'interface et le modèle de tâche, qui permet, par la définition de scénarii directement sur l'application finale, de vérifier que cette exécution reste conforme aux modèles.

4.2 Liens entre modèle de tâches et interface utilisateur

Notre objectif est de définir une liaison entre le modèle de tâches ayant servi à la réalisation de l'IHM et l'IHM elle-même. Cette mise en relation permettra de vérifier la conformité de l'IHM au modèle. Le modèle de tâches est donc constitué de tâches. Une tâche est définie par Normand comme « étant un but que l'utilisateur vise à atteindre assortie d'une procédure (ou plan) qui décrit les moyens pour atteindre ce but ». Un but est un état du système que l'utilisateur souhaite obtenir. De façon intuitive, il apparaît que les modèles de tâches permettent d'exprimer une partie de l'interaction. Par exemple une tâche de « saisie de valeur » est nécessairement une interaction sur l'IHM. Dans cette sous-section, nous établissons formellement des liens entre modèles de tâches et interfaces. Ces liens existent entre les tâches élémentaires du modèle de tâches et les actions élémentaires de l'interface. Cela, parce que lors de la simulation, le modèle de tâches est analysé suivant l'exécution ou les activations successives des tâches élémentaires de ce modèle. Que signifie donc une tâche

élémentaire et action élémentaire ? Nous les déterminons dans le contexte de notre prise en compte du modèle de tâches dans la conception des IHM à la phase terminale.

4.2.1 Tâches élémentaires du modèle de tâches

Une tâche est une activité (considérée comme) nécessaire, ou utilisée pour atteindre un objectif en utilisant un moyen donné. Une tâche est en général décomposable en sous-tâches jusqu'au niveau des actions.

Une tâche élémentaire correspond à une action (système ou utilisateur) élémentaire non décomposable ou non décomposée du modèle de tâches. C'est une tâche n'impliquant pas de résolution de problème ou de structure de contrôle (alternatives, répétition, etc.). Elle est en générale focalisée sur un ou des objets. La tâche élémentaire est atteinte par les actions d'un but terminal (désignant un état « terminal » du système).

Les tâches élémentaires sont des tâches feuilles du modèle de tâches. Elle ne peut être de type « Abstrait », et est obligatoirement de type « Utilisateur », « Interactif », ou « Système ».

En suivant la décomposition hiérarchique des tâches, seules les tâches élémentaires apparaissent susceptibles de concerner une action sur l'IHM, comme nous l'avons souligné précédemment. Un lien avec la planification hiérarchique adopté en Intelligence Artificielle et qui constitue l'un des principes du fonctionnement du modèle du processeur humain est une argumentation supplémentaire confirmant le positionnement de la tâche élémentaire.

Du modèle de tâche de notre étude de cas, on peut déduire les tâches élémentaires suivantes : *Entrer_valeur*, *Choisir_DeviseA*, *Valider_bouton*, *Choisir_DeviseB* et *Obtenir_Résultat*.

4.2.2 Actions élémentaires de l'interface utilisateur

Une interface utilisateur est une couche applicative permettant à l'utilisateur d'interagir avec le noyau de l'application. Nous avons dit précédemment que c'est à travers l'interface que l'utilisateur exerce les différentes actions lui permettant d'exécuter une tâche.

L'IHM, ensemble de composants de présentation réagissant aux actions de l'utilisateur, peut prendre en compte plusieurs techniques d'interactions comme nous l'avons présenté plus haut (cf. § 4.1.1). Les actions élémentaires sont définies comme les interactions unitaires de l'utilisateur ou du système. Une interaction unitaire peut être soit du type d'interaction utilisé, soit du type système. Le type d'interaction que nous considérons le plus répandu et que nous utilisons est le type interaction à partir d'un dispositif d'entrée standard : soit le clavier, soit la souris. L'interaction unitaire du type système est en général la mise à jour de l'affichage suite à un changement de l'état interne du système. Le changement de l'état interne peut être aussi vu comme une action élémentaire. L'action élémentaire est une fonction de réalisation de l'action de l'utilisateur. Elle peut être le fruit d'une interaction élémentaire ou d'une suite d'interactions élémentaires dans la réalisation d'une tâche donnée.

On peut dire que les actions élémentaires sur l'interface sont celles qui sont exercées par l'utilisateur lors de ses actions, et celles qui sont programmées en retour pour refléter l'état de l'application.

Concrètement, en se conformant au modèle de tâches de notre exemple, nous obtenons l'interface utilisateur de la Figure 56. Les actions élémentaires définissables sur cette interface se résument à la saisie d'une valeur dans la *zone 1*, au clic sur l'un des deux "comboBox"

pour le choix de la devise (*zone 2* définie la devise à convertir et *zone 3* la devise de conversion à obtenir) et au clic sur le bouton « *Convertir* ». L'affichage du résultat (dans la zone 4) est une action du système. Toute autre action élémentaire issue d'une interaction quelconque de l'utilisateur n'a pas d'effet sur l'état du système et par conséquent ne sera pas considérée.

Figure 56 : Interface de Conversion de devises.

4.2.3 Liens entre tâches élémentaires du modèle de tâches et actions sur l'interface utilisateur

Partant de l'idée intuitive d'une équivalence potentielle entre les feuilles d'un arbre de tâches raffiné jusqu'aux actions du système et certains éléments de l'interface, nous avons cherché à établir plus précisément le lien existant entre eux.

Les tâches de type « Abstrait » sont des éléments structurants du modèle de tâches. Elles sont essentielles à la définition de la dynamique du modèle (et donc à la réalisation de la simulation) car elles portent la plupart des opérateurs et attributs nécessaires pour définir cette dynamique (opérateur de décomposition, multiplicité, interruption, etc.). Malgré cela, elles ne rentrent pas en ligne de compte dans l'écriture de scénarii. En effet, ces derniers ne sont constitués que d'enchaînement de tâches « feuilles » du modèle, qui ne peuvent être de type « Abstrait », comme nous l'avons dit plus haut.

Les tâches de type « Utilisateur » et de type « Système » ne donnent pas lieu à une interaction de l'utilisateur sur l'interface. Les premières ne sont liées au système que par le fait qu'elles requièrent, de la part du système, la mise à disposition des informations nécessaires à leur accomplissement. Ce sont en général des tâches de réflexion. Un lien entre ce type de tâches et l'interface ne pourrait se faire qu'à travers des états de l'interface. Les tâches de type « Système » correspondent à une action de l'application, qui engendre normalement (principe d'observabilité) un retour d'information vers l'utilisateur. Ce retour d'information, qui constitue une transition entre deux états de l'interface, est un bon candidat pour établir un lien avec la tâche.

Les tâches de type « Interactif » sont les candidats naturels au lien avec l'interface. Elles correspondent à une interaction entre l'utilisateur et le système, donc obligatoirement à au moins une action du premier sur le deuxième.

D'une manière pratique, cette correspondance entre tâches élémentaires et transitions de l'interface s'établit par le développeur de l'application. Le développeur définit sous la forme d'un tableau, les correspondances possibles entre les tâches élémentaires du modèle de tâches et les traitements de son interface. Le tableau de correspondance lui permet, à chaque implémentation d'un abonnement ou d'un traitement à partir d'un composant de l'interface, de lier ce dernier à la tâche prescrite du modèle de tâches. L'implémentation technique de cette correspondance est l'objet de la sous-section 4.3.2.

Notons cependant que la relation qui s'établit entre le modèle de tâches et l'application n'est en aucun cas une bijection. Rien n'empêche de prévoir la réalisation de la tâche par deux dispositifs d'interaction. Ce serait par exemple le cas si l'on décidait de doubler le bouton « *Convertir* » par un item de menu. De même, rien n'interdit d'utiliser la même interaction pour des rôles différents en fonction du contexte. La conséquence de ce point est que le lien est fonction de ce contexte. Nous discuterons ce point dans la suite.

La correspondance entre les tâches élémentaires et les actions de l'interface de l'étude de cas est établie de façon simple. La tâche « *Entrer_valeur* » correspond sur l'interface à la saisie d'une valeur numérique dans le champ de texte (*zone 1*) prévu à cet effet. Les tâches « *Choisir_DeviseA* » et « *Choisir_DeviseB* » correspondent à l'appui sur l'un des comboBox (représentés sur l'interface par la *zone 2* et *zone 3*) pour définir les différentes devises de la conversion. Enfin, la tâche système est représentée par l'affichage de la valeur convertie dans *zone 4*. La table de correspondance est ainsi établie (Tableau 4).

Tableau 4 : Correspondance entre actions élémentaires de l'interface et tâches du modèle de tâches.

Actions élémentaires de l'interface utilisateur	***Tâches élémentaires du modèle de tâches***
Une saisie dans la zone 1	« *Entrer_valeur* »
Un clic sur le combo de la zone 2	« *Choisir_DeviseA* »
Un clic sur le combo de la zone 3	« *Choisir_DeviseB* »
Un clic sur le bouton « Convertir »	« *Valider_Bouton* »
Affichage du résultat dans la zone 4	« *Obtenir_Résultat* »

Le tableau de correspondance dispose de deux autres colonnes dans lesquelles sont associés à chaque tâche, l'objet manipulé dans le modèle, et le numéro d'identification (idTask). Ces éléments n'interviennent pas directement dans la programmation du lien entre l'action élémentaire de l'interface et la tâche élémentaire. Ils sont associés aux tâches à la phase suivante. Nous y reviendrons dans la section suivante lors de la génération de scénarii.

À l'état actuel du système, la table de correspondance est un fichier texte (utilisé par le module d'enregistrement de scénario) que le développeur doit mettre en place de la manière suivante :

- le fichier doit se nommer « *TableCorrespondance.txt* » et doit être placé dans le même dossier que l'application ;
- chaque correspondance est définie en une seule ligne dans l'ordre suivant : le nom de la tâche, son identifiant, suivi de l'objet associé, de la pré-condition, de la post condition. Il est obligatoire de renseigner les deux premiers champs, les autres ne sont pas obligatoires.

Après avoir explicité cette mise en correspondance par l'exemple de l'étude de cas, la question est de savoir à quel niveau du système cet enregistrement des actions utilisateur est effectué. Le choix du niveau est important en fonction du but recherché. Plus l'enregistrement s'effectue à un niveau élémentaire dans le dialogue entre l'utilisateur et l'application, et plus il sera dépendant des changements intervenant dans l'interface utilisateur de l'application. Comme nous l'avons montré, les actions élémentaires à prendre en compte sont les fonctions de réalisation des actions de l'utilisateur que nous devons considérer lors de l'enregistrement des interactions. Ce ne sont pas les actions de l'utilisateur sur l'interface applicative qui sont enregistrées en réalité, mais ce sont les différentes opérations ou fonctions liées aux objets de

l'interface graphique. Il s'agit là d'un enregistrement au niveau syntaxique. Cela signifie que les actions élémentaires de l'interface utilisateur sont situées au niveau syntaxique de l'application. À quel niveau se situent donc les tâches élémentaires du modèle de tâches ? Dans le modèle de tâches, c'est l'ordonnancement des tâches qui définit leur disponibilité ou leur activation. Si une tâche s'active après une autre, alors une logique de raisonnement est respectée et vu qu'il existe en général des conditions, le résultat est nécessairement significatif. Les tâches élémentaires sont donc au niveau sémantique dans le modèle de tâches. On utilise un lien sémantique vers le modèle de tâches alors que l'espionnage se fait au niveau syntaxique dans l'application.

Nous allons maintenant montrer comment on peut réellement tester une application à partir de son lien avec le modèle de tâches. C'est l'objet de la section ci-dessous.

4.3 Génération des scénarii à partir de l'interface

Un scénario peut être défini comme une utilisation particulière du système dans un contexte précis. Le modèle de tâches, lui au contraire est une description globale. Du point de vue de ce dernier, un scénario est en fait une succession de tâches élémentaires sans structuration autre que la séquence. Vérifier si le système est conforme au modèle de tâches consiste à s'assurer que la succession des tâches respecte les conditions posées par les opérateurs du modèle de tâches. Cette vérification peut se faire en utilisant un outil existant dans un outil comme K-MADe (ce serait aussi le cas avec CTTe) : l'outil de simulation. Si les scénarii issus de l'utilisation du système peuvent être exécutés avec succès par le simulateur de K-MADe, cela signifie qu'ils sont conformes au modèle de tâches.

4.3.1 Définition des liens

Notre démarche s'appuie sur l'approche présentée dans le chapitre 3. Nous allons enrichir cette boîte à outils, dans le respect des choix faits pour elle, à savoir en limitant le travail du programmeur. Ainsi, il s'agit ici de permettre au développeur d'établir directement et le plus simplement possible le lien entre son code (donc l'IHM réelle de l'application finale) et le modèle de tâches.

Au travers des différents composants de la boîte à outils, PbDToolkit, intégrant le principe de l'enregistrement et de rejeu, le développeur définit facilement les liaisons établies dans sa table de correspondance. Cet établissement se fait lors de la définition des abonnements, technique principale de traitement des événements en Swing.

L'implémentation de la technique de correspondance entre les tâches élémentaires du modèle de tâches et les actions élémentaires de l'interface utilisateur peut se faire par un double abonnement ou par la modification d'une méthode par un ajout d'attribut. En effet, il est possible de créer une nouvelle ligne d'abonnement après chaque abonnement de composant ayant une entrée dans la table de correspondance. Il est aussi possible de surcharger des méthodes existantes afin d'adjoindre la liaison avec les tâches correspondantes. Dans le chapitre précédent, nous avons eu à discuter du problème du double abonnement. Il est lourd à mettre en place. Nous avons donc opté pour la solution alternative. En effet, pour faciliter cette définition, on redéfinit les composants graphiques de manière à leur ajouter au niveau des interfaces d'écoute, un autre attribut de type String. Cet attribut de type String correspond au nom réel de la tâche élémentaire correspondante dans le modèle de tâches.

Dans la sous-section 4.3.2 nous avons montré que seules les tâches d'interactions et les tâches systèmes sont sujettes à être liées aux actions élémentaires de l'interface. Pour chacune de ses tâches, comment se met en pratique cette implémentation ?

4.3.2 Implémentation des liens pour les tâches interactives

Le mécanisme de base permettant de programmer une boîte à outils événementielle telle que Swing est celui des événements. Afin de programmer la réaction du système aux actions de l'utilisateur, le programmeur doit "abonner" un traitement à chaque objet interactif susceptible d'être actionné par l'utilisateur. Ce mécanisme d'abonnement est le candidat idéal pour l'établissement du lien entre l'application et le modèle de tâches. Cependant, il ne peut concerner que la catégorie de tâches d'interactions du modèle de tâches, qui correspondent effectivement à une action de l'utilisateur sur le système.

Dans notre exemple sur le convertisseur de devises, les tâches concernées sont les tâches de déclenchement de la conversion (*Valider_Button* et *Choisir_DeviseB*), la tâche de saisie de la valeur (*Entrer_valeur*), la tâche de sélection de la devise à convertir (*Choisir_DeviseA*).

Les deux premières tâches sont équivalentes et correspondent, d'un point de vue purement Swing, au traitement de l'événement ActionEvent, base du comportement d'un JButton. Créer un lien entre le code et le modèle de tâches consiste dans le cas présent à associer le déclenchement de l'événement ActionEvent sur le bon bouton avec l'identification de la tâche élémentaire correspondante. Pour cela, nous avons spécialisé les composants d'interface Swing utilisés en surchargeant leur(s) méthode(s) d'abonnement. À chaque composant, nous avons ainsi rajouté la possibilité d'adjoindre une chaîne de caractères, désignant le nom de la tâche. Cela est sans conséquence du moment où nous ne modifions pas la structure du composant.

Dans le cas du JButton, il s'agit d'ajouter un paramètre (chaîne de caractères) permettant de représenter l'identifiant de la tâche correspondante dans le modèle de tâches. La classe PbDButton est donc de nouveau étendue au niveau de sa méthode d'abonnement comme suit :

```
public class PbDButton extends JButton {
...
public void addActionListener(ActionListener action,
        String taskName) {...}
}
```

Code 12 : Classe PbDButton étendue avec la prise en compte du nom de la tâche

Du point de vue du développeur qui code son application, le seul travail qu'il a à faire est de préciser lors de l'association du listener la tâche élémentaire, ce qui donne l'appel pour le bouton « Convertir » correspondant à la tâche élémentaire « *Valider_Button* » du modèle de tâches :

```
boutonC=new PbDButton ("Convertir");
boutonC.addActionListener(monActionListener, "Valider_Button");
```

Code 13 : Création et abonnement de PbDButton

Cette solution n'interdit pas à l'utilisateur d'utiliser les autres méthodes d'abonnement telles que les classes anonymes, ou l'auto-abonnement, puisque le Listener en lui-même n'est pas concerné par la modification.

Notons que la solution que nous proposons ici s'apparente à celle proposée par (Tarby, 2006), qui utilise la technique de la programmation par aspects. Cependant, cette dernière technique nécessite une étape supplémentaire de programmation, susceptible de générer des erreurs par omission : l'insertion de traces par la programmation par aspects. Il s'agit d'insertion de lignes de codes supplémentaires pour désigner les traces dans le système. L'effort de programmation peut s'avérer considérable et la maîtrise d'une autre technique de programmation est nécessaire : les aspects.

Dans le cas de la tâche de saisie, l'étude du composant d'interface classique JTextField montre qu'il existe plusieurs façons de gérer les événements. Le fonctionnement de base consiste à gérer la saisie par l'intermédiaire de l'événement ActionEvent, déclenché par l'appui sur la touche return du clavier. Cependant, un programmeur avisé en Swing évitera le plus souvent la programmation de ce seul événement, car dans ce cas, l'application ne sera pas capable de fournir un feedback approprié à la saisie tant que l'utilisateur n'aura pas frappé cette touche.

Dans notre exemple, il serait souhaitable de programmer un feedback proactif sur le bouton et le comboBox de la zone 2 (de l'interface), qui interdit à l'utilisateur de déclencher une conversion (désactivation du bouton, et du comboBox) lorsque le champ de saisie n'est pas convertible, et qui active ces mêmes composants lorsqu'il l'est. Dans ce cas, l'utilisateur devra programmer des événements différents, qui sont en fait accessibles par la hiérarchie d'héritage dans le JTextField. Une solution plus complète consistera à utiliser l'architecture MVC des composants Swing pour programmer la réaction du composant au niveau du Model, ce qui garantit une réaction à tout changement de la donnée contenue dans le JTextField. Ces différentes considérations expliquent pourquoi, pour permettre au programmeur de demeurer libre de ses choix, il nous a fallu surcharger les listeners disponibles au niveau des composants, y compris ceux qui sont disponibles à travers l'héritage.

4.3.3 Implémentation des liens pour les tâches systèmes

Les tâches de la catégorie Système sont d'une toute autre nature que les tâches d'interactions. Comme nous l'avons vu précédemment, elles se caractérisent principalement par un retour d'information (feedback) vers l'utilisateur. Le mécanisme utilisé pour les tâches d'interactions n'est donc pas utilisable. L'idée ici est de fournir une solution qui soit la plus légère d'utilisation pour le programmeur, afin de réduire les risques d'oublis ou d'erreurs. Plusieurs solutions différentes, au nombre de trois, sont envisageables.

Une première solution consisterait à considérer un composant comme responsable du feedback d'une tâche système au maximum. Il suffit alors de surcharger tous les composants avec une fonction permettant d'enregistrer, pour un composant d'interface donné, le nom de la tâche système associée. L'inconvénient de cette méthode est qu'elle restreint un composant à un seul usage, ce qui peut sembler contraignant à l'usage.

Une autre solution consisterait à surcharger tous les modifieurs des composants de la boîte à outils, afin de leur adjoindre la chaîne de caractères correspondant à la tâche invoquée. Le mécanisme est dans ce cas analogue à celui utilisé pour les tâches d'interactions. Le problème majeur de cette solution est la lourdeur du développement de la boîte à outils. Alors que les événements, et donc les listeners différents, sont en nombre relativement limités, les

fonctions d'accès aux composants, eu égard à la relation d'héritage, sont particulièrement nombreuses.

La solution que nous avons privilégiée, toujours dans le but de minimiser l'intervention du développeur, mais également de proposer une solution acceptable du point de vue de l'implémentation, consiste à fournir une primitive statique permettant au programmeur de spécifier explicitement dans son code qu'il active une fonction système. Dans notre exemple, cela donne :

```
TaskTestToolkit.execSystemTask(
                    "Obtenir_Résultat");
```

Code 14 : Fonction système pour l'activation d'une tâche système

La responsabilité de la définition du lien reste au programmeur, qui peut très bien oublier de la déclarer. Mais la simplicité et l'universalité de la solution compensent largement cette limite.

4.3.4 Génération des scénarii et test de l'interface

Nous avons déjà écrit que le scénario est en fait un enchaînement de tâches élémentaires. Dans K-MADe, la manipulation de scénarii distingue deux aspects : l'enregistrement de scénarii et le rejeu de scénarii. L'enregistrement consiste à simuler un arbre de tâches en choisissant, tout au long de ces différents états, les actions à appliquer d'une part et les valeurs particulières d'autre part. Le rejeu d'un scénario consiste à parcourir un chemin particulier donné par le scénario dans le modèle de tâches K-MAD.

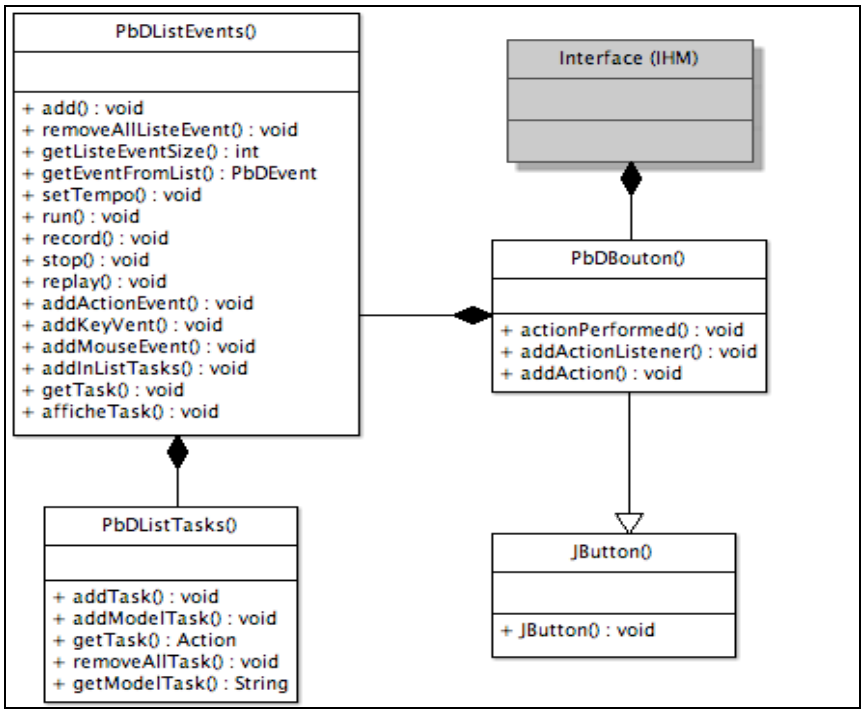

Figure 57 : Extrait du diagramme UML de PbDToolKit

En fonction du but recherché par le test, plusieurs aspects peuvent être vérifiés : atteignabilité de la tâche dans le modèle (est-ce que toutes les tâches disponibles à un instant donné lors de la simulation sont accessibles ?), atteignabilité d'un état particulier dans les

objets du modèle concret (à partir d'une exécution, est-il possible d'obtenir une valeur donnée fixée en fonction des données d'entrées, et cela à travers le modèle de tâches ?), finalisation complète de la suite de tâches dans l'arbre de tâches (toutes les tâches sont bien exécutables suivant l'ordre des interactions de l'utilisateur, et cela jusqu'à la fin); etc.

Pour générer un scénario à partir de l'application, les noms des tâches insérées par le développeur sont accumulés dans une liste au fur et à mesure de l'exécution de l'IHM. À chaque interaction significative de l'utilisateur (c'est-à-dire instrumentée par le concepteur), le système engendre la tâche élémentaire correspondante que l'on insère dans la liste. L'ordre d'insertion détermine l'ordre d'ordonnancement des tâches.

La mise en place de la liste des noms des tâches est liée à l'existence d'une classe rajoutée dans *PbDToolkit* ainsi que de certaines méthodes dans la classe *PbDListEvent()* (Figure 57). La nouvelle classe, *PbDListTask()*, fournit l'ensemble des opérations nécessaires à l'enrichissement d'une liste et à l'écriture de son contenu dans un fichier XML suivant une DTD (Document Type Definition) en se servant du fichier de la table de correspondance. Dans notre cas, la DTD est celle de K-MADe.

Pendant la durée d'une exécution, l'utilisateur réalisera un certain nombre de tâches élémentaires pour atteindre son but. Le résultat de l'espionnage de cette exécution construit donc automatiquement un scénario. Cette phase est tout à fait analogue à la phase d'enregistrement d'un scénario dans l'environnement K-MADe. L'utilisateur peut enregistrer son scénario. Pour cela, dans le menu « *Fichier* » de l'interface du contrôleur de *PbDToolkit*, l'item « *Enregistrer* » propose deux sous actions : « *Evénements* » et « *Tâches* ». Le premier, déjà décrit dans la dernière section du chapitre précédent permet d'enregistrer les actions utilisateur dans le cadre d'un rejeu (objectif PsE). L'item « *Tâches* » donne la possibilité à l'utilisateur d'enregistrer un scénario après une exécution de l'application.

```
<!ELEMENT scenario (execute | pass | interrupt | resume | cancel)* >
<!ATTLIST scenario name CDATA #REQUIRED >
 <!ATTLIST scenario date CDATA #IMPLIED >
<!ATTLIST scenario comment CDATA #IMPLIED >
<!ATTLIST scenario idTask CDATA #REQUIRED >
<!ATTLIST scenario nameTask CDATA #REQUIRED >
<!ELEMENT execute(userExecutingConstraint?, userPreValue*,
    userPreConcreteObject*, userPostValue*, userPostConcreteObject*,
                          userIterValue*, userIterConcreteObject*)>
<!ATTLIST execute idTask CDATA #REQUIRED >
 <!ATTLIST execute nameTask CDATA #REQUIRED >
<!ELEMENT userExecutingConstraint EMPTY >
<!ATTLIST userExecutingConstraint idUser CDATA #REQUIRED >
<!ATTLIST userExecutingConstraint nameUser CDATA #REQUIRED >
```

Figure 58 : Extrait de la DTD de K-MADe

Le fichier est enregistré suivant une traduction en XML selon la DTD défini par K-MADe (Figure 58). Ce fichier XML (voir exemple sur la Figure 59) est ensuite rejoué dans l'espace de simulation de l'environnement K-MADe. Les valeurs des contraintes n'étant pas directement liées aux objets lors de l'enregistrement du scénario seront celles du contexte du modèle de tâches à sa dernière simulation dans l'environnement. Le déroulement du rejeu se fait suivant l'animation du modèle de tâches et les erreurs dans le scénario sont détectées ou présentées de façon simple dans l'environnement : la simulation se termine, ou la tâche suivante n'est pas atteignable. L'exécution est une simulation normale sur le modèle de tâches.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE scenario SYSTEM "file:KMADEScenrio.dtd">
<scenario comment="Scenario Test" date="Under Development"
idTask="#4" name="Under Development" nameTask="Nom de la tâche">
<execute idTask="#13" nameTask="Saisie_valeur"/>
<execute idTask="#16" nameTask="Saisie_valeur"/>
<execute idTask="#28" nameTask="Choisir_DeviseA"/>
<execute idTask="#22" nameTask="Choisir_DeviseB "/>
<execute idTask="#10" nameTask="Obtenir_Résultat"/>
</scenario>
```

Figure 59 : Exemple de fichier XML après une exécution

Notons qu'il serait très facile d'exploiter, en plus de la notion de tâches, les vérifications autour des objets telle qu'elle est prévue dans K-MAD (pré et post-conditions). Il suffit, en même temps que l'on transmet les noms des tâches, de transmettre la valeur des objets concernés par la tâche. Ceci permettra à l'outil K-MADe d'utiliser ces valeurs pour ses évaluations de pré et post-conditions. Ceci est une perspective probable du travail car la DTD actuelle du format d'échange de K-MADe n'inclut pas les valeurs d'objets, ce qui interdit leur utilisation dans K-MADe.

5 Conclusion

La PsE, par l'utilisation de sa technique d'enregistrement et de rejeu, s'avère extrêmement prometteuse pour la vérification et la validation des IHM. Les approches intégrant la PsE sont de plus en plus nombreuses comme en témoigne l'étude sur les outils existants ainsi que les démarches de réalisation des tests. L'efficacité des outils de cette approche confère une possibilité d'investissement dans l'amélioration des travaux.

La solution présentée dans ce chapitre, bien que très simple dans son principe, permet de relier une application concrète avec un modèle de tâches censé la représenter. La vérification de la conformité de l'application au modèle de tâches permet de montrer de façon simple des erreurs de conception. Par exemple, la réalisation de la tâche de conversion suppose la saisie d'une valeur à convertir valide. Cette propriété est exprimable dans le modèle K-MAD par l'intermédiaire d'une pré-condition liée aux tâches « *Valider_Bouton* » et « *Choisir_DeviseB* », qui engendre une interdiction de celles-ci tant que l'on n'a pas commencé la tâche de saisie. Si l'application n'implémente pas cette vérification, il sera possible de créer un scénario qui sera non conforme au modèle.

Le lien en fonction du contexte n'est pas un problème majeur. Il engendre simplement une multiplication des scénarii à chaque choix dans le sens application vers modèle de tâches, alors qu'il n'a pas d'incidence dans l'autre sens. Néanmoins, on peut reprocher à cette méthode d'être plus destructive que constructive. En effet, les résultats les plus faciles à mettre en évidence sont des non-conformités au modèle de tâches : dès qu'un scénario n'est pas conforme, il arrête le simulateur.

À l'inverse, il n'est possible de prouver la complétude de l'application par rapport aux tâches prescrites qu'en bâtissant un jeu de scénarii couvrant tous les chemins du modèle de tâches, ce qui est toujours très difficile en test. Une réponse à cette question peut-être apportée par l'utilisation des techniques préconisées par Memon, qui devraient permettre de générer des scénarii en nombre, pour ensuite les confronter au modèle de tâche.

De plus, le diagnostic d'erreur est très pauvre : en l'état actuel du simulateur de K-MADe, qui, rappelons-le, n'a pas été conçu pour cet usage, il est difficile de comprendre où se situe l'erreur. Mais le simulateur de K-MADe affiche des messages d'erreur de non-exécution d'une tâche. Une remontée manuelle est nécessaire à l'interface pour cibler la cause. Tout cela suppose un travail d'envergure sur K-MADe.

Conclusion générale

1 Bilan

Le travail que nous venons de présenter s'inscrit dans le cadre de développement d'outils pour incorporer les techniques de PsE dans le développement des applications interactives. Il vient enrichir et compléter une famille de systèmes de PsE très variés. Dans cet objectif, nous avons étudié ces systèmes. Nous avons proposé de les regrouper en quatre catégories principales : l'assistance, les outils de conception, l'apprentissage de la programmation et les outils de PsE. C'est au niveau des outils de PsE que nous apportons notre contribution principale.

Au vu des différents systèmes de PsE que nous avons étudiés, il est clair qu'ils présentent à la fois des caractéristiques communes et des différences certaines. Plusieurs méthodes très différentes sont utilisées pour créer des Programmes sur Exemple. La plus répandue consiste à créer un programme à la manière d'une macro, c'est-à-dire déclencher l'enregistrement des commandes, démontrer la tâche puis arrêter l'enregistrement : cette méthode est bien adaptée si l'utilisateur est déjà familier avec le concept de macro et s'il a une tâche précise à automatiser. Une seconde méthode consiste à enregistrer en permanence les commandes de l'utilisateur et à proposer de créer un programme lorsqu'une répétition est rencontrée. Cette approche est bien adaptée aux utilisateurs débutants dans la mesure où elle ne requiert aucune connaissance de la programmation et l'utilisateur n'a pas toujours conscience ou réalise parfois trop tard qu'il peut créer un programme pour automatiser la tâche qu'il est en train d'accomplir. Enfin une troisième méthode adopte un comportement intermédiaire dans la mesure où elle enregistre toutes les commandes de l'utilisateur et permet à celui-ci de créer une macro en sélectionnant les commandes à inclure dans la macro.

Chacune de ces méthodes propose donc ses propres avantages et à l'heure actuelle il n'est pas clair de savoir quelle est la méthode la plus adaptée pour la PsE. La réponse à cette question ne peut être affirmée que si une étude est menée auprès des utilisateurs. Un système général de construction de la PsE doit donc être capable de supporter ces trois méthodes puisqu'un même mécanisme d'inférence peut être utilisé pour ces trois approches. Un autre point concerne la définition des arguments d'une procédure et le retour d'une valeur. Il n'y a pas de commandes spécifiques à invoquer pour définir et passer les arguments d'une macro. Cette approche est par contre limitée puisque les arguments ne peuvent être que des objets sélectionnables.

Nous avons proposé un concept de développement d'application intégrant la PsE à partir de l'extension de la boîte à outils Swing : PbDToolkit. Il se base sur les principes de base de la PsE : l'enregistrement et le rejeu. Nous avons proposé une technique d'implémentation du système de PsE à partir de l'enregistrement/rejeu qui lui se base sur la

file des événements. PbDToolkit sous-entend la scrutation de la file des événements pour l'enregistrement des événements de bas niveau tandis qu'il lui est nécessaire d'être proche des composants pour pouvoir effectuer ses fonctions à un haut niveau. En effet, le concept basé sur l'extension de la boîte à outils, étend les composants de ce dernier de sorte à leur ajouter la possibilité de communiquer avec des composantes du système pour la notification d'événements reçus. Ainsi, le concept de PbDToolkit redéfinit l'ensemble des composants Swing ainsi que la majeure partie des écouteurs appropriés de ces widgets.

Nous avons implémenté un système simple et facile à utiliser. PbDToolkit peut fonctionner en mode externe de l'application. Il fonctionne aussi en interne avec une prise en main complète du système par le développeur, qui peut transmettre cette liberté de manipulation à l'utilisateur final de l'application réalisée. En externe, il intercepte tous les événements bas niveau au travers de la file d'événements système. Le choix de manipulation pour l'utilisateur se résume à enregistrer ses interactions, à les sauvegarder et à les rejouer. En interne, PbDToolkit offre la possibilité d'enregistrement à deux niveaux. Le premier niveau est le bas niveau qui est sur le même principe que l'utilisation en externe. Seulement là, le développeur dispose de plus de fonctions à exploiter et à mettre au profit de l'utilisateur final. Le deuxième niveau porte sur les événements de haut niveau. PbDToolkit récupère au niveau de chaque abonnement l'événement engendré. Il permet au développeur de s'abstraire du bas niveau et obtenir les résultats fonctionnels et le comportement des widgets sans passer par les interactions de très bas niveau.

Afin de maximiser les possibilités de PbDToolkit, nous avons utilisé le concept dans le cadre de la validation des IHM à partir du modèle de tâches. En effet, nous avons établi un lien entre l'interface utilisateur de l'application et le modèle de tâches prescrit à son développement. Le développeur de l'application définit une table de correspondance entre les deux composants. Le modèle de tâches est relié à l'interface utilisateur par l'intermédiaire de ses tâches élémentaires (ou actions) qui correspondent aux actions élémentaires de l'utilisateur sur l'interface. La table de correspondance contient pour chaque tâche donnée, l'identité, le nom de la tâche, l'objet utilisé, les pré et post-conditions éventuelles et l'action élémentaire de l'interface représentée.

À partir de ce lien, nous générons des scénarii à l'exécution de l'application. À chaque interaction de l'utilisateur, le système engendre le nom de la tâche correspondante. Cette tâche et son objet ainsi que ces conditions (si elles existent) sont placés dans une file pour constituer le scénario d'exécution. L'utilisateur peut stocker les informations à travers un fichier et créer ainsi un vrai scénario exécutable sur le modèle de tâche. Le modèle utilisé pour l'implémentation est K-MAD et le fichier de scénario est un fichier XML suivant la DTD de l'environnement graphique K-MADe.

Nous avons ainsi fourni la possibilité de vérifier la conformité d'une IHM à son modèle de tâches. Aussi, on peut déduire aussi l'usage effectif de l'application par rapport à l'usage prévu. Enfin, on peut déterminer l'évolution du comportement de l'utilisateur lors de l'exécution d'une application.

2 Perspectives

L'outil fourni est simple d'utilisation et riche en maniabilité. Néanmoins, un certain nombre de points nous semblent intéressants et correspondent à autant de perspectives de recherche :

1. L'amélioration du système en fonctionnement externe : intégrer la prise en compte des deux niveaux d'enregistrement même si l'application est lancée de façon indépendante. En effet, PbDToolkit ne peut enregistrer que les événements de bas niveau en mode externe. Il ne scrute que la file des événements système lors de cette exécution. Étendre ses capacités pour pouvoir écouter les composants de l'application à un haut niveau semble nécessaire. Aussi, la proposition de plus d'outils d'interaction avec l'enregistrement et le rejeu de ce mode ne pourrait qu'accroître la convivialité d'utilisation de l'outil.

2. En fonctionnement interne, un accroissement du nombre de fonctions disponibles au développeur donne une possibilité de couvrir mieux ses besoins et réduire encore plus ses efforts de programmation. La fonctionnalité la plus fondamentale à ce niveau est l'exploitation de l'historique pour détecter les possibles boucles d'interaction de l'utilisateur. En effet, la mise en place une fonction de scrutation de l'historique à la recherche de motif (à considérer comme une séquence d'événements) permettra de signaler toute tâche itérative de la part de l'utilisateur. Une automatisation simple pourrait être proposée alors que l'utilisateur ne l'avait pas prévue. Mais avant, il faudra pouvoir agir sur la valeur des objets afin de prendre en compte les entrées de données d'exécution. Cela devrait s'étendre aussi au niveau du test par la possibilité de renseigner la valeur d'un objet lors de la simulation.

3. La perspective principale se situe au niveau du test de validation d'interface. Une automatisation de l'établissement du lien entre les tâches élémentaires du modèle de tâches et les actions élémentaires de l'interface utilisateur réduirait considérablement les risques d'erreurs de mise en correspondance, surtout au niveau des objets des tâches et de leurs conditions. Cette automatisation passera par la mise en place d'une interface intermédiaire entre l'environnement K-MADe et l'application. Il s'agira d'une part de modifier la structure de représentation des fichiers de données dans K-MADe en permettant leur récupération dynamiquement. D'autre part, l'application implémentée devra etre réliée par l'intermédiaire d'une interface à ces fichiers pour la transmission des actions utilisateur. En suivant la même logique, pourquoi ne pas pouvoir suivre directement une simulation simultanée du modèle de tâches à l'exécution de l'IHM ? Pour chaque interaction entre le système et l'utilisateur, le simulateur de K-MADe présentera la tâche activée dans le modèle de tâches.

4. Une généralisation du concept à tous les types d'applications, toute application développée dans tout autre langage, constitue la dernière des perspectives.

Bibliographie

[1] Aho, A., Monica, L., Ravi, S., and Jeffrey, U. (2007) Compilateurs : principes, techniques et outils. Pearson Education.

[2] Aït-Ameur, Y., Baron, M., and Kamel, N. (2003) Utilisation de techniques formelles dans la modélisation d'Interfaces Homme-Machine. Une expérience comparative entre B et Promela/SPIN. 6th International Symposium on Programming and Systems ISPS 2003, Algérie, pp. 57-66.

[3] Balbo, S. (1994) Évaluation ergonomique des interfaces utilisateur : un pas vers l'automatisation. IMAG. Université Joseph Fourier, Grenoble, p. 206.

[4] Balbo, S., Draheim, D., Lutteroth, C., and Weber, G. (2006) Projet WAUTER (Website Automatic Usability Testing EnviRonment.

[5] Baron, M. (2003) Vers une approche sûre du développement des Interfaces Homme-Machine (Thèse). Université de Poitiers, Poitiers, p. 255.

[6] Baron, M., Lucquiaud, V., Autard, D., and Scapin, D. (2006) K-MADe : un environnement pour le noyau du modèle de description de l'activité. In: Robert, J.-M., and David, B. (eds) IHM'06. ACM Publishers, Montréal, Canada, pp. 287-288.

[7] Bass, L., and Coutaz, J. (1991) Developing Software for the User Interface. Addison-Wesley.

[8] Bass, L., Hardy, E., Hoyt, K., Little, R., and Seacord, R. (1988) The ARCH Model : SEEHEIM Revisited, The Serpent run time architecture and dialog model. Carnegie Melon University.

[9] Bass, L., Pellegrino, R., Reed, S., Sheppard, S., and Szezur, M. (1991) The Arch Model : Seeheim revisited. User Interface Developper's Workshop.

[10] Bastide, R. (2000) PetShop : a tool for the formal specification of CORBA systems. Conference on object Oriented Programming Systems Languages and Applications. ACM, Minneapolis, United States, p. 167.

[11] Bastien, C., and Scapin, D. (1993) Ergonomic Criteria for the Evaluation of Human-Computer Interfaces. INRIA.

[12] Baudel, T. (2002) Les principaux services d'une boîte à outils. Contribution au GDR 13 / GT ALF, Workshop IHM, Paris Sud, p. 17.

[13] Bauer, M.A. (1979) Programming by Examples. Articial Intelligence 12:1-21.

[14] Beaudouin-Lafon, M. (2004) Designing interaction, not interfaces. Advanced Visual Interfaces, Gallipoly, Italie, pp. 15-22.

[15] Berti, S., Correani, F., Mori, G., Paternò, F., and Santoro, C. (2004) TERESA: a transformation-based environment for designing and development multi-device interface. In: York, A.N. (ed) Conference on Human Factors in Computing Systems - CHI'04. ACM NY., Vienna, Austria, pp. 793-794.

[16] Blanch, R. (2005) Architecture logicielle et outils pour les interfaces hommes-machines graphiques avancées. Paris XI, Orsay, p. 174.

[17] Boehm, B.W. (1988) A spiral model of software developement and enhancement. Articial Intelligence 21:61-72.

[18] Boshernitsan, M., Graham, S.L., and Hearst, M.A. (2007) Aligning development tools with the way programmers think about code changes. In: ACM-CHI (ed) Conference on Human Factors in Computing Systems, CHI'07. ACM Digital Library, San José, California, USA, pp. 567-576.

[19] Bourguin, G., Lewandowski, A., and Tarby, J.-C. (2006) Vers des composants logiciels orientés tâches. IHM. ACM Press, Montréal, Québec, pp. 215-218.

[20] Brooks, P.A., and Memon, A.M. (2007) Automated GUI testing guided by usage profiles. In: ACM (ed) Twenty-second IEEE/ACM International conference on Automated Software Engineering. ACM Press, Atlanta, Georgia, USA, pp. 333-342.

[21] Brun, P., and Beaudouin-Lafon, M. (1995) A taxonomy and evaluation of formalisms for the specification of interactive systems. In: Kirby, M.A.R., Dix, A.J., and Finlay, J.E. (eds) Conference of the British Human-Computer Interaction Group (HCI'95). Cambridge University Press, University of Huddersfeild, UK, pp. 197-212.

[22] Bryce, R.C., and Memon, A.M. (2007) Test suite prioritization by interaction coverage. Workshop on Domain specific approaches to software test automation. ACM New York, Dubrovnik, Croatia.

[23] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., and Vanderdonckt, J. (2002) Platicity of User Interfaces: A Revised Reference Framework. Tamodia, Bucarest.

[24] Chen, J.-H., and Weld, D.S. (2008) Recorvering from errors during programming by demonstration. In: Library, A.D. (ed) Intelligent User Interfaces 2008. ACM Digital Library, Maspalomas, Gan Canaria, Spain, pp. 159-168.

[25] Coutaz, J. (1990) Interfaces Homme-Ordinateur, Conception et Réalisation. Dunod Informatique, Paris.

[26] Curtis, B., and Hefley, B. (1994) A Wimp No More: The Maturing of User Interface Engineering. Interactions:23-34.

[27] Cypher, A. (1991) Eager: Programming Repetitive Tasks by Example. Human Factors in Computing Systems (CHI'91). ACM/SIGCHI, New Orleans, Louisiana, pp. 33-39.

[28] Cypher, A. (1993a) Eager: Programming Repetitive Tasks by Demonstration. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 205-217.

[29] Cypher, A. (1993b) Introduction: Bringing Programming to End Users. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 1-11.

[30] Cypher, A. (1993c) (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts.

[31] Cypher, A., Kosbie, D.S., and Maulsby, D. (1993) Characterizing PBD Systems. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 467-484.

[32] D'Ausbourg, B. (1998) Using Model Checking for the Automatic Validation of User Interface Systems. In: Markopoulos, P., and Johnson, P. (eds) Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98). SpringerWienNewYork, Abingdon, UK, pp. 242-260.

[33] D'Ausbourg, B., and Cazin, J. (1999) Using TRIO Specifications to Generate Test Cases for an Interactive System. Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal, pp. 143-160.

[34] D'Ausbourg, B., Durrieu, G., and Roché, P. (1996) Deriving a Formal Model of an Interactive System from its UIL Description in order to Verify and Test its Behaviour. In: Bodart, F., and Vanderdonckt, J. (eds) Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96). Springer-Verlag, Namur, Belgium, pp. 105-122.

[35] Depaulis, F., Guittet, L., and Martin, C. (2003) Apprends ce que je fais. 15ème Conférence Francophone sur l'Interaction Homme-Machine. ACM Press, Caen, pp. 236-239.

[36] Faaborg, A., and Lieberman, H. (2006) A goal-oriented web browser. In: ACM-CHI (ed) Conference on Human Factors in Computing Systems, CHI'06. ACM Digital Library, Montréal, Québec, Canada, pp. 751-760.

[37] Girard, P. (1992) Environnement de Programmation pour Non-Programmeur et Paramétrage en Conception Assistée par Ordinateur : le système LIKE. LISI/ENSMA. Université de Poitiers, Poitiers, France, p. 195.

[38] Girard, P. (2000) Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur. LISI/ENSMA. Université de Poitiers, Poitiers, p. 92.

[39] Girard, P., and Pierra, G. (1995) Structures de contrôle générales en Programmation par Démonstration. In: Palanque, P. (ed) Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'95). Cépaduès, Toulouse, pp. 61-68.

[40] Girard, P., Potier, J.-C., and Pierra, G. (1996) EBP: Example-Based programming in parametrics. Grenoble.

[41] Glinert, E.P. (ed) (1990) Visual programming environments: paradigms and systems. IEEE Computer Society Press, Los Alamitos, California.

[42] Guibert, N. (2006) Validation d'une approche basée sur l'exemple pour l'initiation à la programmation. LISI / ENSMA. Université de Poitiers, Poitiers, p. 246.

[43] Guibert, N., Guittet, L., and Girard, P. (2005) Validation d'une approche "basée sur exemples" pour l'apprentissage de la programmation. 17e Conférence Francophone sur l'IHM. AFIHM, Toulouse, France.

[44] Halbert, D. (1984) Programming by Example. University of California, Berkeley, p. 121.

[45] Halbert, D. (1993) SmallStar: Programming by Demonstration in the Desktop Metaphor. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusuetts, pp. 102-123.

[46] Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. (2007) Programming by a sample: rapidly creating web applications with d.mix. In: Library, A.D. (ed) User Interface Software and Technology (UIST'07). ACM Digital Library, Newport, Rhode Island, USA, pp. 241-250.

[47] Hartson, H.R., and Hix, D. (1992) The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. International Journal of Man-Machine Studies 8(3):181-203.

[48] Jambon, F. (2002) From Formal Specifications to Secure Implementations. In: Kolski, C., and Vanderdonckt, J. (eds) Computer-Aided Design of User Interfaces (CADUI'2002). Kluwer Academics, Valenciennes, France, pp. 43-54.

[49] Jambon, F., Brun, P., and Aït-Ameur, Y. (2001) Spécifications des systèmes interactifs (chapitre 6). In: Kolski, C. (ed) Analyse et conception de l'I.H.M. / Interaction Homme-Machine pour les S.I. vol.1. Hermès Science, Paris, France, pp. 175-206.

[50] Jambon, F., Girard, P., and Aït-Ameur, Y. (2000) Interactive System Safety and Usability enforced by the Development Process: the FADEC User Interface Case Study. In: Hohnson, C., Palanque, P., and Paternò, F. (eds) Safety and Usability Concerns in Aeronautics (SUCA 2000) IFIP WG 13.5 Workshop within HCI-Aero'2000, Toulouse, France.

[51] Jérôme, V. (2004) Génération automatique de cas de test guidée par les propiétés de sûreté. Informatic. Université Joseph Fourrier, Grenoble, France.

[52] Kahn, K. (2001) How Any Program Can Be Created by Working with Examples. In: Lieberman, H. (ed) Your Wish is My Command, pp. 21-44.

[53] Kosbie, D.S., and Myers, B.A. (1994) Extending Programming by Demonstration With Hierarchical Event Histories. EWHCI'94, St. Petersburg, Russia, pp. 147-157.

[54] Kurlander, D., and Feiner, S. (1992) A History-Based Macro by Example System. ACM Symposium on User Interface Software and Technology (UIST'92). ACM/SIGCHI, Monterey, California, pp. 99-115.

[55] Lecolinet, E. (1999) Réification et réplication dans les interfaces graphiques : le toolkit Ubit. IHM'99, Montpellier, pp. 9-12.

[56] Lieberman, H. (1993a) Mondrian: a Teachable Editor. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 341-360.

[57] Lieberman, H. (1993b) Tinker: A Programming by Demonstration System for Beginning Programmers. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 49-66.

[58] Lieberman, H. (2001) Your Wish is my command. Morgan Kaufmann.

[59] Lieberman, H., Nardi, B.A., and Wright, D. (1998) Grammex: Defining Grammar by Example. In: ACM (ed) Human Factors in Computing Systems (CHI'98). ACM/SIGCHI, Los Angeles, California, pp. 11-12.

[60] Lieberman, H., Paternô, F., and Wulf, V. (2006) End-User Development. Springer, The Netherlands.

[61] Limbourg, Q., and Vanderdonckt, J. (2003) Comparing Task Models for User Interface Design (Chapter 6). In: Diaper, D., and Stanton, N. (eds) The Handbook of Task Analysis for Human-Computer Interaction. Lawrence Erlbaum Associates.

[62] Lucquiaud, V. (2005a) Proposition d'un noyau et d'une structure pour les modèles de tâches orientés utilisateurs. In: AFIHM (ed) 17th French-speacking conference on Human-computer interaction, Toulouse, France, pp. 83-90.

[63] Lucquiaud, V. (2005b) Sémantique et Outil pour la Modélisation des Tâches Utilisateur: N-MDA. Poitiers, p. 285.

[64] Lucquiaud, V., Scapin, D., and Jambon, F. (2002) Outils de modélisation des tâches utilisateurs : exigences du point de vue utilisation. IHM'02. AFIHM - ACM NY USA, Poitiers, France, pp. 243-246.

[65] MacDonald, A. (1982) Visual programming. Datamation 28:132-140.

[66] Macias, J.A., and Castells, P. (2007) Providing end-user facilities to simplify ontology-driven web application authoring. Interacing with Computers 19:563-585.

[67] Maulsby, D.L., Witten, I.H., and Kittlitz, K.A. (1989) Metamouse : Specifying Graphical Procedures by Example. SIGGRAPH'89, Boston, pp. 127-135.

[68] McDaniel, R.G., and Myers, B.A. (1999) Gamut: Creating Complete Applications Using Only Programming by Demonstration. CHI'99

[69] Memon, M.A., and Lou, S.M. (2003) Regression testing of GUIs. In: ACM (ed) Foundations of Software Engineering. ACM New York, Helsinki, Finland, pp. 118-127.

[70] Memon, M.A., Martha E. Pollack, and Lou, S.M. (1999) Using a goal-driven approach to generate test cases for GUIs. In: IEEE-CS (ed) International Conference on Software Engineering. IEEE Computer Society Press Los Alamitos, Ca, USA, Los Angeles, California, United States, pp. 257-266.

[71] Memon, M.A., Martha E. Pollack, and Lou, S.M. (2000) Automated test oracles for GUIs. ACM SIGSOFT Software Engineering Notes 25:30-39.

[72] Memon, M.A., Martha E. Pollack, and Lou, S.M. (2001) Hierarchical GUI test case generation using automated planning. IEEE Trans. Software Engeniering 27(2):144-155.

[73] Modugno, F., and Myers, B.A. (1993) Graphical Representation and Feedback in a PbD System. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 415-422.

[74] Myers, A.B., McDaniel, R.G., and Wolber, D. (2000) Intelligence in demonstrational interfaces. Communication of the ACM 43:82-89.

[75] Myers, B.A. (1986) Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. Human Factors in Computing Systems (CHI'86). ACM/SIGCHI, New-York, pp. 59-66.

[76] Myers, B.A. (1988) Creating User Interface by Demonstration. Academic Press.

[77] Myers, B.A. (1990) Taxonomies of Visual Programming and Program Visualization. Journal of Visual Languages and Computing 1:97-123.

[78] Myers, B.A. (1992) Report on the end-user programming working group. In: Myers, B.A. (ed) Languages for Developing User Interfaces. Jones & Bartlett, Boston, pp. 343-366.

[79] Myers, B.A. (1993a) PERIDOT: Creating User Interfaces by Demonstration. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 125-154.

[80] Myers, B.A. (1993b) Why are User Interface Difficult to Design & Implement. Carnegy Melon University, Pittsburg.

[81] Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A., and Doane, P. (1997) The Amulet Environment: New Models for Effective User Interface Software Development. IEEE Transactions on Software Engineering, 23:347-365.

[82] Norman, D.A. (1986) User Centered System Design. Lawrence Erlbaum Associates.

[83] Normand, N. (1992) Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateur à leur réalisation. Université Joseph Fourrier, Grenoble, France.

[84] Palanque, P. (1997) Spécifications formelles et systèmes interactifs : vers des systèmes fiables et utilisables. LIS/FROGIS. Université de Toulouse I, Toulouse, p. 187.

[85] Paternò, F., Mancini, C., and Meniconi, S. (1997) ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. IFIP TC13 human-computer interaction conference (INTERACT'97), Sydney, Australia, pp. 362-369.

[86] Paternò, F., Mori, G., and Galimberti, R. (2001) CTTE: An Environment for Analysis and Development of Task Models of Cooperative Applications. ACM CHI 2001. ACM Press, Seattle.

[87] Paternò, F., and Santoro, C. (2002) One model, many interfaces. In: Kolski, C., and Vanderdonckt, J. (eds) Computer-Aided Design of User Interfaces (CADUI'2002). Kluwer Academics, Valenciennes, Frances, pp. 143-154.

[88] Patry, G. (1999) Contribution à la conception du dialogue Homme Machine dans les applications graphiques interactives de conception technique : le système GIPSE. LISI/ENSMA. Université de Poitiers, Poitiers, p. 199.

[89] Patry, G., and Girard, P. (1999) GIPSE: a Model-Based System for CAD. In: Vanderdonkt, J., and Puerta, A. (eds) Third Conference on Computer-Aided Design of User Interfaces (CADUI'99). Kluwer Academics, Louvain-la-Neuve, Belgique, pp. 61-72.

[90] Piernot, P.P., and Yvon, M.P. (1993) The AIDE Project: An Application-Independent Demonstrational Environment. In: Cypher, A. (ed) Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Massachusetts, pp. 383-402.

[91] Pierra, G., Potier, J.-C., and Girard, P. (1996) The EBP system : Example Based Programming for Parametric Design. In: Teixeira, J., and Rix, J. (eds) Modelling and Graphics in Science and Technology. Springer-Verlag, pp. 124-140.

[92] Potier, J.-C. (1995) Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP. LISI/ENSMA. Université de Poitiers, p. 140.

[93] Prabaker, M., Bergman, L., and Castelli, V. (2006) An evaluation of using programmming by demonstration and guided walkthrough techniques for authoring and utilizing documentation. In: ACM-CHI (ed) Conference on Human Factors in Computing Systems, CHI'06. ACM Digital Library, Montréal, Québec, Canada, pp. 241-250.

[94] Renaud, P., Phillipe, J., and Seinturier, L. (2004) Programmation orientée aspect pour Java/J2EE. Eyrolles, Paris.

[95] Repenning, A., and Perrone, C. (2001) Programming by Analogous Examples. In: Lieberman, H. (ed) Your Wish is My Command, pp. 351-370.

[96] Richard, K.S., and Daniel, P.S. (1997) A method to automate user interface testing using variable finite state machines. FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing. IEEE Computer Society., Washington, DC, USA, p. 80.

[97] Sanou, L. (2007) Validation directe de la conformité d'une application interactive avec son modèle de tâches. In: AFIHM (ed) 19e Conférence Internationnale sur l'IHM. ACM Press, Paris, France, pp. 249-252.

[98] Sanou, L., Caffiau, S., Girard, P., and Guittet, L. (2008a) Example usage evaluation for the programming learning in the MELBA environment. In: IADIS-MCCSIS (ed) Interface and Human-Computer Interaction (IHCI). IADIS-MCCSIS, Amsterdam, The Netherland, pp. 35-42.

[99] Sanou, L., Girard, P., and Guittet, L. (2005) Introducing Programming by Demonstration techniques at the toolkit level: a case study. In: Associates, L.E. (ed) HCI International. CD-ROM by Lawrence E. A., Las Vegas, Nevada, USA.

[100] Sanou, L., Girard, P., and Guittet, L. (2006a) Comparaison de deux méthodes pour implémenter la programmation sur exemple. In: AFIHM (ed) IHM'06. ACM Press, Montréal, Canada, pp. 265-268.

[101] Sanou, L., Girard, P., and Guittet, L. (2006b) La programmation sur exemple : principe, utilisation et utilité pour les applications interactives. In: ESTIA, E.I. (ed) Ergo'IA, Bidart-Biarritz, pp. 201-208.

[102] Sanou, L., Girard, P., and Guittet, L. (2006c) Vers un outil pour la réalisation de système de programmation sur exemple. In: AFIHM (ed) RJC-IHM'06. AFIHM, Anglet, France.

[103] Sanou, L., Girard, P., Guittet, L., and Caffiau, S. (2008b) Tester la conformité d'une interface homme-machine à son modèle de tâches. In: AFIHM (ed) IHM'08. ACM Press, Metz, Université Paul Verlaine, pp. 159-162.

[104] Sanou, L., Guittet, L., and Caffiau, S. (2008c) La programmation sur exemple pour l'automatisation des tests d'interfaces. In: INNOVATION, E.E. (ed) Ergo'IA ESTIA & ESTIA INNOVATION, Bidart/Biaarritz, France, pp. 255-256.

[105] Scapin, D. (1986) Guide ergonomique de conception des interfaces homme-machine. INRIA Rocquencourt.

[106] Scapin, D., and Pierret-Golbreich, C. (1989) MAD : Une méthode analytique de description des tâches. Colloque sur l'Ingénierie des Interfaces Homme-Machine (IHM'89), Sophia-Antipolis, France, pp. 131-148.

[107] Shepherd, A. (1989) Analysis and training in information technology tasks. In: Diaper, D. (ed) Task Analysis for Human-Computer Interaction. Ellis Horwood, Chichester, USA, pp. 15-55.

[108] Shneiderman, B. (1983) Direct Manipulation: a Step beyond Programming Languages. IEEE Computer 16:57-69.

[109] Shneiderman, B. (1998) Designing the User Interface. Addison-Wesley.

[110] Smith, D.C. (1977) A Computer Program to Model and Stimulate Creative Thought. Birkhauser, Basel.

[111] Smith, D.C., and Cypher, A. (1995) KidSim: Child Constructible simulation. Imagina'95, Monte-Carlo, Février, pp. 87-99.

[112] Smith, D.C., Cypher, A., and Tesler, L. (2001) Novice Programming Comes of Age. In: Lieberman, H. (ed) Your Wish is My Command, pp. 7-20.

[113] Smith, D.C., Kimball, R., Verplank, B., and Harslem, E. (1982) Designing the Star User Interface. Byte 7:242-282.

[114] Tarby, J.-C. (2006) Evaluation précoce et conception orientée évaluation. In: ESTIA.INNOVATION, E. (ed) Ergo'IA 2006, Bidart/Biarritz France, pp. 343-346.

[115] Texier, G. (2000) Contribution à l'ingéniérie des systèmes interactifs : Un environnement de conception graphique d'applications spécialisées de conception. Université de Poitiers, Poitiers, p. 230.

[116] Thévenin, D. (2001) Adaptation en Interaction Homme-Machine : le cas de la plasticité. Université Joseph Fourier, Grénoble, p. 213.

[117] Tuchinda, R., Szekely, P., and Knoblock, C.A. (2008) Building Mashups by example. In: Library, A.D. (ed) Intellignet User Interface 2008. ACM Digital Library, Maspalomas, Gan Canaria, Spain, pp. 139-148.

[118] Wells, D. (2006) Extreme Programming: a gentle introduction.

[119] Whitley, K.N., Novick, L.R., and Fisher, D. (2006) Evidence in favor of visual representation for the dataflow paradigm: an experiment testing LabVIEW's. International Journal of Human-Computer Interaction 64:281-303.

[120] Wilson, S., and Jonhson, P. (1996) Bridging the Generation Gap: From Task to User Interface Designs. In: Vanderdonckt, J. (ed) Computer-Aided Design of User iterface (CADUI'96), Namur, Belgium, pp. 77-94.

[121] Winston, W.R. (1970) Managing the development of large software systems: Concept and techniques. In: Press, I.C.S. (ed) WESCOn, Los Alamitos, pp. 1-9.

[122] Wolber, D. (1996) Pavlov: Programming by Stimulus-Response Demonstration. In: Tauber, M. (ed) Human Factors in Computing Systems (CHI'96). ACM/SIGCHI, Vancouver, Canada, pp. 252-269.

[123] Wong, j., and Hong, J.L. (2007) Making mashups with marmite: towards end-user programming for the web. In: ACM-CHI (ed) Conference on Human Factors in Computing Systems, CHI'07. ACM Digital Library, San José, California, USA, pp. 1435-1444.

[124] Woods, W. (1970) Transition Network Grammars for Natural Language Analysis. Communications of the ACM 13:591-606.

Définition et réalisation d'une boîte à outils générique dédiée à la Programmation sur Exemple

Présentée par :

Loé SANOU

Sous la direction de **Patrick Girard** et **Laurent Guittet**

Résumé: L'implémentation d'un système intégrant la Programmation sur Exemple (PsE) demande au développeur de mettre à disposition de l'utilisateur final des outils d'assistance lors de la réalisation des tâches. Pour le développeur, cela passe par la mise à disposition des différents services à partir de l'interface utilisateur de l'application. Le système doit donc fournir des interfaces particulières, car non seulement l'objectif fonctionnel de l'application ne doit pas changer, mais surtout parce que les techniques de la PsE doivent être naturellement intégrées. Un système de PsE est difficile à implanter, et pourtant, la plupart possèdent des éléments en commun parmi lesquels on trouve une représentation des actions utilisateur, un historique des actions, et parfois un algorithme d'apprentissage symbolique opérant sur l'historique. Nous favorisons la création d'un tel système en fournissant les outils nécessaires sous forme d'une boîte à outils par extension de Swing. Les développeurs peuvent bâtir, avec un minimum d'effort, des applications mettant en œuvre les techniques de la PsE. Les principaux services de base ont été identifiés et définis : enregistrement des actions utilisateur, rejeu des actions et des techniques utilisables pour la mise en œuvre d'applications types. Ils ont été prototypés à travers l'outil *PbDToolkit,* ouvrant la voie vers la simplification de la mise en œuvre des applications de PsE. En utilisant *PbDToolkit*, il n'est pas nécessaire d'implémenter les fonctionnalités de base car toutes les opérations y sont déjà implémentées avec la liberté d'usage et d'exploitation offerte aux développeurs. *PbDToolkit* est instrumenté pour permettre de vérifier la conformité d'une IHM à son modèle de tâches. Le concept établit un lien entre les tâches élémentaires du modèle de tâches et les actions de l'IHM. Ainsi, à l'exécution, un scénario est généré suivant le format de scénario de l'environnement K-MADe, outil de modélisation utilisé.

Mots-Clés : Programmation sur Exemple, Interaction Homme-Machine, Interface Homme-Machine, Boîte à Outils, Test d'IHM, Modèle de tâches.

Definition and realization of generic toolbox dedicated to Programming by Demonstration

Abstract: The implementation of a system integrating Programming by Demonstration (PbD) requires the developer to provide for the end user, easy to use automation tools to help in task realization. For the developer, it passes through the provision of different services from the application user interface. System must provide special interfaces because not only the functional purpose of application must not change, but also the PbD techniques must be naturally integrated. A PbD system is difficult to implement, yet most have elements in common among them are user actions representation, historic action, and sometimes a symbolic learning algorithm operating on the historic. We favor creation of such systems by providing the necessary tools in a toolbox by extension of Swing. Developers can build, with minimal effort, applications implementing the PbD techniques. The main basic services have been identified and defined: user actions recording and replay, and technology used for implementing standard applications. They have been through the prototype tool *PbDToolkit*, paving the way towards simplifying the PbD applications implementation. Using *PbDToolkit*, it is not necessary to implement the basic features as all operations are currently implemented with the freedom of use and exploitation offered to developers. *PbDToolkit* is instrumented to verify the compliance of a GUI to its task model. The concept establishes a link between the basic tasks of the task model and the GUI actions. Thus, at execution, a scenario is generated following the format of the K-made environment scenario, the modeling tool used.

Keywords: Programming by Demonstration, Human Computer Interaction, Human Computer Interface, Toolkit, HCI test, Task model.