UNIVERSITÉ DE MONTRÉAL

UN ENVIRONNEMENT DYNAMIQUE DE DÉVELOPPEMENT (EDD) POUR LE PROTOTYPAGE RAPIDE D'INTERFACES GRAPHIQUES

ANDRÉ JODOIN

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AOÛT 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé

UN ENVIRONNEMENT DYNAMIQUE DE DÉVELOPPEMENT (EDD)
POUR LE PROTOTYPAGE RAPIDE D'INTERFACES GRAPHIQUES

présenté par : JODOIN André

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. GIULIANO Antoniol, Ph.D., président

M. DESMARAIS Michel C., Ph.D., membre et directeur de recherche

M. ROBILLARD Pierre-N., Ph.D., membre

REMERCIEMENTS

J’aimerais remercier tous ceux qui m’ont aidé tout au long de ce projet de recherche. Érick Dupuis, Serge Ferland, et Martin Picard de l’Agence spatiale canadienne pour m’avoir permis de réaliser cette maîtrise et pour m’avoir encouragé. Mes collègues de travail de l’Agence spatiale canadienne, Régent L’Archevêque, Pierre Allard, Pierre Langlois, Stéphane Beaudry et Mathieu Landry, pour m’avoir conseillé et avoir partagé avec moi leurs idées. Je veux remercier aussi Louis Granger de l’École Polytechnique de Montréal pour son aide et ses conseils, et mon directeur de recherche, Michel Desmarais, aussi de l’école Polytechnique de Montréal, pour m’avoir permis de présenter ce projet de recherche à deux conférences, pour sa patience, pour son aide et pour ses conseils.

Enfin, je remercie mes proches de m’avoir encouragé pendant que je réalisais ces travaux.

RÉSUMÉ

Le développement d’interfaces graphiques est une tâche complexe qui exige beaucoup de ressources spécialisées et de temps. Dans bon nombre de projets, cette tâche est laissée à la fin et, conséquemment, les résultats ne répondent pas toujours aux attentes des utilisateurs.

Ceci peut s'expliquer par le fait que l’utilisateur n’a pas participé aux phases d’analyse et de conception de l’architecture du système sous-jacente à l’interface graphique. L’utilisateur est un expert qui connaît bien les tâches qui devront être accomplies et il doit être invité à participer à la définition des besoins, notamment en ce qui a trait à l’ergonomie des interfaces homme-machine (IHM). Pour y parvenir, il est préférable de définir les exigences et de concevoir avec les utilisateurs les IHM, bien avant de concevoir l'architecture du système que l’on veut contrôler. Cette architecture sera ainsi assujettie aux besoins qu’ont les utilisateurs dans l’exécution de leurs tâches. Le prototypage des IHM permet aux utilisateurs de bien faire connaitre leurs besoins aux développeurs et de s’assurer que ceux-ci bénéficieront d’une IHM qui répond à leurs exigences. Les utilisateurs ne peuvent exprimer leurs besoins autrement qu'en ayant recours à une maquette avec laquelle ils peuvent interagir. Il est donc nécessaire d’avoir accès à des outils performants permettant le prototypage des IHM, à savoir les outils les plus efficaces, les plus simples et les plus complets possibles.

Les environnements de développement logiciel disponibles sur le marché offrent des outils d’aide au développement d’IHM. Ces outils permettent de manipuler directement les objets graphiques en y donnant accès à l’aide d’une palette ou d’un autre mécanisme. Ils permettent aussi de gérer la mise en page de vues et génèrent le squelette du code source des méthodes de traitement des évènements qui doivent être complétées par le programmeur. Malgré que ces outils soient très utiles, le développement des IHM exige tout de même beaucoup de travail et de connaissances, notamment parce que les objets graphiques sont reliés à l'application par la programmation à même les méthodes d'écoute d'évènements de ces objets. Cette façon de faire rend ardue l'utilisation d'une même IHM pour contrôler divers procédés, en plus de nécessiter beaucoup de connaissances en programmation. De plus, les comportements ainsi définis ne sont pas d'emblée réutilisables.

L'objectif global de ce projet de recherche est de faciliter la tâche de l’utilisateur concepteur d'IHM en lui offrant une plateforme de prototypage rapide des IHM à l’aide de laquelle il peut

tester avec des maquettes les exigences de l'IHM et les valider. Cet objectif vise particulièrement l'utilisateur qui, bien qu'il soit un expert dans son domaine, n'est pas un programmeur. Dans ce but, une architecture innovatrice et flexible de développement d'interfaces graphiques nommée « environnement dynamique de développement » (EDD) est proposée. L'EDD permet d'intégrer les objets graphiques (*widgets*) par manipulation directe et de les rendre « dynamiques », dans le sens où ils sont exécutés dans ce même environnement. En plus de permettre au concepteur de faire rapidement l'essai du comportement de l'interface, l'EDD offre une très grande flexibilité d'intégration de nouveaux objets graphiques, tout en minimisant le besoin de codage. Les comportements définis sont réutilisables et peuvent être associés à plusieurs événements.

L'aspect dynamique des composants et des comportements est ce qui distingue le plus l'EDD des autres environnements de développement d'IHM. Les environnements comme Microsoft Visual Studio et .Net, NetBeans de Sun Micro Systems ou encore Eclipse, proposent une approche en trois étapes : programmation, compilation et exécution. L'approche proposée ici est différente, car toute modification apportée à l'IHM est applicable immédiatement et peut-être testée directement dans l'environnement sans avoir à relancer l'EDD. En effet, celui-ci est à la fois l'éditeur visuel de l'IHM, l'environnement de développement Java et l'IHM même. Les modifications effectuées par manipulation directe à partir de l'éditeur visuel de l'IHM et les modifications apportées au code source Java contribuant au contrôle de l'IHM sont appliquées de façon dynamique et sont aussitôt sauvegardées.

De plus, cette plateforme offre une architecture fondée sur la configuration modèle-vue-contrôle (MVC), ce qui a pour résultat de maximiser la réutilisabilité des vues développées par le concepteur d'IHM. Le choix de la configuration MVC et de l'application dynamique des modifications a pour objet d'accélérer le processus de développement des interfaces, d'améliorer la qualité de leur conception et aussi d'assurer le prototypage des IHM tôt dans le processus de développement logiciel.

L'architecture modèle-vue-contrôle (MVC) de l'EDD permet de développer des interfaces malgré l'absence du système à contrôler, c'est-à-dire en n'utilisant que les parties modèle et contrôle de la configuration MVC et en substituant au système une souche (un remplacement, une ébauche) dont l'interface de programmation est la même que celle de celui-ci. La souche pourra être remplacée par le vrai noyau fonctionnel, une fois celui-ci disponible. De la même façon, il

est possible de contrôler au choix à partir d'une même vue des systèmes dont les interfaces de programmation sont identiques. Par exemple, il serait possible ce contrôler un système robotique et une simulation d'un tel système par la même IHM.

L'EDD est une extension de la plateforme Eclipse dont les plugiciels augmentent l'outillage tout en continuant d'offrir les nombreux avantages propres à son architecture. Les caractéristiques ajoutées complémentent l'environnement de développement Java d'Eclipse, en offrant, entre autres, une couche d'abstraction dans l'assemblage des comportements reliant l'interface graphique aux procédés, tout en favorisant leur séparation par le biais de l'architecture MVC.

L'EDD introduit les concepts de systèmes, de cibles et de contextes d'exécution allant dans ce sens. Le système est une interface abstraite à laquelle sont branchés les objets graphiques, tandis que la cible représente une adaptation proposée d'un système. Enfin, le contexte définit à quelle cible est associé un système. Le concepteur ou l'utilisateur peut, en pleine exécution, changer le contexte de façon à rediriger l'IHM vers un autre procédé, c'est-à-dire une implémentation différente du système sous son contrôle. Cette dernière caractéristique, qui a pour but de promouvoir la réutilisation des interfaces graphiques, offre une flexibilité accrue au niveau de l'aiguillage de l'application pendant son exécution.

L'EDD a comme caractéristique importante de permettre au développeur de fournir un objet graphique dynamique obtenu d'un plugiciel Eclipse à travers un point d'extension. Une fois le nouveau composant détecté, il est ajouté à la palette d'objets graphiques; son interface de programmation devient alors publique et directement visible au concepteur de l'IHM. Celui-ci peut alors prendre connaissance des événements qu'il peut gérer et connecter le composant à d'autres objets dans l'application en développement par manipulation directe, par la saisie de propriétés ou par l'utilisation de méthodes propres au composant. Il est aussi possible de faire basculer l'environnement dans le mode exécution et de valider d'emblée le comportement du nouveau composant graphique.

Un autre des objectifs visés par ce projet de recherche est de permettre la programmation dynamique d'interfaces et de classes Java qui sont automatiquement détectées, compilées et chargées par la machine virtuelle Java durant l'assemblage, sans qu'il soit nécessaire de redémarrer l'EDD. De cette façon, il est possible de faire usage de ces interfaces et classes Java immédiatement en les reliant à l'interface graphique par la programmation visuelle. Cette

fonctionnalité offre au concepteur la possibilité de développer un système complexe, tout en lui permettant de le brancher à l'interface graphique toujours sans devoir relancer la plateforme.

De façon à démontrer la faisabilité de l'EDD et à en valider l'utilité, on a élaboré à l'Agence spatiale canadienne (ASC) une implémentation qui offre toutes les fonctionnalités nécessaires à son utilisation en milieu opérationnel. Cette implémentation a permis de développer plusieurs IHM toujours en fonction aujourd'hui, entre autres l'application MARVIN qui sert à tester des fichiers de paramètres destinés au bras robotique de la Station spatiale internationale (SSI). Malgré les nombreux autres outils disponibles sur le marché, on envisage à l'heure actuelle d'utiliser l'EDD pour le développement de nouveaux projets. Son utilisation a débuté en 2006 avec une IHM de contrôle du robot CART que l'on utilise dans le cadre de divers projets de recherche à l'ASC. Son développement s'est ensuite poursuivi, et depuis, plusieurs améliorations ont été apportées pour résoudre les problèmes auxquels sont confrontés les utilisateurs, ainsi que pour répondre à leurs observations, On envisage également d'apporter certaines autres améliorations. On envisage, entre autres, de proposer un nouveau mode de programmation visuel en réponse aux observations des utilisateurs voulant que l'assemblage d'appel de méthodes dans le cadre de la présentation actuelle est difficile. En effet, l'assemblage des comportements, qui se fait à l'aide d'un arbre, n'est pas intuitif lorsque l'arbre est trop profond. Une visualisation semblable à celle proposée par les diagrammes d'interactions UML est envisagée.

L'implémentation actuelle permet de manipuler directement des objets graphiques. Elle offre la possibilité de définir des systèmes, des cibles et des contextes, et de les associer à des applications ou des classes, ainsi que de sélectionner le contexte d'exécution à activer. Elle permet également d'assurer la programmation dynamique d'interfaces et de classes Java que l'on peut utiliser immédiatement. Il est possible d'associer les systèmes aux objets graphiques à l'aide de comportements dont la définition est effectuée à l'aide d'un assistant de programmation visuelle conçu à cette fin. En dernier lieu, il est possible d'intégrer rapidement de nouvelles librairies d'objets graphiques et de les tester dans l'environnement.

Une démonstration du prototype initial de l'EDD a eu lieu à la conférence Interaction Homme-Machine (IHM) 2006 de Montréal et une communication de nature industrielle a été faite à ce sujet à la conférence IHM 2009 de Grenoble.

ABSTRACT

The development of graphical interfaces is a complex task that requires a lot of specialized resources and time. In many projects, this task is left to the end and, consequently, the results do not always meet users' expectations.

This can be explained by the fact that the user has not participated in the software analysis and design phases of the system architecture behind the user interface. The user is an expert who is familiar with the tasks to be performed and must be invited to participate to in the requirement definition, particularly with respect to those requirements that affect human factors. In order to achieve this, it is best to define with the users the requirements and design of the Human-Computer Interface (HCI) before designing the system to be monitored. This architecture is thus constrained by the needs that users have in performing their tasks. HCI prototyping allows users to communicate their needs to developers and ensure that they get an HCI that meets their requirements. Users can't express their needs other than by using a mockup with which they can interact. It is therefore necessary to have access to effective tools for HCI prototyping. Tools that are most effective, simple and complete as possible.

The software development environments available on the market today provide tools to assist in development of HCIs. These tools allow drag-and-drop of graphical components they make available using a pallet or other mechanism. They also manage the layout of views and generate skeleton source code for event handling methods to be completed by the programmer. Despite these strengths are helpful assistants, development of HCIs still requires much work and knowledge, particularly because the graphical components are linked to the application through event handling methods contained in the HCI code. This approach makes it difficult to use the same HCI to control various processes in addition to requiring a lot of programming knowledge. Moreover, resulting behaviors are not easily reusable.

The overall objective of this research project is to facilitate the task of the user by offering a platform for rapid prototyping of HCIs which he can use to test and validate HCI requirements using mockups. This objective is aimed particularly at domain experts who are not necessarily programmers. To this end, an innovative and flexible architecture for the development of graphical interfaces called the Dynamic Development Environment (DDE) is proposed. The DDE provides the capability to drag-and-drop graphical components (widgets) and allows them to be

"dynamic" in the sense that they run in that environment. It allows the designer to quickly test the behavior of the interface and offers great flexibility to incorporate new graphical components, while minimizing the need for coding. Behaviors defined in the DDE are reusable and can be associated with multiple events.

The dynamic aspect of the components and behavior is the primary distinction of the DDE in relation to other HCI development environments. Environments such as Microsoft Visual Studio, NetBeans or Eclipse offer a three-step approach involving coding, building and execution. The approach proposed here is different in the sense that any modification to the HCI is immediately applied and can be directly tested in the environment without the need to restart the DDE. In fact, the DDE is the visual editor, the Java development environment and the HCI itself at the same time. Both changes made by drag-and-drop in the visual editor and changes to the Java source code contributing to the control of the HCI are dynamically applied immediately after being saved.

In addition, this platform provides an architecture based on the Model-View-Control (MVC) pattern, which maximizes the reusability of views developed by HCI designer. The choice of the MVC pattern and the dynamic application of the changes are designed to accelerate interface the development process, improve the quality of their design and also encourage HCI prototyping early in the software development process.

The Model-View-Control (MVC) architecture of the DDE enables the development of interfaces, despite the absence of the control system, that is to say the model and control parts of the MVC pattern, substituting a stub (replacement, draft) whose programming interface is the same as the real system. The stub can be replaced by the real system core when available. Similarly, it is possible to control different systems offering identical application programming interfaces (API) from a single view. For example, a robotic system could be controlled by the same HCI as a simulation of this system.

The DDE extends the Eclipse platform by offering plug-ins to augment its tooling while maintaining the many benefits of its architecture. The added features are an addition to the Eclipse Java Development Toolkit (JDT), among other things, offering a layer of abstraction in assembly behavior between the HCI processes, while facilitating their decoupling through the MVC architecture.

The DDE introduces the concepts of systems, targets and execution contexts. The system is an abstract interface that can be connected to DDE widgets, while the target is an implementation of a proposed system. Finally, the context defines which target is associated with a system. The designer or the user may, during runtime, change the context to redirect the HCI to another process, that is to say, another implementation of the system under its control. This latter feature is designed to promote the reuse of graphical interfaces and offers greater flexibility in the control of the application during its execution.

An important feature of the DDE is the possibility it offers to the developer to provide a dynamic widget obtained from an Eclipse plug-in extension point. When the new widget is detected, it is added to the palette and its API becomes public and visible directly in the HCI designer. The user can discover what events the widget fires and can connect them to other widgets through visual programming by using property getters, property setters and method calls. It is also possible to switch the environment in the run mode and immediately validate the behavior of the new widget.

Another of the objectives of this research project is to enable dynamic programming of interfaces and Java classes which are automatically detected, compiled and loaded by the Java Virtual Machine (JVM) during the assembly without having to restart the DDE. In this way, it becomes possible to use these interfaces and Java classes immediately by connecting to the HCI through visual programming. This feature provides the designer the ability to develop a complex system while allowing it to connect to the HCI without ever having to restart the platform.

In order to demonstrate the feasibility and validate the usefulness of the DDE, an implementation that offers all the features required for its use in an operational environment has been developed at the Canadian Space Agency (CSA). It was used to develop several HCIs still in use today. Notably, the application MARVIN which tests parameter files for the robotic arm of the International Space Station (ISS). Despite the many alternatives available on the market, the use of the DDE for the development of new projects is presently being considered. Its use began in 2006 with an HCI for controlling the CART robot used in various research projects at the CSA. Its development is being pursued and since then several improvements have been made to address the problems encountered by users and respond to their comments. Also, some improvements are planned for the future. Among other things, a new mode of visual

programming is proposed to address the comments of users to the effect that the assembly of method calls in the current presentation is difficult. Indeed, the assembly of behaviors is done using a tree and is not intuitive when the tree is too deep. A display mode similar to that proposed by UML interaction diagrams is considered.

The current implementation allows for widget drag-and-drop style interactions. It offers the ability to define systems, targets and contexts, and to associate them with specific applications or classes and to select the execution context to activate. It also allows dynamic programming of interfaces and Java classes that can be used immediately. It is possible to integrate systems with widgets using behaviors defined using a visual programming tool. Finally, it is possible to rapidly add widget libraries and to use them afterwards.

The initial prototype of the DDE was demonstrated at the *Interaction Homme-Machine* (IHM) conference in 2006 in Montreal, and an industrial paper was presented on this topic at the IHM conference in 2009 in Grenoble.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ASC	Agence spatiale canadienne
CME	Cadriciel de modélisation Eclipse
CSA	*Canadian Space Agency*
CVS	*Concurrent Versions System*
DDE	*Dynamic Development Environment*
EDD	Environnement dynamique de développement
EUD	*End-User Design*
HCI	*Human-Computer Interface*
IHM	Interface Homme-Machine
IPA	Interface de programmation d'applications
JDK	*Java Development Kit*
JDT	*Java Development Toolkit*
JVM	*Java Virtual Machine*
ME	*Mobile Edition*
MVC	Modèle-vue-contrôle
Nb	Nombre
QAW	Ateliers d'attributs de qualité
RCP	*Rich Client Platform*
SQL	Langage d'interrogation standard
SSI	Station spatiale internationale
SVG	*Scalable Vector Graphics*
SWT	*Standard Widget Toolkit*
UIML	Langage de balisage d'interface utilisateur
UML	Langage de modélisation unifié
VB	*Visual Basic*
XML	Langage de balisage extensible
XUL	Langage d'interface utilisateur XML

LISTE DES ANNEXES

INTRODUCTION

Le prototypage des interfaces homme-machine (IHM) permet aux utilisateurs de bien faire connaître leurs besoins aux développeurs et de s'assurer que ceux-ci obtienne une IHM qui répond à leurs exigences. Il est donc nécessaire d'avoir accès à des outils performants permettant ce prototypage, à savoir des outils les plus efficaces, les plus simples et les plus complets possibles. La problématique reliée au prototypage rapide des IHM n'est pas nouvelle. On n'a qu'à penser au système Druid (Singh, Kok, et Ngan, 1990) présenté au début des années 1990 pour le prototypage rapide d'IHM. Depuis, plusieurs outils de prototypage ont vu le jour, aussi bien sur le plan théorique que commercial. Sur le plan théorique, de nombreux prototypes ont été proposés, comme, par exemple, les prototypes TkZinc et INDIGO qui offrent des architectures de collaboration entre concepteurs graphiques et programmeurs (Blanch, Beaudoin-Lafon et Conversy, 2005) (Chatty, Lecoanet et Sire, 2004), ou encore le Squidy, un environnement qui permet d'unifier une grande variété de dispositifs très hétérogènes présenté dans le cadre de travaux portant sur la conception interactive d'interfaces multimodales (König, Rädle et Reiterer, 2010). Sur le plan commercial, il existe de nombreux environnements de développement, les plus connus étant Eclipse, Visual Studio et NetBeans. Non seulement ces outils ont-ils permis de progresser en ce qui concerne la séparation interface-noyau fonctionnel, ils ont également contribué à faciliter la mise en forme de l'interface et la structuration du code. Il s'agit maintenant d'outils dont les développeurs d'interfaces ne peuvent plus se passer.

Dans le cadre du présent projet de recherche, nous présentons une architecture innovatrice et flexible de développement d'interfaces graphiques que nous avons choisi d'appeler « Environnement dynamique de développement » ou EDD. L'EDD permet d'intégrer les objets graphiques (*widgets*) par manipulation directe et de les rendre «dynamiques», dans le sens où ils sont exécutés dans ce même environnement. Une première version de ce système a été présentée à la conférence Interaction Homme-Machine (IHM) 2006 de Montréal (Jodoin, L'Archevêque, Allard, et Desmarais, 2006) et une communication de nature industrielle a été faite à ce sujet à la conférence IHM 2009 de Grenoble (Jodoin et Desmarais, Environnement dynamique de développement (EDD) pour le prototypage rapide d'interfaces graphiques, 2009).

Dans une perspective d'utilisation, l'EDD se distingue des autres environnements de développement, comme Netbeans, .Net et Eclipse avec son « *Visual Editor* », en offrant la

possibilité à l'utilisateur de tester l'IHM à mesure qu'il la construit sans étapes supplémentaires de compilation et d'exécution. Il permet au concepteur de faire rapidement l'essai du comportement de l'interface et offre une très grande flexibilité d'intégration de nouveaux objets graphiques, tout en minimisant le besoin de codage.

L'environnement est implanté en Java et est intégré au cadre d'Eclipse. Il repose sur l'utilisation de la réflexivité de Java et sur les principes des langages déclaratifs d'IHM, comme le LGIU (langage de gestion d'interface utilisateur) (Abrams, Phanouriou et Batongbacal, 1999), mieux connu sous le sigle UIML.

Il a été conçu et développé à l'ASC dans le but d'offrir aux utilisateurs experts un système leur permettant de créer rapidement des prototypes d'IHM pour le contrôle de systèmes robotiques. L'ASC développe de nombreux projets en robotique et avait besoin d'un environnement de développement simple et efficace qui permettrait aux utilisateurs de développer rapidement des prototypes d'IHM au tout début des projets et avant même que les systèmes à contrôler ne soient complètement définis.

Dans ce contexte, les utilisateurs visés sont les spécialistes des opérations spatiales, qui sont généralement des scientifiques ou des ingénieurs. La plupart d'entre eux possèdent des connaissances de base en informatique qui leur permettent une plus grande flexibilité d'utilisation de l'EDD, celui-ci leur permettant d'avoir recours aux outils de programmation Java en complément aux outils de programmation visuelle et de manipulation directe.

Comme autre caractéristique utile aux utilisateurs et absente des environnements actuels, il y la possibilité de réaliser la conception complète d'une IHM de contrôle d'un système connecté par une interface de programmation d'applications (IPA). Il est donc possible de concevoir l'IHM sans que le système soit disponible; il suffit tout simplement de le remplacer provisoirement par une souche possédant la même IPA. Cette caractéristique avait été proposée à l'ASC pour la station de contrôle de la boîte à outils d'autonomie des opérations spatiales (Dupuis, Doyon et Martin, 2004) à l'ASC.

Au moment de la conception de l'EDD, il est devenu rapidement évident qu'un tel environnement pouvait servir à développer des IHM pour plusieurs autres types de systèmes. Il a

notamment été utilisé pour développer des IHM de contrôle de robots (CART et NORCAT) et pour développer l'IHM d'une application automatisée de test d'IHM pour simuler les interactions utilisateur (MARVIN).

CHAPITRE 1 - REVUE DE LITTÉRATURE

Le développement d'un environnement de prototypage qui offre des outils puissants et flexibles d'assemblage d'IHM par des utilisateurs non programmeurs dans un contexte d'exécution dynamique exige la prise en compte de plusieurs technologies, disciplines et concepts. L'objet de ce chapitre est de présenter un survol de la littérature sur les aspects pertinents d'un tel développement. Ces aspects sont les suivants :

1. La promotion par le prototypage de la participation de l'utilisateur non programmeur à la spécification des exigences de l'IHM. L'utilisateur non programmeur est l'expert dans le domaine pour lequel une IHM doit être développée. Il doit participer tôt à la spécification des exigences de l'IHM et l'on sait depuis fort longtemps que les exigences à cet égard doivent être précisées lorsque l'utilisateur dispose d'une maquette fonctionnelle de l'interface de l'application (Seffah, Gulliksen et Desmarais, 2005), même s'il lui faut modifier sensiblement le cas échéant les exigences originales (Hennipman, Oppelaar et Van Der Veer, 2008).

2. Les environnements de développement et de prototypage. On survole les environnements de développement disponibles aujourd'hui, les critères d'évaluation de ces environnements et les défis qui sont d'actualité. Il faut également réviser les facteurs ergonomiques qui entrent en jeu au moment de cette conception.

3. La programmation par l'utilisateur non programmeur. L'environnement de prototypage doit offrir aux utilisateurs qui ne sont pas nécessairement des programmeurs la possibilité d'effectuer l'assemblage des vues graphiques et des comportements de l'IHM. L'EDD permet aux utilisateurs qui n'ont pas nécessairement des compétences avancées en programmation d'effectuer cet assemblage, ce qui impose des contraintes de conception propres aux utilisateurs non programmeurs.

4. Les langages déclaratifs. La majorité des IDE avancés sont fondés sur un langage de définition qui permet de préciser les IHM par le biais d'un langage de modélisation. Ces

langages permettent, entre autres, de développer des outils flexibles qui partagent une même compréhension du modèle de l'IHM. C'est l'approche utilisée par l'EDD.

5. L'abstraction, la plasticité et la flexibilité permettent de définir des systèmes flexibles malléables qui s'adaptent à leur contexte d'utilisation.

1.1 Participation de l'utilisateur à la spécification des besoins

L'utilisateur est un expert qui connaît bien les tâches à accomplir qu'il faut inviter à participer à la définition des besoins, notamment en ce qui concerne l'ergonomie des interfaces homme-machine (IHM). Il faut faciliter la participation de l'utilisateur tôt dans le processus de spécification des exigences de l'IHM en lui offrant une plateforme permettant le prototypage rapide des IHM à l'aide de laquelle il peut tester ces exigences avec des maquettes et les valider.

Les études montrent que les interactions sont faibles entre les concepteurs d'IHM et leurs clients, les utilisateurs qui sont des experts dans le domaine, durant la phase de spécification des applications logicielles (Vukelja, Müller et Opwis, 2007). Les concepteurs adoptent rarement une approche axée sur le client (Scaffidi, Myers et Shaw, 2007). De plus, ils possèdent peu de connaissances propres au domaine des IHM et ne prennent pas toujours les décisions d'architecture qui favorisent l'utilisabilité des IHM (Bass et Kates, 2001). L'expertise offerte par les utilisateurs devrait pouvoir mieux contribuer à la spécification des IHM.

Comme solution à ce problème, des efforts sont déployés pour promouvoir la collaboration entre les concepteurs et les utilisateurs durant les activités de spécification et de gestion des exigences (Rashid, Weisenberger et Meder, 2008). Il faut notamment assurer le développement d'outils graphiques de représentation des exigences dans le but de favoriser les échanges entre les concepteurs et les utilisateurs au cours des activités reliées à la gestion de celles-ci.

Ces études nous indiquent qu'il est important de s'assurer d'une plus forte participation des utilisateurs à la spécification, à la conception et au développement des applications.

Les esquisses ou modèles en vraie grandeur sont des outils de communication indispensables de la spécification et de la conception de l'IHM qui favorisent la participation des utilisateurs tôt dans ce processus. Les esquisses tracées à la main constituent la forme la plus répandue de

réalisation de cette pratique. Certains outils informatiques offrent une version électronique des maquettes papier; SILK en est un exemple (Landay et Myers, 1995).

Les prototypes d'IHM sont des maquettes haute fidélité qui offrent une solution de rechange intéressante aux maquettes tracées à la main dans la mesure où les outils utilisés pour les produire présentent des avantages supplémentaires par rapport à la technique de traçage. DRUID, un environnement de prototypage rapide pour la démonstration des interactions (Singh Kok et Ngan, 1990) offrait déjà des assistants d'assemblage par manipulation directe des prototypes d'IHM à partir de composants graphiques. Sa principale caractéristique était de permettre de générer le code source de l'application à partir du modèle créé par l'utilisateur, de façon à permettre son exécution pour ensuite en tester les interactions.

Les environnements de prototypage sont des assistants à l'exploration des exigences précieux. On les utilise dans le cadre d'une approche de conception itérative fondée sur l'expérience (Hennipman, Oppelaar et Van Der Veer, 2008). L'utilisateur améliore progressivement le prototype en l'utilisant pour la recherche des fonctionnalités nécessaires.

Grâce à ces travaux d'amélioration, les outils de prototypage mènent à la production de prototypes multifidélités (Memmel, Vanderdonckt et Reiterer, 2008) en permettant de fournir des objets interactifs à n'importe quel niveau de fidélité. Le prototypage à fidélité mixte, qui consiste à mélanger simultanément différents niveaux de fidélité à l'intérieur d'un même prototype, permet une transition dynamique et souple de ces niveaux de fidélité par le biais de l'évolution d'un prototype et d'objets graphiques (Vanderdonckt et Coyette, 2007). Une approche similaire est offerte par l'EDD qui propose des mécanismes dynamiques d'ajout et de remplacement des composants graphiques. Les composants graphiques basse fidélité peuvent être introduits au début et être remplacés ultérieurement par des composants de plus haute fidélité.

1.2 Environnements de développement

Les premiers environnements de développement d'IHM sont apparus dans les années 1970 et le sujet a fait l'objet de nombreuses recherches depuis (Myers, Hudson et Pausch, 2000). Certaines études portent sur les environnements de développement qui permettent aux concepteurs de créer des IHM sans avoir à programmer et à faire l'apprentissage de nombreux cadriciels (Singh, Kok et Ngan, 1990). Par exemple, l'environnement de conception d'interfaces utilisateur UIDE (*User*

Interface Design Environment) utilise le modèle du noyau fonctionnel de l'application pour générer automatiquement une IHM (Foley, Kim et Kovacevic, 1991). D'autres travaux ont suivi (Sukaviriya, Foley et Griffith, 1993) dans lesquels on propose plutôt d'utiliser le modèle de l'application pour faciliter la conception de l'IHM par le développeur en simplifiant la connexion de l'IHM au noyau fonctionnel.

Ces travaux ont marqué profondément les développeurs d'IHM actuels. De nos jours, les IHM sont pratiquement toutes conçues à l'aide d'environnements de développement. Les environnements les plus utilisés sont Microsoft Visual Studio.Net (Microsoft, Le site de Microsoft Visual Studio. http://msdn.microsoft.com/en-us/vstudio), Eclipse (Eclipse.org) et Netbeans (Netbeans.org). Mentionnons également Labview (National Instruments), un environnement très souvent utilisé pour le développement de simulations qui renferme également des outils de développement d'IHM.

En plus des environnements de développement susmentionnés, de nombreux outils de prototypage sont disponibles sur le marché. Ils varient des outils de développement de maquettes aux outils de développement de prototypes, voire d'IHM complètes. Les deux environnements suivants sont d'intérêt dans le cadre des travaux présentés ici :

- GUI Design Studio (Carrera Software). Environnement de prototypage par manipulation directe qui offre une architecture fondée sur les composants.
- Microsoft Expression Blend (Microsoft, Page web de Microsoft Expression Blend). Environnement complet qui permet de passer du prototype à l'application. Il s'intègre à Microsoft Visual Studio (Microsoft, Le site de Microsoft Visual Studio http://msdn.microsoft.com/en-us/vstudio) et comprend l'outil Sketchflow pour le développement de prototypes basse fidélité.

Ces deux environnements exigent le déploiement du prototype pour en permettre l'exécution. Myers, Hudson et Pausch (2000) proposent une méthode d'évaluation de l'utilité de ces environnements. Les cinq aspects suivants sont évalués à l'aide de cette méthode :

1. Les éléments de l'IHM auxquels s'applique l'environnement : les outils sont efficaces dans tous les aspects de l'assemblage de l'IHM.

2. Le seuil et le plafond : le niveau de difficulté d'apprentissage par rapport à ce que l'outil permet d'accomplir.
3. La voie offrant la plus faible résistance : l'aptitude de l'environnement à orienter le développeur dans la bonne direction.
4. La prévisibilité : les environnements qui comportent des éléments d'automatisme peuvent parfois être imprévisibles.
5. Les cibles en évolution : les environnements ont du mal à suivre l'évolution des technologies et des IHM.

Myers *et al*. présentent aussi une liste des technologies qui ont eu du succès : entre autres, les systèmes de fenêtrage et les boîtes à outils comme, p. ex., Motif et X-Windows, les langages d'évènements, les systèmes graphiques interactifs et la manipulation directe, les systèmes de composants (OLE de Microsoft et *Java Beans* de Sun), les scripts et les langages interprétés, l'hypertexte et l'internet, et la programmation orientée objet. Ces technologies sont toujours employées pour le développement d'IHM.

L'emploi de technologies éprouvées pour la conception des environnements de développement a eu, bien entendu, une incidence positive sur l'utilisabilité de ces environnements, mais il faut aussi tenir compte des facteurs ergonomiques qui influent sur la conception d'une IHM. Bass et Kates (2001) présentent une démarche qui permet d'établir des liens entre les aspects de l'utilisabilité et l'architecture des IHM. Ils proposent une liste étoffée de scénarios dans le cadre desquels les facteurs ergonomiques ont un impact direct sur l'architecture et présentent des patrons de conception en guise de solutions. La capacité d'assembler des groupes de commandes, de récupérer un mot de passe perdu, de défaire et de refaire des opérations en sont quelques exemples.

Dans une publication subséquente, Bass et John (2003) proposent une classification hiérarchique fondée sur l'importance de l'incidence des avantages offerts par les scénarios proposés sur le plan de l'utilisabilité et présentent un tableau de corrélation entre les tactiques architecturales, les avantages sur le plan de l'utilisabilité et l'importance de leur incidence (voir aussi Bass et Kates, 2001).

Ces travaux mènent à une approche qui permet d'améliorer le processus de développement logiciel sur le plan de l'ergonomie liée à la réalisation des IHM. C'est ce qu'ont proposé Rafla, Robillard et Desmarais (Rafla, Robillard et Desmarais, 2007) en présentant une adaptation des ateliers d'attributs de qualité (QAW) régie par les facteurs ergonomiques et en décrivant son intégration au processus de développement logiciel.

Comme autre caractéristique favorisant l'efficacité des environnements de développement, il y a la capacité de modifier des applications pendant leur exécution. Le cycle d'édition de l'ensemble code/compilation/lancement de l'application est de toute évidence un artéfact des plateformes traditionnelles de développement dont l'étape de compilation s'impose comme entrave à l'efficacité du développeur. Pour y pallier, la possibilité d'appliquer des modifications à l'IHM pendant son exécution est offerte par la boite à outils d'interaction Squidy (Auer, Pölz et Biffl, 2009), ce qui a pour résultat de réduire les changements de contextes d'interaction dans l'exercice de prototypage. Selon Ruyter et Sluis (Ruyter et Sluis, 2006), les environnements d'EUD doivent offrir une rétroaction rapide à l'utilisateur, ce qui est le cas des environnements dynamiques de conception d'IHM; un environnement offre une rétroaction rapide lorsqu'il n'interrompt pas l'exécution de l'application pour apporter des changements au code.

En effet, hormis le fait que la compilation semble une étape inutile du cycle de conception, la motivation derrière la conception d'environnements dynamiques est plus souvent associée au besoin de ne pas interrompre l'exécution d'applications critiques offrant des services à de nombreux utilisateurs ou encore au besoin de réduire le coût de la maintenance des logiciels (Pukall, Kästner et Götz, 2009); néanmoins, le progrès réalisé profite au développement des environnements de prototypage. Le chapitre 3 du présent mémoire explique en détail comment l'EDD pousse cette approche encore plus loin en combinant plusieurs techniques permettant d'offrir un environnement de prototypage d'IHM où toutes les modifications sont appliquées et activées au moment de l'exécution. Une de ces techniques consiste à permettre la programmation Java dynamique.

Les langages de programmation comme Java, par exemple, grâce à leur réflectivité, permettent de substituer facilement les parties d'un programme informatique pendant son exécution. Il convient de souligner que les machines virtuelles Java actuelles n'offrent pas cette possibilité. Il est, par contre, relativement simple de bénéficier de cette possibilité en utilisant une instrumentation qui

permet de passer outre le comportement par défaut du chargeur de classes. Selon des études récentes, il faudrait faire en sorte que les futures machines virtuelles Java offrent cette possibilité; il faudrait pour ce faire en améliorer l'architecture (Gregersen, Simon et Jorgensen, Towards a Dynamic-Update-Enabled JVM, 2009).

Une approche semblable est intégrée à l'environnement dynamique de développement Jpie (Goldman, 2003), lequel offre la possibilité, grâce à ses classes Java dynamiques, de développer l'application pendant son exécution.

Les techniques proposées à ce jour pour le chargement de classes Java modifiées pendant l'exécution d'une application consistent à utiliser un chargeur de classes différent de celui qui a déjà été utilisé pour charger la classe ou à utiliser le même chargeur de classes en modifiant le nom de la classe (Gregersen et Jorgensen, Dynamic update of Java applications—balancing change flexibility vs programming transparency, 2009). Le coût de ce type d'instrumentation en termes de performance de l'exécution est relativement faible; il n'est généralement associé qu'à la performance de l'IPA de l'environnement de prototypage. L'EDD propose une technique plus efficace encore, laquelle consiste à utiliser un chargeur de classes qui lit les classes à même leurs fichiers et décèle les changements à ces fichiers de façon à pouvoir les recharger automatiquement au besoin.

Une autre technique de modification d'une application en cours d'exécution consiste à se servir des langages déclaratifs utilisés pour leur grande flexibilité durant la conception et l'exécution. Les modèles qui en découlent sont malléables, car ils sont interprétés pendant l'exécution (Szekely, Sukaviriya et Castells, 1995). Il est possible de les modifier à même leur représentation en mémoire (Lehmann, Blumendorf et Albayrak, 2010). On traite de ces langages en 1.4.

L'utilisation des langages réflectifs et déclaratifs est directement applicable à nos travaux, mais il existe plusieurs autres techniques qui permettent de modifier une application en cours d'exécution. Par exemple, la programmation orientée aspects permet aussi de modifier ainsi une application (Previtali et Gross, 2006). Ce type de programmation isole dans les aspects les comportements partagés par plusieurs classes. L'aspect définit l'endroit où doit avoir lieu l'exécution et le point de jointure un endroit où d'autres comportements peuvent s'insérer; cette programmation est donc particulièrement bien adaptée aux modifications en cours d'exécution.

Auer, Pölz, et Biffl (Auer, Pölz et Biffl, 2009) proposent le UMLet, un environnement hautement dynamique dans lequel l'utilisateur peut apporter des modifications au modèle de l'application, lesquelles seront appliquées pendant que celle-ci est en cours d'exécution. On utilise l'UML pour représenter le modèle. Ainsi, pour créer un nouveau composant graphique à l'aide de l'UML, il suffit à l'utilisateur d'en préciser les caractéristiques; il pourra ensuite l'utiliser immédiatement. Cette démarche est possible grâce à une implémentation d'environnement d'EUD, le UMLet, qui répond au critère de succès énoncé par Ruyter (Ruyter et Sluis, 2006), à savoir que les environnements d'EUD doivent permettre à l'utilisateur de bénéficier d'une rétroaction rapide. L'exercice se limite à la création dynamique de nouveaux composants graphiques.

L'environnement proposé par Auer, Pölz, et Biffl est celui qui se rapproche le plus de l'environnement présenté dans cet ouvrage, mais il ne s'agit que d'une démarche partielle appliquée à la création de composants graphiques réutilisables. L'environnement que nous proposons est un environnement de prototypage d'IHM complet (Jodoin, L'Archevêque, Allard et Desmarais, 2006; Jodoin et Desmarais, L'environnement dynamique de développement (EDD) pour le prototypage rapide d'interfaces graphiques, 2009). Ce qui le distingue des autres environnements semblables, c'est que tout le développement et toutes les modifications apportés par l'utilisateur sont intégrés de façon dynamique au prototype de l'IHM sans nécessiter son redéploiement, c'est-à-dire pendant qu'il est en marche.

Malgré tout ce progrès et l'entrée en scène de nombreux environnements de développement, le développement des IHM des générations futures reste difficile et fait écho aux problèmes antérieurs (Olsen et Klemmer, 2005). Plusieurs défis demeurent d'actualité : les besoins d'interactions continues et multiutilisateurs, la collaboration multidisciplinaire, l'intégration de nouveaux dispositifs d'interaction et la capacité de stimuler la créativité plutôt que de simplement faciliter le développement.

On a tenté, dans le cadre de plusieurs travaux récents, de relever ces défis. TkZinc et INDIGO proposent des architectures de collaboration entre concepteurs graphiques et programmeurs (Blanch, Beaudoin-Lafon et Conversy, 2005; Chatty, Lecoanet et Sire, 2004). Le SVG (*Scalable Vector Graphics*) standard permet au concepteur graphique d'utiliser ses propres outils pour produire en partie l'IHM avec la même représentation que celle qu'utilise le programmeur. Squidy est une bibliothèque qui permet l'unification d'une grande variété de dispositifs très

hétérogènes. Les boites à outils typiques offrent des solutions pilotes pour des dispositifs spécifiques et l'ajout de nouveaux dispositifs exige d'apporter des modifications à la spécification des applications. En favorisant l'abstraction des dispositifs d'entrée-sortie, Squidy se veut une solution plus générique sur le plan de l'interaction. Elle est présentée dans le cadre de travaux portant sur la conception interactive d'interfaces multimodales (König, Rädle et Reiterer, 2010).

1.3 Programmation par des utilisateurs non programmeurs

L'expression *end-user programming* ou programmation par des utilisateurs non programmeurs a été introduite il y a longtemps (Nardi, 1993); les concepts d'EUD (*End-User Design*) et d'EUSE (*End-User Software Engineering*) en découlent. La programmation par des utilisateurs non programmeurs est un paradigme qui a pris de plus en plus d'importance au cours des dernières années (Lieberman, Paterno et Klann, 2006). De nos jours, la plupart des gens connaissent les technologies informatiques, mais ne possèdent pas les connaissances nécessaires pour développer ou modifier leurs applications à l'aide des langages de programmation. Il s'agit là d'un défi qui exige la mise en place d'un nouveau paradigme fondé sur une approche multidisciplinaire faisant appel à plusieurs domaines d'expertise, comme le génie logiciel, les interactions humain-machine, le travail collaboratif assisté par ordinateur, domaines entre lesquels il y a peu d'interactions de nos jours.

De plus en plus de logiciels offrent des interfaces dans lesquelles l'utilisateur peut définir ses propres applications; c'est le cas, par exemple, des scripts VB de la suite bureautique de Microsoft, des formules et des scripts de chiffriers ou encore de l'OLAP, un environnement dans lequel l'utilisateur peut spécifier des interrogations en langage SQL.

En 2012, on comptera plus de 3 millions de programmeurs aux États-Unis, ainsi que plus de 55 millions d'utilisateurs de chiffriers et des bases de données faisant usage de formules mathématiques et d'interrogations en SQL (Ko, Abraham et Beckwith, 2009). Des millions d'utilisateurs utilisent Flash pour développer des prototypes d'IHM et développer des simulations dans MATHLAB.

Historiquement, on n'avait pas vraiment intérêt à effectuer des travaux de recherche et de développement dans le domaine. Mais récemment, avec l'arrivée des technologies du Web et l'omniprésence des dispositifs mobiles, cette tendance s'est inversée.

Entre autres, comme aspect de plus en plus étudié en vue d'améliorer la flexibilité des environnements d'EUD, il y a la possibilité pour l'utilisateur de modifier le comportement d'une application en cours d'exécution. Par exemple, les langages comme Java, grâce à leurs caractéristiques de réflectivité, facilitent grandement cette démarche en permettant d'invoquer les fonctionnalités du programme de façon programmatique (les objets sont instanciés et les méthodes sont appelées par réflectivité). L'approche proposée offre à l'utilisateur une flexibilité accrue en lui permettant de modifier certaines parties du code d'une application en cours d'exécution. On a déjà traité ci-dessus de cette capacité de modification d'une application en cours d'exécution.

On a identifié quatre approches allant dans ce sens et une cinquième est proposée par Auer, Pölz,et Biffl (2009) :

La première approche consiste à permettre d'augmenter la fonctionnalité d'une application à l'aide de plugiciels (Éclipse, Excell, Mozilla, pour n'en nommer que quelques-uns). Le principal inconvénient de cette approche découle du fait que ces plugiciels doivent être développés de façon externe. Par contre, cette approche permet la modification transparente de l'application en cours d'exécution.

La seconde approche permet de spécifier des expressions et des fonctions définies par l'utilisateur, comme les interrogations en langage SQL d'OLAP, par exemple, ou encore les expressions mathématiques dans Excel. Habituellement, l'utilisateur a un accès limité aux données et aux fonctionnalités. L'avantage de cette approche est également de permettre d'apporter ces modifications pour une application qui est en cours d'exécution.

Pour la troisième approche, les applications peuvent offrir des langages et des cadriciels de scripts (VBA pour MS Office, AppleScript pour MacOS) avec un accès limité aux fonctionnalités disponibles dans l'application par le biais d'une IPA.

La quatrième approche découle du fait que les logiciels libres, en exposant leur code source, permettent aux utilisateurs programmeurs de l'examiner et de le modifier au besoin. Avec cette approche, l'application doit être relancée après avoir été modifiée.

Avec l'approche proposée par Auer, Pölz et Biffl (2009), qui se situe entre les approches 2 et 3 précédentes, l'IPA de l'outil est rendue complètement accessible. Cette approche n'offre que la possibilité de modifier les aspects de flexibilité prévus à cette fin. De plus, les modifications sont apportées à l'application en cours d'exécution. Elle est inédite, du fait de son utilisation du langage de modélisation UML pour représenter les comportements des applications.

Une autre façon de faciliter l'EUD est de développer des environnements multimodaux qui offrent plusieurs modes visuels de programmation à l'utilisateur. Ces modes, qui peuvent être présentés simultanément, contribuent à accroître l'efficacité de programmation. Il est possible d'offrir et de synchroniser plusieurs modes visuels d'assistants à la programmation (Bellynck, 2000). Ces modes visuels peuvent être graphiques ou textuels, d'autres permettant la programmation par l'exemple ou encore par voie de démonstration. Il est nécessaire de s'assurer que ces modes visuels sont synchronisés lorsqu'on les utilise en parallèle.

Il se pourrait fort bien que certains utilisateurs possèdent les connaissances requises pour la programmation à l'aide d'un langage informatique. Toutefois, dans le but de leur faciliter la tâche ou pour ceux qui ne possèdent pas de telles connaissances, il convient de substituer la programmation visuelle à la programmation effectuée à l'aide d'un tel langage. Plusieurs taxonomies des langages de programmation visuelle ont été réalisées (Price, Baeker et Small, 1993 ; Myers B., 1990). Myers rapporte qu'à l'aide de la programmation visuelle, des utilisateurs non programmeurs sont parvenus à développer des applications très complexes sans connaissances informatiques particulières.

Grâce à ces innovations, on peut passer des applications faciles d'utilisation aux applications faciles à développer (Beringer, Fischer, Mussio, Myers, Patern et Ruyter, 2008). Un des défis lié au développement de ces environnements est de permettre aux utilisateurs qui ont peu de connaissances en informatique de développer ou de modifier leurs propres applications. Toutefois, ce défi ne vise pas uniquement les utilisateurs, mais bien les analystes de processus et les experts du domaine de l'application, ainsi que les ergonomes qui peuvent contribuer au travail des utilisateurs en les aidant à définir les contenus et les processus des IHM. Il faut proposer aux utilisateurs des représentations plus intuitives des concepts et des éléments logiciels d'interaction. Sous cet aspect, l'EDD se veut un environnement hybride qui propose des mécanismes de composition intuitifs par la programmation visuelle et la manipulation directe aux utilisateurs non

programmeurs, et un environnement de développement complet dont peuvent bénéficier les utilisateurs ayant les connaissances informatiques nécessaires pour effectuer la programmation.

Les besoins associés au débogage sont nés du paradigme de la programmation par les utilisateurs non programmeurs. Ces besoins ont vu le jour avec les chiffriers et les erreurs contenues dans les innombrables applications financières qui y ont été développées (Panko, 1998). De multiples stratégies ont été proposées pour aider au débogage des applications (Grigoreanu et Burnett, Design Implications for End-User Debugging Tools: A Strategy-Based View, 2009; Grigoreanu, Burnett et Robertson, A Strategy-Centric Approach to the Design of End-User Debugging Tools, 2010).

Plusieurs méthodes de visualisation de localisation des erreurs ont été recensées (Ruthruff, Creswick et Burnett, 2003; Jones et Harrold, 2005). En ce qui concerne les assistants de débogage actifs à l'intérieur d'une application en cours d'exécution (souvent codés à même l'application), ils servent à repérer les erreurs en temps réel et doivent être conçus de façon à aviser l'utilisateur dès que des erreurs sont repérées. Il existe deux types d'interruptions utilisées lors de notification (Robertson, Prabhakararao et Burnett, 2004). L'interruption négociée sert à aviser l'utilisateur sans l'obliger à agir sur-le-champ. Par contre, l'interruption immédiate oblige l'utilisateur à poser immédiatement une action. On pensera aux dialogues affichés que l'utilisateur doit fermer avant de poursuivre ses tâches lorsque survient une erreur.

Certaines études proposent aussi des mécanismes d'évaluation automatique de la qualité des logiciels qui peuvent servir d'assistants au débogage dans les environnements d'EUD. Bouktif, Sahraoui et Giuliano (2006) proposent une approche de jumelage des modèles de qualité offrant des expertises différentes. Cette approche a été appliquée au développement d'un modèle de qualité dont l'expertise consiste à évaluer la stabilité des logiciels orientés objets.

Enfin, notons les raisons pour lesquelles le débogage reste difficile (Brandt, Guo et Lewenstein, 2009). Un projet peut comporter plusieurs technologies et plusieurs langages de programmation ou de représentations visuelles. Le code source est souvent mal encapsulé, c'est-à-dire qu'il existe trop de relations entre les éléments logiques de l'application. De plus, l'utilisateur refuse souvent d'investir le temps nécessaire à l'apprentissage d'outils puissants, mais complexes, utilisés comme assistants au débogage. Connaissant bien ces difficultés, l'utilisateur expérimenté se

montrera souvent proactif en planifiant l'incorporation dans son code d'éléments de débogage, comme les instructions d'affichage de l'état de l'application en prévision d'éventuelles erreurs.

1.4 Langages déclaratifs

On a de plus en plus tendance à utiliser des langages déclaratifs pour la conception d'IHM. Il suffit de penser aux formulaires utilisés par Netbeans, ou encore au langage XUL, à l'aide duquel l'IHM de Mozilla (Firefox) et de plusieurs des plugiciels Mozilla ont été développés. L'EDD utilise son propre langage déclaratif défini à l'aide du cadriciel de modélisation CME, aussi connu sous le sigle EMF (*Eclipse Modeling Framework*). Comme pour XUL, le langage résultant est fondé sur XML et permet de définir les vues graphiques et les comportements de l'IHM. Le paradigme de la conception fondée sur les modèles utilisant des langages déclaratifs a été introduit durant les années 1990 (Szekely, Sukaviriya et Castells, 1995). On décrivait déjà à cette époque plusieurs de ses avantages :

- Il offre de puissants outils durant la conception et l'exécution. Le langage déclaratif permet de définir un modèle qui constitue une représentation commune pouvant être comprise par l'environnement de développement et d'exécution, ce qui facilite la création d'outils qui comprennent le modèle et qui aident à sa manipulation.
- Il favorise la cohérence et la réutilisabilité. Toutes les parties du système se partageant les mêmes connaissances du modèle, il est possible d'assurer la cohérence de ses interfaces et de permettre la réutilisation de ses sous-ensembles.
- Il permet une conception hâtive. Le modèle incite les concepteurs à exprimer les raisons sous-jacentes à leurs décisions en matière de conception.
- Il favorise une conception itérative. Le modèle peut être modifié de façon progressive.

Depuis, plusieurs langages déclaratifs ont été proposés (da Silva, 2001), entre autres UIML (langage de balisage de l'interface utilisateur), un langage déclaratif fondé sur XML (langage de balisage extensible) qui permet de créer des spécifications d'IHM indépendantes de la plateforme d'exécution (Abrams, Phanouriou et Batongbacal, 1999), UsiXML, un langage déclaratif pour la spécification des IHM sensibles à leur contexte d'exécution (Limbourg, Vanderdonckt et Michotte, 2004) et GrafiXML, un langage déclaratif fondé sur UsiXML qui permet de spécifier

des IHM à la fois sensibles à leur contexte d'exécution et indépendantes de la plateforme d'exécution (Michotte et Vanderdonckt, 2008).

L'EDD utilise son propre langage de définition de l'IHM; nous y reviendrons au chapitre suivant.

1.5 Abstraction, plasticité et flexibilité

L'abstraction est suggérée pour aider au développement d'environnements flexibles et programmables par l'utilisateur. Dans ce sens, l'EDD intègre le cadriciel d'Eclipse qui offre plusieurs mécanismes flexibles faisant usage d'abstraction et les propose pour mieux aider l'utilisateur dans son travail de conception de l'IHM.

L'abstraction concrète (abstraction à partir d'exemples concrets) permet de paramétrer les applications et est utilisée comme technique de programmation par l'utilisateur non programmeur (Balaban, Barzilay et Elhadad, 2002). On distingue deux types d'abstraction concrète, soit l'abstraction par les valeurs et l'abstraction par les structures. L'abstraction par les valeurs permet d'agir sur un ensemble d'éléments dont les valeurs ont des caractéristiques communes. Par exemple, l'utilisateur pourrait vouloir modifier la largeur de tous les objets graphiques dont la couleur de fond est le gris.

L'abstraction par les structures permet d'agir sur un ensemble d'éléments dont les structures ont des caractéristiques communes. On pourrait imaginer, par exemple, que l'utilisateur pourrait vouloir changer la couleur de fond de tous les boutons contenus dans des groupes, sans affecter la couleur des boutons situés à l'extérieur des groupes.

Il y a plusieurs exemples de ces deux types d'abstraction dans les outils de refactorisation proposés par les environnements de développement comme Eclipse (Eclipse.org). Par exemple, l'action de renommer une variable s'applique non seulement partout où la valeur de son nom apparaît, mais une analyse du contexte, ainsi que la structure et les dépendances, permettent de s'assurer qu'il s'agit de la bonne variable et non d'une autre variable du même nom.

La plasticité est une autre caractéristique des IHM flexibles et malléables. Le terme plasticité est inspiré de l'aptitude des matériaux à se déformer de façon non élastique, sans rupture, à savoir conservant leur utilité (Calvary, Coutaz et Thevenin, 2002). Appliqué aux IHM, le terme plasticité décrit l'aptitude de l'IHM à s'adapter à un contexte d'utilisation tout en conservant son utilité.

L’avenue de la mobilité conjuguée à la multiplication des dispositifs d’entrée et sortie a créé de nouveaux besoins en matière d’adaptabilité des IHM aux divers contextes d’utilisation. La plasticité exige de prendre en considération de nouveaux modèles dans la conception des IHM. Entre autres, les modèles de l’environnement, de la plateforme matérielle et de l’évolution de l’IHM deviennent des éléments nécessaires à la conception d’applications plastiques. Bien que la plasticité des IHM soit pertinente dans le contexte des travaux présentés ici, elle ne fait pas partie du sujet traité.

Au-delà des contextes d’utilisation associés aux considérations matérielles, la nature changeante du contexte d’utilisation des applications sur le plan des activités et des tâches connexes exige que ces applications soient flexibles de façon à pouvoir s’y adapter. L’architecture à base de composants est proposée comme solution. Bien que ce type d’architecture soit typiquement associé à la réutilisabilité des composants offerte au programmeur pendant les phases de conception et de développement, il a aussi servi à développer des environnements flexibles sur le plan de l’exécution en permettant aux utilisateurs de les personnaliser par la manipulation des leurs composants (Wulf, Pipek et Won, 2007).

CHAPITRE 2 - OBJECTIFS DE RECHERCHE

2.1 Objectif global

Le présent projet de recherche a pour objectif global de faciliter la participation de l'utilisateur tôt dans le processus de spécification des exigences de l'IHM, en permettant notamment le développement de maquettes interactives où de nouveaux composants peuvent être définis et intégrés dynamiquement à l'IHM et où il peut y avoir interchange de noyau fonctionnel entre un simulateur et le système opérationnel. Ce contexte est particulièrement pertinent pour l'Agence spatiale canadienne, mais il s'applique d'emblée au développement de plusieurs types de systèmes. Le principe consiste essentiellement à offrir une plateforme permettant le prototypage rapide et fidèle des IHM à l'aide de laquelle il peut tester avec des maquettes et valider ces exigences tôt dans le processus de développement. Il va de soi que la possibilité de basculer d'un environnement de simulation à un environnement opérationnel ou à un autre noyau fonctionnel quelconque offre d'autres avantages, par exemple, pour les tests ou la formation. Toutefois, nous nous sommes fixés comme objectif principal de faciliter le prototypage, ce qui a mené à la détermination en priorité des lignes directrices de conception de l'EDD.

L'utilisateur qui est particulièrement visé est l'expert dans le domaine pour lequel une IHM doit être conçue. Il ne s'agit pas nécessairement d'un programmeur, auquel cas il se limitera à la manipulation de certains aspects de l'interactivité. S'il est programmeur, alors la plateforme lui offrira en outre les possibilités habituelles des environnements de programmation.

2.2 Objectifs spécifiques

Pour atteindre l'objectif global exposé plus haut, l'EDD doit posséder les caractéristiques nécessaires pour faciliter la tâche de l'utilisateur dans l'exercice de prototypage d'IHM.

Les objectifs spécifiques de présent projet de recherche correspondent en quelque sorte aux principaux éléments de la spécification de l'EDD. Ces objectifs sont les suivants :

1. Il faut concevoir un environnement complet qui s'applique à tous les aspects de l'assemblage de l'IHM, à savoir la structure du projet d'IHM, l'assemblage graphique, le

couplage au noyau fonctionnel, la logique des interactions et le déploiement du produit qu'est l'IHM.

2. L'IHM doit s'exécuter à même l'environnement de développement et les modifications apportées par l'utilisateur doivent pouvoir être appliquées dynamiquement et immédiatement sans devoir relancer la plateforme (Ruyter et Sluis, 2006). Cette caractéristique, qui favorise le développement rapide du prototype, est celle qui permet à l'EDD de se démarquer des autres environnements de développement et de prototypage d'IHM.
3. Des outils de programmation visuelle doivent être disponibles pour permettre le couplage de l'IHM au noyau fonctionnel et pour permettre le développement de la logique d'interaction par un utilisateur non programmeur (Myers B., 1990), laquelle précise comment les composants graphiques interagissent entre eux et avec le noyau fonctionnel de l'application.
4. L'utilisateur qui possède les connaissances nécessaires doit pouvoir programmer des interfaces et des classes, et les intégrer de façon dynamique dans l'IHM sans devoir relancer son exécution.
5. L'IHM doit être modélisée à l'aide d'un langage de modélisation favorisant l'instrumentation (outils) de l'environnement et la réutilisabilité des parties de l'IHM (Szekely, Sukaviriya et Castells, 1995).
6. L'IHM doit pouvoir être aiguillée de façon dynamique vers différents noyaux fonctionnels utilisant la même IPA. Ces mécanismes d'aiguillage servent d'assises au polymorphisme du noyau fonctionnel. C'est par le biais de cette caractéristique que le cadriciel de l'EDD se distingue des autres.

CHAPITRE 3 - FONDEMENTS ET ARCHITECTURE DE L'EDD

Dans le présent chapitre[1], on décrit l'EDD, son architecture et ses caractéristiques, et on expose comment l'objectif global de conception d'un environnement de prototypage d'IHM complet, flexible et dynamique qui permet à l'utilisateur de participer tôt à la spécification des exigences et à la conception de l'IHM a été atteint.

De façon à préciser les fondements et l'architecture qui ont permis d'atteindre chacun des objectifs spécifiques énoncés en 2.2, on traite en détail des choix qui ont été faits au moment de la conception de l'EDD. Le chapitre suivant traite de la rétroaction obtenue à la suite d'essais d'utilisation de l'EDD.

3.1 Architecture et conception

Les environnements de développement logiciel disponibles sur le marché offrent des outils d'aide au développement d'IHM. Ces outils permettent de manipuler directement des objets graphiques qu'ils rendent accessibles à l'aide d'une palette ou d'autres mécanismes. Ils permettent aussi entre autres de gérer la mise en forme des écrans et génèrent le code source des méthodes de traitement des évènements qui doivent ensuite être complétées par le programmeur. Bien que ces assistants soient très utiles, le développement des IHM exige encore beaucoup de travail et de connaissances en informatique, notamment parce que les objets graphiques sont reliés au noyau fonctionnel par la programmation à même les méthodes de traitement des évènements. Cette façon de faire rend ardue l'utilisation d'une même IHM pour contrôler divers procédés. Elle ne propose pas de mécanismes ou d'architectures consacrés à la définition d'IHM pour laquelle les systèmes à contrôler sont polymorphes ; en effet, il faut une architecture qui ne tient compte que de l'interface du noyau fonctionnel et non pas de ses implémentations. Cette caractéristique ne fait pas défaut chez l'EDD, ce qui le distingue des autres environnements. Il est, bien sûr, possible de coder l'aspect polymorphe du contrôle exercé par l'IHM sur le noyau fonctionnel,

[1] Certains passages de ce chapitre sont fortement inspirés de la communication de nature industrielle présentée à la conférence IHM 2009 de Grenoble (Jodoin et Desmarais, Environnement dynamique de développement (EDD) pour le prototypage rapide d'interfaces graphiques, 2009).

mais le fait que la caractéristique susmentionnée soit offerte par l'EDD rend ce type de construction beaucoup plus intuitif.

Enfin, l'approche « programmation des méthodes de traitement des événements » exige beaucoup de connaissances en programmation et les comportements auxquels elle mène ne sont pas d'emblée réutilisables. Les comportements de l'IHM sont représentés dans l'EDD par des objets qui peuvent être connectés facilement à tout événement; il suffit de les associer graphiquement à celui-ci.

Le fait de déplacer plusieurs aspects de la construction d'IHM de la programmation traditionnelle à la manipulation directe et à la programmation visuelle permet aux utilisateurs experts de participer à la conception et au développement des IHM (Myers B., 1990). Les utilisateurs peuvent, grâce au prototypage rapide, adapter avec précision l'IHM aux tâches qu'ils doivent accomplir.

La communication entre l'utilisateur, les constituants graphiques et les constituants logiques s'établit à partir de la levée d'événements qui provoquent l'activation des comportements. Ce mode de communication est très compatible avec le développement d'IHM par manipulation directe et programmation visuelle (Myers, Hudson et Pausch, 2000). En effet, il est facile d'établir la relation entre les actions de l'utilisateur et la réaction du système. De plus, les comportements définis sont réutilisables et peuvent être associés à plusieurs événements.

L'utilisateur n'a qu'à définir les comportements de l'IHM en reliant graphiquement les événements qui peuvent être levés par les composants graphiques aux diverses fonctionnalités du noyau fonctionnel. L'EDD offre des assistants qui utilisent la réflexivité de Java pour proposer à l'utilisateur les méthodes de l'interface du noyau fonctionnel qu'il peut brancher aux événements levés par les composants graphiques.

L'aspect dynamique des composants et des comportements est la caractéristique qui permet vraiment à l'EDD de se démarquer des autres environnements de développement d'IHM. Les environnements, comme Microsoft Visual Studio et .Net, ou encore NetBeans de Sun Micro Systems ou Eclipse, proposent une approche en trois étapes : programmation, compilation et exécution. L'approche proposée ici est différente, car toute modification apportée à l'IHM est

applicable immédiatement et peut-être testée directement dans l'environnement sans avoir à relancer l'EDD. En effet, celui-ci est à la fois l'éditeur d'IHM visuel, l'environnement de développement Java et l'IHM même. Les modifications effectuées par manipulation directe à partir de l'éditeur visuel de l'IHM et les modifications apportées au code source Java contribuant au contrôle de l'IHM sont appliquées à l'IHM de façon dynamique aussitôt qu'elle sont sauvegardées par l'utilisateur.

De plus, l'architecture de cette plateforme est fondée sur le patron modèle-vue-contrôleur (MVC). Dans le contexte de l'EDD, le principal avantage de cette approche découle de la séparation entre les vues, les données et les interactions entre l'IHM et le noyau fonctionnel. En effet, cette séparation facilite l'entretien et l'amélioration de l'EDD sans invalider les vues déjà construites par l'utilisateur ; elle favorise également leur réutilisabilité.

L'EDD est à la fois le noyau et l'environnement de développement de l'IHM. Ses caractéristiques en font un environnement qui est non seulement facile à utiliser, mais aussi un cadriciel à partir duquel il est facile de développer l'IHM. Beringer *et al.* suggèrent que les objectifs visés dans le domaine des interactions homme-machine évoluent dans ce sens (Beringer, Fischer, Mussio, Myers, Patern et Ruyter, 2008).

Il est important de souligner que le noyau fonctionnel est le résultat d'un cycle de développement distinct et que l'EDD a pour objectif d'offrir un environnement de développement d'IHM. Il suffit d'utiliser l'IPA du noyau fonctionnel pour assembler les interactions de l'IHM. Or, il est possible de produire tôt une souche du noyau fonctionnel possédant la même IPA, souche à l'aide de laquelle l'IHM pourra être développée. La souche est remplacée par le vrai noyau fonctionnel, une fois celui-ci disponible.

3.1.1 EDD – Une extension de la plateforme Eclipse

Tel qu'il est illustré à la Figure 3-1, l'EDD est constituée de plugiciels qui complémentent l'atelier de développement Java d'Eclipse en offrant différents points d'extension. L'EDD offre, entre autres, un assistant de création de projet, un éditeur central de la configuration de l'IHM, un éditeur graphique de création des vues graphiques par manipulation directe, une palette de composants graphiques, des éditeurs de propriétés, des vues de survol (arbre de la page), des

valideurs, des actions pour la fonction couper/coller et pour la création des comportements par programmation visuelle et un point d'extension permettant aux plugiciels externes de fournir des bibliothèques de nouveaux composants graphiques.

Les projets de l'EDD sont assemblés par l'assistant de création de projets qui crée les répertoires et les fichiers constituant le squelette du projet. Ces répertoires se situent dans l'espace de travail de la plateforme. Le projet qui est créé joue simultanément plusieurs rôles ; en effet, il agit à la fois comme plugiciel Eclipse, comme projet de type *Rich Client Platform* (RCP), comme projet Java et comme projet EDD.

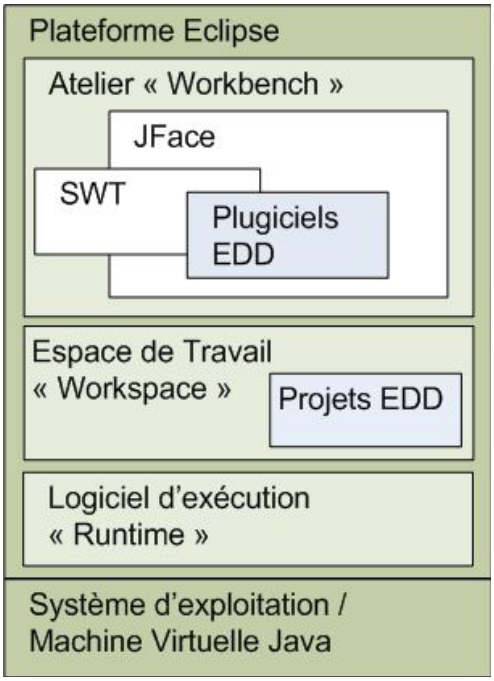


Figure 3-1 : EDD comme extension de l'environnement de développement Java d'Eclipse

Les pages ou vues créées par l'utilisateur sont conservées en mémoire par l'EDD dans des objets faisant partie d'un modèle de type « Cadriciel de modélisation Eclipse » (CME), aussi connu sous le sigle EMF. Le modèle CME est à son tour encodé en format XML au moment de la sauvegarde sur fichier dans le répertoire du projet. Les fichiers, au moment de l'ouverture d'un projet EDD, sont décodés de leur format XML pour reconstituer le modèle CME en mémoire. Le rendu graphique de l'IHM découle de l'interprétation du modèle par l'EDD et des comportements d'affichage de chacun des composants graphiques du modèle. En permettant une interprétation adaptée à l'environnement cible, cette approche offre une flexibilité supplémentaire ; en effet, le

langage de modélisation proposé pourrait être interprétés par d'autres implémentations que celle fournie par l'EDD.

Les pages de l'IHM sont des assemblages graphiques qui peuvent être affichés dans des vues, des dialogues ou encore des fenêtres séparées de l'IHM. On traite plus à fond de ce concept en Pages de l'IHM.

Il s'agit d'une approche qui permet de déployer les IHM dans des espaces interactifs hétérogènes et dynamiques et qui s'apparente à celle de l'ingénierie de l'IHM dirigée par les modèles. Le modèle permet à l'EDD d'interpréter de façon dynamique les comportements et la mise en page de façon à permettre leur utilisation immédiate par l'utilisateur. De plus, cette approche permet d'interpréter le modèle de façon à répondre aux besoins de plateformes hétérogènes; il convient toutefois d'indiquer que cette fonction n'a pas été intégrée à l'EDD. On pourrait, par exemple, déployer l'IHM sur un portable ou sur une page web. L'approche est différente de celles employées par d'autres extensions Eclipse, comme le «Visual Editor », dont le produit qui est le code Java et SWT de l'IHM ne peut être déployé que sur les plateformes compatibles avec le langage Java.

Comme l'EDD intègre plusieurs éléments de la plateforme de développement Eclipse, l'architecture de ces éléments permet de composer avec certains aspects ergonomiques identifiés dans les scénarios de Bass (Bass & Kates, 2001). En voici quelques exemples :

1. **Capacité de modification d'interfaces.** Les perspectives d'Eclipse permettent de choisir les vues qui sont affichées, de les dimensionner et de les positionner. L'EDD offre une perspective par défaut qui peut être modifiée.

2. **Capacité d'accomplissement simultané de plusieurs tâches**. L'environnement offert par Eclipse permet de modifier simultanément plusieurs ressources dans des éditeurs distincts.

3. **Capacité de prédiction de la durée d'une tâche.** Les assistants de compilation et d'exportation d'Eclipse affichent une barre de progrès qui indique le progrès en pourcentage au fur et à mesure de l'accomplissement de la tâche.

4. **Capacité d'accessibilité aux vues.** Eclipse offre les perspectives et un menu principal qui donnent accès à toutes les vues. Par le biais de son éditeur central de la configuration de l'IHM (voir 3.3.3), l'EDD offre l'accès aux pages créées dans le cadre du projet d'IHM.

5. **Capacité d'annulation et de répétition des actions**. Tel qu'il indiqué plus loin dans ce paragraphe, le cadriciel de modélisation Eclipse (CME) permet d'annuler et de répéter des actions.

Tous les modèles de données, incluant ceux des pages et des comportements, ont été conçus à l'aide du cadriciel de modélisation Eclipse (CME), et tous les mécanismes offerts par l'EDD pour modifier ces modèles utilisent des commandes et des piles de commandes du CME qui se branchent automatiquement sur l'environnement pour annuler et répéter les actions connexes. Les modèles CME permettent également de sérialiser et de désérialiser les modèles contenus dans des fichiers à l'aide du langage de modélisation du plugiciel du CME en format XML. Grâce à cette approche, on a pu réaliser l'objectif qu'on s'était fixé de modéliser l'IHM à l'aide d'un langage de modélisation favorisant l'instrumentation (outils) de l'environnement et d'assurer la réutilisabilité des parties de l'IHM (Szekely, Sukaviriya et Castells, 1995).

Les modèles sont composés d'arbres qui représentent les vues et les comportements de l'IHM et il est facile d'en extraire des branches pour les réutiliser. On est ainsi en mesure d'offrir les mécanismes servant à copier et à coller des parties de l'IHM à l'aide d'outils flexibles. L'utilisation de ces outils est décrite plus en détail en 3.3.4. Un survol des modèles CME de l'EDD est présenté en Survol de la conception des modèles CME.

Pour modéliser l'IHM en utilisant le modèle CME, l'EDD se sert de son propre langage déclaratif d'IHM. À la manière de XUL, ce langage est fondé sur XML; il permet de définir les vues graphiques et les comportements de l'IHM. Bien entendu, l'EDD offrant des outils d'assemblage de l'IHM par manipulation directe et par programmation visuelle, l'utilisateur n'a pas à se soucier de la représentation textuelle de l'IHM dans ce langage, celle-ci étant entièrement prise en charge par l'EDD et le CME. D'autre part, ce langage pourrait être utilisé par d'autres applications pour créer des IHM.

L’utilisation du cadriciel Eclipse et du langage Java pour le développement de l’EDD a eu pour résultat de faire de celui-ci un environnement multiplateforme (Meskens, Vermeulen et Luyten, 2008). L’EDD peut être employé sur les mêmes plateformes que celles qu’utilise Eclipse, à savoir Linux, Microsoft Windows et MacOs de Apple; il convient toutefois d’indiquer qu’il n’a pas été testé sur MacOs. Java étant déjà offert en version « mobile » avec Java ME, le consortium Eclipse envisage actuellement de développer une version mobile de son cadriciel, ce qui permettrait d’y adapter l’EDD et de rendre possible son exécution sur des systèmes interactifs mobiles (Ali, 2003). Grâce au langage de modélisation CME utilisé par l’EDD, il serait possible d’offrir des implémentations de composants graphiques adaptées à ces plateformes.

3.1.2 Couplage de l’IHM au noyau fonctionnel

L’un des facteurs de conception de l’EDD consiste à accroître l'outillage de la plateforme Eclipse, tout en profitant des nombreux avantages de son architecture. Les caractéristiques ajoutées offrent des fonctionnalités supplémentaires qui simplifient la conception, qui permettent une application immédiate des modifications apportées aux vues et aux comportements de l’IHM par l’utilisateur et qui permettent également de relier l’IHM à différents noyaux fonctionnels possédant une interface commune.

En effet, en remplaçant le noyau fonctionnel par une souche (remplacement factice) dont l’interface de programmation est la même que celle du noyau fonctionnel réel, il est possible de concevoir l’IHM sans que le noyau fonctionnel soit disponible. La souche peut être remplacée par le noyau fonctionnel, une fois celui-ci disponible. Il est également possible de contrôler à partir d’une même vue des noyaux fonctionnels dont les interfaces de programmation sont identiques. Par exemple, il est possible de contrôler un système robotique et une simulation de celui-ci par la même IHM.

On peut utiliser une ou plusieurs interfaces Java comme modèle(s) de noyau fonctionnel auquel ou auxquels l’utilisateur fera constamment référence pour concevoir l’interface graphique. En procédant ainsi, on s’assure que les architectures de l’IHM et du noyau fonctionnel demeurent compatibles tout au long du processus de conception (Sukaviriya, Foley et Griffith, 1993). De plus, comme il ne s’agit que d’un modèle du noyau fonctionnel, la forme concrète du noyau peut être polymorphe et correspondre à des formes différentes compatibles avec le modèle.

Dans ce sens, l'EDD introduit les concepts de systèmes, de cibles et de contextes d'exécution. Le système représente une interface définissant une IPA dont le but est d'exposer les méthodes avec lesquelles peuvent interagir les objets graphiques. La cible représente une implémentation de système proposée et le contexte définit la cible à laquelle est associé le système. Pour changer le procédé qui est contrôlé, le concepteur ou l'utilisateur peut sélectionner en cours d'exécution un contexte différent de celui qui est utilisé par l'IHM. Cette dernière caractéristique, qui a pour but de promouvoir la réutilisation des interfaces graphiques, permet de relier l'IHM à l'un ou à l'autre des noyaux fonctionnels représentés par les cibles en sélectionnant, même en cours d'exécution de l'IHM, le contexte correspondant.

De façon concrète, tel qu'il est illustré à la Figure 3-2, le système est représenté par une interface Java, la cible par une classe Java qui réalise cette interface et le contexte par la relation entre le système et la cible, c'est-à-dire entre une interface Java et une classe qui entraîne sa réalisation.

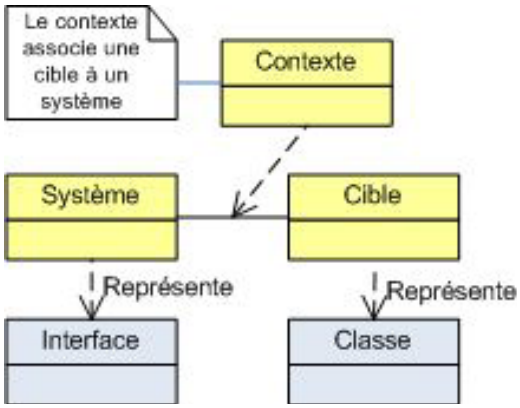


Figure 3-2 : Association tertiaire système-cible-contexte

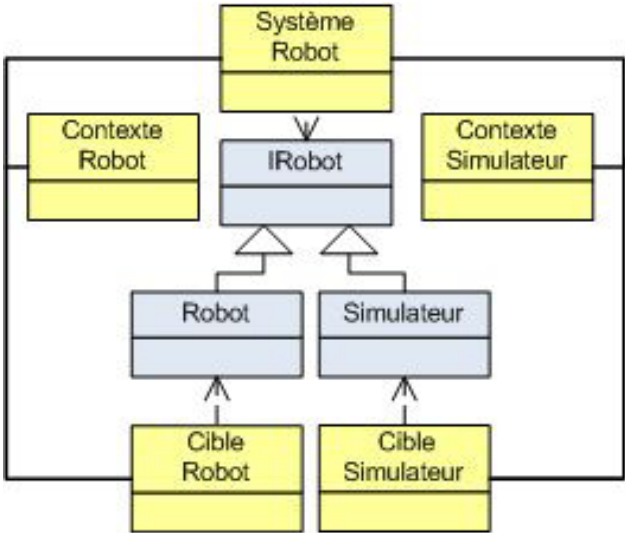


Figure 3-3 : Exemple de relation système-cible-contexte

Un exemple d'utilisation des contextes d'exécution est présenté à la Figure 3-3. Le système `Robot` définit l'interface qui assure le contrôle du robot. Ce système est associé à une interface Java `IRobot` qui en définit l'IPA. Deux réalisations de cette interface Java sont présentées, à savoir les classes `Robot` et `Simulateur`. Ces deux classes, qui sont des réalisations de l'interface `IRobot,` sont associées aux cibles correspondantes `Robot` et `Simulateur`. L'une communique avec un robot et en permet le contrôle, tandis que l'autre remplace le robot par un simulateur. En choisissant le contexte `Robot`, l'utilisateur configure l'IHM pour qu'elle s'interface directement avec le robot, et en choisissant le contexte `Simulateur`, il fait en sorte que l'IHM contrôle un simulateur du robot.

L'EDD permet de changer de façon dynamique le contexte d'exécution à tout moment. Les comportements étant construits à partir de l'IPA du système (interface Java) et non de celle des cibles (réalisation de l'interface), il est possible d'aiguiller les appels de méthodes vers les classes sélectionnées par le contexte.

Pour accéder aux interfaces et aux classes qui font partie de l'IPA du noyau fonctionnel, l'utilisateur doit intégrer le noyau fonctionnel à son IHM en l'ajoutant aux dépendances de son projet, que celui-ci soit défini dans un fichier d'archive Java (fichier JAR) ou dans un plugiciel.

Le dialogue de sélection des classes lui permettra alors de sélectionner les interfaces et les classes à partir d'une liste englobant tout ce qui fait partie des dépendances du projet.

3.1.3 Composants graphiques et non graphiques actifs

L'EDD a comme caractéristique importante de permettre au développeur de concevoir un objet graphique dynamique rendu disponible à l'utilisateur par le biais d'un point d'extension Eclipse défini à cette fin. Une fois le nouveau composant détecté, il est ajouté à la palette d'objets graphiques; son interface programmatique devient alors publique et directement visible au concepteur de l'IHM. Celui-ci peut alors prendre connaissance des événements qu'il peut gérer et associer le composant à d'autres objets dans l'IHM en développement par manipulation directe, par la saisie de propriétés ou par l'utilisation de méthodes propres au composant. Il est aussi possible de basculer l'environnement dans le mode exécution et de valider d'emblée le comportement du nouveau composant graphique.

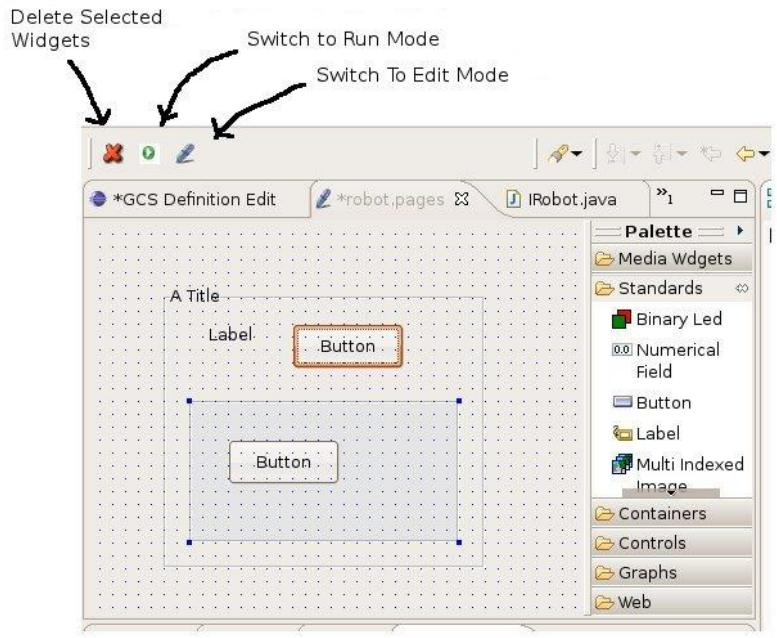


Figure 3-4 : Éditeur de pages

L'éditeur de pages est présenté à la Figure 3-4. On peut apercevoir le canevas sur lequel est assemblée la page à partir de composants graphiques, un quadrillage servant de guide pour l'ajustement de leur emplacement et de leurs dimensions, les outils permettant d'effacer les composants sélectionnés et de faire basculer l'éditeur du mode édition au mode exécution, et la palette de composants graphiques.

Les composants offerts par défaut par l'EDD sont présentés à l'annexe 1. La plupart d'entre eux sont des composants graphiques. Toutefois, l'IPA des composants, définie par l'interface Java `GCSWidget,` permet de préciser si un composant doit s'afficher ou non en mode exécution. En mode édition, les composants non graphiques sont représentés par une étiquette portant leur nom et leur type, de façon à permettre à l'utilisateur de le repérer aux fins de sélection. Ces composants ont les mêmes caractéristiques que les composants graphiques, c'est-à-dire qu'ils peuvent lever des évènements et être configurés selon leurs propriétés. Par contre, on ne peut pas les dessiner. L'EDD offre par défaut deux composants de ce type : la minuterie et la connexion à une base de données.

En effet, c'est grâce à l'interface Java `GCSWidget` si les composants peuvent être implémentés et s'il est possible pour l'EDD de les intégrer et d'interagir avec eux. Cette interface permet de créer les bibliothèques de composants réutilisables.

Un tableau des méthodes de l'interface `GCSWidget` réalisées par les composants est présenté à l'annexe 2, où est décrite l'utilité de chacune d'elles.

3.1.4 Programmation visuelle

La programmation visuelle est l'un des objectifs visés par le présent projet. Elle doit permettre à l'utilisateur non programmeur de développer certains aspects des interactions dans l'IHM.

Le mode de programmation visuelle proposé est accessible par le biais de l'éditeur graphique de pages qui s'accompagne d'un survol du modèle de la page. On y retrouve l'arbre des composants graphiques (et non graphiques) et la liste des événements qu'ils lèvent. Le menu contextuel des évènements offre plusieurs choix liés à des actions pouvant leur être rattachées. Par exemple, il pourrait s'agir de l'appel d'une méthode propre à un système. Dans ce cas, l'utilisateur choisit un système et l'EDD, grâce à la réflexivité de Java, procède à l'inspection de l'interface Java pour révéler ses méthodes et leurs signatures. Une fois la méthode sélectionnée, ses paramètres sont ajoutés à la vue en survol et une action devra être sélectionnée pour chacune de façon à lui attribuer une valeur. L'utilisateur pourra alors choisir entre la valeur d'une propriété, d'une constante ou d'une conversion numérique ou faire appel à la méthode d'un système pour attribuer une valeur au paramètre.

Les actions qui sont rattachées aux évènements sont accompagnées de résolveurs. Les résolveurs sont les objets qui exécutent l'action. On décrit plus en détail les résolveurs et leurs actions en Survol de la conception des modèles CME, « Survol de la conception des modèles du CME ».

Les composants qui offrent une ou des interfaces Java à partir de l'IPA des composants sont considérés comme des systèmes par l'EDD. Les méthodes propres au composant peuvent alors être rattachées aux comportements définis par l'utilisateur.

Plusieurs éléments de l'IHM peuvent lever des évènements auxquels des actions peuvent être rattachées :

1. Les composants des pages de l'IHM. Le tableau A-1 présente une liste des évènements levés par les composants offerts par défaut par l'EDD.

2. L'EDD même. L'EDD lève des événements dès qu'il modifie son état d'exécution. Ces états sont les suivants : en initialisation, en démarrage, en exécution, en arrêt, en pause et en retour d'exécution.

3. Les pages de l'IHM. L'ouverture et la fermeture d'une page lèvent des évènements. Il est possible de lever deux évènements lorsqu'une page est fermée, soit « page fermée, opération acceptée », soit « page fermée, opération annulée ». Ces deux évènements sont utiles lorsque la page sert de dialogue. Si la page se résume à une simple vue, l'évènement « page fermée, opération acceptée » est levé dès que la page est fermée.

4. Les systèmes qui héritent de la classe `GCSEventSourceWithActionMap` (une classe qui offre la structure permettant d'associer les actions aux évènements) donnent lieu à la déclaration d'une liste statique des évènements qu'ils peuvent lever et auxquels l'utilisateur peut associer des actions.

L'utilisateur a accès aux évènements levés par l'EDD, par les systèmes et par les pages du survol de l'éditeur central de la configuration de l'IHM dont il est question en 3.3.3. Il a accès aux évènements levés par les composants d'une page du survol de l'éditeur de pages.

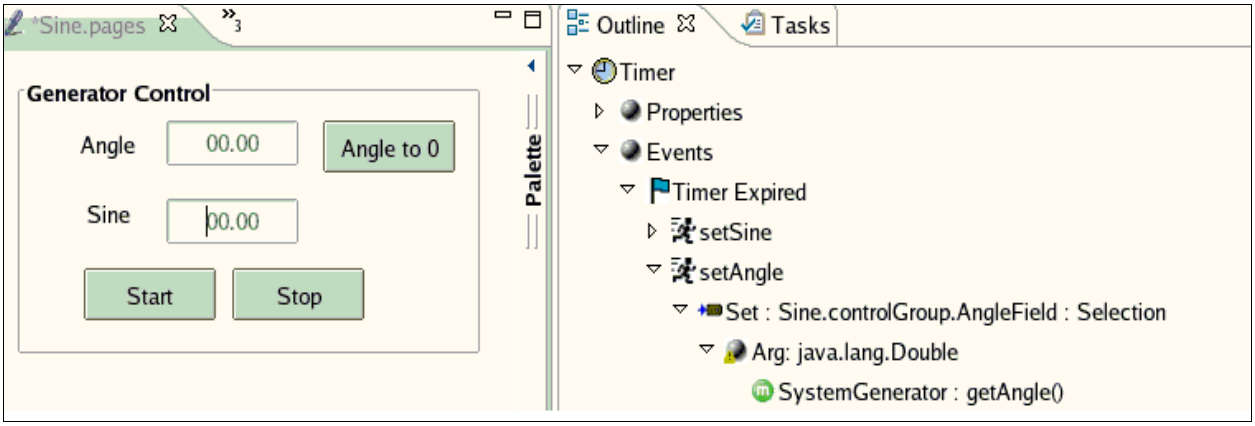


Figure 3-5 : Comportement en construction par la programmation visuelle

La Figure 3-5 illustre un comportement en construction. L'exemple proposé consiste en une IHM très simple de contrôle d'un générateur de signal sinusoïdal, à savoir le noyau fonctionnel. La page en construction permet de démarrer et d'arrêter le simulateur, et de remettre le signal à zéro. Elle affiche la valeur du signal et la valeur de son sinus.

Le comportement démontré est celui de l'affichage du signal sinusoïdal sur la page. La page comporte aussi une minuterie qui lève périodiquement l'évènement « `Timer Expired` ». L'utilisateur rattache à cet évènement l'attribution d'une valeur à une propriété d'un composant. Il choisit, à l'aide d'un dialogue conçu à cette fin, le composant (un champ numérique) et la propriété (sa valeur) à laquelle il veut attribuer une valeur. Cette action est ajoutée à l'arbre avec son paramètre d'entrée. Pour attribuer une valeur à ce paramètre, il y ajoute un appel de méthode. Il spécifie le système et la méthode à appeler. Il s'agit ici de la méthode `getAngle` du système `SystemGenerator`. L'assemblage du comportement est complété et la valeur du signal sera rafraîchie périodiquement par la suite.

Dans ce mode de programmation visuelle, les comportements de l'IHM doivent être assemblés de haut en bas pour préciser les paramètres auxquels il faut attribuer des valeurs. Toutefois, ces comportements doivent être exécutés de bas en haut pour attribuer les valeurs aux paramètres des actions avant leur exécution.

Ce mode d'assemblage peut toutefois être laborieux, si l'arbre de comportement devient trop profond. La Figure 3-6 propose un mode d'interaction visuelle de rechange.

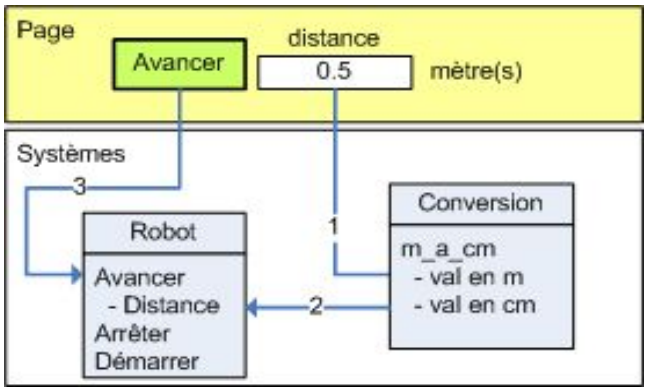


Figure 3-6 : Alternative au mode de programmation visuelle de l'EDD

L'exemple proposé ici consiste en une IHM utilisée pour contrôler un robot. Le comportement illustré permet de spécifier une distance en mètres et de lancer la commande au robot d'avancer de cette distance. Le robot accepte la distance en centimètres.

Ce mode de programmation visuelle est en fait une représentation graphique de l'arbre du comportement équivalente à un diagramme d'interaction UML. L'utilisateur utilise le clic de la souris pour associer les éléments du comportement.

L'ordre dans lequel l'assemblage est effectué a peu d'importance. Ce qui est illustré n'est qu'un des 6 scénarios d'assemblage possibles. Dans ce scénario, la valeur de la distance entrée par l'utilisateur a fait d'abord l'objet d'une conversion de mètres en centimètres. Le résultat de la conversion est ensuite attribué au paramètre `distance` de la méthode `avancer` du système `Robot`. Finalement, l'évènement « bouton pressé » du bouton `Avancer` est associé à l'exécution de la méthode `avancer` du système `Robot.`

Ce mode de programmation visuelle n'a pas été implanté dans l'EDD, mais il fait partie des futures améliorations envisagées, car il comporte plusieurs avantages sur le mode programmation visuelle en arbre couramment offert :

1. L'utilisateur peut placer tous les intervenants du comportement dans l'éditeur de façon à mieux visualiser le problème. Ce mode de visualisation est plus intuitif.

2. L’ordre d’assemblage n’a pas d’importance. Il faut simplement que l’assemblage du comportement soit complet et correct en fin d’exercice. L’utilisateur peut découvrir le comportement à mesure qu’il le construit.

3. Le paradigme n’est pas directement associé aux notions de programmation et d’appel de méthodes. C’est une présentation plus naturelle de la séquence des intéractions.

4. L’interaction est plus rapide, car l’utilisateur sélectionne les intervenants et les associe à l’aide de la souris.

3.1.5 Programmation Java dynamique

Un autre objectif visé consiste à permettre la programmation dynamique d'interfaces et de classes Java qui soient automatiquement détectées, compilées et chargées par la machine virtuelle Java durant l'assemblage, sans devoir redémarrer l'EDD. De cette façon, il devient possible de faire immédiatement usage de ces interfaces et classes Java en les reliant à l'interface graphique. Cette fonctionnalité offre au concepteur la possibilité de développer un système complexe, tout en lui permettant de le brancher à l’interface graphique sans devoir relancer la plateforme.

À cette fin, l’EDD est muni d’un chargeur de classe dynamique qui permet le chargement des classes directement à partir des fichiers (fichiers dont le nom se termine par l’extension « .class ») situés dans le répertoire du projet EDD. Il permet aussi le chargement des classes accessibles par le chemin de classes défini par le projet même. Cette caractéristique n’est pas normalement offerte par la plateforme Eclipse, laquelle exige d’être lancée avec un nouveau chemin de classes défini par la configuration du projet.

Le chargeur de classe dynamique est dérivé du chargeur de classe d’un plugiciel de l’EDD qui comporte des dépendances à tous les autres plugiciels de l’EDD. De cette façon, il lui est possible de localiser toute classe référée par le code de l’utilisateur ou par l’EDD même. En outre, cette technique donne à l’utilisateur accès à l’IPA de programmation de l’EDD et permet de remplacer les classes déjà chargées par la machine virtuelle par des versions modifiées sans avoir à redémarrer l’environnement (Gregersen, Simon et Jorgensen, Towards a Dynamic-Update-Enabled JVM, 2009).

3.1.6 IPA de programmation de l'EDD

Pour offrir plus de flexibilité encore à l'utilisateur, l'EDD offre une IPA permettant de faire appel à ses fonctionnalités par la programmation. Par exemple, le code de l'utilisateur peut recevoir en référence un composant graphique et en modifier les propriétés. Il peut aussi simuler le lancement d'un événement de ce composant. Ceci pourrait permettre, par exemple, de développer une série de tests automatisés dans lesquels l'utilisateur n'a pas à intervenir. Cette capacité permet aussi d'interagir avec les systèmes ou de contrôler les états de l'EDD.

Non seulement l'EDD offre-t-il son IPA, il rend toutes ses classes accessibles grâce à l'architecture de plugiciels d'Eclipse et à sa gestion des dépendances. Ainsi donc, l'utilisateur est en parfait contrôle de l'EDD. Bien qu'avantageux, cela comporte des risques, car l'utilisateur pourrait créer des états qui pourraient être à l'origine d'erreurs d'exécution.

3.1.7 Pages de l'IHM

Les pages de l'IHM sont assemblées sur des canevas du cadriciel graphique SWT à l'aide de l'éditeur décrit en 3.3.4 à partir desquels l'utilisateur peut créer des vues de l'IHM, des dialogues ou même des fenêtres (grâce aux perspectives flexibles d'Eclipse qui permettent de détacher les vues de l'environnement). Elles sont perçues par l'EDD comme des composants graphiques de l'IHM, mais aussi comme des systèmes qui peuvent lever des évènements : page ouverte, page fermée/opération acceptée, page fermée/opération annulée (voir l'explication en 3.1.4). À ces évènements peuvent se rattacher des actions. De plus, l'EDD offre les actions qui peuvent être associées à tout évènement de l'IHM lui permettant d'ouvrir et de fermer une page.

3.1.8 Validation du développement de l'IHM

Les interactions définies par l'utilisateur sont continuellement validés par l'EDD, qui s'assure que des valeurs ont été attribuées à tous les paramètres et qu'elles sont compatibles avec les types afférents à la méthode appelée. Dès qu'une erreur est détectée, l'EDD marque la méthode d'une icône indiquant une erreur. De cette façon, l'utilisateur peut déterminer qu'il a effectué l'assemblage du comportement correctement.

En ce qui concerne le développement des codes Java par l'utilisateur, l'EDD intègre la plateforme de développement Java d'Eclipse, c'est-à-dire la JDT, qui offre une suite d'outils très

performants pour la validation de la syntaxe et la gestion des dépendances du code source. Ces outils sont activés automatiquement par l'éditeur de code source Java.

Ces assistants de validation permettent d'accroître l'efficacité du développement effectué par l'utilisateur et favorisent l'utilisabilité de l'environnement (Bass et Kates, 2001).

3.1.9 Structure du projet

On décrit au tableau A-4 la structure du projet de l'IHM créé à partir de l'assistant de création de projet offert par l'EDD, lequel assistant est décrit en 3.3.1.

Le projet comporte les répertoires et les fichiers nécessaires au développement du projet de l'IHM, à la gestion des ressources, ainsi qu'à la définition de l'IHM, de ces pages et des fichiers Java développés dans ce contexte.

3.1.10 Survol de la conception des modèles CME de l'EDD

L'EDD est composé de plusieurs plugiciels Eclipse conçus à partir des modèles de données qui ont tous été développés à l'aide du cadriciel de modélisation (CME). À noter que le sigle *GCS* est utilisé pour désigner l'EDD dans les codes sources et que ce sigle est utilisé comme préfixe dans la désignation de certains types. De plus, le nom des plugiciels de l'EDD débute toujours par « *csa.gcs* », pour identifier le projet et sa provenance, et se termine par le nom du plugiciel en question. Les modèles sont définis dans des plugiciels distincts, dont voici la liste :

- Classes communes, `csa.gcs.common`
- Noyau de l'EDD, `csa.gcs.core`
- Pages, `csa.gcs.pages`
- Interactions IHM-noyau, `csa.gcs.systems`
- Classes communes des comportements, `csa.gcs.common.reflect`
- Modèle avancé des comportements, `csa.gcs.reflect`

Ces modèles sont en fait les métamodèles qui définissent le contenu des modèles de l'IHM créés à l'aide de l'EDD.

Le CME offre plusieurs méthodes pour définir les modèles. L'outil le plus intuitif est maintenant l'éditeur graphique de diagrammes offert par ce cadriciel. Lorsque nos travaux ont débuté, la méthode la plus efficace consistait à annoter le squelette de chacune des interfaces du modèle dans son code source et de faire appel au générateur de modèle du CME.

Les diagrammes simplifiés qui sont présentés ici ont été conçus à partir des modèles du CME à l'aide de la refactorisation offerte par l'éditeur graphique de ce cadriciel. Nous nous sommes efforcés de présenter les concepts qui permettent le mieux de comprendre les modèles.

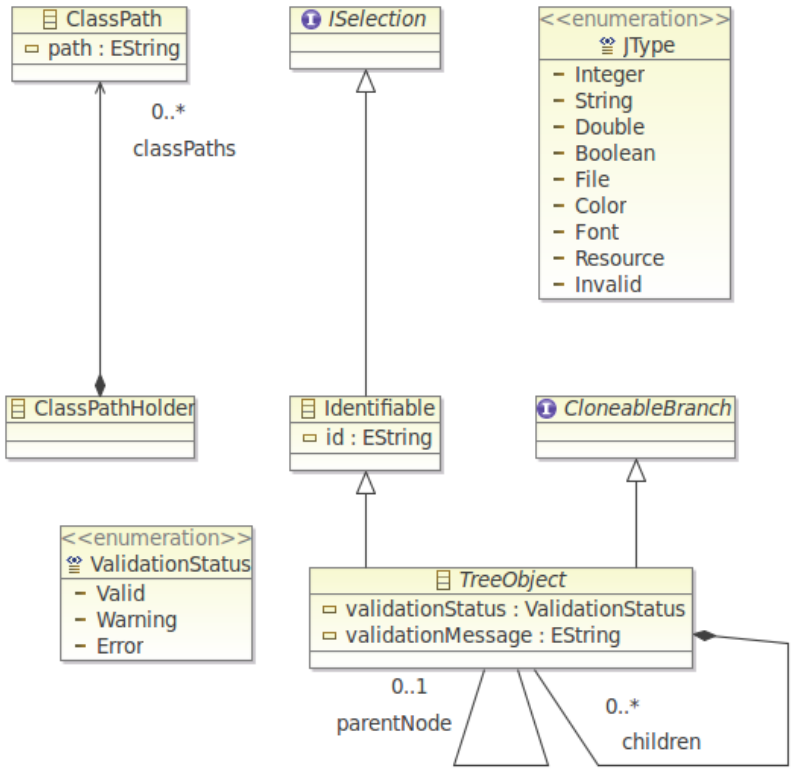


Figure 3-7 : Classes communes, `csa.gcs.common`

On présente à la Figure 3-7 le modèle des classes communes. Ce modèle du CME définit les classes de base desquelles sont dérivées plusieurs classes des autres modèles et quelques classes utilitaires. La classe `TreeObject` implémente l'interface `Identifiable` et l'interface `CloneableBranch`. Elle représente un nœud dans un arbre identifiable par un nom qualifié et possède la structure récursive nécessaire à l'assemblage des branches d'un arbre ; et il permet de

se cloner. L'interface `Identifiable` est un type d'`ISelection` de la plateforme Eclipse permettant à tous les nœuds des arbres définissant les modèles d'IHM d'être sélectionnés. Les nœuds sont marqués de leur état de validation avec leur `ValidationStatus`.

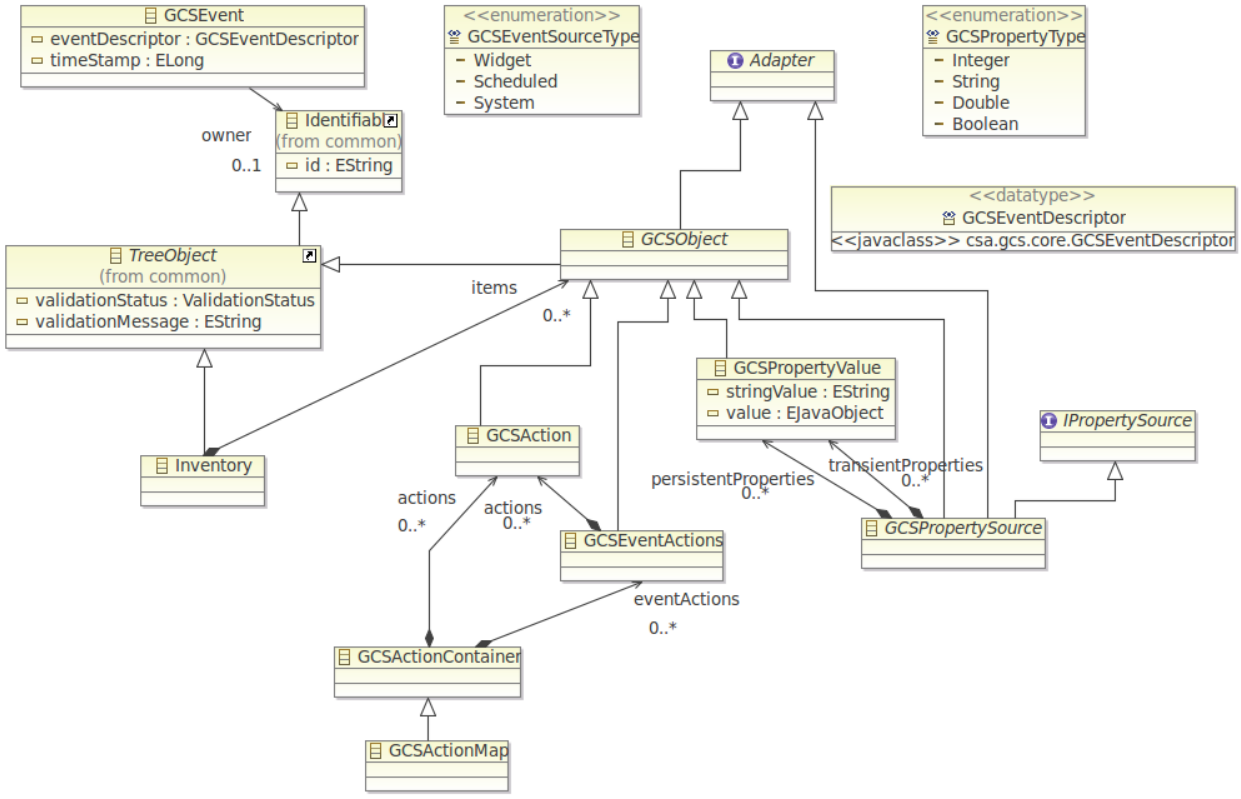


Figure 3-8 : Noyau de l'EDD, `csa.gcs.core`

On présente à la figure 3-8 le modèle définissant les classes du noyau de l'EDD. La classe `GCSObject` définit davantage les objets qui composent l'IHM, lesquels sont des spécialisations du type `TreeObjet` présenté plus haut et de l'interface `Adapter` du CME qui leur offre la possibilité d'écouter le modèle et les annonces de modifications à celui-ci. La classe `GCSPropertySource` définit davantage les objets du type `GCSObject` qui offrent des propriétés qui permettent de réaliser l'interface `IPropertySource.` Celle-ci lui permet de communiquer ses propriétés aux vues d'Eclipse, comme la vue des propriétés. Ce mécanisme est aussi utilisé par l'EDD pour afficher ces propriétés dans les vues de survol. Le modèle spécifie aussi les actions (`GCSAction`) et les événements (`GCSEvent`) associés à l'aide des

dictionnaires d'actions (`GCSActionMap`) qui servent de contenants (`GCSActionContainer`) pour ces associations (`GCSEventAction`).

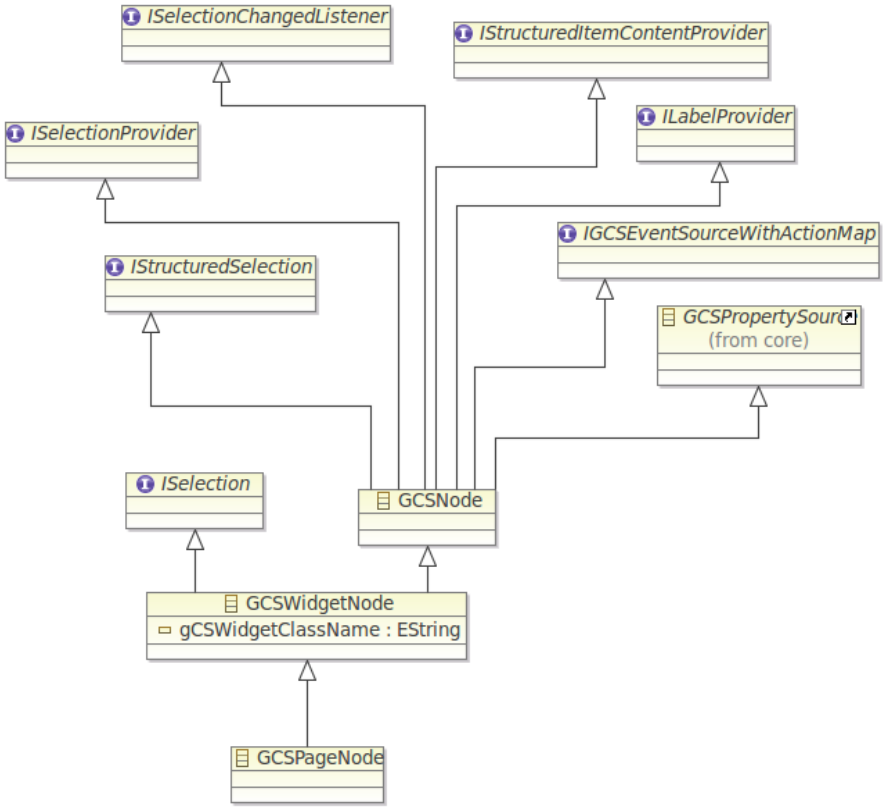


Figure 3-9 : Pages, `csa.gcs.pages`

On présenter à la figure 3-9 le modèle des pages. L'arbre définissant le modèle d'une page est représenté par la classe `GCSPageNode` qui spécialise `TreeObject` présenté plus haut par le biais de la hiérarchie suivante (de bas en haut). GCSPageNode est une spécialisation de `GCSWidgetNode` qui représente les composants graphiques d'une page, incluant la page même. Les composants graphiques (`GCSWidgetNode`)sont des composants de la page (graphique ou non, `GCSNode`) qui offrent des propriétés (GCSPropertySource) et qui sont des objets EDD instrumentés par le CME (`GCSObject`) dérivés de la classe de base des nœuds d'arbre (`TreeObject`) qui sont à leur tour identifiables (`Identifiable`) et peuvent être clonés (`CloneableBranch`). Bien que l'héritage de ce modèle soit profond, il se prête à la réutilisation de ses parties partout dans le modèle.

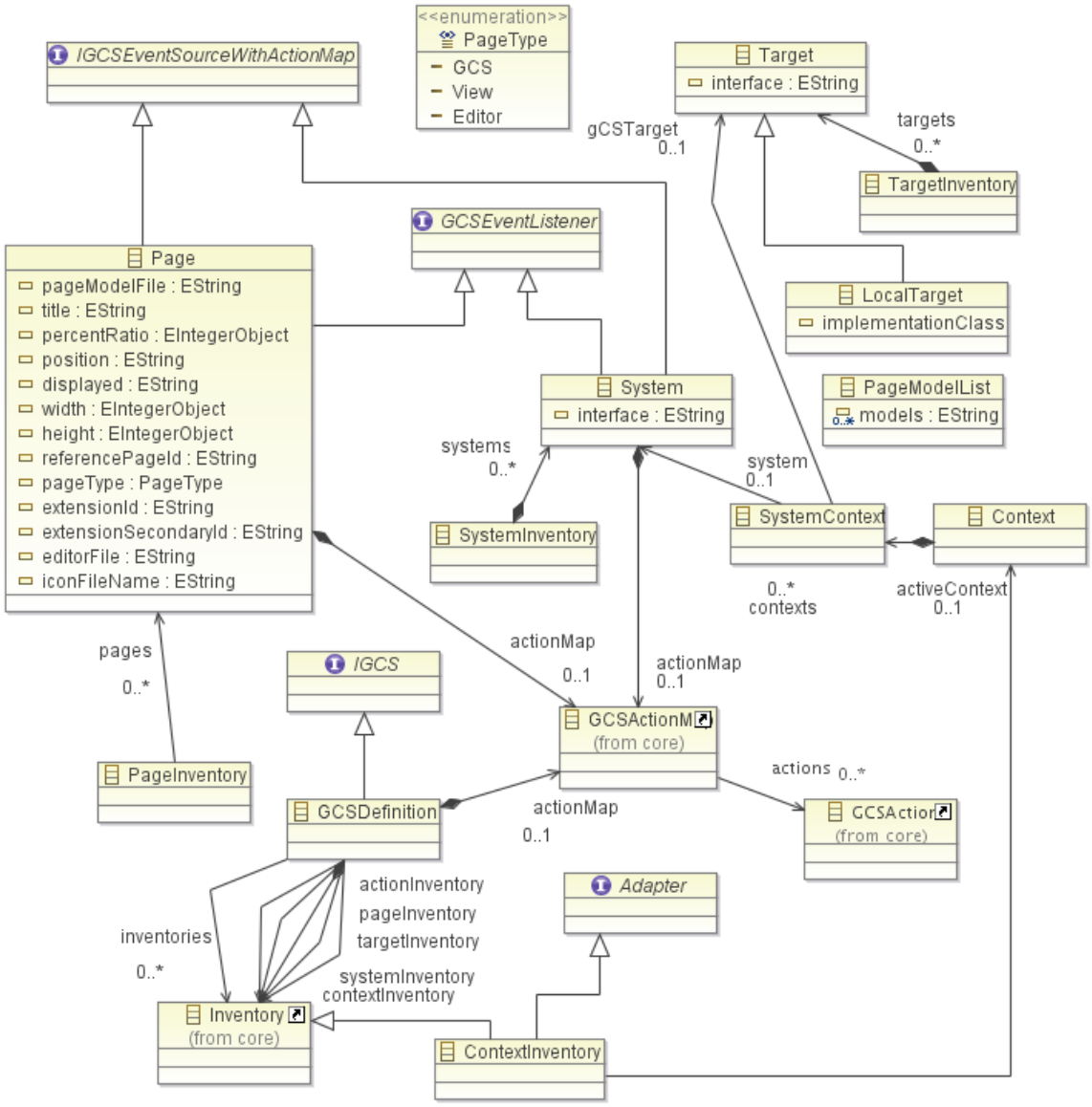


Figure 3-10 : Interactions IHM-noyau, `csa.gcs.systems`

Le modèle des interactions entre l'IHM et le noyau fonctionnel sont présentés à la Figure 3-10. Il définit les concepts de « système, cible et contexte » décrits en détail en 3.1.2. Il convient de souligner quelques points intéressants. La page (`Page`) présentée ici est une coquille permettant d'associer la page modélisée (`GCSPageNode`) au fichier qui renferme sa définition et aux paramètres de sa mise en page. La définition de l'IHM (`GCSDefinition`) sert de contenant aux inventaires des pages, des systèmes et des cibles (`Target`). Les pages et les systèmes

peuvent lever des évènements et contiennent les dictionnaires d'association entre les évènements et les actions qu'ils réalisent grâce aux propriétés de l'interface `IGCSEventSourceWithActionMap`.

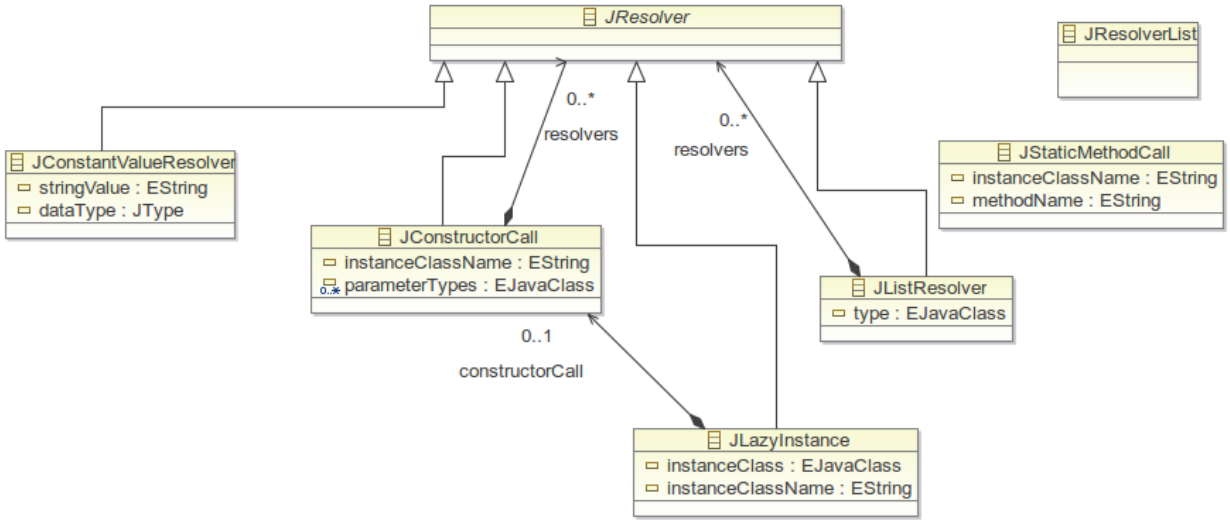


Figure 3-11 : Classes communes des comportements, `csa.gcs.common.reflect`

L'assemblage des interactions et du couplage de l'IHM au noyau fonctionnel est possible à l'aide de la programmation visuelle. C'est grâce à la réflectivité de Java et à la transparence du modèle de l'IHM qu'il est possible de combiner les interactions des éléments de l'IHM entre eux et avec ceux du noyau fonctionnel pendant l'exécution de l'EDD.

La programmation visuelle est un objectif visé par l'EDD qui consiste ici à assembler des appels de méthodes et à les conjuguer à la lecture et à l'écriture des propriétés des composants de l'IHM. Ces interactions sont associées aux évènements levés par les composants de l'IHM, par les systèmes développés à même le projet de l'IHM, par ceux qui font partie du noyau fonctionnel et par ceux qui sont levés par l'EDD. Pour ce faire, il faut disposer d'un modèle de ces interactions; les éléments principaux d'un tel modèle de base sont présentés dans le diagramme de classes UML de la Figure 3-11. Pour bien décrire les mécanismes qui permettent la programmation visuelle, le couplage de l'IHM au noyau fonctionnel et la flexibilité au niveau de l'aiguillage de l'IHM, on explique ici en détail les modèles de classes des comportements.

On retrouve au sommet du diagramme l'interface `JResolver` qui est à la base du modèle d'interaction et que nous appelons « résolveurs ». Cette interface définit la méthode `resolve` qui reçoit en argument un chargeur de classe et retourne un objet qui est la solution. Le chargeur n'est utile qu'aux résolveurs qui instancient des objets. `JLazyInstance` en est un exemple. Il instancie l'objet à sa première utilisation et retourne le même objet les fois suivantes que sa méthode `resolve` est appelée ; il utilise pour ce faire la représentation du constructeur de la classe à instancier offerte par son `JConstructorCall`, un résolveur de constructeur de classe.

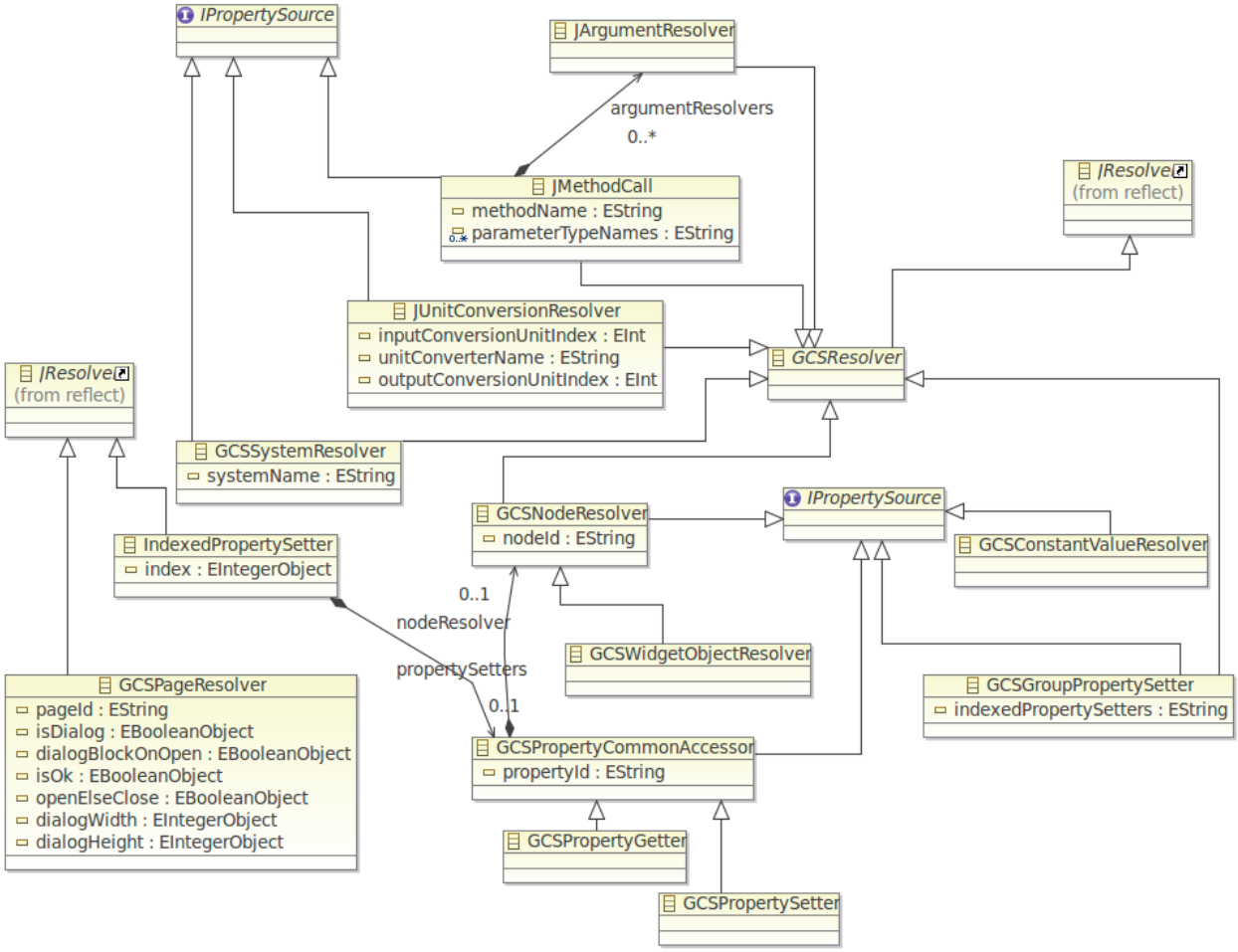


Figure 3-12 : Modèle avancé des comportements, `csa.gcs.reflect`

Le modèle qui en découle est plus raffiné et offre des résolveurs qui permettront d'assembler les comportements. Ce modèle avancé est présenté à la Figure 3-12. En effet, la classe `GCSResolver` est une spécialisation de `JResolver`. Elle hérite de `GCSObject`, une spécialisation de `TreeObject`, du type qui représente les nœuds des arbres dans l'EDD.

`GCSResolver` est la classe qui permet l'assemblage des arbres de comportements. Cette classe est à la fois un résolveur et un noeud dans l'arbre du comportement. Plusieurs résolveurs de ce type sont introduits, entre autres `JMethodCall`, le résolveur qui permet d'effectuer des appels de méthodes. JMethodCall possède une liste de `JArgumentResolver`, les résolveurs de paramètre de méthodes. Chaque résolveur de paramètres possède un quelconque résolveur qui lui fournira l'objet servant de valeur au paramètre. Le modèle possède une structure récursive qui engendre l'assemblage des arbres. Ces arbres de comportements sont exécutés de bas en haut, car il faut attribuer des valeurs aux paramètres avant d'appeler les méthodes. Par contre, ils doivent être assemblés de haut en bas pour exposer les paramètres auxquels on rattache des résolveurs.

Le résolveur d'appels de méthodes (`JMethodCall`) possède aussi un résolveur qui permet de trouver l'objet duquel la méthode doit être appelée. Il s'agira soit du `GCSNodeResolver` (résolveur de nœud), qui retourne un composant de l'IHM, ou encore de `GCSSystemResolver` (résolveur de système), qui retourne l'objet associé à un système. Le résolveur de système doit consulter l'EDD pour connaître le contexte actif et identifier la cible qui possède l'objet (voir système-cible-contexte en 3.1.2). Pendant l'assemblage de l'appel de méthode, la cible n'est pas utilisée; c'est plutôt le système qui est consulté pour connaître l'interface Java sur laquelle porte l'introspection pour en découvrir les méthodes. Par contre, la cible est utilisée au moment de l'exécution de l'appel de méthode, car elle fournit l'implémentation du système qui est associée au contexte. C'est ce mécanisme qui fournit la flexibilité au niveau de l'aiguillage de l'IHM vers la cible à contrôler, un des objectifs visés par ce projet.

Les résolveurs de propriétés `GCSPropertyGetter` et `GCSPropertySetter` permettent d'accéder respectivement en lecture et en écriture aux propriétés des composants de l'IHM.

Enfin, l'EDD offre des résolveurs pour compléter le mécanisme qui permettent d'introduire des constantes et des conversions numériques.

Les comportements ainsi développés sont associés aux évènements et, lorsqu'un évènement est levé, le gestionnaire d'exécution (`SystemManager`) en est avisé et active tour à tour chacun des comportements connexes. Seuls les comportements qui ne sont pas marqués d'une erreur par le système de validation sont exécutés, ce qui permet à l'EDD de poursuivre l'exécution de

l'IHM en évitant d'activer les comportements erronés. Bass et Kates (Bass et Kates, 2001) proposent cette caractéristique de la tolérance aux erreurs pour améliorer l'utilisabilité des IHM.

Tel qu'il est décrit en 3.1.4, les arbres de comportements sont créés à partir des évènements levés par les composants graphiques et non graphiques de l'IHM, ou à partir de ceux levés par les systèmes ou encore de ceux levés par l'EDD à l'aide d'actions (`GCSAction`) rendues disponibles à l'utilisateur par le biais du menu contextuel.

3.2 Génération d'un produit statique

Le projet créé par l'EDD comporte aussi un fichier de produit. Ce fichier permet à l'utilisateur, par le biais de l'assistant de déploiement de produits Eclipse, de déployer l'IHM en un produit complètement autonome dans lequel tous les plugiciels nécessaires à son exécution sont inclus. Dans ce contexte, l'IHM ne s'exécute plus dans un environnement de développement, mais plutôt dans un environnement d'exécution dénudé des outils de l'atelier Eclipse et de ceux de l'EDD. Seuls les plugiciels permettant l'exécution des pages construites à l'aide de l'EDD y sont présents. Ce mode de déploiement n'est utile que lorsque la construction de l'IHM est terminée et que le produit est prêt à être utilisé aux étapes de production ou d'opération, selon l'objectif visé.

3.3 Vues offertes par l'EDD

Nous décrivons ici les différentes vues et différents éditeurs offerts par l'EDD. Ces vues et éditeurs complémentent l'environnement de développement d'Eclipse et permettent d'atteindre l'objectif visé, à savoir de fournir à l'utilisateur tous les outils nécessaires à l'assemblage d'une IHM complète.

Les images présentées ci-dessous sont des instantanés d'écran de l'EDD en cours d'exécution; le sigle GCS (*Generic Control Station*) est utilisé pour désigner l'EDD.

3.3.1 Assistant de création de projet

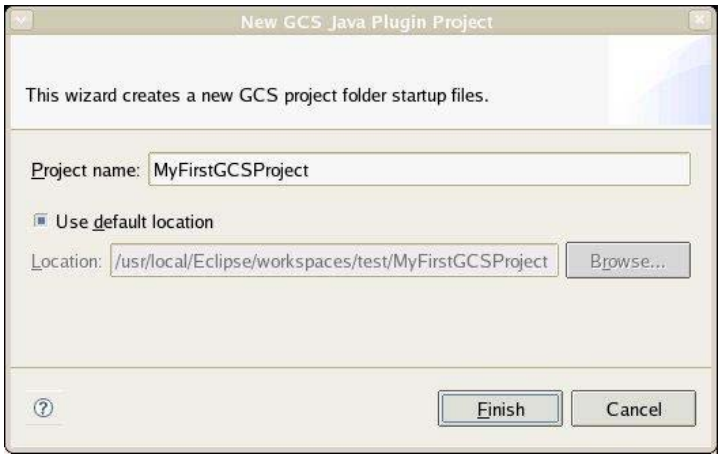


Figure 3-13 : Assistant de création de projet

Le dialogue de l'assistant de création de projet d'IHM de l'EDD est présenté à la figure 3-13. L'utilisateur n'a qu'à entrer le nom du projet et l'assistant crée le projet selon la structure décrite en 3.1.9.

3.3.2 Explorateur de projet

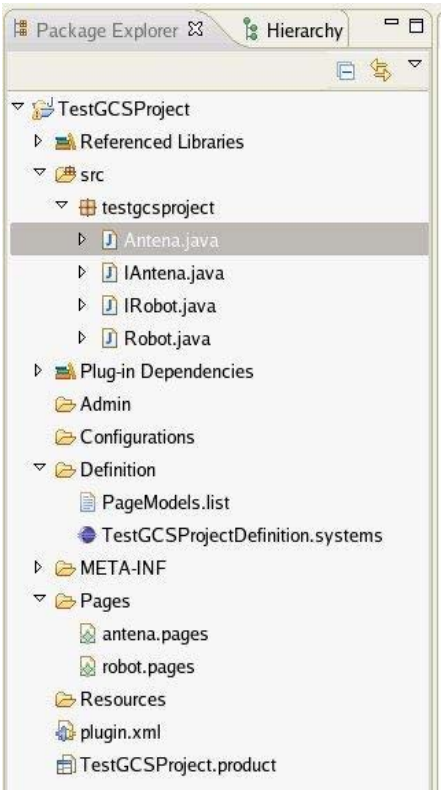


Figure 3-14 : Explorateur de projet

La vue présentée à la Figure 3-14 permet de naviguer dans la structure du projet; il s'agit de celle offerte par la JDT d'Eclipse qui donne accès à tous les éditeurs et assistants de façon contextuelle selon la sélection.

3.3.3 Éditeur central de la configuration de l'IHM

L'éditeur central de la configuration de l'IHM est au cœur du développement du projet de l'IHM. Il présente quatre tableaux qui permettent respectivement de définir les systèmes, les cibles, les contextes et les pages. Il est accompagné d'une vue de survol permettant de visualiser la liste de ces éléments, des évènements que chacun peut lever et d'attacher des comportements aux évènements levés par l'EDD, les systèmes et les pages.

Nous présentons ci-après tour à tour les quatre tableaux de cet éditeur.

Le tableau « Systèmes », qui permet de créer, de modifier et de supprimer des systèmes, est présenté à la Figure 3-15. L'interface (ou la classe) Java du système est sélectionnée à partir du dialogue de sélection offert dans ce but par l'environnement de développement Java d'Eclipse qui est intégré à l'EDD. Ce dialogue donne accès aux classes et aux interfaces Java présentes dans les voies faisant partie des dépendances du projet de l'IHM.

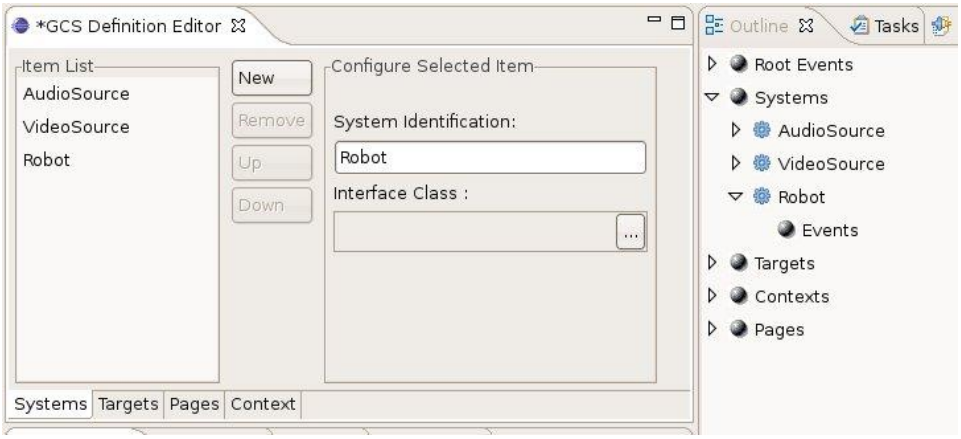


Figure 3-15 : Tableau « Systèmes » de l'éditeur central de la configuration de l'IHM

Le tableau « Cibles » qui permet de créer, de modifier et de supprimer des cibles, est présenté à la Figure 3-16. La classe Java de la cible est aussi sélectionnée à partir du même dialogue de sélection de classes que celui utilisé pour la sélection de l'interface du système. Les interfaces réalisées par la classe sélectionnée sont automatiquement détectées par l'EDD et ajoutées à la définition de la cible. L'utilisateur peut, par la suite, modifier manuellement cette liste à l'aide du dialogue produit à cette fin.

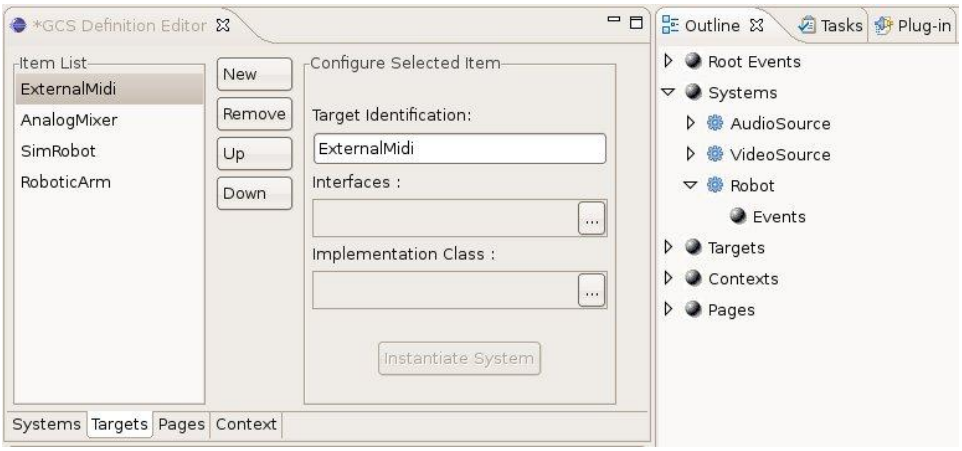


Figure 3-16 : Tableau « Cibles » de l'éditeur central de la configuration de l'IHM

Le tableau « Contextes », qui permet de créer, de modifier et de supprimer des contextes, est présenté à la Figure 3-17. L'utilisateur doit sélectionner la cible correspondant à chaque système dans chaque contexte. Il est aussi possible de préciser dans ce tableau le contexte qui est actif.

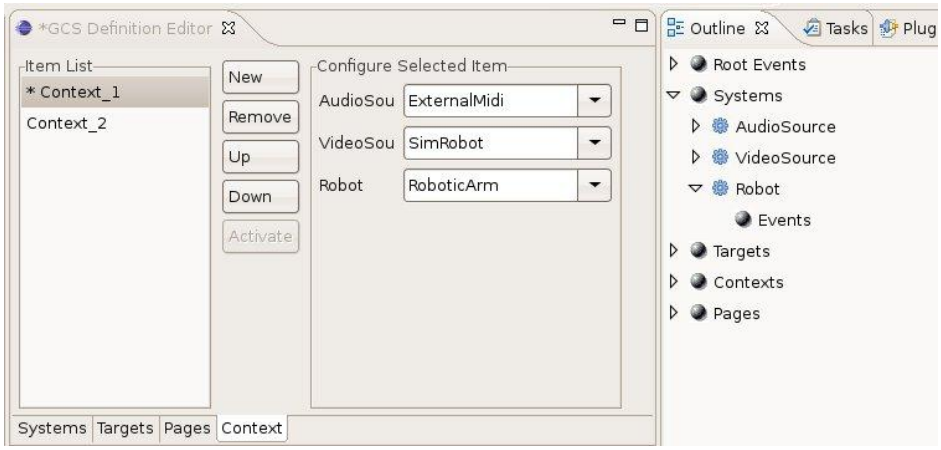


Figure 3-17 : Tableau « Contextes » de l'éditeur central de la configuration de l'IHM

Enfin, le tableau « Pages », qui permet de créer, de modifier et de supprimer des pages de l'IHM, est présenté à la Figure 3-18.

La page créée peut-être associée à un modèle EDD, c'est-à-dire à une page qui sera développée à l'aide de l'éditeur de pages de l'EDD, ou encore à une vue ou à un éditeur fourni par un plugiciel Eclipse externe. Cette caractéristique permet d'intégrer à l'IHM des produits conçus à l'extérieur de l'EDD.

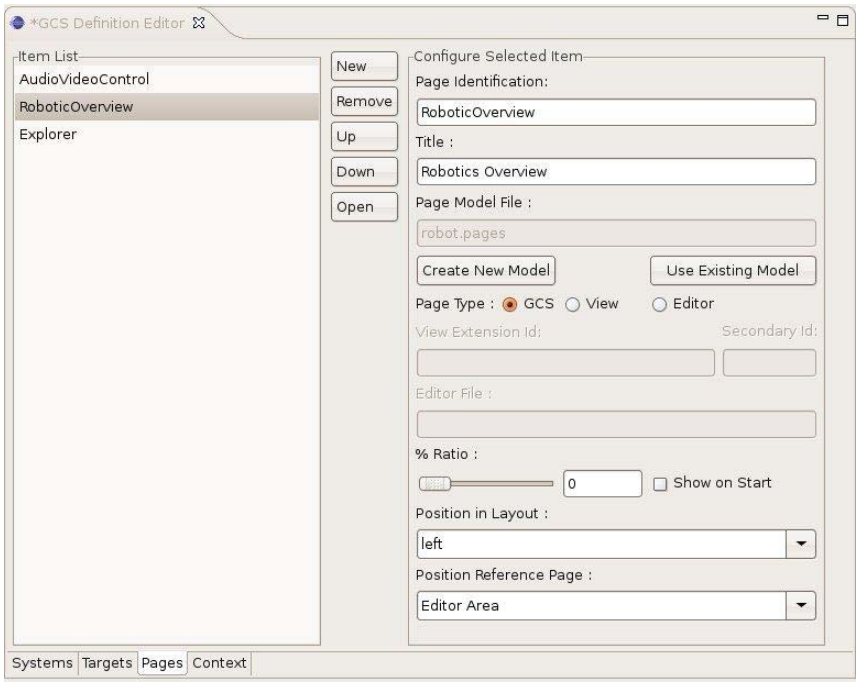


Figure 3-18 : Tableau « Pages » de l'éditeur central de la configuration de l'IHM

3.3.4 Éditeur de pages

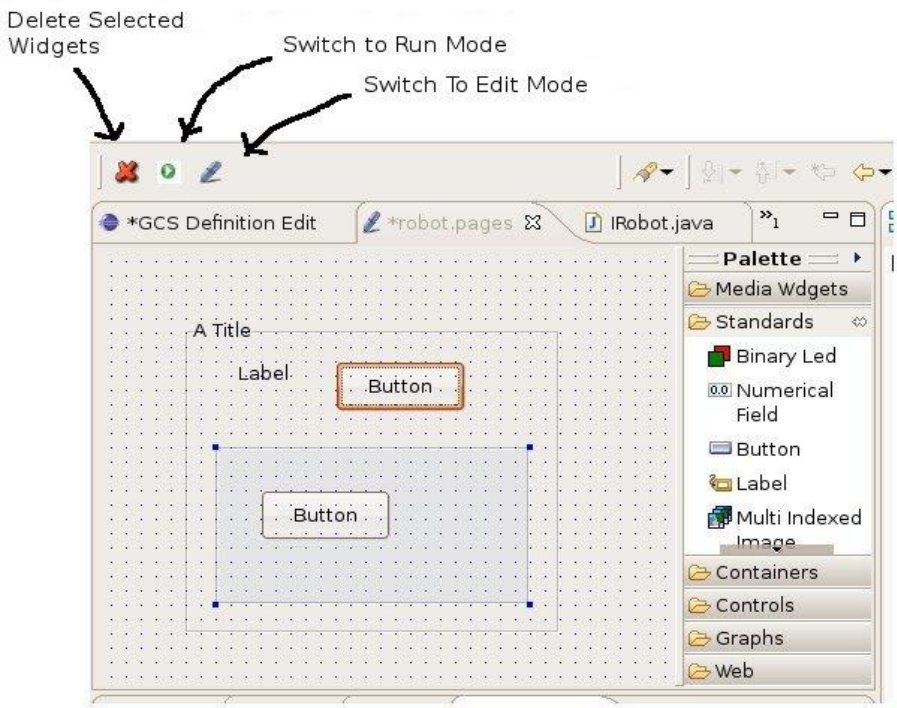


Figure 3-19 : Éditeur de pages en mode édition

L’éditeur de pages en mode édition est présenté à la

Figure 3-19. Il est également présenté à la Figure 3-5, accompagné de la vue de survol du modèle où apparaît un comportement en développement.

Dans ce mode, la palette de composants est active et permet l’ajout de composants au canevas par manipulation directe. À l’aide de la souris, l’utilisateur peut déposer les composants, les déplacer, les sélectionner, les modifier et les supprimer. Il peut aussi modifier les propriétés d’un composant ou de plusieurs d’entre eux par le biais de l’éditeur de propriétés en les sélectionnant sur le canevas.

L’éditeur de pages donne accès à un mécanisme intelligent qui permet de copier/coller un composant ou plusieurs composants d’une même page ou de pages différentes. Ce mécanisme est qualifié d’intelligent parce qu’il procède au clonage des composants, de la structure de l’assemblage et de leurs comportements. Les comportements sont analysés de façon à reproduire et à cloner les interactions entre les composants faisant partie du groupe sélectionné et de façon à préserver la direction des interactions avec les composants à l’extérieur du groupe.

En permettant de réutiliser l’information contenue dans le modèle, ce mécanisme est très utile pour accélérer le développement d’une IHM (Bass et Kates, 2001).

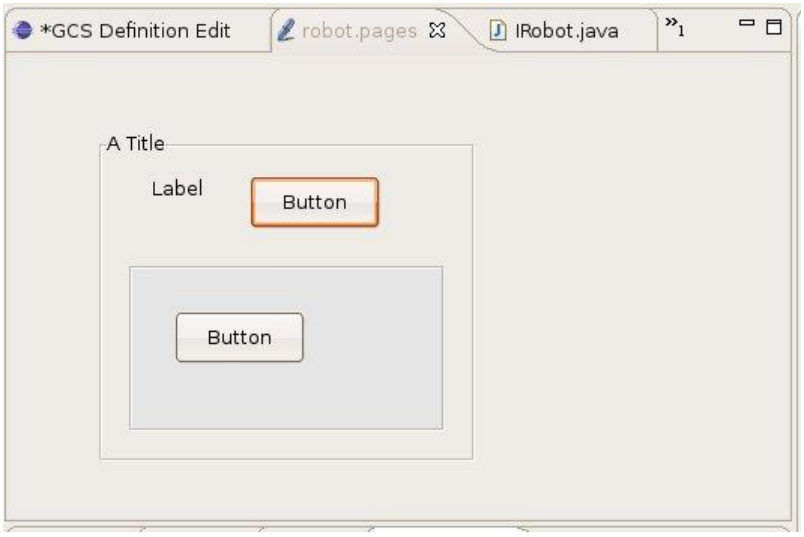


Figure 3-20 : Éditeur de pages en mode exécution

On présente à la Figure 3-20, l'éditeur de pages basculé dans le mode exécution. Dans ce mode, les composants sont actifs et peuvent lever des évènements.

3.3.5 Palette de composants

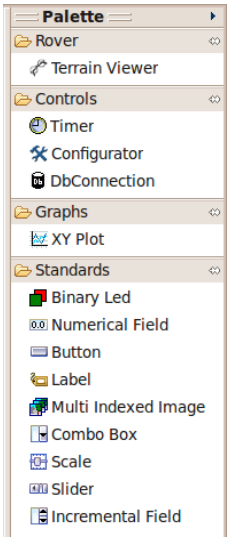


Figure 3-21 : Palette de composants graphiques

La palette de composants graphiques de l'EDD apparaît dans l'éditeur de pages lorsque celui-ci est en mode édition. Il donne accès par manipulation directe aux composants graphiques fournis par les plugiciels qui offrent des extensions du type « bibliothèque de composants » et qui contiennent des classes de réalisation de l'interface GCSWidget. Ce mécanisme est expliqué plus en détail en 3.1.3 et en 3.3.4.

3.3.6 Gestion de l'organisation des vues

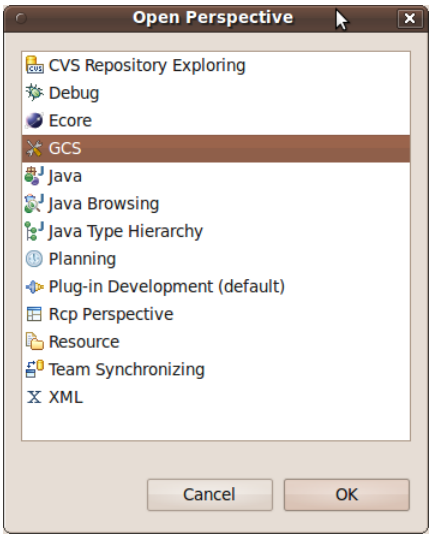


Figure 3-22 : Perspective de l'EDD

L'EDD offre une perspective par défaut de la mise en page des vues utiles au développement de l'IHM. Eclipse offre à l'utilisateur les outils nécessaires à la personnalisation de cette perspective.

3.3.7 Éditeur de propriétés

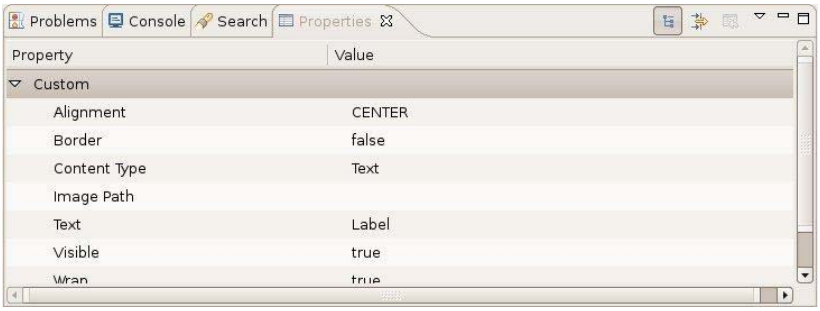


Figure 3-23 : Éditeur de propriétés

L'éditeur de propriétés présenté à la Figure 3-23 est inclus dans la perspective EDD décrite en 3.3.6. L'EDD utilisant les mécanismes de sélection du cadriciel d'Eclipse, les propriétés des objets dont la sélection est active dans la vue courante y sont affichées automatiquement; l'utilisateur peut les modifier à l'aide de l'éditeur spécifique offert pour chaque propriété.

La sélection multiple est permise et entraîne l'affichage des propriétés communes aux objets sélectionnés. Cette caractéristique touche un aspect ergonomique qui consiste à accroître

l'efficacité de l'utilisateur en lui permettant de cumuler les données et de modifier plusieurs composants à la fois (Bass et Kates, 2001). Par exemple, l'utilisateur pourrait vouloir changer la police de toutes les étiquettes qui figurent sur une page.

3.3.8 Préférences

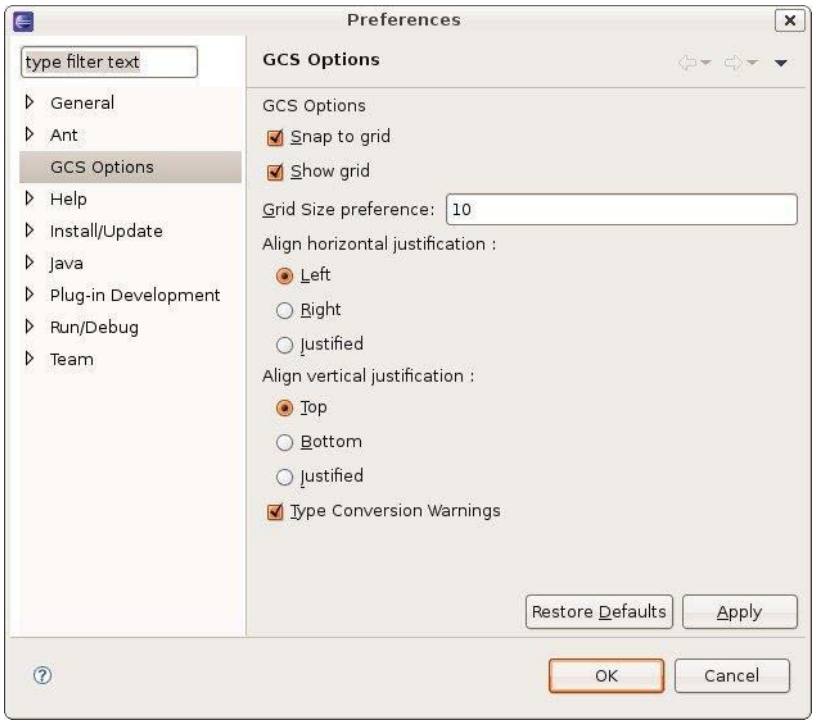


Figure 3-24 : Dialogue des préférences

L'EDD permet de personnaliser le mode édition de l'éditeur de pages à l'aide du mécanisme de préférences d'Eclipse. Il permet entre autres d'afficher un grillage d'alignement, d'en spécifier la granularité et d'activer le mode connexion dans lequel les composants graphiques se connectent au grillage lorsque déposés ou déplacés.

CHAPITRE 4 - RÉSULTATS ET DISCUSSION

De façon à démontrer la faisabilité de l'EDD et à en valider l'utilité, on a développé à l'ASC une implémentation qui offre toutes les fonctionnalités nécessaires à son utilisation en milieu opérationnel.

On présente au tableau A-3 de l'annexe 4 des statistiques détaillées sur cette implémentation. Elle compte au total 29 plugiciels Eclipse comportant 67 447 lignes de codes, 15 de ces plugiciels comportant 49 600 lignes de codes ayant été créés à la main; environ 29 760 de ces lignes ont été codées à la main et les autres générées à l'aide du CME. Ces plugiciels se greffent à ceux d'Eclipse pour former l'EDD, c'est-à-dire l'environnement de développement et d'exécution d'IHM. Ces plugiciels, qui sont génériques et réutilisables, accompagnent chacun des IHM pour en permettre l'exécution et le développement.

L'EDD compte 7 modèles CME à partir desquels environ 40 % de ses codes sources ont été produits automatiquement. Il offre 8 éditeurs et assistants et 25 composants graphiques accompagnés d'éditeurs spécialisés et de valideurs de leurs propriétés. À ces composants graphiques peuvent s'ajouter les composants graphiques importés de bibliothèques supplémentaires. Seul l'EDD propose une implémentation de ces composants. Par contre, le langage déclaratif employé pour décrire les IHM permettrait leur interprétation et leur exécution dans d'autres environnements proposant des implémentations équivalentes de ces composants.

Un manuel de l'utilisateur d'une centaine de pages accompagne l'EDD; on y décrit les concepts de bases de l'EDD, on y présente une vingtaine de scénarios d'utilisation sous forme de tutoriels et on y explore une dizaine de situations problématiques et les voies de contournement permettant de les éviter.

Au total, trois IHM complètes destinées à des applications opérationnelles ont été développés à l'aide de l'EDD : l'IHM de CART, celle de NORCAT et celle de MARVIN. On les décrit plus bas. De plus, une douzaine de prototypes expérimentaux ont été développés à des fins de démonstration, d'expérimentation et d'apprentissage.

L'utilisation des premières versions a débuté en 2006 avec une IHM de contrôle du robot CART utilisé dans le cadre de divers projets de recherche à l'ASC. Dans ce contexte, les mécanismes de

l’EDD associés au polymorphisme du noyau fonctionnel ont été mis à profit en permettant à l’utilisateur d’aiguiller l’IHM, soit vers le système de contrôle du robot CART, soit vers son simulateur, en choisissant le contexte d’exécution connexe. L’IHM a permis de contrôler le vrai robot et a été testée avec le simulateur.

L’EDD a été utilisée directement par environ 10 étudiants de différentes disciplines d’ingénierie qui n’étaient pas nécessairement des programmeurs. Ces utilisateurs œuvraient soit dans le département de robotique, soit dans celui des opérations de la Station spatiale internationale (SSI), ou encore dans le département du segment terrestre. Dans le cas de l’application MARVIN, les utilisateurs, qui étaient des spécialistes en opérations spatiales, ont utilisé l'EDD par l'entremise d'étudiants pour développer les maquettes et les améliorer de façon itérative par le biais d’ateliers de révision. On traite de MARVIN un peu plus loin dans ce chapitre.

On résume ici les commentaires de ces utilisateurs. Ils ont été reçus de façon informelle tout au long de l’utilisation de l’EDD et ont permis de résoudre des problèmes ou d’apporter des améliorations. Plusieurs commentaires ont été reçus verbalement, d’autres par courriel, et certains ont été documentés dans Mantis. Les problèmes se situent règle générale au niveau des comportements fautifs des outils de manipulation directe ou de programmation visuelle, ou encore des bogues dans les composants graphiques. La grande majorité de ces problèmes ont été résolus sur le champ au début du développement de l’EDD. Quelques problèmes se sont manifestés au niveau du chargement dynamique des classes et ont été résolus aussitôt.

On a également reçu des rétroactions concernant l’utilisabilité de l’EDD. Le manuel de l’utilisateur de 100 pages témoigne entre autres du fait que l’EDD est un outil puissant, mais qu’il exige un certain apprentissage de ses fonctionnalités et de la méthodologie d’assemblage proposée. De nombreux commentaires au sujet de l’utilisation de l’EDD ont servi à parfaire le manuel de l’utilisateur. Toutefois, il serait utile d’ajouter à l’EDD des services d’aide en ligne et d’autres outils contextuels pour simplifier davantage la tâche des utilisateurs.

La rétroaction positive la plus souvent reçue concerne la rapidité avec laquelle une IHM peut-être développée à l’aide de l’EDD, notamment en ce qui a trait à la manipulation directe et à l’aspect dynamique de l’assemblage qui permet de modifier l’IHM sans devoir relancer la plateforme. C’est cette dernière caractéristique qui distingue l’EDD des autres environnements de développement. Une autre rétroaction positive concerne les mécanismes de copier et coller

intelligents qui savent rediriger les comportements copiés de façon contextuelle. Ces mécanismes favorisent la réutilisation des parties de l'IHM.

Certains commentaires portent sur les perfectionnements qui seront abordés plus tard. On proposera entre autres un nouveau mode de programmation visuelle en réponse au commentaire des utilisateurs voulant que l'assemblage d'appel de méthodes dans la présentation actuelle est laborieux. En effet, l'assemblage des comportements se fait à l'aide d'un arbre et n'est pas intuitif lorsque l'arbre est trop profond. Une visualisation semblable à celle proposée par les diagrammes d'interactions UML est envisagée.

Les utilisateurs ont aussi relevé une lacune au niveau de la transparence de l'état du système. L'utilisateur doit pouvoir observer facilement l'état du système (Bass et Kates, 2001). En effet, les mécanismes qui permettent de changer l'état de l'EDD sont disponibles, mais l'utilisateur doit se souvenir de ses actions pour connaître l'état d'exécution de l'EDD. Dans une situation idéale, il faudrait pouvoir afficher une représentation de cet état à un endroit de l'environnement qui est toujours visible.

Une des applications qu'a servi à développer l'EDD est l'IHM de l'application MARVIN mentionnée plus haut. MARVIN sert à tester des fichiers de paramètres destinés au bras robotique de la SSI. Le noyau fonctionnel de MARVIN installe et exécute des tests automatiques qui ont été configurés par l'utilisateur pour simuler les interactions de l'astronaute sur l'IHM de contrôle du bras robotique à bord de la station spatiale. Les applications de tests font la collecte des données pendant leur exécution et rapportent les résultats à MARVIN. MARVIN produit ensuite un rapport qui permet à l'utilisateur d'analyser les résultats.

Bien que l'expérience ait été constructive, les utilisateurs de l'EDD ont relevé certains problèmes importants. Notamment, en l'absence de toute contrainte au niveau de la méthodologie de design, on a doté les interactions entre l'IHM et le noyau fonctionnel de MARVIN d'une trop grande fonctionnalité et d'un niveau d'intelligence trop élevé. Comme conséquence directe, les concepteurs ont éprouvé des difficultés à localiser la source des erreurs décelées.

On a tiré plusieurs leçons de cette expérience. On s'est rendu compte, par exemple, qu'il est préférable de limiter au minimum les interactions entre l'IHM et le noyau fonctionnel. De plus, il faut s'assurer que l'IPA du noyau fonctionnel est complète et simple et qu'elle favorise une intégration rapide de l'IHM et du noyau fonctionnel en offrant des méthodes dont les arguments

nécessitent peu de traitement. Dans ce sens, plusieurs méthodes du noyau fonctionnel de MARVIN exigent que des objets soient construits et transférés en paramètres aux méthodes. Il aurait été préférable que ces arguments soient simples et que le travail de construction des objets complexes soit effectué à même le noyau fonctionnel. Enfin, l'objectif de l'EDD étant d'offrir un environnement de prototypage rapide d'IHM, il est préférable que le noyau fonctionnel soit complètement testé et validé avant de l'intégrer à l'IHM.

L'EDD a aussi été utilisée pour la conception d'un prototype d'IHM de contrôle d'une perceuse destinée à la planète Mars dans le cadre du projet NORCAT. Le prototype a été développé à l'ASC par le département du segment terrestre. Il a été utilisé pour contrôler la perceuse à partir d'une salle de contrôle de l'ASC à St-Hubert sur une surface analogue à celle de Mars située à Hawaii. L'expérience s'est conclue avec succès et des modifications supplémentaires seront apportées à l'IHM en vue de l'exécution d'expériences futures.

Malgré les nombreux autres systèmes disponibles sur le marché, on envisage à l'heure actuelle à l'ASC d'utiliser l'EDD pour le développement de nouveaux projets.

CHAPITRE 5 - CONCLUSION

L'implémentation de l'EDD présentée dans ce mémoire a permis de répondre aux objectifs visés (chapitre 2) en offrant une plateforme permettant le prototypage rapide des IHM à l'aide de laquelle l'utilisateur peut tester avec des maquettes une IHM et en valider les exigences. L'architecture est innovatrice, est flexible et permet à l'utilisateur de participer tôt à la spécification de l'IHM par prototypage. Il s'agit d'une implémentation mature et bien documentée de l'EDD qui nous aura permis d'en vérifier la faisabilité et d'en valider l'utilité.

Nous avons présenté les fondements et l'architecture de l'EDD. Nous avons décrit comment sont abordés tous les aspects de l'assemblage d'une IHM et comment l'EDD permet à l'utilisateur d'apporter des modifications et de les appliquer dynamiquement et immédiatement sans avoir à relancer la plateforme. Cette caractéristique distingue cette plateforme des autres environnements de développement et de prototypage d'IHM qui proposent un cycle d'édition code/compilation/lancement de l'application dont l'étape de compilation constitue une entrave à l'efficacité du développeur.

L'objectif de permettre à un utilisateur non programmeur de prototyper les IHM a été démontré grâce à cette implémentation de l'EDD qui offre un environnement hybride dans lequel l'utilisateur non programmeur dispose d'outils de programmation visuelle et de manipulation directe pour la conception des vues graphiques et des comportements et qui offre toutes les possibilités usuelles des environnements de programmation aux utilisateurs programmeurs. Il utilise un langage déclaratif pour la modélisation de l'IHM, langage à l'aide duquel la réutilisation des vues graphiques et de leurs comportements est possible grâce, entre autres, aux mécanismes copier/coller qui redirigent les comportements collés selon le contexte. Il permet aussi l'aiguillage de l'IHM par le biais de mécanismes de polymorphisme du noyau fonctionnel.

Tel qu'on l'a indiqué au chapitre précédent, l'EDD a permis de concevoir plusieurs IHM. Les utilisateurs qui ont conçu ces IHM nous ont fourni de précieux commentaires qui ont servi à améliorer l'EDD et à cerner les améliorations à apporter à cet environnement.

On a également observé certaines des limites de l'EDD. On a constaté notamment qu'en l'absence de toute contrainte au niveau de la méthodologie, l'utilisateur peut développer des interactions trop complexes qui rendent difficile leur débogage. De plus, rien ne nous assure que le noyau fonctionnel et son IPA, tel qu'ils sont fournis à l'utilisateur, sont complètement dépourvus d'erreurs et validés, dans quel cas il est difficile d'établir la source des erreurs à l'exécution.

Les utilisateurs ont indiqué que le mode de programmation visuelle des comportements était peu intuitif et difficile à utiliser lorsque les arbres de comportements deviennent trop profonds. Une autre visualisation devra être proposée, par exemple, un mode de programmation utilisant une visualisation semblable aux diagrammes d'interactions UML.

Les futurs travaux pourraient apporter de nouvelles fonctionnalités qui favoriseraient encore plus l'efficacité de l'utilisateur en matière de conception des prototypes d'IHM. Entre autres, on pourrait faire en sorte que les patrons de conception d'IHM accélèrent le développement des parties de l'IHM traitant de problèmes pour lesquels des solutions ont été identifiées par le biais des patrons utilisés (Sinnig, Gaffar, Reichart, Forbrig et Seffah, 2005). Ces patrons offrent des solutions à des problèmes connus réutilisables. L'approche inverse pourrait aussi être appliquée en permettant à l'utilisateur de sélectionner des parties de l'IHM pour en faire des patrons qui pourront être réutilisés par la suite pour résoudre des problèmes similaires.

Enfin, nos travaux n'ont qu'effleuré le sujet du débogage par l'utilisateur non programmeur en offrant des éléments de validation automatiques qui permettent d'attirer l'attention de l'utilisateur sur certaines erreurs d'assemblage. Une recherche plus poussée pourrait permettre d'offrir des moyens flexibles et intuitifs pour prévenir les erreurs ou les communiquer à l'utilisateur, voire même des moyens qui lui proposeraient des solutions de façon automatique (Ruthruff, Creswick et Burnett, 2003; Jones et Harrold, 2005).

BIBLIOGRAPHIE

Abrams, M., Phanouriou, C. et Batongbacal, A. L. (1999). UIML: an appliance-independent XML user interface language. Elsevier Science B.V. .

Ali, M. P.-Q. (2003). Building Multi-Platform User Interfaces With UIML. In *A. Seffah et H. Javahery (eds.) Multiple User Interfaces: Engineering and Application Framework.* John Wiley and Sons.

Auer, M., Pölz, J. et Biffl, S. (2009). END-USER DEVELOPMENT IN A GRAPHICAL USER INTERFACE SETTING. *Proceedings of the 11th International Conference on Enterprise Information Systems, ICEIS 2009,* (pp. 5-14). Milan, Italie.

Balaban, M., Barzilay, E. et Elhadad, M. (2002). Abstraction as a means for end user computing in creative. *IEEE Transactions on Systems* , 640-653.

Bass, L. et John, B. (2003). Linking usability to software architecture patterns : through general scenarios. *The Journal of Systems and Software , 66,* 187-197.

Bass, L. et Kates, B. (2001). *Achieving usability through software architecture.* CMU/SEI-2001.

Bellynck, V. (2000). SYNCHRONISATION DE PLUSIEURS MODES VISUELS POUR AIDER A LA PROGRAMMATION DES UTILISATEURS NON-PROGRAMMEURS. *Colloque sur la multimodalité, Mai 2000, IMAG.* Grenoble.

Beringer, J., Fischer, G., Mussio, P., Myers, P., Patern, B. et Ruyter, B. D. (2008). The Next Challenge: from Easy-to-Use to Easy-to-Develop. Are You Ready? *Proceedings of the Conference on Human Factors in Computing Systems (CHI'08),* (pp. 2257-2260).

Blanch, R., Beaudoin-Lafon, M. et Conversy, S. (2005). INDIGO : une architecture pour la conception. *IHM.* Toulouse.

Bouktif, S., Sahraoui, H. et Giuliano, A. (2006). Simulated Annealing for Improving Software Quality Prediction. *GECCO'06.* Seattle, Washignton, É-U.

Brandt, J., Guo, P. et Lewenstein, J. (2009). End-user software engineering Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software* , 18-24.

Calvary, G., Coutaz, J. et Thevenin, D. (2002). Plasticity of User Interfaces: A Revised Reference Framework. *Academy of Economic Studies of Bucharest Academy of Economic Studies of Bucharest*, (pp. 127-134). Bucharest.

Carrera_Software. (n.d.). *Page web de Carrera Software: GUI Design Studio*.

Chatty, S., Lecoanet, P. et Sire, S. (2004). Revisiting Visual Interface Programming: Creating GUI Tools for Designers and Programmers. *UIST'04 proceedings , 6* (2).

da Silva, P. P. (2001). User Interface Declarative Models and Development Environments: A Survey. *DSV-IS 2000, LNCS 1946*, (pp. 207-226).

Dupuis, E., Doyon, M. et Martin, E. (2004). Autonomous Operations for Space Robots. *55th International Astronautical Congress.* Vancouver.

Eclipse.org. (n.d.). *Le site web d'Eclipse. http://eclipse.org*.

Foley, J., Kim, W. et Kovacevic, S. (1991). UIDE-An Intelligent User Interface Design Environnement. Dans *Architectures for Intelligent Interfaces: Elements and Prototypes.*

Goldman, K. J. (2003). A Demonstration of JPie: An Environment for Live Software Construction in Java. *OOPSLA'03.* Anaheim, Californie.

Gregersen, A. R. et Jorgensen, B. N. (2009). Dynamic update of Java applications—balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.: Res. Pract. 2009 , 21*, 81-112.

Gregersen, A. R., Simon, D. et Jorgensen, B. N. (2009). Towards a Dynamic-Update-Enabled JVM. *RAM-SE'09.* Gênes, Italie.

Grigoreanu, V. et Burnett, M. (2009). *Design Implications for End-User Debugging Tools: A Strategy-Based View.* Oregon State University Technical Report, http://ir.library.oregonstate.edu/jspui/handle/1957/12443.

Grigoreanu, V., Burnett, M. et Robertson, G. G. (2010). A Strategy-Centric Approach to the Design of End-User Debugging Tools. *CHI 2010.* Atlanta, Georgie, É-U.

Hennipman, E.-J., Oppelaar, E.-J. R. et Van Der Veer, G. C. (2008). Rapid and rich prototyping: proof of concepts for experience. *ECCE'08.* Madère.

Jodoin, A. et Desmarais, M. (2009). L'Environnement Dynamique de Développement (EDD) pour le prototypage rapide d'interfaces graphiques. *IHM'09.* Grenoble.

Jodoin, A., L'Archevêque, R., Allard, P. et Desmarais, M. (2006). Un environnement dynamique de développement qui intègre des composants graphiques actifs. *IHM'06.* Montréal.

Jones, J. A. et Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *ASE'05.* Long Beach, Californie, États-Unis.

Ko, A., Abraham, R. et Beckwith, L. (2009). The State of the Art in End-User Software Engineering. *ACM Computing Surveys.*

König, W., Rädle, R. et Reiterer, H. (2010). Interactive design of multimodal user interfaces, Reducing technical and visual complexity. Dans *Multimodal User Interfaces* (pp. 197-213). Springer.

Landay, A. et Myers, B. (1995). Interactive sketching for the early stages of user interface design. *Conference on Human Factors in Computing Systems: Proceedings of the SIGCHI conference on Human factors in computing systems*, (pp. 43-50). Denver, Colorado, É-U.

Lehmann, G., Blumendorf, M. et Albayrak, S. (2010). Development of Context-Adaptive Applications on the Basis of Runtime User Interface Models. *EICS'10.* Berlin.

Lieberman, H., Paterno, F. et Klann, M. (2006). *End-User Development: An Emerging Paradigm.* Springer.

Limbourg, Q., Vanderdonckt, J. et Michotte, B. (2004). USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. *Proc. of 9th IFIP Working Conf. on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004*, (pp. 55-62). Hambourg.

Memmel, T., Vanderdonckt, J. et Reiterer, H. (2008). Multi-fidelity User Interface Specifications. *DSVIS 2008, LNCS 5136*, (pp. 43-57).

Meskens, J., Vermeulen, J. et Luyten, K. (2008). Gummy for Multi-Platform User Interface Designs: Shape me, Multiply me, Fix me, Use me. *AVI'08.* Napoli, Italie.

Michotte, B. et Vanderdonckt, J. (2008). Grafixml, A multi-target user interface builder based on UsiXML. *Proc. ICAS 2008. IEEE Computer Society Press.* Los Alamitos.

Microsoft. (n.d.). *Le site de Microsoft Visual Studio. http://msdn.microsoft.com/en-us/vstudio*.

Microsoft. (n.d.). *Page web de Microsoft Expression Blend*. extrait de http://www.microsoft.com/expression/products/Blend_Overview.aspx

Myers, B. (1990). Taxonomies of Visual Programming and Program Vizualization. *Journal of Vizual Languages and Computing , 1(1).*

Myers, B., Hudson, S. E. et Pausch, R. (2000). Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction , 7* (1), 3-28.

Nardi, B. (1993). A Small Matter of Programming: Perspectives on End User Computing. *Cambridge, MA: The MIT Press* .

National_Instruments. (n.d.). *Le site web de Labview. http://www.ni.com/labview/.*

Netbeans.org. (n.d.). *Le site web de Netbeans. http://netbeans.org*.

Olsen, D. R. et Klemmer, S. R. (2005). The Future of User Interface Design Tools. *CHI 2005.* Portland, Oregon.

Panko, R. (1998). What we know about spreadsheet errors. *Journal of End User Computing* , 15-21.

Previtali, S. C. et Gross, T. (2006). Dynamic Updating of Software Systems Based on Aspects. *22nd IEEE International Conference on Software Maintenance (ICSM'06).*

Price, B., Baeker, R. et Small, I. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing ,* 211-266.

Pukall, M., Kästner, C. et Götz, S. (2009). Flexible Runtime Program Adaptations in Java - A Comparision. *Otto-von-Guericke-Universität, Magdeburg* .

Rafla, T., Robillard, P. et Desmarais, M. (2007). A method to elicit architecturally sensitive usability requirements: its integration into a software development process. *Software Quality Journal, 15(2),* pp. 117-133.

Rashid, A., Weisenberger, J. et Meder, D. (2008). Bringing developers and users closer together: The openproposal story. *In Multikonferenz Wirtschaftsinformatik.*

Robertson, T., Prabhakararao, S. et Burnett, M. (2004). Impact of Interruption Style on End-User Debugging. *CHI 2004.* Vienne, Autriche.

Ruthruff, J., Creswick, E. et Burnett, M. (2003). End-User Software Visualizations for Fault Localization. *ACM Symposium on Software Visualization.* San Diego, CA.

Ruyter, B. et Sluis, R. (2006). Challenges for end-user developpement in intelligent environnements. Dans *End-User Developpement* (pp. 243-250). Springer.

Scaffidi, C., Myers, B. et Shaw, M. (2007). Trial by water: creating Hurricane Katrina “person locator” web. Dans *In Weisband S., Leadership at a Distance: Research in Technologically-Supported Work (ed).* Lawrence Erlbaum.

Seffah, A., Gulliksen, J. et Desmarais, M. (2005). An introduction to human-centered software engineering: Integrating usability in the development process. Dans A. Seffah, J. Gulliksen, et M. Desmarais, *Human-Centered Software Engineering - Integrating Usability in the Development Process* (pp. 3-14). Heidelberg: Springer.

Singh, G., Kok, C. H. et Ngan, T. Y. (1990). Druid: A System for Demonstrational Rapid User Interface Development. *ACM* .

Sinnig, D., Gaffar, A., Reichart, D., Forbrig, P. et Seffah, A. (2005). PATTERNS IN MODEL-BASED ENGINEERING. In R. Jacob, & al., *Computer-Aided Design of User Interfaces* (pp. 197-210). Kluwer Academic Publishers.

Sukaviriya, P. N., Foley, J. D. et Griffith, T. (1993). A Second Generation User Interface Design Environment: The Model and The Runtime Architecture. *Interchi'93.*

Szekely, P., Sukaviriya, P. et Castells, P. (1995). Declarative interface models for user interface construction tools: the MASTERMIND approach. *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction* , (pp. 120-150).

Vanderdonckt, J. et Coyette, A. (2007). Modèle, méthodes et outils de support au prototypage multi-fidélité des interfaces graphiques. *Revue d’Interaction Homme-Machine , 8* (1), pp. 91-123.

Vukelja, L., Müller, L. et Opwis, C. (2007). Are Engineers Condemned to Design? A Survey on Software Engineering and UI Design in Switzerland. *INTERACT 2007, LNCS 4663, Part II*, (pp. 555-568).

Wulf, v., Pipek, V. et Won, M. (2007). Component-based tailorability: Enabling highly flexible Software Applications. *International Journal of Human-Computer Studies , 66,* 1-22.

ANNEXE 1 – Composants graphiques et non graphiques offerts par l'EDD

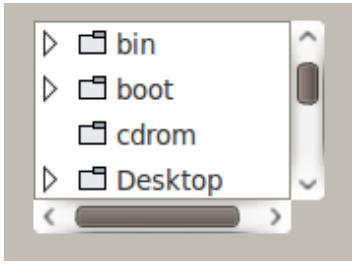
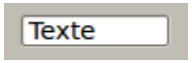
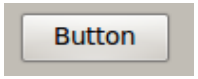
On présente ici la liste des composants graphiques et non graphiques offerts par défaut par l'EDD. Tous ces composants apparaissent dans la palette de construction. L'utilisateur peut complémenter cette liste à l'aide de plugiciels qui contribuent des composants graphiques et non graphiques par le biais du point d'extension `WidgetLibrary` de l'EDD en offrant des classes qui implémentent l'interface `GCSWidget`. L'interface `GCSWidget` est présentée à l'annexe 2. La méthode `showInRunMode` permet de déterminer si un composant est graphique ou non. La méthode `getInterfaces` permet aux composants de déclarer des méthodes.

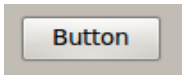
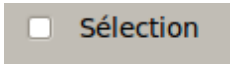
Chaque composant est identifié et brièvement décrit; les événements qu'il lève et ses propriétés sont résumés et une représentation graphique appropriée est présentée.

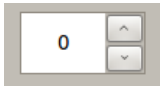
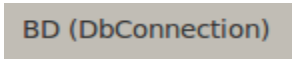
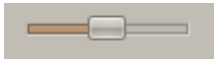
Les propriétés communes à tous les composants, soit la position, les dimensions, la police et la couleur du texte, la couleur de fond et l'identificateur unique, ne sont pas présentées ici.

Les composants graphiques sont soit dessinés, soit construits à partir de composants SWT. SWT offre aussi un contenant qui permet d'utiliser SWING. L'apparence des composants qui sont fondés sur des composants SWT correspond à celle de l'environnement d'exécution. L'apparence des images proposées dans la présente annexe correspond à celle des composants de la distribution Ubuntu de Linux sous le thème *Human*.

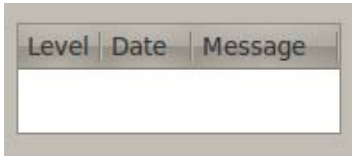
Tableau A-1 : Composants graphiques et non graphiques offerts par défaut par l'EDD.

Nom	**Description**	**Événements**	**Paramètres**	**Image**
Arbre de navigation	Un composant exclusif à l'EDD qui permet de naviguer dans le système de fichiers à l'aide d'un arbre, Il offre des méthodes pour accéder aux propriétés du/des fichier(s) sélectionné(s)	-La sélection a changé -Double clic	-Si l'affichage est statique ou interactif -Chemin à la racine de l'arbre -Si la sélection multiple est permise -Chemin de la (des) dernière(s) sélection(s)	
Barre de défilement	Barre de défilement	-La position a changé	-Position -Orientation	
Barre de progrès	Barre de progrès	-La position a changé	-Position en pourcentage -Si la valeur est indéterminée (la barre bouge de gauche à droite) -Orientation	
Boîte combinée	Un composant qui offre une liste de sélections et qui affiche la dernière sélection.	-Le texte a changé -La sélection a changé -La liste a changé	Liste des valeurs	
Boîte de texte	Boîte de texte	-Le texte a changé	-Valeur du texte -Si le texte doit s'enrouler	
Bouton	Un bouton	-Enfoncé -Relâché	-Étiquette -S'il s'agit d'un bouton poussoir	

Nom	Description	Événements	Paramètres	Image
Bouton de commande	Un bouton permettant d'exécuter une commande de système d'exploitation (lancer une application). Le composant suit l'état d'exécution de la commande et ajuste	-Bouton enfoncé -Bouton relâché -Commande lancée -Exécution interrompue de l'intérieur -Exécution interrompue de l'extérieur	-Chaîne de caractères de la commande -Chaîne de caractères de la liste d'arguments de la commande -Étiquette et couleur du bouton lorsque la commande est en exécution -Étiquette et couleur du bouton lorsque la commande est en arrêt	
Bouton radio	Boutons radio placés dans un même groupe; la sélection devient exclusive, un seul bouton est sélectionné à la fois	-La sélection a changé	-Valeur de la sélection -Groupe	
Cadran	Un composant qui présente un cadran d'affichage de valeur numérique avec zones limites	-Valeur entrée dans la zone (normale, d'avertissement, d'alerte inférieure et supérieure)	-Surface à afficher (en degrés, limites inférieure et supérieure) -Couleur de chaque zone -Valeur du début de chaque zone -Couleur de l'aiguille -Position de l'aiguille	
Case à cocher	Un composant pour activer une sélection. Lorsque plusieurs cases à cocher sont présentes dans un même groupe, le choix n'est pas exclusif et plusieurs peuvent être sélectionnées	-Sélectionné -Non-sélectionné	-Étiquette -Si la case est cochée	

Nom	Description	Événements	Paramètres	Image
Champs numériques à incrément	Un composant qui permet de spécifier une valeur et d'y ajouter ou d'en retirer une valeur « delta » à l'aide des boutons vers le haut ou vers le bas	La valeur a changé	-Delta à appliquer -Valeur	
Configurateur	Un composant qui permet de sauvegarder et de récupérer les propriétés des composants de l'IHM dans des fichiers personnalisés. Lorsqu'une configuration sauvegardée est sélectionnée, les valeurs sont appliquées aux propriétés des composants pour les remettre dans l'état où ils étaient au moment de la sauvegarde.	-La sélection a changé -La liste des propriétés a changé	Liste des propriétés à sauvegarder et valeurs par défaut à appliquer (un éditeur spécifique est offert)	
Connexion à une base de données	Un composant non graphique qui permet la connexion à une base de données; il offre des méthodes de contrôle et d'accès aux données	Aucun	Paramètres de connexion	
Échelle	Échelle permettant d'afficher ou de spécifier une valeur numérique	-La valeur a changé	-Valeur -Minimum et maximum -Orientation	
Étiquette	Une étiquette	Aucun	-Texte de l'étiquette -Si le texte doit s'enrouler	

Nom	**Description**	**Événements**	**Paramètres**	**Image**
Groupe (contenant)	Un groupe	Aucun	Titre	
Image	Une image	Aucun	Chemin du fichier de l'image	
Indicateur binaire	Un composant qui représente un de deux états par sa couleur	-Allumé -Éteint -L'état a changé	-Couleur pour chacun des 2 états -Index de l'état courant, soit 0 ou 1	
Minuterie	Un composant non graphique qui lève des évènements à des intervalles spécifiés	-Intervalle de temps atteint	-Intervalle de temps -Si la minuterie est en marche	
Multi-images indexées	Un composant qui permet d'afficher une image par son index. Un éditeur spécifique est offert pour construire la liste d'images. Il peut servir à représenter des états ou à produire des animations à l'aide d'une minuterie.	-L'index a changé	-Liste d'images et leurs index -Index sélectionné	

Nom	Description	Événements	Paramètres	Image
Panneau de journal	Un composant qui permet d'afficher des messages. Il s'agit d'un journal Java. Le composant offre des méthodes pour ajouter des messages et spécifier leur niveau de gravité (information, avertissement, alerte) ainsi que de nombreuses méthodes utilitaires	Aucun	-Texte du panneau -Nombre maximum de messages	
Sélection d'un fichier	Un composant qui permet d'effectuer la sélection d'un fichier à l'aide d'un dialogue. Il offre des méthodes pour accéder aux propriétés du/des fichier(s) sélectionné(s)	-La sélection a changé -Bouton enfoncé -Bouton relâché -Sélection acceptée -Sélection annulée	-Étiquette du bouton -Si la sélection multiple est permise -Chemin de la (des) dernière(s) sélection(s)	
Tableau de journal	Un composant identique au panneau de journal, mais présenté dans un tableau qui peut être classé par colonne. La largeur et la position des colonnes peuvent être modifiées.	Aucun	-Texte du panneau -Nombre maximum de messages	

Nom	Description	Événements	Paramètres	Image
Valeur numérique avec zones limites	Un composant exclusif à l'EDD affichant une valeur numérique formatée et représentant les zones limites par des couleurs	-La valeur a changé -Valeur entrée dans la zone (normale, d'avertissement, d'alerte inférieure et supérieure)	-Valeur -Couleur de chaque zone -Valeur du début de chaque zone -Format numérique (expression standard, exemple: `##0.##E0` pour cette image)	

ANNEXE 2 – Méthodes de l'interface Java `GCSWidget` implémentée par les composants

Cette annexe présente l'interface Java `GCSWidget` qui permet de créer des bibliothèques de composants réutilisables. Elle permet à l'EDD d'intégrer les composants à la palette et d'interagir avec eux par le biais de ses méthodes implémentées par les composants.

Tableau A-2 Méthodes de l'interface `GCSWidget` réalisée par les composants.

Nom de la méthode	**Valeur de retour**	**Liste des arguments**	**Description/Utilité**
`Init`	-	Le gestionnaire d'évènement et celui d'accès aux propriétés	L'EDD transmet au composant les gestionnaires à l'aide desquels il pourra lever des évènements et annoncer les changements de valeur de ses propriétés.
`getControl`	Le contrôle SWT	Le composite SWT qui sera le parent	L'EDD demande au composant de se créer graphiquement
`getPropertyDescriptors`	Une liste des descripteurs de propriétés	-	L'EDD demande la liste des descripteurs de propriétés du composant avec laquelle il pourra construire l'éditeur de propriétés. Chaque descripteur, en plus de fournir les informations sur type et la valeur par défaut de la propriété, fournit un éditeur spécifique au type de la propriété. (Un dialogue de sélection de couleur ou de police, par exemple)

Nom de la méthode	**Valeur de retour**	**Liste des arguments**	**Description/Utilité**
`getEventDescriptors`	Une liste des descripteurs d'évènements	-	L'EDD demande la liste des descripteurs d'évènements pour lever le composant. L'EDD pourra les afficher à l'utilisateur pour qu'il y attache des comportements.
`propertyChanged`	Un contrôle SWT	Le nom de la propriété modifiée, son ancienne et sa nouvelle valeur	Le composant est avisé du changement de la valeur d'une de ses propriétés (de l'extérieur). Il peut y réagir et retourner le contrôle SWT qui le représente s'il a dû le substituer.
`Refresh`	-	-	L'EDD demande au composant de se redessiner.
`getDefaultWidth/ getDefaultHeight`	La valeur	-	L'EDD demande les dimensions par défaut du composant qu'il utilisera lorsque l'utilisateur le dépose sur une page.
`getLabelProvider`	Un producteur d'étiquette	-	L'EDD demande le producteur d'étiquette duquel il obtiendra l'image et l'étiquette servant à représenter le composant sur la palette.

Nom de la méthode	**Valeur de retour**	**Liste des arguments**	**Description/Utilité**
`isPropertyApplicable`	Une valeur booléenne	Le nom d'une propriété	L'EDD interroge le composant pour savoir si la propriété lui est applicable. Il s'agit des propriétés qui sont génériques (positions, dimension, etc.).
`getInterfaces`	Une liste d'interfaces Java	-	Le composant peut offrir une liste d'interfaces Java qu'il implémente. L'EDD pourra offrir cette liste à l'utilisateur au moment de la construction des interactions avec le composant.
`showInRunMode`	Une valeur booléenne	-	Le composant non graphique indique « non » et l'EDD n'appellera pas ses méthodes graphiques.
`isWidgetContainer`	Une valeur booléenne	-	Le composant qui est un contenant (pour d'autres composants) indique « oui » et l'EDD utilise cette information dans ses recherches par le biais de l'arbre qui représente la composition de l'IHM.

ANNEXE 3 – Environnement de développement de l’EDD

Nous avons développé l’EDD dans le cadre de nos fonctions dans le groupe de robotique de l’Agence spatiale canadienne. Le développement s’est fait dans Eclipse sous Linux avec la machine virtuelle de Sun (maintenant Oracle). La compatibilité à Microsoft Windows a été vérifiée à chaque modification de l’EDD.

La configuration initiale était la suivante :

- Eclipse 3.1
- Linux Ubuntu 6.06 (Dapper Drake)
- Sun Java JDK 1.5
- Windows 2000 (pour la vérification de la compatibilité)

L’EDD est en mode maintenance à l’Agence spatiale canadienne. La maintenance est assurée par le département du secteur terrestre. La configuration est la suivante :

- Eclipse 3.5.1
- Linux Red Hat Enterprise 5
- Java JDK 1.6 (Oracle)
- Windows 2000 (pour la vérification de la compatibilité)

Nous poursuivons l’expérimentation avec l’EDD dans le groupe de robotique. Dans ce contexte, la configuration de l’environnement de développement varie et a peu d’importance.

Tout au long du développement et de la maintenance de l’EDD, le contrôle des versions des codes sources a été assuré par CVS (*Concurrent Versions System*). Mantis a servi à la documentation et la gestion des bogues.

ANNEXE 4 – Statistiques sur l'implémentation proposée

On présente ici quelques statistiques sur les codes sources de chacun des 29 plugiciels qui composent l'EDD et les valeurs combinées pour l'ensemble d'entre eux. Les statistiques ont été compilées à partir des plugiciels Metrics et Mylin dans Eclipse et incluent les valeurs suivantes :

- Nombre de modules
- Nombre de classes
- Nombre de méthodes
- Nombre de lignes de code,
- Profondeur moyenne de l'héritage
- Complexité cyclomatique moyenne (mesure de McCabe)

Les plugiciels générés à l'aide du CME ont été identifiés séparément. Les autres plugiciels ont été créés à la main, mais comportent un certain pourcentage de codes sources générés à l'aide du CME. Il est difficile d'établir ce pourcentage avec précision, car la plupart des classes générées ont été modifiées par la suite. On pourrait indiquer de façon prudente qu'environ 40 % des codes sources des plugiciels créés à la main ont été générés à l'aide du CME..

En résumé, l'EDD est composé de 29 plugiciels comportant 67 447 lignes de codes, 15 de ces plugiciels comportant 49 600 lignes de codes ayant été créés à la main; environ 29 760 de ces lignes ont été codées à la main. Bien entendu, il s'agit ici d'un ordre de grandeur.

Un effort supplémentaire a été déployé pour la conception des 7 modèles CME à partir desquels environ 40 % des codes source de l'EDD ont été produits automatiquement.

Enfin, un manuel de l'utilisateur d'une centaine de pages décrit les concepts de bases de l'EDD, présente une vingtaine de scénarios d'utilisation sous forme de tutoriels et explore une dizaine de situations problématiques et les voies de contournement permettant de les éviter.

Tableau A-3 Statistiques sur les codes sources de l'EDD

Nom du plugiciel	**Nb de modules**	**Nb d'interfaces**	**Nb de classes**	**Nb de méthodes**	**Nb de lignes de codes**	**Profondeur moyenne de l'héritage**	**Complexité cyclomatique moyenne (McCabe)**
Plugiciels de l'EDD créés à la main							
csa.gcs.common	6	11	22	162	1 830	2.82	2.15
csa.gcs.common.conversion	1	1	14	36	590	1.14	1.25
csa.gcs.common.reflect	4	9	22	215	2 021	4.50	2.10
csa.gcs.common.ui	2	1	15	111	1 366	3.73	2.18
csa.gcs.common.validation	1	1	16	65	537	2.06	1.80
csa.gcs.core	6	18	57	523	5 114	3.70	1.90
csa.gcs.core.ui	2	0	8	55	592	2.25	1.90
csa.gcs.pages	6	5	21	341	3 392	4.33	2.11
csa.gcs.rcp	1	0	9	43	962	1.56	2.37
csa.gcs.reflect	5	16	43	543	6 039	4.70	2.49
csa.gcs.systems	6	30	28	662	5 597	4.10	2.00
csa.gcs.systems.ui	10	1	65	609	8 181	2.62	2.14
csa.gcs.widgets.control	6	9	28	266	3 444	3.36	2.21
csa.gcs.widgets.core	1	1	4	30	253	2.00	1.55
csa.gcs.widgets.standard	7	3	73	540	9 682	2.23	2.00
Total	**64**	**106**	**425**	**4 201**	**49 600**	**3.01**	**2.01**

Nom du plugiciel	**Nb de modules**	**Nb d'interfaces**	**Nb de classes**	**Nb de méthodes**	**Nb de lignes de codes**	**Profondeur moyenne de l'héritage**	**Complexité cyclomatique moyenne (McCabe)**
Plugiciels de l'EDD générés à l'aide du CME							
csa.gcs.common.edit	2	0	9	54	459	3.11	1.52
csa.gcs.common.editor	2	0	10	83	1 592	3.40	2.59
csa.gcs.common.reflect.edit	1	0	11	92	858	4.55	1.55
csa.gcs.common.reflect.editor	1	0	8	80	1 575	3.38	2.66
csa.gcs.core.edit	2	0	13	102	931	4.23	1.61
csa.gcs.core.editor	1	0	8	80	1 578	3.38	2.66
csa.gcs.pages.edit	1	0	7	49	403	5.14	1.50
csa.gcs.pages.editor	1	0	8	80	1 581	3.38	2.66
csa.gcs.reflect.edit	1	0	17	165	1 882	6.35	1.68
csa.gcs.reflect.editor	1	0	8	80	1 583	3.38	2.66
csa.gcs.systems.edit	1	0	17	159	1 674	4.29	1.60
csa.gcs.systems.editor	1	0	11	90	1 642	3.55	2.47
csa.gcs.widgets.control.edit	1	0	7	54	508	4.29	1.56
csa.gcs.widgets.control.editor	1	0	8	80	1 581	3.38	2.66
Total	**17**	**0**	**142**	**1 248**	**17 847**	**3.98**	**2.10**
Statistiques combinées des plugiciels créés à la main et des plugiciels générés à l'aide du CME							
Grand total	**81**	**106**	**567**	**5 449**	**67 447**	**3,49525**	**2.05420476**

ANNEXE 5 – Structure du projet EDD de l’IHM

L’assistant de création de projets de l’EDD permet de créer un projet de plugiciel Java Eclipse marqué de la nature EDD. Sa structure est décrite au tableau A-4.

Tableau A-4 Structure du projet EDD de l'IHM.

Nom du répertoire ou du fichier	**Description**
+ <Nom du projet>	Répertoire à la racine du projet dans Eclipse.
+ Referenced Libraries	Liste des bibliothèques dont dépend le projet.
+src + <package> - <java files>	Répertoire des codes sources Java développés par l'utilisateur.
+ Plugin Dependencies	Liste des plugiciels dont dépend le projet.
+ Admin	Répertoire contenant les fichiers de gestion de l'EDD.
+ Configurations	Répertoire où sont conservées les configurations créées à partir du composant graphique « Configurateur » présenté au Tableau A-1.
: + Definition - PageModels.list - <Project Name>Defition.systems	Fichier de définition de l'IHM et de la liste des modèles de pages créées par l'utilisateur.
+ META-INF	Ce répertoire contient les fichiers de configuration de l'IHM en tant que plugiciel Eclipse et peut-être modifié à l'aide de l'éditeur de manifeste offert par Eclipse.
+ Pages -<Model1>.pages -<Model2>.pages	Répertoire contenant les modèles de pages créées par l'utilisateur.
+ Resources	Répertoire où l'utilisateur peut placer des ressources comme des images.

Nom du répertoire ou du fichier	**Description**
- <Project Name>.product	Fichier permettant de définir un produit RCP qui peut être utilisé pour exporter une version indépendante et statique de l'IHM.