UNIVERSITÉ DU QUÉBEC À MONTRÉAL

# APPROCHE POUR LA DÉFINITION D'APPLICATIONS WEB RICHES MULTIPLATEFORME

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

STÉPHANE BOND

OCTOBRE 2008

# REMERCIMENTS

J'aimerais premièrement remercier mon directeur, Louis Martin pour sa disponibilité, son soutien et la patience dont il a su faire preuve au cours des nombreuses années sur lesquelles s'est étalé ce projet de maîtrise.

Je tiens aussi à remercier mon employeur, le CRIM, pour la latitude qui m'a été laissée dans la réalisation de plusieurs mandats, ayant permis d'effectuer des avancées dans ces travaux de recherches. Merci particulièrement à mon chef d'équipe, Robert Bolduc, pour les heures qui m'ont été accordées afin de mener à terme ce projet. Merci aussi à ma collègue Sylvie Trudel pour son aide quant à l'évaluation des résultats obtenus.

Remerciement particulier également à ma famille qui a toujours été présente pour m'appuyer dans les différents projets que j'ai entrepris.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

| | |
|---|---|
| ADO | ActiveX Data Objects |
| AJAX | Asynchronous JavaScript And XML |
| AOP | Aspect-oriented programming |
| API | Application programming interface |
| ASP | Active Server Pages |
| AUI | Abstract User Interface |
| AUIML | Abstract User Interface Markup Language |
| AWT | Abstract Window Toolkit |
| BD | Base de données |
| CAS | Central Authentication Service |
| CFP | Cosmic Function Point |
| CLI | Common Language Infrastructure |
| CLR | Common Language Runtime |
| COM | Component Object Model |
| COSMIC | Common Software International Consortium |
| COSMIC-FFP | COSMIC Full Function Point |
| CPU | Central Processing Unit |
| CRIM | Centre de recherche informatique de Montréal |
| CRUD | Create, Read, Update et Delete |

| | |
|---|---|
| CSS | Cascading Style Sheets |
| CUI | Concrete User Interface |
| DBCP | Database Connection Pool |
| DDL | Data Definition Language |
| DHTML | Dynamic HTML |
| DOM | Document Object Model |
| ECMA | European Computer Manufacturers Association |
| EJB | Enterprise Java Beans |
| EL | Expression Language |
| ETL | Extract, transform, load |
| FUI | Final User Interface |
| GTK | The GIMP Toolkit |
| GTKML | The GIMP Toolkit Markup Language |
| GWT | Google Web Toolkit |
| HTML | HyperText Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IDE | Integrated development environment |
| IOC | Inversion Of Control |
| ISO | International Organization for Standardization |
| J2EE | Java 2 Enterprise Edition |
| JA-SIG | Java Architectures Special Interest Group |
| Java EE | Java Enterprise Edition |
| JCP | Java Community Process |
| JDBC | Java database connectivity |

| | |
|---|---|
| JDO | Java Data Objects |
| JIT | Just-in-time |
| JMS | Java Message Service |
| JNDI | Java Naming and Directory Interface |
| JPA | Java Persistance API |
| JSP | JavaServer Pages |
| JSTL | JavaServer Pages Standard Tag Library |
| JTA | Java Transaction API |
| JVM | Java Virtual Machine |
| LZX | Laszlo Application Language |
| MDA | Rapid application development |
| MDI | Multiple Document Interface |
| MFC | Microsoft Foundation Class |
| MVC | Modèle Vue Controlleur |
| MXML | Macromedia Extended Markup Language |
| ODBC | Open Database Connectivity |
| OLAP | Online Analytical Processing |
| PC | Personal Computer |
| PDA | Personal digital assistant |
| PHP | PHP Hypertext Preprocessor |
| PIM | Platform-independent model |
| POJO | Plain Old Java Object |
| PSM | Platform-specific model |
| RAD | Rapid application development |

| | |
|---|---|
| RCP | Rich Client Platform |
| RDF | Resource Description Framework |
| RMI | Java Remote Method Invocation |
| ROLAP | Relational Online Analytical Processing |
| RPC | Remote procedure call |
| SDI | Single Document Interface |
| SMTP | Simple Mail Transfer Protocol |
| SOAP | Simple Object Access Protocol |
| SQL | Structured Query Language |
| SVG | Scalable Vector Graphics |
| SWF | Shockwave Flash |
| SWT | Standard Widget Toolkit |
| TCP/IP | Transmission Control Protocol / Internet Protocol |
| TDI | Tabbed document interface |
| UI | User interface |
| UIDL | User Interface Description Language |
| UIML | User Interface Markup Language |
| UML | Unified Modeling Language |
| UsiXML | USer Interface eXtensible Markup Language |
| VoiceXML | Voice Extensible Markup Language |
| W3C | World Wide Web Consortium |
| WAF | Web Application Format |
| WAP | Wireless Application Protocol |
| WML | Wireless Markup Language, |

| | |
|---|---|
| WPF | Windows Presentation Foundation |
| WYSIWYG | What You See Is What You Get |
| XAML | Extensible Application Markup Language |
| XBL | XML Binding Language |
| XIML | Extensible Interface Markup Language |
| XML | Extensible Markup Language |
| XML-RPC | XML-Remote procedure call |
| XPath | XML Path Language |
| XPFE | Cross-Platform front-end |
| XPI | Cross-Platform Install |
| XSL | Extensible Stylesheet Language |
| XSWT | XML/SWT page description language |
| XUL | XML User Interface Language |
| ZKML | ZK Markup Language |

# RÉSUMÉ

Les termes client Web riche ou interface riche sont utilisés pour désigner l'interface utilisateur d'une application Web qui comprend des fonctionnalités et des méthodes d'interactions similaires à celles des interfaces utilisateurs conventionnelles. Un client Web riche assure une part du traitement de l'application. Cela peut aller de la validation de saisies jusqu'à la prise en charge complète des interactions avec l'utilisateur. Il doit donc être doté d'une certaine intelligence, c'est-à-dire que du code, décrivant son comportement ainsi qu'une part de la logique d'affaire de l'application, doit pouvoir y être exécuté. L'utilisation de méthodes d'interactions avancées (comme le glisser-déplacer, la saisie semi-automatique ou l'utilisation de contrôles « widgets ») implique aussi des capacités de traitement plus poussées que pour les clients Web standards.

Il existe une multitude de technologies pouvant êtres utilisées pour le développement d'un client Web riche. Le principal problème relié à cette situation est que les projets basés sur une technologie de présentation deviennent dépendants de cette dernière. Un changement de technologie implique alors la perte des investissements relatifs au développement de la partie client. Le fait d'avoir à supporter plusieurs plateformes de présentation implique aussi généralement d'avoir à maintenir plusieurs versions distinctes du client de l'application.

Ces travaux couvrent les approches existantes et la conceptualisation d'une nouvelle approche permettant de définir, indépendamment d'une technologie de présentation, le volet client d'une application. Celle-ci a été expérimentée à l'intérieur de trois projets concrets présentés sous forme d'étude de cas. Une revue des plateformes d'exécution contemporaines pour les clients Web riches y est effectuée, suivi d'un état de l'art couvrant les méthodes existantes pour la définition d'interfaces utilisateur. Les travaux s'intéressant à la définition d'interfaces utilisateurs à l'aide de dialectes XML sont également couverts, de même que les architectures couramment utilisées pour la définition d'un client Web riche.

Les résultats obtenus à l'intérieur des études de cas auront permis de montrer la faisabilité de l'approche ainsi que de mesurer certains avantages de celle-ci selon différents critères de qualité.

# CHAPITRE I

# PROBLÉMATIQUE ET MÉTHODE DE RECHERCHE

## 1.1 Problématique

Les premières générations d'applications Web se contentaient de présenter des interfaces utilisateurs simples, en format HTML, générées entièrement côté serveur. Aujourd'hui, la tendance est d'offrir, via le Web, des applications clientes dont l'expérience utilisateur et les fonctionnalités disponibles sont équivalentes à celles retrouvées dans une application locale conventionnelle. C'est ce que l'on nomme un « Client Web riche ».

Puisque, par définition, une application Web est hébergée sur un serveur et est accessible via un navigateur, celle-ci n'a pas à être installée sur chacun des postes de travail. Le client Web doit donc pouvoir s'exécuter dans un environnement logiciel déjà présent sur les postes des utilisateurs. Il s'agit de la « Plateforme d'exécution ».

Au cours des dernières années, nous avons vu apparaitre une multitude de nouvelles plateformes d'exécutions pouvant êtres utilisées par les clients Web riches. À chacune de ses plateformes peuvent s'ajouter plusieurs technologies de présentations et un grand nombre de cadres de développement. Les développeurs d'aujourd'hui ont donc à choisir parmi un éventail de technologies imposant lorsqu'ils débutent un projet. Le principal problème relié à ce surnombre de combinaisons est que les développements effectués deviennent, dans la très grande majorité des cas, dépendants d'une seule plateforme. C'est-à-dire que les investissements reliés à la confection d'un client riche sous une plateforme donnée seront en majeure partie perdus advenant un changement technologique. L'introduction rapide, dans un court délai, d'un très grand nombre de technologies pour le développement d'applications

Web augmente d'autant plus le risque relié au choix d'une plateforme d'exécution. Certaines technologies pourraient cesser d'évoluer, laissant la place à d'autres qui s'imposeraient comme norme au cours des prochaines années.

La figure 1.1 expose la problématique en montrant qu'une part importante des développements réalisés pour une application Web est spécifique à la technologie utilisée du côté du client.

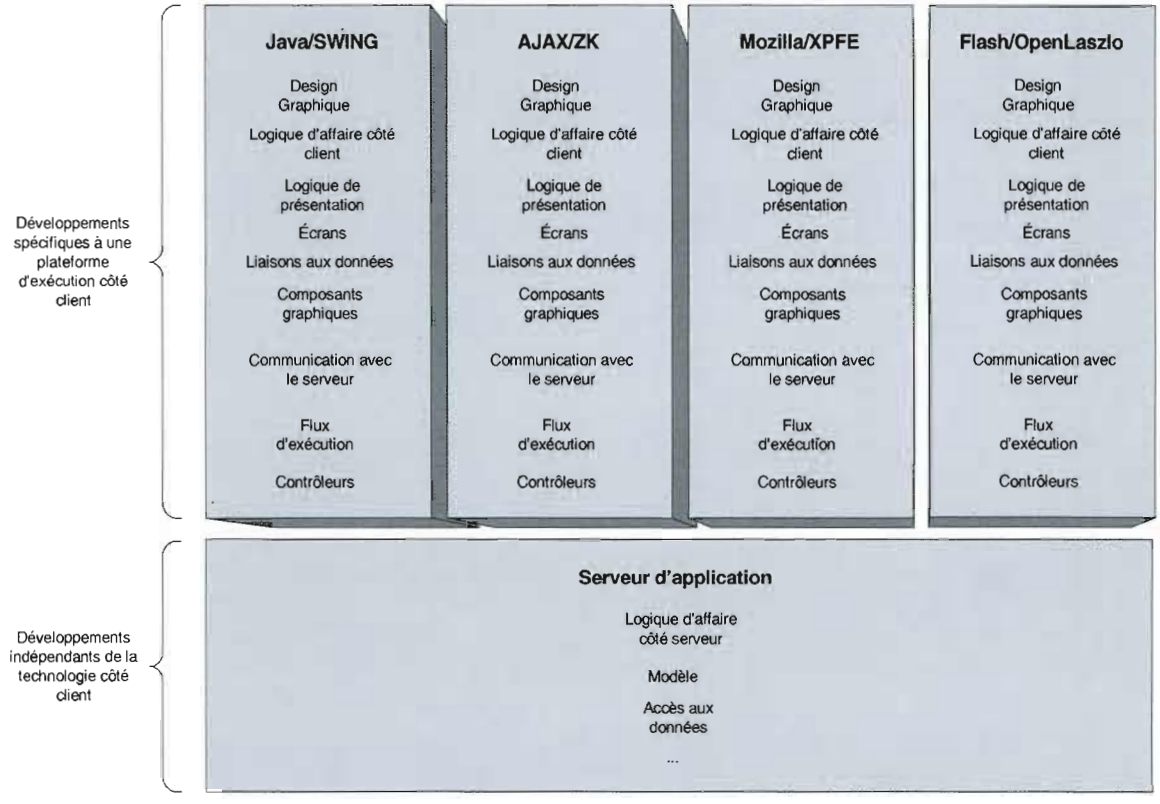


**Figure 1.1** Développements spécifiques à une plateforme d'exécution

Les développements reliés aux clients ne se limitent donc pas qu'aux interfaces utilisateurs puisque chaque technologie de clients Web riches peut exploiter ses propres méthodes de communications, ses propres contrôleurs et ses propres méthodes de définition pour la logique côté client. Les fonctionnalités disponibles peuvent aussi varier d'une

technologie à une autre, ce qui a pour effet d'accentuer les différences entre des clients de technologies différentes.

Le même problème se pose lorsqu'on veut développer une application offerte pour plus d'une plateforme. Prenons l'exemple d'une application devant offrir un client riche Java ainsi qu'une interface HTML classique. Les interfaces utilisateurs, la logique de présentation, les méthodes de communications et les contrôleurs seront, dans la plupart des cas, spécifiques à chacun des clients et devront êtres développés et maintenus en double.

## 1.2 Objectifs recherchés

L'objectif ultime visé par ces travaux de recherche est l'élaboration d'une approche de définition pour les clients Web riches permettant le support de plusieurs plateformes. Une telle approche devrait ainsi combler la problématique exposée en rendant indépendants d'une technologie de présentation les développements relatifs au client.

Le développement et la mise au point d'un environnement complet pour le déploiement et l'exécution d'applications sous plusieurs plateformes pourraient s'avérer un projet d'une très grande envergure. Pour cette raison, le présent travail devra se concentrer sur un des enjeux principaux relié au problème : la définition d'interfaces utilisateurs indépendantes d'une technologie de présentation. Les prototypes développés devront donc être basés sur un modèle indépendant de plateforme, mais pourront, aux fins de l'expérimentation, n'offrir qu'un seul choix de plateforme. Le tout devra tout de même offrir la possibilité d'être étendu ultérieurement pour le support de nouveaux types de clients.

Nous avons fait le choix d'expérimenter une nouvelle approche de développement, basée sur les méthodes et concepts qui devront êtres étudiés dans une première phase. Celle-ci a pour but d'approfondir les domaines rattachés aux clients Web riches et de nous procurer une vue d'ensemble de ce qui se fait présentement dans le domaine couvert par la problématique. Une étude des plateformes d'exécutions et des méthodes existantes pour la définition d'interfaces utilisateurs devra être réalisée. Les architectures couramment utilisées pour le déploiement d'applications Web riches devront aussi êtres couvertes afin de mettre en contexte le client et ses interactions avec les autres composants d'une application.

### 1.2.1 Critères à respecter

La solution retenue devra être exploitable à l'intérieur de projets concrets. De ce fait, d'autres critères importants devront êtres respectés, dont la maintenabilité et la rapidité de développement. Ils devront aussi se baser sur des cadres de développement reconnus et s'adapter aux choix technologiques des différents projets dans lesquels la solution sera exploitée.

#### 1.2.1.1 Support de plusieurs plateformes

En réponse à la problématique, l'approche retenue devra être conçue pour supporter plusieurs plateformes et technologies de présentation à partir d'une définition unique. Pour respecter ce critère, la définition de l'application devra donc être indépendante de toute technologie de présentation.

#### 1.2.1.2 Temps de développement

Les travaux d'expérimentations étant effectués à l'intérieur de projets concrets, avec un nombre restreint d'heures de travail accordées, le temps de développement s'impose comme une des principales exigences. Le temps accordé pour la réalisation d'un projet utilisant l'approche sélectionnée ne pourra être supérieur aux délais qui auront été prévus pour la réalisation à l'aide de méthodes conventionnelles. Lors de la conceptualisation de l'approche, une évaluation approximative de l'effort de développement pourra être utilisée comme base pour le respect de ce critère.

#### 1.2.1.3 Maintenabilité

Les projets impliqués dans les travaux d'expérimentations devront pouvoir continuer d'évoluer et d'être améliorés au fil des années. L'application de modifications, la correction d'erreurs et l'ajout de nouvelles fonctionnalités devront pouvoir être effectués dans un temps raisonnable, similaire à ce qui aurait été requis avec l'utilisation de technologies conventionnelles. Le temps d'apprentissage ne devra pas être trop élevé et les modifications ne devront pas entraîner de changements trop importants au niveau des modèles et des fichiers sources des applications.

## 1.3 Méthode utilisée pour la recherche

Les travaux couverts par ce mémoire se sont échelonnés sur plus de 2 ans. Ils ont été réalisés en lien avec la problématique, mais aussi avec l'objectif de répondre à des besoins concrets dans des projets réels. Suite à l'identification de la problématique de recherche, la première étape fut d'effectuer une revue de l'existant en matière de plateforme d'exécution, de méthodes de définition d'interface utilisateur, des approches existantes pour la génération d'interfaces utilisateurs et des architectures d'application Web riches. Ce n'est qu'après cette première phase que fut sélectionnée l'approche à expérimenter. Celle-ci a ensuite été détaillée et reprise à l'intérieur de deux études de cas afin d'en montrer la faisabilité.

### 1.3.1 Revue de l'existant

Cette partie consiste à effectuer un état de l'art sur les différents domaines rattachés à la problématique. Dans un premier temps, une synthèse des principales plateformes d'exécutions pour clients Web riches a été effectuée, suivi d'un cadre d'analyse pouvant êtres utilisé pour le choix d'une plateforme à l'intérieur d'un projet donné. Un état de l'art pour les méthodes de définition d'interface utilisateur a ensuite été réalisé, couvrant les approches connues pour le développement de tout type d'interfaces utilisateurs.

Nous avons alors identifié l'utilisation de modèles descriptifs en XML comme étant la méthode de définition la plus appropriée pour rendre indépendante d'une plateforme d'exécution la définition d'un client graphique. Une revue des dialectes existants a alors été effectuée couvrant du même coup les travaux déjà publiés en matière de définition d'interfaces utilisateur basés sur un modèle XML. Une comparaison détaillée de chacun de ces dialectes a été réalisée en fonction d'une grille de critères définie au préalable.

Puisque le développement d'un client Web riche ne se limite pas qu'aux interfaces graphiques, une étude des architectures types retrouvées avec chacune des plateformes a finalement été réalisée. Cette étape aura permis d'offrir une vue d'ensemble de tous les composants se rattachant au client, nous aidant ainsi à identifier les éléments devant êtres pris en considération pour la mise au point d'une approche de définition indépendante d'une technologie d'exécution de clients Web riches.

### 1.3.2 Conceptualisation d'une solution

Cette étape présente une synthèse des concepts couverts et la conceptualisation d'une solution pouvant, en théorie, répondre aux besoins en matière de définition d'interfaces utilisateurs indépendantes d'une technologie de présentation. Les choix de conception effectués visaient également à répondre à l'ensemble des objectifs décrits précédemment tout en respectant les exigences et choix technologies déjà établis pour les projets à réaliser. Une présentation de la technologie Spring, utilisée comme base pour la mise en pratique de l'approche présentée a ensuite été effectuée.

### 1.3.3 Études de cas

Les développements présentés dans les études de cas ont été réalisés entièrement par l'auteur de ce mémoire en lien avec ces travaux de recherches. L'expérimentation a été effectuée à l'intérieur de projets réels menés par le Centre de Recherche Informatique de Montréal (CRIM). Les prototypes développés sont ici des applications clientes génériques, prenant en entrée une description du client indépendante d'une technologie de présentation, pouvant êtres utilisés à l'intérieur d'applications concrètes. Les études de cas ont été réalisées en trois grandes étapes :

1. Analyse et conception du modèle indépendant de plateforme à l'aide d'UML,
2. Développement du prototype prenant en entrée une instance du modèle,
3. Développement des applications concrètes basées sur le prototype développé.

Un premier prototype de définition de client graphique indépendant d'une plateforme de présentation a été conçu dans le cadre d'un projet d'entrepôt de données pour le développement d'un client destiné à l'édition de données, de tables de correspondance et de configuration. Ce même prototype a pu être réutilisé intégralement dans un deuxième projet, cette fois destiné à l'alimentation d'une base de données rattachée à un portail d'entreprise. Pour ce prototype, le niveau de fonctionnalités requis n'était pas très élevé et seule une première implémentation, pour le format HTML classique, a été développée.

Une deuxième génération de modèle d'interface indépendante de plateforme, cette fois-ci beaucoup plus riche, a ensuite été mise au point et utilisée dans une application

destinée à la génération d'horaires d'écoles. Le client graphique basé sur ce nouveau prototype comportait une logique de présentation très élaborée et a permis d'explorer davantage les limites de l'approche sélectionnée.

Une évaluation des résultats a finalement été effectuée et les données obtenues ont été mesurées selon plusieurs critères, incluant une évaluation de la productivité pour chacune des plateformes. Une comparaison avec des projets similaires, mais basés sur des approches de développement conventionnelles a pu servir de base pour l'évaluation de ces résultats.

# CHAPITRE II

# LES PLATEFORMES D'EXÉCUTION POUR CLIENT WEB RICHES

## 2.1 Introduction

La plateforme d'exécution prend en charge le traitement et le rendu de l'interface utilisateur du client. Elle peut supporter un ou plusieurs langages et reçoit l'application sous forme de scripts (interprétés) ou de code octet (exécuté par une machine virtuelle). La plateforme d'exécution gère aussi les mécanismes de communication avec le serveur et offre aux applications des bibliothèques utilitaires et d'affichage. Elle agit comme cadre (framework) pour le développement de la partie client des applications. Les formats utilisés pour la distribution d'un client riche sont généralement spécifiques à une seule plateforme, ce qui fait qu'un client riche (sous forme de script ou de code octet) n'est pas portable d'une plateforme d'exécution à une autre.

Cette section présente une synthèse des principales plateformes d'exécution de client Web riche. Seules les technologies permettant l'exécution de client Web sous forme d'applications en ligne, donc sans installation préalable, seront abordées. Tout comme pour les clients riches conventionnels, les clients Web riches sont principalement dédiés aux ordinateurs personnels, ce qui explique l'orientation commune des technologies couvertes.

Les plateformes .Net, HTML/JavaScript, Flash, Java et Mozilla seront d'abord présentées. Un cadre d'analyse des technologies sera ensuite proposé pour en permettre la comparaison. Le tout se terminera par une réflexion sur le choix d'une plateforme pour le développement d'un client Web riche.

## 2.2 Synthèse des plateformes d'exécution

### 2.2.1 Microsoft .NET

La technologie .Net a été lancée en 2002 par Microsoft et est aujourd'hui présente sur les nouveaux PC équipés de Windows. Elle est principalement constituée d'une machine virtuelle (CLR, *Common Language Runtime*) et d'un cadre de développement. L'infrastructure pour le support et la définition des langages supportés par .Net (CLI, *Common Language Infrastructure*) est standardisée par l'organisation ECMA. Les langages VisualBasic, C++ et C# sont, entre autres, supportés. La technologie d'exécution et une grande part des bibliothèques faisant partie du cadre .Net sont propriétaires à Microsoft (Gibbons, 2002).

Cette technologie peut être utilisée pour le développement d'applications lourdes conventionnelles ou pour le développement d'applications Web s'exécutant côté serveur. La possibilité de distribuer des applications lourdes sous forme d'applet via le Web en fait une plateforme pour l'exécution de client riche.

Les interfaces d'un client riche .Net ont l'avantage de se conformer au guide de style de MS Windows en offrant le paradigme des fenêtres et les mêmes composants « widgets », ce qui contribue à en faciliter l'utilisation et l'apprentissage. Les applets sont distribués en format « bytecode » et exécuté à l'aide de du CLR, leur permettant ainsi d'être indépendants d'une architecture matérielle. Le « bytecode » est compilé avant exécution à l'aide d'un compilateur juste à temps « JIT », offrant un niveau de performance à l'exécution similaire à celui des applications Windows natives. L'exécution étant prise en charge par la machine virtuelle, le tout s'effectue dans un environnement dit « géré » contrôlant l'accès aux ressources et renforçant ainsi la sécurité.

La communication entre le client et le serveur peut s'effectuer de différentes façons. Puisque la technologie a la capacité de supporter la plupart des protocoles de communication IP existants, un très haut niveau de flexibilité est donc offert. La communication entre l'applet et le serveur se fait généralement à l'aide des services Web.

Le développement .Net repose principalement sur un outil : Visual Studio. Ce dernier offre des facilités de débogages avancées telles que la possibilité de suivre une exécution pas

à pas. Certains langages dont VB.Net permettent la modification du code en cours d'exécution, ce qui en fait un environnement bien adapté pour le développement rapide d'application. Les outils de développement Microsoft sont aussi réputés comme étant facile à apprendre.

La définition des interfaces utilisateur sous .Net est principalement effectuée à l'aide de la bibliothèque WinForms. Il s'agit d'une couche d'objets encapsulant les composants Win32 standard. De ce fait, une application .Net bénéficie des mêmes fonctionnalités qu'une application Windows conventionnelle en matière d'interface riche : fenêtres, contrôles, menus, glisser-déplacer, etc. L'affichage peut être modifié en cours d'exécution. On peut définir une interface WinForm par programmation (à la manière d'une interface Swing du monde Java) ou à l'aide d'un éditeur visuel enregistrant les composants de l'interface et leurs propriétés sous forme d'un fichier de définition.

Les interfaces définies sous .Net restent rattachées à l'environnement Microsoft et peuvent difficilement s'exécuter ailleurs que sous Windows, limitant ainsi la compatibilité avec d'autres systèmes. Une alternative au framework .Net connu sous le nom de Mono offre une implémentation libre de WinForms qui n'est toutefois pas mature à l'heure actuelle. De plus, bon nombre de bibliothèques restent une propriété de Microsoft et n'offrent pas d'implémentations alternatives pour le moment.

La technologie .Net est à la base de la stratégie qu'entend mettre de l'avant Microsoft au cours des prochaines années. Le framework Avalon, aussi connu sous le nom de WPF (Windows Presentation Foundation), remplacera les interfaces MFC actuelles. WPF propose un moteur de rendu graphique plus avancé et intégrant des fonctions de 2-D, de 3-D, de multimédia, de transparence, et d'animation des interfaces. Le tout basé sur XAML, un langage de définition d'interface utilisateur développé par Microsoft. Cette nouvelle génération de client, toujours basé sur .Net, pourra être distribuée par le Web, en faisant ainsi une technologie d'exécution de client Web riche des plus complètes. Le produit Microsoft Expression permet la composition d'animations et de contenu multimédia similaire à Flash et pouvant être exécuté sous .Net. Le framework Indigo, devant aussi accompagner Windows

Vista, propose une méthode unifiée pour la communication entre les paliers d'une application basée sur les services Web (David, 2005).

### 2.2.2 HTML/JavaScript (AJAX)

Le terme AJAX (*Asynchronous JavaScript and XML*) est apparu pour la première fois au début 2005. Il repose sur des technologies utilisées depuis longtemps soit HTML, DOM et JavaScript. Un client riche AJAX est pris en charge directement par le navigateur. Il prend la forme d'une page HTML dont une partie de la logique de présentation est codée en JavaScript. Les scripts contenus dans la page peuvent interagir directement avec le serveur Web à l'aide d'une bibliothèque permettant l'exécution de requêtes HTTP. On obtient donc comme résultat une application JavaScript capable d'envoyer et de recevoir des données à un serveur Web et de mettre à jour l'interface de l'utilisateur en modifiant le DOM (*Document Object Model*) de la page HTML. En y ajoutant les possibilités d'interface HTML dynamiques issues de DHTML, le navigateur constitue une plateforme complète pour l'exécution d'un client Web riche (Zakas, McPeak et Fawcett, 2007).

Le principe de la communication via JavaScript sur lequel est basé AJAX est utilisé par des applications Web depuis déjà plusieurs années. C'est le cas du client Web de Microsoft Exchange qui exploitait une technique similaire dès la fin des années 1990. Le regain de popularité que connaît aujourd'hui cette technologie est en partie attribuable à la normalisation des navigateurs Web et à la sortie de plusieurs produits, dont Google Maps et Gmail. L'apparition de nombreux cadre de développement, bibliothèques et outils étiquetés AJAX a aussi eu pour effet d'alimenter l'engouement suscité par cette technique.

La communication asynchrone permet à l'utilisateur de poursuivre la navigation lorsqu'un accès au serveur est en cours. Il peut, par exemple, lancer plusieurs actions simultanément à différents endroits de l'application. Les sections concernées seront mises à jour lors de la réception de leurs données respectives provenant du serveur. Cette situation est rendue possible grâce au fait que l'interface utilisateur reste chargée et se met à jour au lieu d'être complètement régénérée. Avec une application Web conventionnelle, chaque interaction implique le transfert et l'affichage d'une nouvelle page, ce qui bloque l'interface le temps du traitement.

Les langages et technologies utilisés pour le développement des interfaces Web sont des plus ouvertes. Les navigateurs Web supportent principalement HTML et CSS pour la définition des interfaces : deux langages standardisés par l'organisme W3C. DOM, XML et plusieurs autres technologies reliées pouvant être exploitées dans un client riche HTML relèvent aussi de cet organisme. Pour sa part, le langage JavaScript, utilisé pour l'exécution d'opérations côté client, est aussi une norme ouverte standardisée par l'association ECMA sous le nom d'ECMAScript. En reposant sur des normes ouvertes, AJAX se trouve à être la technologie la plus portable puisqu'elle n'exige qu'un navigateur Web, ce qu'on retrouve sur pratiquement tous les systèmes d'exploitation. L'exécution correcte d'un client AJAX dépend toutefois de la compatibilité du navigateur avec les technologies utilisées (Paulson, 2005).

Les bibliothèques permettant l'exécution de requêtes HTTP à l'aide de JavaScript ne sont pas normalisées. Celles-ci peuvent différer d'un navigateur à l'autre, ce qui nécessite le traitement de cas particulier en fonction du navigateur utilisé (Zakas, McPeak et Fawcett, 2007). Un groupe de travail du W3C, le WebAPIs, est en charge de faire de l'objet XMLHttpRequest un nouveau standard pour pallier à cette lacune.

Le langage HTML intègre plusieurs fonctionnalités permettant la composition d'interface utilisateur. Ce langage, d'abord conçu pour la définition de document texte avec hyperliens, s'est vu ajouter plusieurs composants favorisant le dynamisme. Ce sont les fonctionnalités issues de DHTML qui permettent l'interaction avec une page HTML. Elles consistent en l'ajout d'attributs tel que « onclick » ou « onmouseover » permettant de prendre en charge des événements déclenchés par l'utilisateur. Ces événements peuvent être traités à l'aide de JavaScript et entraîner des modifications à la page. Les éléments « div » combinés aux techniques de positionnement de CSS permettent la superposition de fragments de contenu à l'aide de couches. L'ensemble de ces fonctionnalités permet de simuler la plupart des fonctionnalités d'une interface riche telle que les fenêtres, les menus ou le glisser-déplacer.

Un client Web riche AJAX peut-être codé directement à l'aide des langages HTML et JavaScript. Le développement d'application en JavaScript est souvent déconseillé à cause des

faiblesses au niveau du débogage et du fait que le comportement du code peut être différent d'une implémentation (navigateur) à l'autre.

Il existe une multitude de technologies permettant de simplifier le développement et le déploiement d'un client AJAX. Parmi les outils de développement, notons le AJAX ToolKit d'IBM intégrant à Eclipse des fonctionnalités de débogage avancé pour du code JavaScript ainsi que le support pour plusieurs bibliothèques AJAX. Il y a aussi la solution de Sun basée sur Java Studio Creator et JSF et offrant une approche orientée composant (*Widget*) pour la composition de pages AJAX.

L'entreprise TIBCO (tibco.com, 2007) offre un environnement de développement entièrement Web conçu spécifiquement pour le design d'interfaces AJAX. L'IDE offre aussi des facilités d'édition et de débogage du code JavaScript. Une particularité de cet outil est qu'il génère un fichier XML de définition d'interface qui sera interprété par le « TIBCO *General Interface framework* » du côté du serveur Web. Le serveur est alors responsable de connaître l'état du client Web et d'interroger les ressources (serveurs d'applications, services Web) pour alimenter le client Web.

Le framework ZK (zkoss.org, 2007), distribué sous licence libre, utilise un principe similaire. Ce dernier va encore plus loin en éliminant tout recours au langage JavaScript lors de la phase de développement. Les interfaces ZK doivent être codées en XML et les actions en Java. Le serveur ZK génère une page HTML/JavaScript pour le client et conserve la responsabilité d'exécuter les actions de l'utilisateur, mais la page générée est mise à jour de façon asynchrone à la AJAX.

Le Google Web Toolkit (code.google.com, 2007) est aussi un framework pour le développement d'application Web AJAX en Java. Il est distribué sous licence libre (Apache Licence 2.0) et permet la génération des interfaces et du JavaScript depuis du code Java conventionnel. Pour la définition des interfaces, il suffit d'instancier une structure d'objets ainsi que leurs propriétés, à la manière d'une interface Swing ou AWT. Le tout est exécuté par le serveur qui gère la communication avec le client tout en conservant l'état de l'interface côté serveur.

Le développement par composant d'une interface AJAX implique l'utilisation de bibliothèques existantes contenant des composants ou « widgets » réutilisables. Les produits Yahoo! *User Interface*, Dojo, DHTMLGoodies et Scriptaculous offrent de tels composants JavaScript pouvant s'intégrer à différentes applications. Ceux-ci se veulent compatibles avec les navigateurs les plus répandus. Leur utilisation permet de limiter la quantité de code à écrire et ainsi éviter les erreurs.

Les clients Web HTML/JavaScript sont interprétés et non exécutés. Cette situation peut influencer la performance lorsqu'on est en présence d'une grande quantité de code JavaScript. Concernant la sécurité, les règles en vigueur sont celles du navigateur. Ce dernier est en charge d'autoriser les opérations lors de l'interprétation du script.

Parmi les désavantages rattachés à l'utilisation d'AJAX, notons que JavaScript, bien que basé sur une norme ECMA, peut offrir un comportement variable d'un navigateur à l'autre. Le code développé devrait donc être testé sur plusieurs environnements avant de pouvoir être considéré comme pleinement fonctionnel. De plus, la modification par l'utilisateur des paramètres de sécurité de son navigateur peut influencer l'exécution du client. En tenant compte de tous les environnements pouvant être utilisés, le nombre de combinaisons différentes s'en trouve très élevé. L'utilisation de fonctionnalités AJAX dans une page limite aussi ses possibilités d'affichage par certains médias comme les PDA (*Personal Digital Assistant*) ou en mode accessibilité, pour les non-voyants.

Un autre argument en défaveur d'AJAX est le fait que la métaphore de la page, utilisée dans le Web conventionnel, n'est pas appropriée pour l'exécution d'une application. La commande « Précédent » bien connue des utilisateurs perd alors son sens et son utilisation peut entraîner des problèmes au niveau du lien entre le client et le serveur (Garrett, 2005).

Les technologies à la base des clients AJAX continueront d'évoluer. Le WebAPIs Work Group du W3C a pour mandat de standardiser l'objet XmlHttpRequest, qui sera bientôt supporté sur la plupart des navigateurs Web. La version 3 de CSS, toujours en préparation, offrira des fonctions plus avancées pour le formatage des pages comme la gestion des colonnes et l'alignement vertical. Cette version permettra aussi des formats de texte avancés

et davantage de possibilités pour les arrières plans et les bordures. Du côté de JavaScript, une version nommée ECMAScript 4 fait l'objet de propositions depuis l'année 2000. Cette version, aussi connue sous le nom de JavaScript 2, vise à se rapprocher des autres langages de programmation objet en se rapprochant de Java, C# et Python. Il intégrera, entre autres, le concept de package, les constructeurs, les itérateurs, les interfaces et corrigera plusieurs failles de JavaScript, notamment au niveau de la gestion des types (Eich, 2006).

Le problème avec l'évolution des technologies des navigateurs Web est que leur temps de pénétration est très élevé. On doit compter un délai de 3 ans entre le moment où la technologie est lancée et le moment où elle peut commencer à être utilisée.

### 2.2.3 Flash

La technologie Flash a été lancée en 1996 par Macromedia. Elle prend la forme d'un connecteur (plug-ins) intégré au navigateur Web. Les premières versions furent axées sur la présentation d'animation pour les pages d'ouverture de sites Web « Splash Screen ». Un outil d'édition servait alors à la conception et à la compilation de l'animation qui était ensuite téléchargée et exécutée par le connecteur. C'est en 2000 qu'on vit apparaître les premiers mécanismes de génération de contenu Flash par un serveur permettant ainsi l'affichage de contenu dynamique. Au cours de la même période, le langage ActionScript fut intégré à la plateforme. Celui-ci, calqué sur ECMAScript, permit de compléter la solution pour le développement de clients intelligents. Flash est aujourd'hui reconnu comme un joueur important pour l'exécution de client Web riche (Hosea, 2006).

Le taux de pénétration de la technologie Flash, supérieur à 98% (Brown, 2007), en fait le connecteur internet le plus répandu. Le format Flash est contrôlé par la firme Macromedia (maintenant propriété d'Adobe). L'entreprise développe et distribue aussi son connecteur Flash intégrable aux navigateurs. Le format de fichier étant public et disponible auprès de Macromedia, d'autres entreprises peuvent distribuer leur propre lecteur Flash ou leur éditeur de contenu produisant des fichiers compatibles avec cette technologie. Le fait qu'un seul fournisseur ait le contrôle sur les spécifications permet de conserver un haut niveau d'uniformité entre les systèmes, ce qui réduit considérablement les problèmes d'incompatibilité.

Les fichiers SWF sont compilés dans un format bytecode. Le lecteur agit donc comme une machine virtuelle exécutant le fichier en fonction du système. Il existe des lecteurs compatibles avec la plupart des systèmes d'exploitation (Windows, Mac, Unix). L'exécution étant contrôlée par la machine virtuelle, la sécurité s'en trouve renforcée puisque celle-ci s'occupe de gérer l'accès aux ressources du système.

La technologie offre un grand nombre de fonctionnalités, principalement au niveau graphique. Elle permet la transparence, le traitement d'images, les animations, la sonorisation et le glisser-déplacer en plus des contrôles traditionnels retrouvés dans les formulaires comme les menus déroulants ou les champs de saisies. La technologie Flash MX supporte aussi plusieurs formats de compression vidéo, ce qui permet d'en faire un lecteur multimédia. De nombreux exemples du potentiel de Flash en matière d'interface utilisateur peuvent être tirés du domaine du jeu vidéo en ligne.

Au moins deux plateformes distinctes permettent de concevoir et de desservir des clients Web riches. Il s'agit de Flex, la plateforme propriétaire d'Adobe, et Open Laszlo, une alternative ouverte dont le développement est principalement assuré par l'entreprise Laszlo System.

Open Laszlo (laszlosystems.com) se greffe à un serveur d'application pour générer des interfaces utilisateur en format Flash. Il utilise un langage XML pour la définition des interfaces, soit LZX, et une version réduite de JavaScript permettant de coder le comportement du client. Le fichier SWF transmis au client peut être généré une seule fois dans une version autonome, statique, ou être généré dynamiquement en fonction de la session. Après avoir été chargé par le connecteur Flash, le client peut communiquer avec le serveur Laszlo par le protocole HTTP ou accéder directement à un serveur d'application à l'aide de SOAP ou XML-RPC. Un outil de développement développé par IBM et intégré à Eclipse permet le développement rapide d'interface Laszlo et prend en charge des fonctions de débogage avancées lors de l'exécution du client.

Flex de Macromedia est venu en réponse à Laszlo. Il permet aussi la génération de client SWF à l'aide d'un langage de définition XML, MXML, et l'exécution de code à l'aide

d'ActionScript, similaire à JavaScript. Le serveur Flex se base sur la technologie J2EE et supporte JSP pour la génération des clients. Il est rattaché à un serveur d'application ColdFusion. La communication entre le client et le serveur d'application s'effectue principalement à l'aide des services Web et d'un format d'échange propriétaire utilisant toujours HTTP. La version 2 de Flex et de son langage ActionScript 3 se rapproche d'un langage orienté objet de haut niveau en supportant les packages, l'héritage, les interfaces et un typage fort. Les outils de développement disponibles permettent le débogage avancé et l'édition visuelle (WYSIWYG). Ces derniers sont maintenant intégrés à l'environnement de développement Eclipse (Anderson, 2005).

Parmi les limites de cette technologie, notons le fait que l'exécution d'un client Flash consomme davantage de ressources, tant au niveau de la mémoire que de la puissance de calcul, sur les postes de travail. Son temps chargement est aussi plus élevé que celui d'un client HTML conventionnel. Avec l'augmentation de la performance des réseaux et des PC, cette situation se trouve toutefois à être de moins en moins un problème.

Les premières versions de Flash et de ses outils de développement ne supportaient que la métaphore du scénario d'animation ce qui ne convenait pas au développement d'applications. Pour cette raison, Flash conserve toujours la réputation d'être un accessoire pour la présentation de contenu multimédia ou d'animations (Noda et Helwig, 2005).

Cette technologie est parmi les mieux positionnées actuellement pour diffuser du contenu multimédia au travers le Web. L'entreprise pourra profiter de sa présence sur le Web pour venir répondre à de nouveaux besoins. Le virage pris par Macromedia avec Flex et ActionScript montre clairement que le développement et la distribution d'applications Web riches sont au cœur de sa stratégie. Notons aussi une convergence entre les applications de format Flash et HTML puisque les versions 2 de Flash et 3.2 de Laszlo supportent tous deux la génération de client riche AJAX depuis les mêmes fichiers de définition XML.

### 2.2.4 Java

La plateforme Java, développée par Sun au début des années 1990, avait comme premier objectif de permettre la distribution, au travers un réseau, d'applications pouvant être

exécutées sur n'importe quel système d'exploitation. Elle fit son apparition comme technologie Internet en 1994 avec HotJava, un navigateur Web de Sun, pour ensuite être intégrée à Netscape un an plus tard. La machine virtuelle Java (JVM) présente aujourd'hui un taux de pénétration de près de 90% (thecounter.com) et est fréquemment utilisée pour ajouter des fonctionnalités avancées aux interfaces Web sous forme d'applets.

Un client Java permet l'exécution d'interfaces riches offrant autant de fonctionnalités qu'un client natif conventionnel. De plus, l'application client Java bénéficie de toutes les possibilités de la plateforme telle que la puissance de l'orienté objet et de Java EE pour l'accès aux bases de données et la sécurité. Cette technologie peut être utilisée pour le développement de client lourd selon le modèle 2-Paliers (*2-Tiers*), mais offre aussi tout le nécessaire pour le développement de client léger pourvu d'une interface riche et distribuée par le Web sous forme d'applet. Différents découpages sont donc possibles entre le traitement effectué par le client et celui effectué par le serveur (Domenig, 2006).

Un client Java est compilé dans un format intermédiaire, le bytecode, qui est exécuté par la JVM. Cette dernière peut procéder à l'exécution, soit avec un interpréteur, soit en code natif en y effectuant une compilation juste à temps (*Just In Time*). La sécurité est assurée par le fait que l'exécution est effectuée dans un environnement contrôlé et que les accès au système d'exploitation sont gérés par la machine virtuelle. Un applet peut être signé numériquement à l'aide d'un certificat de sécurité assurant l'authenticité du client téléchargé.

Java est à la fois une plateforme d'exécution et un langage autour duquel il existe plusieurs normes. L'organisme en charge de la standardisation et de l'évolution de la technologie Java est le JCP (Java Community Process). Les normes et recommandations sont mises au point par des groupes d'experts provenant de différents milieux. Celles-ci sont ouvertes et peuvent généralement compter sur différentes implémentations, à la fois libre et propriétaire. La machine virtuelle la plus utilisée est celle de Sun distribuée gratuitement. IBM, BEA et Apache ont aussi leur machine virtuelle Java. On retrouve des machines virtuelles Java sur la plupart des systèmes d'exploitation.

Une des forces de la plateforme Java se trouve au niveau des outils de développement. En effet, plusieurs outils permettent le développement d'applications et d'interface Java. Notons, entre autres, JBuilder de Borland, NetBeans de Sun, Eclipse d'IBM et JDeveloper d'Oracle. Ces outils intègrent tous des fonctionnalités évoluées simplifiant le développement tel que la complétion de code ou la possibilité de tracer l'exécution d'un programme.

La définition d'une interface Java se fait à l'aide d'objets encapsulant les éléments qui la composent. Il existe trois bibliothèques permettant la définition d'interfaces : AWT, SWING et SWT. AWT fut de la première génération. Il exploite directement les composants offerts par le système d'exploitation, ce qui le rend léger et performant. En n'offrant que les composants qui sont disponibles sous toutes les plateformes, les possibilités d'interface sont toutefois limitées et le rendu peut différer d'une plateforme à l'autre. SWING est purement Java et son rendu ne dépend pas du système d'exploitation. Il offre davantage de possibilités et un guide de style uniforme sur toutes les plateformes. Son exécution est toutefois plus lente puisqu'il n'utilise pas les composants natifs. SWT, développé par IBM et utilisé dans le projet Eclipse, se veut un compromis entre les deux. Il utilise les composants natifs lorsque possible et génère lui-même les composants manquants. Il offre un rendu plus uniforme entre les plateformes qu'AWT et une meilleure performance que SWING. Alors qu'AWT et SWING sont standardisés et font partie de la JVM, SWT nécessite l'installation d'une bibliothèque native, ce qui limite ses possibilités de distribution par le Web.

Différentes méthodes de génération d'interface sont disponibles en Java. AUIML d'IBM (Arhelger, Hanson et Erwin, 2004) offre un langage de définition d'interface indépendante de plateforme basé sur XML et permettant, entre autres, la génération de client Java. SwiXML et XSWT offrent des dialectes XML, respectivement pour la définition d'interface SWING et SWT.

D'autres technologies proposent des techniques spécialisées pour la définition de client léger Java pour le Web. C'est le cas de Canoo (canoo.com) dont l'architecture permet l'exploitation d'une interface Java communiquant avec un serveur Web. Le client ne

comporte que la logique de présentation et est étroitement lié à une partie serveur alimentant les listes de valeurs et exécutant les actions. Le produit Thinlet offre une stratégie similaire.

Parmi les désavantages de la plateforme Java, notons le fait que l'installation et la mise à jour de la JVM doivent être effectuées sur tous les systèmes des utilisateurs pour maintenir la compatibilité avec les clients développés. Un facteur ayant freiné la progression des applets Java est la performance. Le temps de chargement de l'environnement Java nécessaire au lancement d'un applet est élevé et les performances à l'exécution peuvent être décevantes selon le système utilisé. Des améliorations au niveau des versions les plus récentes de la JVM de Sun combinée à des méthodes de mise en cache des applets (Richardson, 2007) contribuent toutefois à améliorer la performance. Autre point, bien que plusieurs outils permettent de faciliter cette tâche, le développement d'une interface Java est plus complexe que le développement d'une page HTML par exemple.

L'amélioration des performances obtenue par l'augmentation des capacités physiques des postes de travail, l'optimisation effectuée sur la JVM et le support du multimédia devrait maintenir Java comme une solution complète pour l'exécution de clients riches (Chen et Ma, 2004). D'autres technologies reliées comme le Java Web Start de Sun continueront d'encourager le développement d'application de bureau sous cette plateforme (Richardson, 2007).

### 2.2.5 Mozilla

Le cadre d'exécution Mozilla (Mozilla Application Framework), aussi connu sous le nom de XPFE (Cross-Platform Front End) offre les outils pour la distribution et l'exécution d'applications multiplateformes. Il a été développé dans le cadre du projet Mozilla, mis de l'avant par Netscape avec l'objectif de développer une nouvelle génération de navigateur Web fondée sur les standards du Web et du code source libre. Ce cadre d'exécution est, entre autres, à la base des navigateurs Netscape 6 et ultérieurs, de la suite Mozilla, du navigateur FireFox et du client courriel Thunderbird. Il comprend plusieurs technologies XML pour la définition d'application et d'interface utilisateur. Le fait que ces technologies puissent être

exploitées en ligne, à la manière d'une page Web, fait du cadre d'exécution de Mozilla une plateforme d'exécution de client Web riche (McFarlane, 2004).

Les technologies de définition d'application livrées avec le cadre d'exécution sont propres à Mozilla et ne sont actuellement pas exploitables sous d'autres plateformes. Elles sont donc réservées aux navigateurs basés sur Mozilla. Ces derniers occupent aujourd'hui une part de marché variant entre 10 et 15%, reposant principalement sur FireFox. Ce navigateur est disponible sur la plupart des systèmes d'exploitation, dont Windows, Max OS et Linux.

Le langage XUL (XML User-Interface Language) est utilisé pour la définition des interfaces utilisateur. Ce dialecte XML renferme tous les éléments retrouvés dans une interface riche conventionnelle tels que le concept de fenêtre, les menus et tous les autres composants « widgets » habituels. Un fichier XUL renferme le contenu, la structure et le formatage de l'interface. Il permet l'utilisation du standard CSS2 pour l'habillage des écrans.

Un autre dialecte XML, XBL, pour eXtensible Bindings Language, permet la définition de composants réutilisables à l'intérieur des documents XUL. Le dialecte XBL permet aussi l'héritage entre les composants ainsi que les propriétés et les méthodes. Le langage utilisé pour définir le comportement est JavaScript, le même que pour les pages DHTML. Pour faire le lien entre les éléments d'interface XUL et le comportement défini par XBL, CSS est utilisé.

L'exécution de l'application Web via un navigateur compatible Mozilla se fait en mode interprété; c'est-à-dire que le navigateur charge les fichiers texte / XML pour ensuite effectuer le traitement approprié. La sécurité est aussi gérée par le navigateur. Mozilla ne permet pas à une application XUL d'accéder aux ressources systèmes. Pour ce faire, cette dernière doit être signée à l'aide d'un certificat de sécurité ou installée localement sur le système. Le format XPI permet l'empaquetage de connecteur et facilite leur installation sur un système Mozilla. Il permet l'intégration de nouvelles applications à l'environnement de l'utilisateur. Une fois installées, ces applications peuvent être exécutées en mode déconnecté.

La communication avec un serveur Web peut être effectuée à l'aide de la bibliothèque XmlHttpRequest. Elle permet d'utiliser les méthodes traditionnelles de communication avec

le serveur d'application grâce aux paramètres HTTP. XPCOM, un composant du framework, comprend plusieurs autres bibliothèques de communication. Ces dernières ne sont toutefois disponibles qu'aux applications installées localement.

Le développement d'une application XUL/XBL/JavaScript peut se faire sans outils avancés. Les fichiers peuvent être lancés à l'intérieur d'un navigateur ou à l'aide de l'application XULRunner. Mozilla offre aussi des outils avancés pour le débogage de code JavaScript et pour la validation syntaxique des documents créés. Des outils comme XULMaker offrent des interfaces graphiques pour l'édition des fichiers XUL. L'utilisation des langages XUL, XBL et JavaScript encourage le découpage de l'application en couches, ce qui en facilite la maintenance (McFarlane, 2004).

Le principal désavantage relié à la publication d'applications Web dans un format Mozilla est que son taux de pénétration est faible (<15%), ce qui en limite grandement la portée. Sa part du marché connaît toutefois une augmentation rapide, ce qui fait que son utilisation pour la distribution d'applications deviendra de moins en moins un problème. Un autre désavantage rattaché à la plateforme est que l'accès à la plupart de ses bibliothèques systèmes XPCOM sont proscrites pour les applications en ligne. Un paquetage d'installation doit alors être téléchargé et installé localement, ce qui va à l'encontre du concept d'application Web.

Un groupe de travail du W3C, le Web Application Formats (Web Application Formats Working Group, 2006), s'affère à standardiser les langages XUL et XBL. Ils ont pour mandat de proposer une nouvelle norme pour la définition des interfaces riches. La plateforme Mozilla, agissant déjà comme implémentation de référence pour ces langages, sera sans doute la mieux positionnée pour le support de cette norme. Un autre aspect positif par rapport à l'utilisation de XUL est l'augmentation des parts de marché de FireFox, ce dernier ayant connu une croissance remarquée au cours de l'année 2005. De plus, certaines technologies AJAX comme ZK générant aujourd'hui des interfaces riches HTML planifient le support prochain de XUL comme format de sortie.

## 2.3 Cadre d'analyse pour le choix d'une plateforme

Cette section présente une liste de critères de comparaison ainsi que leurs justifications qui permettront de faire ressortir les différences entre les plateformes. L'étude et la comparaison de ces technologies permettront de sélectionner la meilleure plateforme pour un développement donné. Les critères sont divisés en cinq catégories :

1. Caractéristiques du client,
2. Facilité de développement,
3. Déploiement,
4. Adoption de la plateforme et
5. Aspect commercial.

Ces critères visent la couverture de l'ensemble des paramètres permettant de différencier les plateformes. Il s'agit d'une synthèse d'articles et de cadres de comparaison (Noda et Helwig, 2005), (Domenig, 2006), (Wroblewski et Ramirez, 2006), (O'Rourke, 2004) complétée par les informations recueillies lors de l'étude des plateformes.

### 2.3.1 Caractéristiques du client

**Fonctionnalités disponibles**

Les fonctionnalités disponibles pour la confection d'interface riche comprennent, entre autres, le glisser-déplacer, les menus, les menus contextuels, le concept de fenêtre, la navigation à l'aide du clavier, l'utilisation de composants (*Widgets*), la possibilité de manipuler les composants par programmation et la présence de méthodes graphiques avancées comme la transparence ou le traitement d'image. La présence de ces fonctionnalités est importante selon le type d'application et le type d'interface recherché.

**Composants disponibles**

Il s'agit des composants (widgets) traditionnels comme les boutons, les listes, les menus déroulants, les boîtes de dialogues et les barres de défilement. Les plateformes offrent généralement une gamme de composants natifs pouvant être réutilisés par les clients.

**Support de l'accessibilité**

Il s'agit de savoir si la plateforme facilite l'utilisation de mécanismes favorisant l'accessibilité telle que le changement de polices, les lecteurs d'écrans (screen readers) ou les transcripteurs braille. Le support des fonctions d'accessibilité est souvent une exigence pour le développement de systèmes grand publique.

**Moteur de rendu graphique**

Un moteur de rendu graphique avancé est nécessaire pour le support d'animations ou pour le traitement d'image. Il permet le traitement ainsi que l'optimisation de ces fonctionnalités.

**Support pour le multimédia**

L'intégration de fonctionnalités multimédias à une technologie de client Web riche permet d'en simplifier l'intégration aux interfaces. Un support natif du multimédia facilite la publication de ce type de contenu.

**Mode d'exécution**

L'application peut être soit sous forme de script interprété, soit sous forme de code octet (bytecode). La compilation du client sous forme de code octet permet, entre autres, d'augmenter la robustesse du client en assurant un comportement uniforme sur les machines virtuelles compatibles. La distribution de l'application sous forme de code source devant être interprété laisse ce dernier vulnérable aux variations dans l'implémentation des clients.

**Robustesse**

La robustesse du client peut être influencée par différents facteurs, dont le mode d'exécution, les technologies d'exécution utilisées, la performance, le traitement des entrées ou les techniques de communication. La présence de certaines fonctionnalités au niveau de la plateforme d'exécution comme les options « précédent », « suivant » et « rafraîchir » peuvent influencer négativement la robustesse de la plateforme.

### Sécurité

La sécurité est dépendante de la plateforme d'exécution. Cette dernière se charge de donner accès aux ressources systèmes. Une exécution dans un environnement contrôlé (sandbox) permet de renforcer la sécurité.

### Puissance requise par le client

Ce critère peut s'avérer important selon la clientèle visée et le type d'application dont il est question. Certaines plateformes peuvent consommer beaucoup plus de mémoire et de capacité de traitement que d'autres.

### Temps de chargement du client

Le temps de chargement dépend de la plateforme d'exécution et peut avoir à être pris en considération, toujours selon le type d'application desservie. Un temps de chargement plus élevé peut s'avérer acceptable dans le cas d'une application d'envergure, mais peut devenir un irritant pour l'affichage d'une page utilisée très brièvement.

### Technologie nécessaire côté client

Est-ce qu'une installation doit être effectuée sur les postes de travail ? Si l'installation ou la mise à jour d'un connecteur est requise à l'exécution du client riche, cela peut avoir pour effet d'en limiter la portée.

### Méthodes de communication

Les clients peuvent supporter différentes méthodes de communications avec le serveur. Le support de l'envoi de données initié par le serveur où « server push » permet le support de nouvelles fonctionnalités comme les alertes ou la présentation de données en temps réel. La communication asynchrone entre le client et le serveur permet à l'utilisateur d'effectuer simultanément des actions sur différents composants de son interface. L'utilisation des services Web permet la mise en place de méthodes d'accès standardisée avec le serveur d'application, permettant ainsi leur utilisation avec plusieurs technologies différentes.

**Systèmes d'exploitation supportés**

Certaines plateformes sont dépendantes d'un système d'exploitation. Ce facteur doit donc être pris en considération lorsque l'application est destinée à être accédée par un grand public. La percée du système d'exploitation Linux du côté des ordinateurs personnels ne fait que renforcer la nécessité de supporter plusieurs systèmes.

### 2.3.2 Facilité de développement

**Complexité du développement**

Ce point couvre, entre autres, l'expertise nécessaire au développement d'un client riche sous une plateforme donnée. La complexité du langage et les possibilités de découpage de l'application pour le travail en équipe peuvent influencer ce critère. Une plateforme peut être appropriée pour le développement de clients simples, mais rendre très complexe le développement d'un client d'envergure. Les facilités pour le repérage des erreurs et pour l'exécution de tests doivent aussi être pris en considération.

**Outils disponibles**

Les outils disponibles pour le développement peuvent faire en sorte de simplifier et d'accélérer le processus de développement. Les outils peuvent offrir des facilités au niveau du débogage et des fonctionnalités de complétion automatique du code. La disponibilité d'outils de développement avancé est donc un avantage pour une plateforme d'exécution.

**Méthodes de définition des interfaces**

Les interfaces riches peuvent être définies par programmation, avec un langage descriptif ou à l'aide d'un outil de développement visuel. Une interface définie par programmation peut être plus complexe à gérer et à maintenir qu'un fichier de définition d'interface. Dans certains cas, l'utilisation d'un langage de définition spécialisé permettra l'interopérabilité entre plusieurs plateformes.

### 2.3.3 Déploiement

**Technologies requises côté serveur**

Certaines plateformes d'exécution nécessitent l'utilisation d'une technologie particulière du côté du serveur. Le choix d'une plateforme peut donc être influencé par les choix technologiques effectués au niveau du serveur d'application.

**Puissance requise par le serveur**

Le traitement à effectuer au niveau du serveur dépend de la technologie utilisée. Pour certains clients Web, le serveur devra assumer la génération de l'interface. Dans d'autres cas, le serveur maintiendra en mémoire l'état de l'interface côté client et sera responsable de compiler et de retourner les mises à jour appropriées. Cette variable est donc importante et devra être évaluée en fonction du nombre de connexions simultanées prévues pour le serveur Web.

**Consommation de bande passante**

La taille du client téléchargé, la taille des messages échangés et le nombre de requêtes effectuées sont des facteurs qui influencent la consommation de bande passante. Une consommation trop élevée peut entraîner des coûts supplémentaires chez le fournisseur et une dégradation des performances chez le client. Aussi, l'exploitation d'un client dans un environnement où le taux de latence est élevé peut aussi influencer l'utilisabilité de l'interface riche.

### 2.3.4 Adoption de la plateforme

**Taux de pénétration**

Le taux de pénétration d'une technologie dans un marché se traduit par le pourcentage des systèmes sur lesquels la technologie est présente. Une application destinée au grand public devra reposer sur des technologies répandues alors qu'une application destinée à un groupe d'usagers spécifiques pourra utiliser des technologies moins conventionnelles nécessitant une installation.

**Développeurs**

Le bassin de développeurs spécialisés sous une technologie est généralement proportionnel à son adoption par l'industrie. Avec une technologie répandue, il sera plus facile de trouver des ressources spécialisées pouvant accomplir le développement et la maintenance de système. Au contraire, il sera très difficile de recruter des experts pour une technologie émergente.

**Support**

Il s'agit d'évaluer la présence et la qualité du support pour la plateforme. Une plateforme peu répandue et non supportée peut être à éviter. Pour les projets en logiciel libre, la qualité du support peut être évaluée par la taille de sa communauté.

**Technologies reliées**

La quantité de technologies reliées ou exploitant une plateforme d'exécution est aussi un signe quant à son adoption. Cela inclut les cadres de développement et les outils permettant le développement et le déploiement des clients. La présence de plusieurs alternatives technologiques pour le développement sous une plateforme évite aussi d'être lié à un seul fournisseur.

### 2.3.5 Aspect commercial

**Utilisation de normes ouvertes**

L'utilisation de langages basés sur des normes ouvertes pour la définition et le développement des clients riches permet, en théorie, l'interopérabilité entre des plateformes de différents fournisseurs supportant ces mêmes normes. À l'inverse, l'utilisation de normes propriétaires aura pour effet d'attacher le développement à un seul fournisseur.

**Coûts**

Les coûts rattachés à la plateforme d'exécution, mais surtout à l'infrastructure requise côté serveur peuvent influencer les choix technologiques. Certaines plateformes propriétaires coûtent plus de 15 000$ par CPU comparativement à la gratuité pour plusieurs produits issus du monde du libre.

Ces critères peuvent être utilisés à l'intérieur de la démarche de sélection d'une plateforme pour un projet donné. Après avoir identifié les besoins en matière de client riche, chacun des critères mentionnés devrait être pondéré en fonction des priorités. Le choix de la meilleure plateforme pourra donc varier en fonction du projet.

## 2.4 Conclusion

Cette section aura effectué un survol des principales plateformes d'exécution pour les clients Web riches. Celles-ci offrent des fonctionnalités similaires, mais ont chacune leurs particularités et des avantages qui leur sont propres. Plusieurs critères entreront en ligne de compte dans le choix d'une technologie pour un projet donné et la meilleure plateforme peut être différente d'un projet à un autre. Un développement effectué sous une technologie spécifique ne peut généralement pas être porté vers une autre. Le choix effectué s'avère donc critique puisque les investissements rattachés au développement et à la programmation du client Web en dépendront.

Pour remédier à ce problème, une définition d'interface de plus haut niveau et indépendante d'une plateforme s'avère nécessaire. Les méthodes existantes pour la définition et le développement des interfaces utilisateurs font l'objet du prochain chapitre.

# CHAPITRE III

# LA DÉFINITION DES INTERFACES UTILISATEUR

## 3.1 Introduction

Cette section présente l'état de l'art en matière de définition et de développement des interfaces utilisateur. Dans un premier temps, les principaux concepts reliés à la conception des interfaces utilisateurs seront abordés. Il sera, par la suite, question de leur composition. Les techniques de définition, les approches de développement ainsi que les liens entre l'interface et l'application seront finalement traités.

## 3.2 Les modèles

Dans la conception d'interfaces utilisateurs, un modèle est une représentation de l'interface, ou d'un aspect de l'interface, comportant un certain degré d'abstraction. Alors que l'interface finale est principalement composée du code et de tous les détails permettant son exécution sur une plateforme spécifique, le modèle contiendra plutôt, sous la forme d'un graphique, d'une notation ou d'une description de haut niveau, les informations sur ce que l'on retrouve dans celle-ci.

La génération d'interface basée sur les modèles (ou *model-based*) signifie simplement qu'un modèle est utilisé pour la génération de l'interface finale (Souchon et Vanderdonckt, 2003). Différents types de modèles peuvent être inclus dans le processus de conception des interfaces utilisateur. Un modèle de tâche pour les interactions de haut niveau, un modèle de domaine pour représenter les objets et relations impliqués dans l'interface ou d'autres modèles plus spécifiques peuvent couvrir les différents niveaux d'abstraction des interfaces (Souchon et Vanderdonckt, 2003).

Plus le niveau de détails du modèle est élevé, plus la correspondance avec l'interface finale sera directe. Le modèle généralement utilisé à l'intérieur des dialectes de définition d'interface est le modèle d'interface concret. Celui-ci contient, toujours sous une forme descriptive, les détails sur la présentation et la disposition des composants de l'interface. Le terme « modèle » se référant, par définition, à une représentation simplifiée de quelque chose de complexe (Encarta, 2008), nous pouvons affirmer que plus le « modèle » est détaillé, moins le « modèle » est un « modèle ». Nous utiliserons ici le terme modèle pour désigner toute représentation se situant à un niveau d'abstraction plus élevé que celui du code source.

La conception des interfaces d'une application peut être réalisée à l'aide de diagrammes d'état ou « statecharts » (Horrocks, 1999). Cette notation permet la couverture de tout l'aspect comportement de l'interface. Elle présente le flux d'exécution entre les états, en fonction des actions déclanchées à l'intérieur de l'interface. Par exemple, pour un état donné, le diagramme représentera les actions possibles, pouvant êtres choisis par l'utilisateur. Le modèle de tâche peut, entre autres, être effectué à l'aide de cette notation. Ce type de diagrammes comporte les avantages d'être rapides à écrire et facilement compréhensibles. Il s'agit d'abord d'une méthode pour la conception axée sur la définition du problème. Le modèle pourra ensuite être réutilisé pour le développement de l'application de façon manuelle ou automatique. La réutilisation du modèle pour générer des tests est aussi une pratique courante. Des variantes de ce type de notation seraient les Java Page Flow, Spring Web Flow et les machines à état UML.

### 3.2.1 Le contexte

Le contexte représente l'état des variables ayant un impact sur l'interface devant être présentée. Ces variables correspondent, entre autres, à l'utilisateur, la plateforme et l'environnement (Souchon et Vanderdonckt, 2003), (Limbourg et al., 2004). À l'utilisateur peut être rattaché un profile composé de préférences, de paramètres et d'autorisations d'accès. La langue d'affichage peut aussi y être rattachée. Le support de plusieurs langages implique non seulement l'internationalisation des chaînes de caractères, mais aussi le changement du sens de l'écriture (ex : droite à gauche au lieu de gauche à droite).

La plateforme est définie comme toute combinaison logicielle et matérielle pouvant être utilisée pour l'exécution de l'application. Elle peut par exemple tenir compte de la résolution d'écran ou du système d'exploitation. L'environnement constitue le milieu physique dans lequel se déroule l'exécution. Il peut s'agir de l'éclairage ambiant, du nombre de personnes en cause, du niveau de stress de l'utilisateur... Toutes ces données sont susceptibles d'influencer l'interface qui devra être affichée.

Un langage de définition des interfaces dit sensible au contexte (context-sensitive) permet une adaptation en fonction du contexte. L'adaptation peut être effectuée lors de la génération de l'interface finale (Limbourg et al., 2004) ou au moment de l'exécution (Souchon et Vanderdonckt, 2003).

### 3.2.2 Le mode d'interaction

Une interaction représente un événement déclenché par l'utilisateur via l'interface et ayant un effet sur l'application (Eisenstein, Vanderdonckt et Puerta, 2001). L'interaction est composée d'une entrée (intervention de l'utilisateur) et d'une sortie (réaction de l'application).

Une application dite « multimodale » supporte plusieurs modes d'interaction (Souchon et Vanderdonckt, 2003). Il peut s'agir du crayon, de la souris, du clavier, de la voix ou de toute autre méthode agissant comme mode d'entrée. Un mode d'interaction peut exploiter un format de sortie différent (ex : une interface VoiceXML gère des entrées vocales, mais aussi une sortie vocale), ce qui implique une interface utilisateur différente.

## 3.3 La composition des interfaces

Les éléments suivants entrent dans la définition et dans la conception des interfaces utilisateur :

Les *contrôles* constituent le plus petit dénominateur défini dans une interface. Il s'agit généralement de composants natifs gérés par la plateforme et permettant l'entrée ou la sortie d'information (Guojie, 2005). Il peut prendre la forme, d'une zone de texte, d'un bouton,

d'une étiquette d'une barre de défilement, d'une fenêtre, d'une liste déroulante, de menus,... Les contrôles supportés peuvent varier d'une plateforme à une autre.

Les *composants* agissent comme des contrôles de plus haut niveau. Au lieu de représenter un élément simple, comme un champ texte, le composant peut assumer une part de la logique applicative. Des exemples peuvent être : un calendrier, une liste de fichiers, un contrôle ActiveX. Une approche de développement basée sur les composants vise la réutilisation et l'intégration de composants existants dans la formation des interfaces (Myers, Hudson et Pausch, 2000).

Le *jeu d'outils* (toolkit) définit l'ensemble des contrôles et composants disponibles sur une plateforme et pouvant être utilisés pour la construction d'une interface sur celle-ci (Guojie, 2005). Les composants issus d'une même famille « toolkit » présenteront un comportement et un aspect similaire. Certaines fonctionnalités comme le dialogue d'ouverture de fichier sont définies à l'intérieur du jeu d'outil. Des exemples de « toolkit » sont MFC de Microsoft, à la base de Windows, Cocoa de Apple, à la base de MacOS X, GTK, Swing, Motif, Qt et plusieurs autres.

Le *style* représente les paramètres de présentation permettant de former l'apparence visuelle des composants de l'interface (Souchon et Vanderdonckt, 2003). Il peut s'agir de couleur, de largeur de ligne, de police de caractère, de marge, etc... Les plateformes possèdent généralement un ensemble paramètres de présentation par défaut nommé guide de style. Celui-ci établit les normes en matière d'apparence et de convivialité et est utilisé par les contrôles et composants natifs. Si aucun style n'est spécifié au niveau de la définition de l'interface, le guide de style de la plateforme sera alors utilisé. Certaines plateformes comme Java avec Swing supportent plusieurs apparences ou « skin ». Ce dernier remplace alors les paramètres d'affichages par défaut et peut généralement être modifié lors de l'exécution.

La *disposition* couvre la structure de l'interface ainsi que le positionnement de ses contrôles et composants. Un positionnement absolu spécifie les coordonnés précises de chacun des éléments à l'intérieur d'une fenêtre. La disposition peut aussi se faire dynamiquement en fonction du contenu des composants et de la taille de l'écran. Dans cette

situation, une hiérarchie entre les composants (ex : fenêtre contenant une section contenant une liste...) sera alors utilisée pour la définition de la disposition.

Une part du *comportement* de l'interface peut aussi être incluse dans sa définition. À chacun des contrôles ou des composants peuvent être rattachés des actions à exécuter lors d'un événement quelconque. Les actions sont généralement prises en charge par le code de l'application derrière l'interface. L'application a généralement la possibilité d'altérer ou de mettre à jour l'interface en dehors des mécanismes de définition utilisés lors de la conception.

## 3.4 Les techniques de définition

Une interface définie de manière *programmatique* (Courtaud, 2000) est constituée d'instructions séquentielles qui en effectuent la construction. Elle définit « comment » construire l'interface. Ces instructions, pouvant être écrites dans la plupart des langages, peuvent interagir directement avec les bibliothèques de la plateforme et prendre en charge le dessin des composants à l'intérieur de l'écran. Les langages de programmation objet fournissent généralement des bibliothèques effectuant l'encapsulation des contrôles et composants plateformes sous forme d'objets (Guojie, 2005). Cela a pour effet d'en simplifier la gestion et la manipulation. Des exemples de définition programmatique à l'aide d'une bibliothèque d'objets sont MFC et Forms de Microsoft ou Awt et Swing de Java.

Une définition *déclarative*, à l'aide d'un langage descriptif permet de spécifier ce qui doit se trouver dans l'interface, le « quoi » (Courtaud, 2000). Un fichier de définition présentera, sous une forme structurée plutôt que séquentielle, les éléments constituant l'interface, ainsi que leurs attributs. La plupart des dialectes à base de XML comme UIML, XUL ou XAML effectuent une définition déclarative. Un dialecte peut être ouvert ou fermé (Souchon et Vanderdonckt, 2003). Un dialecte ouvert permettra l'ajout d'éléments ou de nouveaux composants, autre que ceux supportés nativement, alors qu'un dialecte fermé permettra uniquement l'utilisation des éléments prédéfinis. Plusieurs dialectes constituent une représentation XML des bibliothèques de construction d'interface qu'ils supportent.

Une définition *basée sur un modèle* (Limbourg et al., 2004) est utilisée pour représenter les différents aspects d'une interface à l'aide de modèles, tel qu'abordé dans la

présente section. Certains modèles peuvent s'apparenter à une définition de type « déclarative » en spécifiant les éléments retrouvés dans l'interface. Elle comporte généralement un niveau d'abstractions plus élevé couvrant davantage le problème que les détails d'implémentation de l'interface.

La *programmation visuelle* (Boshernitsan et Downes, 2004) fait appel à un outil de conception graphique pour la définition d'interfaces. Cette technique peut aussi être utilisée pour concevoir des algorithmes ou une logique de traitement. L'exemple le plus connu d'éditeur visuel appliqué aux interfaces est le langage Visual Basic de Microsoft. Ce dernier permet de développer une application par l'agencement de composants et la spécification d'attributs (Myers, Hudson et Pausch, 2000). Cette technique permet de simplifier l'apprentissage du langage de programmation, le rendant ainsi plus accessible.

Différentes techniques peuvent être utilisées pour la définition du style et de la disposition à l'intérieur d'un langage déclaratif. Ces informations peuvent se retrouver directement au niveau des éléments sous la forme de divers attributs. Cette technique est utilisée dans XUL, XAML ou LZX. Une autre approche, toujours reliée au dialecte, consiste à définir le style indépendamment de la structure (Courtaud, 2000). Celle-ci permettra le support de styles différents pour les différentes plateformes. Finalement, l'utilisation de fichiers de définition distincte contenant des informations sur l'aspect visuel et le positionnement constitue une autre technique. C'est ce que permet, entre autres, le format CSS.

Les modèles ou « templates » peuvent être utilisés à l'intérieur d'une définition d'interface déclarative. Un modèle permettra de spécifier un agencement de composants pouvant être réutilisé à différents endroits. Les langages UIML et XUL, via XBL, utilisent ce concept qui peut réduire la taille d'une définition d'interface et ainsi en favoriser la maintenabilité.

## 3.5 Les approches de développement

La conception des interfaces peut être présente aux différents niveaux du processus de développement. Dans plusieurs approches, la modélisation des interfaces commence au

niveau de la tâche (Souchon et Vanderdonckt, 2003) qui exprime les interactions et les liens qu'aura l'utilisateur avec l'application. Cette section présente différentes manières d'aborder la conception des interfaces.

### 3.5.1 Human Centred Design Process

La norme ISO 13407 « *Human Centred Design Process for Interactive Systems* » présente une approche générale de conception à l'intérieur de laquelle la création des écrans est effectuée au tout début du processus. Cela se concrétise par la réalisation d'une maquette, à partir des spécifications initiales, qui sera validée par les utilisateurs et retouchée au cours de plusieurs itérations, s'il y a lieu. Certaines méthodes de définition, telle que XIML (Helmes, 2003), permettent de compléter cette approche en exprimant les interfaces dans un format pouvant être réutilisé plus tard dans le processus.

Le processus de développement centré sur l'humain peut aussi faire intervenir plusieurs disciplines telles que l'ergonomie, la santé, la sécurité ou la performance (UsabilityNet, 2006). Les étapes de développement sont de : comprendre et spécifier le contexte d'utilisation, spécifier les exigences liées à l'utilisateur et à l'organisation, proposer des solutions de conception et évaluer les conceptions par rapport aux exigences. Tant que le système ne répond pas aux exigences, le processus est repris à la première étape. L'évaluation des solutions d'un point de vue des interfaces utilisateur peut être effectuée à l'aide de tests d'utilisabilité. Cette démarche, ayant comme premier objectif de centrer le processus de développement sur les besoins, est de plus en plus intégrée à l'intérieur de processus de qualité logicielle (ISO13407, 1999).

### 3.5.2 RAD

Le développement rapide d'application « RAD » a comme principal objectif de parvenir le plus rapidement possible aux résultats attendus (McFarlane, 2004). Cela peut être concrétisé de plusieurs manières telles que : la réutilisation de composants, l'utilisation de Framework et l'utilisation d'outils de développement avancés. Cette technique est principalement utilisée lorsque la principale contrainte à respecter est le temps de développement. Elle met généralement à profit des techniques moins sophistiquées et des

architectures moins flexibles au profit des techniques permettant de produire le plus rapidement possible. Du point de vue de la philosophie derrière RAD, il faut éviter de passer trop de temps sur l'atteinte d'un design parfait, l'objectif est tout simplement de faire en sorte que ça fonctionne (McFarlane, 2004).

Certains outils RAD mettent l'emphase sur le prototypage des interfaces utilisateur. Il s'agit d'un moyen de garantir le respect des besoins et de faciliter l'expérimentation dans les premières phases du développement. Des éditeurs permettant la définition des interfaces utilisateur font généralement partie de ces environnements. Ces derniers doivent permettre beaucoup de flexibilité quant aux possibilités de modifications tout au long du développement.

Le développement RAD met aussi l'emphase sur les solutions « Verticales » (McFarlane, 2004), c'est-à-dire, une solution qui couvre l'ensemble des aspects pour répondre à un besoin spécifique. Cette approche décourage le développement de parcelles de solutions génériques pouvant faire l'objet de réutilisation à différents endroits. Selon la situation, il peut être plus avantageux de développer des composants spécialisés pour répondre à un besoin particulier plutôt que de travailler avec du code générique. L'utilisation de composants existant, déjà développés et prêts à utiliser, est par contre encouragée.

La plateforme de développement Visual Basic de Microsoft est un exemple de développement rapide d'application. Le Framework de développement d'application de Mozilla vise aussi à mettre de l'avant cette approche de développement (McFarlane, 2004).

### 3.5.3 MDA

L'approche MDA (Model-Driven Architecture) vise le passage d'un modèle abstrait, indépendant des plateformes technologiques, vers un modèle concret, dépendant d'une plateforme, à l'aide de transformations successives (Sottet, Calvary et Favre, 2005). Cette approche permet à la fois une diminution de l'effort de développement, en générant une part du code en fonction d'un modèle, et le support de plusieurs plateformes, en permettant l'exploitation de transformations différentes.

La norme MDA définit les différents types de modèles qui entrent dans le processus ainsi que leurs interrelations. On y retrouve les modèles indépendants de plateformes (PIM), les modèles spécifiques à une plateforme (PSM) et le code, le dernier résultat de transformation (Bast, Kleppe et Warmer, 2003). Les PIM comportent un niveau d'abstraction plus élevé, indépendant d'une technologie d'implémentation, donc, ne comportant pas de détails propres aux plateformes spécifiques (ex : diagramme du domaine). Du côté des PSM, on retrouve toujours un modèle abstrait, mais basé sur une nomenclature et une forme spécifique à une plateforme (ex : modélisation d'une base de données relationnelle). Des détails d'implémentations peuvent y être ajoutés (ex : champs indexés pour une bd relationnelle). Finalement, le code généré depuis les PSM offre la description finale de l'application (ex : DDL pour une bd relationnelle). Nous retrouvons, entre autres, des cas d'utilisation UML en tant que PIM pour la génération de différents composants de l'application (Bast, Kleppe et Warmer, 2003).

Les transformations entre les modèles peuvent être automatisées, mais les outils actuels ne permettent pas d'effectuer en totalité les transformations nécessaires pour passer des PIM jusqu'au code. Pour cette raison, des interventions manuelles de la part des développeurs s'avèrent généralement nécessaires lors du passage d'une représentation à une autre (Bast, Kleppe et Warmer, 2003).

Parmi les avantages de l'approche MDA, notons : l'augmentation de la productivité, le respect des besoins, la portabilité, l'interopérabilité ainsi que la facilité de maintenance et de documentation du produit (Bast, Kleppe et Warmer, 2003).

L'augmentation de la productivité est toutefois conditionnelle au niveau d'automatisation obtenu lors des différentes transformations. La définition de ces transformations se trouve à être plus complexe qu'un développement spécifique à une plateforme, nécessitant du même coup un niveau de qualification plus élevé de la part des développeurs. Le meilleur respect des besoins est attribuable au fait que ces besoins sont pris en compte au niveau d'un modèle abstrait, plutôt qu'au niveau du code, recentrant ainsi le processus de développement sur les besoins plutôt que sur les détails d'implémentations. Les aspects portabilité et interopérabilité sont adressés par le fait que les PIM peuvent être

transformés vers plusieurs plateformes et par la possibilité de générer plus que des implémentations à l'aide des modèles. En effet, un modèle pourrait servir à générer des ponts ou connecteurs permettant l'interopérabilité. Les modèles peuvent aussi servir à la génération de la documentation nécessaire à la compréhension de la solution. Finalement, la maintenance peut être facilitée par la possibilité de propager des modifications effectuées sur un PIM jusqu'au code, réduisant du même coup les efforts nécessaires à l'opération.

Certaines initiatives pour appliquer l'approche MDA à la génération d'interface utilisateur ont été réalisées (Sottet, Calvary et Favre, 2005). Cela consiste à utiliser un ou plusieurs modèles, comptant différents niveaux d'abstraction, et d'y appliquer les transformations nécessaires jusqu'à l'interface finale.

## 3.6 L'intégration aux applications

Il existe différentes techniques pour l'établissement des liens entre l'interface utilisateur et le reste de l'application. La logique de l'application se définit principalement de manière programmatique, à l'aide d'un langage de programmation conventionnel. La grande majorité des plateformes permettant l'exécution d'une interface supporteront aussi un ou plusieurs langages ayant les capacités d'altérer l'interface utilisateur au cours de l'exécution. Une part de l'intégration pourra alors être réalisée via la modification de l'interface depuis l'application.

Dans plusieurs langages orientés objet, l'accès aux composants en cours d'exécution peut se faire via les instances correspondant aux fenêtres et aux différents contrôles présents dans l'interface. Les modifications peuvent alors être effectuées par l'accès aux méthodes de ces objets. Les actions à prendre pour la mise à jour de l'interface sont alors encapsulées par l'objet. Il s'agit de la technique utilisée par Swing ou Windows Forms.

Une autre technique répandue pour l'altération en cours d'exécution est celle du DOM (Document Object Model) , un standard du W3C. Il est, entre autres, employé par JavaScript (Almeida *et al.*, 2006) pour l'accès à la structure d'une interface HTML, XUL ou LZX. Le DOM est directement relié à la représentation XML du document. Une bibliothèque DOM permettra d'accéder à l'arborescence d'un document XML chargé et d'en modifier les

éléments et attributs. Le développement d'application interagissant avec le DOM de son interface utilisateur a pour effet de créer un couplage fort entre le code et l'interface, empêchant ainsi le découpage en couches.

Dans plusieurs cas, c'est l'interface qui doit communiquer avec l'application. C'est ce qui se produit lors d'une interaction de l'utilisateur qui doit être diffusée à l'application. La plupart des applications centrées sur une interface utilisateur sont dites « dirigées par les événements » ou « event-driven » (Horrocks, 1999). Dans ce type d'application, le flux d'exécution est dirigé par l'utilisateur à qui s'offre une multitude de possibilités d'interaction. Pour ce faire, différentes techniques peuvent être utilisées.

Certaines plateformes permettent d'implémenter les événements à l'aide de fonctions rattachées directement à l'interface. Lorsqu'un événement survient (ex : un clic), le code rattaché à l'événement est alors exécuté (Myers, Hudson et Pausch, 2000). Cette mécanique est présente sur la plateforme .Net de Microsoft. Le standard DomEvent offre aussi un comportement similaire. Il permet de spécifier le code devant être exécuté lors d'un événement. À noter que, dans les deux cas, le code associé peut prendre en charge l'exécution de l'action ou communiquer l'action à une autre couche de l'application, selon l'architecture en place. Placer la logique applicative directement au niveau du code des événements entraîne un couplage fort entre l'application et son interface, ce qui est une pratique déconseillée.

Une autre possibilité est celle d'utiliser le patron « listener » (Almeida *et al.*, 2006) pour la propagation d'une action au niveau de l'application. Cette pratique est généralement rattachée à un contrôleur d'interface donnant accès à différentes actions pouvant être exécutées au niveau de l'application. Le contrôleur d'interface a la responsabilité de s'enregistrer auprès des événements de l'interface. Plusieurs actions peuvent alors être enregistrées pour un même événement. Lorsqu'un événement survient au niveau de l'interface, les actions inscrites sont alors notifiées. Cette méthode permet un meilleur découpage entre l'interface et le reste de l'application.

Certains dialectes XML permettent la définition d'une part du comportement. Les modifications pouvant être effectuées se limitent généralement à la structure et au contenu de l'interface. Le comportement défini l'est généralement sous forme de règle (Abrams et Helms, 2000) précisant, pour chacune des conditions, quelle modification apporter.

## 3.7 Conclusion

Dans la situation qui nous préoccupe, pour la définition d'interface utilisateur multiplateforme, les approches déclaratives, spécifiant ce qui doit se trouver dans l'interface, semblent être les plus appropriées. Celles-ci se contrentrent sur la définition de l'interface plutôt que sur les instructions permettant d'initialiser l'interface sur une plateforme donnée. Un fichier descriptif de haut niveau exprimant les contrôles utilisés et leur disposition devrait permettre la génération du code pour plusieurs plateformes.

De plus en plus de dialectes pour l'enregistrement de données, l'échange d'information, la configuration ou la définition de contenu graphique utilisent une syntaxe XML. Le standard XML du W3C est de plus en plus répandu. Il offre une syntaxe très flexible et facilement compréhensible (Souchon et Vanderdonckt, 2003), en plus d'être disponible pour la très grande majorité des langages de programmation. On note aussi que la plupart des langages descriptifs existants pour la définition d'interface utilisateur sont basés sur XML. La prochaine section se concentrera donc sur les dialectes XML pour la définition d'interface.

# CHAPITRE IV

# LES DIALECTES XML DE DÉFINITION DES INTERFACES

## 4.1 Introduction

Cette section effectue une revue des approches existantes en matière de définition d'interfaces utilisateurs basées sur des modèles XML. Celles-ci sont à la fois issues de travaux de recherches et de développements commerciaux. Ces dialectes, ou User Interface Description Language (UIDL) offrent les éléments nécessaires pour représenter le contenu ainsi que, dans une moindre mesure, le comportement et autres éléments du modèle à l'aide d'un format descriptif.

## 4.2 Description des dialectes

### 4.2.1 UIML

UIML est un métalangage de définition d'interface utilisateur dont le développement a débuté en 1997, principalement par l'université Virginia Tech et des entreprises affiliées. Il est actuellement en processus de standardisation auprès de l'organisme Oasis.

Un premier objectif d'UIML est de permettre la représentation des exigences, le design et les différentes implémentations pour n'importe quel type d'interface et à l'aide d'un format de fichier unique. Le principal marché visé est celui des applications multimodales permettant le support de plusieurs modes d'interaction. Une application multimodale peut, par exemple, supporter l'interaction à l'aide d'un système vocal, d'un PDA, d'un téléphone cellulaire ou d'une interface Web conventionnelle. La philosophie derrière UIML est donc la suivante : écrire une seule implémentation pouvant servir à 'N' plateformes différentes.

Le format UIML couvre six concepts (Helmes, 2003) :

1. La structure,
2. Le style,
3. Le contenu, le comportement,
4. Les API externes et
5. La correspondance (*mapping*) avec les interfaces concretes.

Les interfaces définies dans ce format sont portables, réutilisables et extensibles. La structure des interfaces pourra être définie au début du processus de développement et être réutilisée par la suite pour la création des interfaces concrètes. Selon l'approche UIML, une interface concrète est générée à partir de la définition UIML et de la correspondance rattachée au type d'interface.

Le format de fichier XML utilisé se veut générique pour permettre la définition de tous les éléments de n'importe quel type d'interface. Aucun composant ou « widget » n'est défini par le langage. La définition de la structure contient la hiérarchie des composants. À chacun des composants devra être rattaché un type. La correspondance entre chacun des types utilisés dans la structure et l'interface concrète est définie par la correspondance. La norme UIML définit 24 types de « widgets » différents devant être supportés au niveau des correspondances avec les différentes plateformes (Puerta et Eisenstein, 2002).

Le dialecte XML contient les éléments suivants (Abrams et Helms, 2000) : <structure>, <style>, <behaviour> et <peers>. Tel que mentionné, la structure définit tout le contenu de l'interface sous une forme hiérarchique. L'élément XML <part> auquel est rattaché un attribut « class » permet de représenter tout widget ou conteneur.

L'élément <style> définit une série de propriétés qui pourra être interprétée lors de la transformation. Ces propriétés sont sous la forme de combinaisons clé-valeur. Il peut s'agir des couleurs, de la police de caractère ou de la taille des composants. Chaque propriété est reliée à un élément <part> et à un élément <structure>.

L'élément <behavior> contient les informations relatives au comportement. Ceux-ci peuvent être exprimés en XML sous forme de règles ou de conditions. Il est aussi possible d'y définir la correspondance avec des fonctions externes pouvant alors être développées dans n'importe quel langage. Pour définir une altération de l'interface devant être effectuée en cours d'exécution, la section <behavior> permet de remplacer des sections de la structure.

L'élément <peers> devra définir chacun des types d'interface concrets supportés par l'interface ainsi que les informations nécessaires à la transformation. Il est possible de définir une correspondance personnalisée au niveau du fichier ou de faire référence à un fichier de correspondance existant. Le dialecte XML permet aussi la définition de "templates" pouvant être réutilisés à différents endroits de la structure.

Exemple de définition UIML :

```
<?xml version="1.0" encoding="UTF-8"?>
<uiml>
    <interface>
        <structure>
            <part class="Frame" id="Frame">
                <part class="Entry" id="leftentry"/>
                <part class="Button" id="copyleft"/>
                <part class="Button" id="copyright"/>
                <part class="Entry" id="rightentry"/>
            </part>
        </structure>
        <style>
            <property part-name="Frame" name="position">5,5</property>
            <property part-name="Frame" name="size">302,150</property>
            <property part-name="leftentry" name="position">5,50</property>
            <property part-name="leftentry" name="size">100,25</property>
            <property part-name="Frame" name="label">Copy</property>
            <property part-name="copyleft" name="label">copy left</property>
            <property part-name="copyright" name="label">copy
      right</property>
        </style>
        <behavior>
            <rule>
                <condition>
                    <event class="ButtonPressed" part-name="copyleft"/>
                </condition>
                <action>
                    <property part-name="rightentry" name="text">
                        <call name="MyTextFs.Swap">
                            <param>
                                <property part-name="leftentry" name="text"/>
                            </param>
```

```
                    </call>
                </property>
            </action>
        </rule>
        <rule>
            <condition>
                <event class="ButtonPressed" part-name="copyleft"/>
            </condition>
            <action>
                <property part-name="rightentry" name="text">
                    <property part-name="leftentry" name="text"/>
                </property>
            </action>
        </rule>
    </behavior>
</interface>
<peers>
    <presentation name="WML">
        <component name="Frame" maps-to="wml:card">
            <attribute name="content" maps-to="wml:card.title"/>
        </component>
        <component name="Entry" maps-to="wml:p">
            <attribute name="content" maps-to="PCDATA"/>
        </component>
    </presentation>
    <presentation name="VoiceXML">
        <component name="Frame" maps-to="vxml:form"/>
        <component name="Entry" maps-to="vxml:block">
            <attribute name="content" maps-to="PCDATA"/>
        </component>
    </presentation>
</peers>
</uiml>
```

Source : (Courtaud, 2000)

### 4.2.2 UsiXML

UsiXML est issu de recherches principalement effectuées par l'université Louvain en Belgique dans le cadre du projet européen Caméléon (Vanderdonckt *et al.*, 2004). Ce dernier propose une approche de développement en plusieurs étapes contenant chacune des artefacts reliés aux interfaces utilisateur (Vanderdonckt *et al.*, 2004). L'approche couvre donc différents niveaux de détails avec différents degrés d'abstraction : le modèle de tâche et de concept (T&C), l'interface abstraite (AUI), l'interface concrète (CUI) et l'interface finale (FUI). Ces quatre couches sont prises en compte par UsiXML.

L'objectif du dialecte UsiXML est de permettre la génération d'interface en fonction du contexte. Le contexte regroupe ici l'utilisateur, la plateforme et l'environnement (Limbourg *et al.*, 2004). Le passage d'un modèle de tâche abstrait à une interface finale est effectué à l'aide de différentes étapes de transformations entre chacune des couches d'abstraction. UsiXML offre des schémas XML pour la représentation de la tâche, du domaine, du contexte, de l'interface abstraite et de l'interface concrète en plus des formats permettant la représentation des transformations.

Le modèle de tâche définit les interactions entre l'utilisateur et le système. Pour chacune d'elles, on y retrouve la fréquence, l'importance et le type d'action à exécuter. Les relations entre les tâches sont aussi représentées à l'aide du dialecte.

Le domaine contient les éléments d'un modèle UML objet conventionnel. Il couvre les classes ainsi que les relations, les attributs et les méthodes qui y sont associés. Des attributs spécifiques facilitant la correspondance avec le modèle de tâche y sont aussi ajoutés.

Le modèle de contexte représente les entités pouvant influencer le comportement de l'interface. La définition du contexte englobe l'ensemble des profils des utilisateurs sous une forme hiérarchique, les propriétés des différentes plateformes supportées et l'environnement d'utilisation avec l'application.

La définition d'interface abstraite ou AUI permet de modéliser une représentation générique des interfaces, indépendante de tout contexte. Elle est principalement constituée d'objets d'interaction abstraits (AIO) spécifiant les éléments qui feront objet d'interaction, en occurrence, les contrôles. Pour chacun d'eux, le langage permet de définir les entrées, sorties ainsi que les informations de navigation et de contrôle. La transition entre les interfaces fait aussi partie de ce modèle.

Le modèle concret définit l'ensemble des éléments de l'interface à l'exception des caractéristiques physiques propres à la plateforme. Le dialecte permet de représenter les contrôles de l'interface, la disposition et le comportement.

Le modèle final ne fait pas partie du langage. Celui-ci est exprimé à l'aide des langages et des formats de données conventionnelles utilisés par les différentes plateformes. Certains outils interprètent directement le modèle concret et génère une interface finale adaptée au contexte au moment de l'exécution.

UsiXML s'intègre à une approche MDA (Model Driven Architecture) pour le développement d'applications. Celle-ci est implémentée à l'aide de mécanismes de transformations et réification (Stanciulescu *et al.*, 2005) permettant de passer d'un modèle à un autre. Plusieurs outils externes rattachés au projet permettent d'effectuer l'édition des modèles ainsi que les différentes transformations. L'ingénierie inverse est aussi possible afin de générer les modèles en fonction d'une application existante.

Exemple de définition UsiXML :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cuiModel  name="MyModel">
    <version modifDate="2004-03-24T17:09:17.402+01:00" xmlns="">7</version>
    <authorName xmlns="">Youri</authorName>
    <window height="500" width="600" name="Formulaire (2/5)" id="window_1">
        <box relativeHeight="100" name="box1_0" id="box1_0">
            <box type="vert" name="boxTodo" id="boxTodo">
                ...
            <box type="horiz" name="box_2_2_2_1" id="box_2_2_2_1">
      <textComponent defaultContent="Sexe" isBold="true" id="label_2"/>
      <comboBox id="comboBox001" name="label_3" isDropDown="true">
          <item id="radiobutton_0" name="radiobutton_0"
      defaultContent="Femme"/>
         <item id="radiobutton_1" name="radiobutton_1"
      defaultContent="Homme"/>
      </comboBox>
              ...
          </box>
       </box>
    </window>
</cuiModel>
```

Source : (Vanderdonckt, 2005)

### 4.2.3 XIML

XIML a été développé par l'entreprise RedWhale et évolue actuellement à l'intérieur d'un forum regroupant plusieurs joueurs de l'industrie. Son objectif est d'établir un format d'échange universel pour la définition des interfaces utilisateurs (Forbrig *et al.*, 2004).

L'adoption de ce format permettrait de rendre les outils d'édition d'interface utilisateur interopérable. Le format vise à regrouper à la fois les volets abstraits, relationnels et concrets des interfaces.

L'approche proposée permet de réutiliser l'information récoltée tout au long du processus de développement. La partie abstraite couvre les modèles de tâches du domaine alors qu'une définition concrète définira la structure et les attributs des composants ou « widgets » utilisés. L'aspect relationnel établit les liens entre les différents éléments du modèle (Puerta et Eisenstein, 2003). En conservant les informations sur les interactions entre ces éléments, XIML peut agir comme une base de connaissance sur l'application utile lors du processus de développement.

Le dialecte XIML permet la définition des modèles suivants (Puerta et Eisenstein, 2003) : la tâche, le domaine, l'utilisateur, le dialogue et la présentation.

Le modèle de tâche contient les processus d'affaires sous forme de hiérarchie entre les tâches <TASK_ELEMENT> et les sous-tâches du même type. Chaque tâche requiert une interaction avec l'utilisateur. L'ordre de présence des éléments XML en définit l'ordre d'exécution.

Le domaine représente une structure hiérarchique de classes et d'attributs. Chacun des attributs ou classes sont décrits à l'aide de l'élément <DOMAIN_ELEMENT> sous forme de combinaison clé, valeur. Il forme l'ontologie de l'interface.

Le modèle utilisateur <USER_MODEL> présente les différentes classes d'utilisateurs ou les utilisateurs concrets, toujours sous une forme hiérarchique. Des caractéristiques peuvent être associées au modèle d'utilisateurs sous forme d'attributs.

Le modèle de présentation décrit la hiérarchie des composants, c'est-à-dire les fenêtres, les composants et les contrôles. Ce modèle ne contient que les informations reliées à la structure et à la disposition des interfaces et ne traite aucunement leur comportement.

Le modèle du dialogue définit les types d'interactions disponibles aux utilisateurs et permet d'affecter ces interactions aux différents composants de l'interface. Les relations entre

les composants y sont aussi présentes. Le tout permet la couverture de la partie comportement.

Exemple de définition XIML :

```
<TASK_MODEL ID="tm1">
    <TASK_ELEMENT ID="t1" name="Make annotation">
        <TASK_ELEMENT ID="t1.1" name="Select location"/>
        <TASK_ELEMENT ID="t1.2" name="Enter note"/>
        <TASK_ELEMENT ID="t1.3" name="Confirm Annotation"/>
    </TASK_ELEMENT>
</TASK_MODEL>

<DOMAIN_MODEL ID="dm1">
    <DOMAIN_ELEMENT ID="d1.1" name="map annotation">
        <DOMAIN_ELEMENT ID="d1.1.1" name="location"/>
        <DOMAIN_ELEMENT ID="d1.1.2" name="note"/>
        <DOMAIN_ELEMENT ID="d1.1.3" name="entered_by"/>
        <DOMAIN_ELEMENT ID="d1.1.4" name="timestamp"/>
    </DOMAIN_ELEMENT>
</DOMAIN_MODEL>

<USER_MODEL ID="umodel">
    <USER_ELEMENT ID="u1.1" NAME="field researcher">
        <USER_ELEMENT ID="u1.1.1" NAME="field supervisor"/>
        <USER_ELEMENT ID="u1.1.2" NAME="field geologist"/>
    </USER_ELEMENT>
    <USER_ELEMENT ID="u1.2" NAME="analyst"/>
</USER_MODEL>

<DIALOG_MODEL ID="im1">
    <DEFINITIONS>
        <RELATION_DEFINITION NAME="is_performed_using">
            <ALLOWED_CLASSES>
                <CLASS REFERENCE="im1" INHERITED="true" SLOT="left"/>
                <CLASS REFERENCE="pm2" INHERITED="true" SLOT="right"/>
            </ALLOWED_CLASSES>
            <ATTRIBUTE_DEFINITION NAME="interaction_technique"
      TYPE="enumeration">
                <DEFAULT>onClick</DEFAULT>
                <ALLOWED_VALUES>
                    <VALUE>onClick</VALUE>
                    <VALUE>onScroll</VALUE>
                    <VALUE>onChange</VALUE>
                    <VALUE>onDoubleClick</VALUE>
                </ALLOWED_VALUES>
            </ATTRIBUTE_DEFINITION>
        </RELATION_DEFINITION>
    </DEFINITIONS>
    <DIALOG_ELEMENT ID="i1.1" NAME="Make annotation"/>
    <DIALOG_ELEMENT ID="i1.2" NAME="Select location">
        <DIALOG_ELEMENT ID="i1.2.1" NAME="Select map point">
            <FEATURES>
                <RELATION_STATEMENT DEFINITION="is_performed_by"
      REFERENCE="2.1.1">
```

```
                <ATTRIBUTE_STATEMENT
    DEFINITION="interaction_technique">onDoubleClick</ATTRIBUTE_STATEMENT>
            </RELATION_STATEMENT>
        </FEATURES>
      </DIALOG>
      <DIALOG_ELEMENT ID="i1.2.2" NAME="Specify latitude"/>
      <DIALOG_ELEMENT ID="i1.2.3" NAME="Specify longitude"/>
  </DIALOG>
  <DIALOG_ELEMENT ID="i1.3" NAME="Enter note"/>
  <DIALOG_ELEMENT ID="i1.4" NAME="Confirm annotation"/>
</DIALOG_MODEL>
```

Source : (Puerta et Eisenstein, 2003)

### 4.2.4 XUL

Le langage XUL, pour XML User-Interface Language, a été développé par Netscape dans le cadre du projet Mozilla. Son objectif principal est de mettre en place un format de définition portable pour la définition d'interfaces riches (McFarlane, 2004), tout comme l'est HTML pour la définition de documents Web.

L'approche proposée est d'inclure, au niveau du dialecte, tous les éléments nécessaires à la description de la structure et des contrôles. Cela permet de définir, à l'intérieur d'un document XML, le contenu, la disposition et le style de l'interface. Le dialecte favorise le découpage

XUL s'exploite avec plusieurs autres langages. CSS est le format utilisé pour séparer l'apparence de la structure. Pour le style de présentation, tous les paramètres supportés par CSS (polices de caractères, couleurs, taille des traits, espacement, ...) peuvent être exploités à l'intérieur d'une interface XUL. Il est possible de définir les éléments CSS au niveau d'une balise « style » ou d'un fichier CSS externe.

Le comportement peut être défini, dans un premier temps, à l'aide d'attributs du type « DOM-Event ». Ces attributs permettent la définition d'une action lorsqu'un événement survient au niveau d'un élément XML (ex : « onclick » sur l'élément « button1 »). Ces événements, une fois interceptés, sont traités à l'aide de JavaScript. D'autres techniques, comme le patron de conception « observer », peuvent aussi être utilisées dans XUL (Deakin, 2006). Des balises définissant l'observation de l'état d'un contrôle font aussi partie du langage.

XBL (pour XML Binding Language) est un dialecte qui accompagne XUL et dont l'objectif est la définition de composants plus complexes ainsi que l'ajout de fonctionnalités reliées au comportement de l'interface. Il agit un peu comme les « templates » retrouvés dans d'autres langages, permettant la réutilisation d'une structure prédéfinie. Ils peuvent aussi être utilisés pour modifier le comportement visuel d'une interface lorsque survient un événement (clique, position du curseur, ...).

XUL supporte, de façon native, la plupart des contrôles retrouvés dans les interfaces conventionnelles telles que, les fenêtres, les boutons, les images, les zones de textes, les listes déroulantes, les tables, les grilles, ... Les menus conventionnels et contextuels sont aussi supportés, tout comme les raccourcis clavier. Des éléments HTML peuvent aussi être incorporés aux interfaces.

Exemple de définition XUL :

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window id="findtext" title="Find Text" orient="horizontal"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <vbox flex="3">
    <label control="t1" value="Search Text:"/>
    <textbox id="t1" style="min-width: 100px;" flex="1"/>
  </vbox>

  <vbox style="min-width: 150px;" flex="1" align="start">
    <checkbox id="c1" label="Ignore Case"/>
    <spacer flex="1" style="max-height: 30px;"/>
    <button label="Find"/>
  </vbox>
</window>
```

Source : http://www.xulplanet.com/tutorials/xultu/

### 4.2.5 XAML

XAML est le format de définition des interfaces utilisateur qui accompagne le framework WinFX de Microsoft, une partie importante du nouveau moteur de présentation de Windows Vista (David, 2005). Son objectif est de couvrir la définition de fenêtres, de contrôles, de texte, de dessin vectoriel, d'animation et d'objets 3D supportés par le

framework. Le style du langage ressemble à XUL et aux autres dialectes définissant les détails propres à une plateforme. Son vocabulaire couvre les contrôles de la plateforme (Bouton, Liste, Menu,...) ainsi que le formatage.

La technique utilisée pour la définition du format des objets diffère des autres dialectes similaires. En plus de permettre la définition de format au niveau des éléments, XAML offre la possibilité de former un style à l'aide de sous-éléments respectant la nomenclature « Parent.Attribut ». Par exemple, pour spécifier un style de fond pour un bouton, le fragment de code suivant peut être utilisé : <Button><Button.Background><SolidColorBrush Color="Blue"/></Button.Background></Button>.

Le concept de texte dont le format variable « Flow Format » est aussi intégré au dialecte. Il permet la définition de texte pouvant être redimensionné et adapté à un format d'écran, à la manière de HTML. Les éléments <Table>, <Paragraph> et <HyperLink> sont destinés à ce type de présentation à l'intérieur d'un document XAML.

Des graphiques vectoriels peuvent être définis à l'intérieur du dialecte. Les éléments <Path>, <Poligon>, <Ellipse>, <Rectangle> et plusieurs autres permettent la création de graphiques. Cette fonctionnalité de la plateforme offre des capacités similaires à SVG du W3C.

WinFX permet l'application de transformations sur tous les éléments pouvant être définis dans XAML. Les contrôles comme les boutons ou les zones de textes peuvent en faire l'objet. Pour faire pivoter un bouton, il suffit d'y ajouter un sous-élément <Button.RenderTransform> contenant un élément <RotateTransform> et les attributs en conséquence.

Des animations peuvent aussi être intégrées au document XAML. Un contrôle XAML peut contenir un sous-élément <Élément.StoryBoard> définissant les animations à exécuter ainsi que le moment auquel ils auront lieu. Au cours d'une animation, un ou plusieurs attributs peuvent être modifiés de façon graduelle. Il est possible de définir des animations complexes affectant plusieurs contrôles simultanément.

La plateforme WinFX comporte également un moteur de rendu 3D, aussi exploitable via XAML. Le dialecte comprend les éléments permettant la définition de modèles géométriques 3D ainsi que les angles de caméra et les sources lumineuses.

La définition du comportement peut se faire de plusieurs manières. Tout d'abord, XAML s'intègre à la technologie .Net. Les interactions effectuées par l'utilisateur peuvent être interceptées par le code rattaché à l'interface utilisateur. Une autre possibilité est d'utiliser les éléments <Trigger> qui permettent, lors d'un événement spécifié, d'altérer l'interface en modifiant les propriétés.

Exemple de définition XAML :

```
<?Mapping XmlNamespace="wf" ClrNamespace="System.Windows.Forms"
      Assembly="System.Windows.Forms"?>
<?Mapping XmlNamespace="wfi" ClrNamespace="System.Windows.Forms.Integration"
      Assembly="WindowsFormsIntegration"?>
<wf:Form Name="Form1" Text="Form1" IsAccessible="False" Capture="False"
      TabIndex="0" DialogResult="None" DesktopLocation="13, 13"
      DesktopBounds="13, 13, 600, 400" ClientSize="592, 373"
      AllowTransparency="False"
      xmlns="http://schemas.microsoft.com/2003/xaml/" xmlns:def="Definition"
      xmlns:wf="wf" xmlns:wfi="wfi">
  <wf:Form.Controls>
    <wfi:ElementHost Name="ElementHost1" TabIndex="0" Size="600, 400"
      Location="0, 0" IsAccessible="False" ClientSize="600, 400"
      Capture="False">
      <Canvas xmlns="http://schemas.microsoft.com/2003/xaml"
      xmlns:def="Definition" Width="600" Height="400" Background="AliceBlue">
        <Canvas.Resources>
          <LinearGradientBrush def:Name="ShineGradient" StartPoint="0,0"
      EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
              <GradientStopCollection>
                <GradientStop Color="#2989CC" Offset="0"/>
                <GradientStop Color="#FFFFFF" Offset="0.5"/>
                <GradientStop Color="#906A00" Offset="0.52"/>
                <GradientStop Color="#D99F00" Offset="0.64"/>
                <GradientStop Color="#FFFFFF" Offset="1"/>
              </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
        </Canvas.Resources>
        <Rectangle RectangleLeft="10" RectangleTop="10" RectangleWidth="560"
      RectangleHeight="350" Fill="{ShineGradient}"/>
      </Canvas>
    </wfi:ElementHost>
  </wf:Form.Controls>
</wf:Form>
```

Source : http://www.devx.com/dotnet/Article/22341/0/page/4

### 4.2.6 LZX

Le langage LZX accompagne la plateforme de distribution d'application Web riche Laszlo. Son développement a débuté en 2001 par l'entreprise LaszloSystem qui rendit sa plateforme Open Source en 2004 (openlaszlo.org).

LZX est un dialecte XML décrivant une interface ainsi qu'une part de la logique applicative sous une forme déclarative, à la manière de XUL ou XAML. Il représente les éléments de l'interface sous une forme hiérarchique et utilise des éléments XML spécifiques à la définition de chacun des contrôles. LZX permet l'utilisation de concepts orientés objet (définition de classes, d'objets et d'attributs) à même le dialecte. Une nouvelle classe, définit dans un document LZX peut être, par la suite, réutilisée comme un élément XML.

Les applications Laszlo s'exécutant principalement dans un environnement Flash, optimisé pour les animations, les effets graphiques et le multimédia, le dialecte LZX couvre une bonne partie de ces fonctionalités. Une balise <animator> permet de paramétrer un effet d'animation en y spécifiant, par exemple, des coordonnées de départ, d'arrivée ainsi que la durée de la séquence. Des éléments XML permettent aussi l'inclusion de séquences sonores ou vidéo à l'application.

La logique applicative est définie à l'aide d'une version allégée de JavaScript. Les normes DOM et DOM-Event sont aussi utilisées. Des modifications à l'interface peuvent donc être apportées en cours d'exécution. Le dialecte supporte également plusieurs éléments dédiés à la communication et à l'accès aux données provenant d'un serveur. Des attributs spécifiant les données ainsi que leur source font partie de certains éléments.

Exemple de définition LZX :

```
<canvas height="135" width="700" debug="true">
   <debug x="200"/>
   <button x="15" y="15"
      onclick="Debug.write( canvas.theField.getText() );">
      Get Text
   </button>
```

```
    <button x="100" y="15"
       onclick="canvas.theField.setText( 'Hello, Laszlo!' );">
          Set Text
    </button>

    <button x="15" y="180"
       onclick="addText();">
          Add Text
    </button>

    <script>
       function addText() {
       var origText = canvas.theField.getText();
       var newText = origText + " And on.";
            canvas.theField.setText( newText );
       }
    </script>

    <text x="45" y="60" width="150" height="75" multiline="true"
       name="theField">Some sample text.</text>
</canvas>

  <canvas debug="true" height="200">
      <debug y="60"/>
      <connection authenticator="anonymous">
          <method event="onconnect">
              Debug.write('connected');
          </method>
          <method event="ondisconnect">
              Debug.write('client disconnected');
          </method>
      </connection>

      <datapointer xpath="connection:disconnectDset:/*[1]">
          <method event="ondata">
              Debug.write('server disconnected');
          </method>
          <method event="onerror">
              Debug.write('disconnect call to server resulted in error');
          </method>
          <method event="ontimeout">
              Debug.write('disconnect call to server timed out');
          </method>
      </datapointer>

      <view>
          <simplelayout axis="x" spacing="5"/>
          <button>connect
              <method event="onclick">
                canvas.connection.connect();
              </method>
          </button>
          <button>disconnect
              <method event="onclick">
                  canvas.connection.disconnect();
              </method>
          </button>
```

```
        <button>client disconnect
            <method event="onclick">
                canvas.connection.clientDisconnect();
            </method>
        </button>
    </view>
</canvas>
```

Source : http://laszlosystems.com

### 4.2.7 Autres dialectes

Il existe plusieurs dizaines d'autres dialectes XML de définition d'interface utilisateur (Cover, 2005). Ceux-ci s'apparentent à ceux couverts dans la présente section. MXML, GTKML, XSWT, AUIML, SWIXNG, ZK, SwiXML ou XAMJ définissent le contenu de l'interface à la manière de XUL.

## 4.3 Approche pour la comparaison des dialectes

L'objectif de la comparaison effectuée est de faire ressortir les forces, les faiblesses ainsi que les différences entre les dialectes de définition d'interfaces. Les critères de comparaison utilisés sont, en partie, basés sur Souchon & Vanderdonckt. Ils ont été complétés pour couvrir l'ensemble des fonctionnalités présentes dans les dialectes présentés.

Les critères de comparaison sont les suivants (Souchon et Vanderdonckt, 2003) :

**Modèles** : Quels sont les modèles couverts par le dialecte ? Il s'agit généralement des modèles de tâche, du domaine, de présentation ainsi que la couverture du comportement.

**Méthode** : Il s'agit de l'approche utilisée pour la spécification de l'interface. Certains dialectes permettent la définition d'un modèle générique pouvant par la suite être adapté aux contextes d'utilisations spécifiques. D'autres permettent la définition d'interfaces spécifiques et nécessitent une nouvelle description pour chacun des contextes.

**Outils disponibles** : Liste des outils de développements supportant le dialecte.

**Langages supportés** : Il s'agit des langages ou des bibliothèques graphiques pour lesquels il est possible de générer du code.

**Plateformes supportées** : Liste des plateformes matérielles (PC, PDA, Vocal...) et logicielles (Java, Mozilla, ...) supportées par le langage.

**Portée** : La portée des définitions exprimées relativement aux plateformes, aux utilisateurs et aux environnements. Il s'agit de déterminer si la définition de l'interface est multiplateforme, multiutilisateur et/ou multienvironnement.

**Niveau d'abstraction** : Est-ce que le dialecte représente une interface concrète, un modèle abstrait ou un métamodèle ?

**Richesse du langage** : Couvre principalement le nombre d'éléments XML offerts par le langage pour la représentation des interfaces.

**Concision** : Ce critère est évalué en fonction du nombre d'éléments nécessaires pour exprimer un concept donné. Certains dialectes nécessiteront beaucoup plus de lignes de code que d'autres pour en arriver à un même résultat.

**Possibilité d'expansion** : Est-ce que le dialecte permet l'ajout de nouveaux éléments XML ?

**Concepts clés** : Liste des concepts spécifique au dialecte.

**Standardisation** : Organisme standardisant le langage. Est-ce un standard ouvert ?

| | UIML | UsiXML | XUL | XAML | LZX | XIML |
|---|---|---|---|---|---|---|
| **Modèles** | Présentation et Comportement | Tâche, Concepts (domaine), Présentation (abstraite et concrète) | Présentation et Comportement | Présentation et Comportement | Présentation et Comportement | Tâche, Domaine, Utilisateur, Présentation, Dialogue (comportement) |
| **Méthode** | Un modèle de présentation générique auquel sont rattachés des détails d'implémentations spécifiques | Modèles de présentation spécifiques basés sur un modèle de tâche générique | Un seul modèle spécifique. Possibilité de plusieurs styles pour un même modèle. | Un seul modèle spécifique | Un seul modèle spécifique | Modèles de présentation spécifiques basés sur un modèle de tâche générique |
| **Outils disponibles** | Harmonia LiquidUI : environnement de développement complet basé sur le dialecte | GrafiXML (environnement de développement), VisiXML (connecteur visio), SketchiXML (éditeur d'interface). PlastiXML (context / tansition), IdealXML (diagrammes / modélisation) | Mozilla / XULRunner (exécution / débogage), XULMaker (éditeur d'interface), EclipseXUL | Visual Studio 2005 | Eclipse IDE for Laszlo par IBM | XIML Editor |
| **Langages supportés** | Java, HTML, WML, VoiceXML, C++, PalmOS, .Net, … (facilement extensible) | HTML, c++, Java, XUL, WML, … (facilement extensible) | XUL (format d'exécution) | WinForms .Net (bibliothèque de définition d'interface) | LZX (format d'exécution) | Java, HTML, WML, ActiveX, … (facilement extensible) |
| **Plateformes supportées** | PDA, mobile, interface vocale, PC | PDA, mobile, interface vocale, PC | PC avec framework Mozilla | PC avec framework .Net sous Windows | PC avec connecteur Flash | PDA, mobile, interface vocale, PC |

| | UIML | UsiXML | XUL | XAML | LZX | XIML |
|---|---|---|---|---|---|---|
| **Portée** | Multiplateforme | Multiplateforme, Multi-contexte | Mono-plateforme | Mono-plateforme | Mono-plateforme | Multiplateforme, Multi-contexte |
| **Niveau d'abstraction** | Modèle abstrait | Méta-modèle | Modèle concret | Modèle concret | Modèle concret | Méta-modèle |
| **Richesse du langage** | 36 tags | Plus de 100 tags pour le modèle de présentation | Plus de 60 tags | 87 tags | Plus de 120 tags | 33 tags |
| **Concision** | Moyen, contient des détails d'implémentation | Fort : selon le modèle, la définition peut être limitée à la tâche | Faible, description détaillée de l'interface | Faible : description détaillée de l'interface | Faible : description détaillée de l'interface | Fort : permet la définition d'éléments personnalisés de haut niveau |
| **Possibilité d'expansion** | Non | Non | Oui | Oui | Oui | Oui |
| **Concepts clés** | Structure de l'interface, division du style, du contenu et du comportement | Plusieurs modèles, approche MDA, transformation gérée par les outils | Description détaillée, contrôles intégrés au langage | Description détaillée, plusieurs jeux de tags (interface utilisateur, graphique, 3d) | Description détaillée, intègre la gestion des sources des données, la gestion des connexions et les animations | Plusieurs modèles, format d'échange et de représentation des interfaces |
| **Standardisation** | Oasis-Open (En cours) / Norme ouverte | Non (Groupe de recherche), Licence ouverte | Mozilla + W3C APP Format / Norme ouverte | Microsoft, Propriétaire | Laszlo, Logiciel libre | Non (Groupe de travail fermé), Sous licence restrictive |

**Tableau 4.1** Tableau comparatif des dialectes XML de définition d'interface utilisateur

## 4.4 Conclusion

Cette grille comparative fait ressortir deux grandes catégories de dialecte : les modèles abstraits, permettant la représentation d'interfaces multiplateforme (UIML, UsiXML et XIML), et les dialectes concrets définissant l'implémentation détaillée d'une interface sur une plateforme donnée (XUL, XAML, LZX).

À première vue, l'utilisation d'un dialecte représentant un modèle abstrait et pouvant supporter plusieurs langages semble être la solution idéale pour répondre à la problématique. En offrant les mécanismes pour transformer un modèle abstrait vers un ou plusieurs formats, le support de multiples plateformes d'exécution d'interface semble simple et bien adapté. En plus des interfaces utilisateurs riches, de telles méthodes de représentation permettent aussi le support de plusieurs modes d'interactions tel que les pages HTML conventionnelles, le WAP pour mobiles ou le VoiceXML pour le mode vocal.

Une définition d'application Web riche avec un dialecte plus abstrait, comme UIML, XIML ou UsiXML, pourrait être transformée vers un autre des dialectes étudiés, comme XUL, XAML ou LZX, comme cela a déjà été fait (Ruiz, Arteaga et Vanderdonckt). Les dialectes concrets, bien que différents au niveau de la syntaxe, restent similaires au niveau des fonctionnalités offertes et des résultats pouvant être obtenus, comme le montre des comparatifs effectués entre XUL et XAML (Alliance, 2004).

La définition du volet graphique d'une application peut donc se faire indépendamment d'une technologie de présentation à l'aide des approches existantes, étudiées dans ce chapitre. Un des points restant, pour couvrir entièrement le client Web riche, est la partie logique d'application, qui est souvent être répartie entre le client et le serveur. Ce volet fait l'objet d'une étude plus approfondie dans le chapitre suivant.

# CHAPITRE V

# LA LOGIQUE D'APPLICATION

## 5.1 Introduction

La définition de la logique applicative peut prendre des formes variées et est généralement définie de manière programmatique. De ce fait, la transformation d'un modèle vers un autre peut s'avérer plus complexe que pour le volet graphique. De plus, dans le contexte d'une application Web, la logique applicative est généralement rattachée aux méthodes de communications couvrant les échanges entre le client et le serveur. La méthode et la technologie utilisée pour la définition d'une application de type client-serveur auront donc inévitablement des incidences sur les développements des deux côtés.

Cette section propose d'étudier plus en détail les méthodes utilisées pour la définition de la logique côté client et serveur ainsi que les méthodes de communications utilisées pour relier les deux. Des exemples d'architectures d'applications AJAX, XUL, Laszlo, Java et HTML seront étudiés plus en détail afin de faire ressortir la manière dont est gérée la logique applicative ainsi que les responsabilités des paliers client et serveur.

## 5.2 L'architecture type d'une application Web

L'ouvrage « *Guide to Web Application and Platform Architectures* » propose une architecture type d'application Web (Jablonski, 2004). Le modèle à 4 paliers : client, contrôleur Web, serveur d'applications et serveur de données est le plus fréquemment rencontré dans les applications d'entreprises.

Le *paliers client* est responsable de la présentation, le rendu HTML dans le cas d'une application Web conventionnelle. Il englobe aussi les moyens de communication utilisés pour l'accès au contrôleur Web et peut aussi prendre en charge une part de la logique applicative. À l'intérieur d'une application Web conventionnelle, son rôle se limite à l'affichage et à la prise en charge des événements déclenchés par l'utilisateur qui seront ensuite transmis au serveur, ce qui élimine le besoin de prise en charge de la logique d'affaires. Par contre, dans le cas d'un client riche, cette dernière pourra occuper une place plus importante au niveau du client.

Le *palier Web* englobe à la fois la partie contrôleur, qui traite les événements provenant du client, et une partie de la présentation dont la responsabilité est de générer le contenu qui devra être affiché par le client. Il peut prendre en charge une part de la logique d'affaire, reliée à la logique de présentation, mais sa responsabilité première est de faire le lien entre ce qui est affiché par le client et l'application.

Le *serveur d'applications* prend en charge toute la logique d'affaire reliée à l'application. Il est, en principe, détaché de toute logique de présentation. Pour plusieurs plateformes Web, le palier Web et le serveur d'applications ne font qu'un pour former un modèle 3 paliers. L'exploitation de la logique d'application sous un palier distinct est principalement pratiquée pour des raisons de mise à l'échelle.

Les *services dorsaux* (back-end) englobent, entre autres, la base de données, les serveurs de courriel, les serveurs de fichiers et annuaires. Ces services ne comportent pas de logique d'application ou de présentation. Ils sont appelés par le serveur d'applications à l'aide d'une méthode de communication appropriée et supportée par les deux partis.

## 5.3 Classification des technologies reliées au Web

Les différents rôles et technologies retrouvés dans une architecture d'application Web peuvent être divisés en modules ou composants, dont les principaux sont : la présentation, la logique d'affaires, l'interaction et la gestion des données. Ces modules peuvent être réutilisés comme point de départ pour la classification des technologies et standards du Web.

Du côté de la *présentation*, le rôle des technologies est de générer le contenu devant être présenté ou d'en effectuer le rendu. Côté serveur, il s'agit des techniques utilisées pour la génération du code HTML alors que, du côté client, ce module englobe les technologies reliées à l'affichage et à la mise en forme. Les technologies CSS, DHTML, Flash et XSL entrent dans cette catégorie.

La *logique d'affaires* peut, dans un premier temps, intervenir au niveau de la gestion des requêtes lancées par le client. Elle peut donc comprendre du code spécifique au contexte d'une application Web. Elle englobe aussi tout ce qui est traitement spécifique à l'application et à son domaine. Les technologies classifiées comme faisant partie de la logique comprennent, entre autres, le PHP, ASP, JSP ou les composants J2EE ou COM de Microsoft pour le traitement côté serveur. Côté client, notons les technologies de script comme JavaScript et VBScript mais aussi les Applets, les ActiveX, les clients Flash, etc. Les procédures stockées peuvent aussi entrer dans cette catégorie.

Les technologies reliées à l'*interaction* ou à la communication permettent l'échange de message et de données entre les différentes parties de l'application. Elles comprennent, entre autres, l'accès aux données (ODBC, JDBC, JDO, ADO, ...), les technologies d'invocation à distance (RPC, RMI, SOAP, ...) et les protocoles de transport et de communication (TCP/IP, SMTP, HTTP, ...). Les mécanismes de gestion des sessions et des transactions entrent aussi dans le module interaction, tout comme les liens entre le serveur d'applications et le serveur de données. Dans leurs représentations d'architectures d'application Web, les auteurs ignorent généralement la mention des technologies d'interaction du client vers la couche Web via requête HTTP, ceux-ci étant triviaux.

Les technologies reliées à la *gestion des données* s'occupent de l'accès et de la manipulation des données. Elles sont responsables de l'exécution des opérations conventionnelles sur les données (lecture, mise à jour, suppression). Dans l'architecture proposée, le rôle de la gestion des données est généralement confié exclusivement au palier « serveur de données », les autres entités concernées comme le serveur d'applications utilisant une technologie d'interaction pour commander les requêtes à exécuter.

Les auteurs définissent aussi des modules couvrant la personnalisation, la sécurité, la sémantique (RDF) et l'interfaçage (import/export), qui sont moins pertinents pour l'analyse des différences entre les architectures des technologies étudiées dans le présent travail.

## 5.4 Comparaison des plateformes et de leurs architectures

Cette classification des technologies reliées au Web, combinées au modèle d'architecture type présenté précédemment permettra de mieux faire ressortir la composition des différentes plateformes Web étudiées jusqu'à présent. Pour chacune d'entre elles, les technologies rattachées à chacun des paliers de leurs architectures seront exposées. La matrice des technologies définie par les auteurs (Jablonski, 2004) et englobant les paliers et les modules couverts précédemment sera utilisée.

L'exercice permettra d'effectuer une comparaison de haut niveau des plateformes et ainsi d'en faire ressortir les différences. Pour commencer, le modèle d'application Web HTML conventionnel sera détaillé, ce qui permettra de l'utiliser comme point de repère et d'être en mesure d'effectuer un parallèle avec les plateformes destinées aux clients Web riches. Dans un deuxième temps, les architectures utilisées par différentes technologies AJAX ainsi que celles rencontrées avec les clients XUL, Flash et Java seront abordées.

Des applications Web destinées à une même technologie de présentation peuvent être basées sur des Framework différents et, par conséquent, utiliser des technologies différentes à tous les niveaux du modèle d'architecture présenté. De plus, deux applications basées sur le même Framework pourront faire l'usage de technologies différentes à l'intérieur des différents modules composant leur architecture. Pour cette raison, des applications existantes serviront de point de repère pour chacun des cas présentés.

### 5.4.1 HTML

L'utilisation de contenu HTML simple comme technologie de présentation implique que la logique se concentre principalement côté serveur. Par l'utilisation du paradigme de la page, rechargée à chaque action de l'utilisateur, le serveur Web génère et retourne l'interface

utilisateur, incluant l'état de l'application. Avec du HTML simple, aucun traitement ne s'exécute par le palier client.

Il s'agit de la méthode utilisée pour pratiquement tous les cadres de développement Web conventionnel. À titre d'exemple, pour illustrer cette approche, nous utiliserons une application développée sous Struts et exploitant les technologies J2EE.

XPetstore-EJB version 3.1.3 (http://xpetstore.sourceforge.net[1])

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | HTML, CSS | | HTTP | |
| **Palier Web** | JSP, JSTL-Taglib, Struts-Taglib, Sitemesh | Servlet, Struts-Actions, Struts-Forms | JNDI (accès couche service) | |
| **Serveur d'applications** | | EJB Services | JDBC | |
| **Services dorsaux** | | | | Base de données |

**Tableau 5.1** Architecture de l'application xpetstore

La décomposition de cette application nous montre que seules les technologies HTML et CSS sont utilisées ici par le client. La méthode d'interaction entre le client et le serveur est triviale (requêtes http vers le serveur et réception de la nouvelle page comme réponse). Du côté du serveur, ce sont des contrôleurs codés à l'intérieur du Framework Struts qui prennent en charge les actions, et interagissent avec la couche application pour la modification du modèle.

### 5.4.2 AJAX

[1] Dernière consultation, août 2007

Les Framework permettant le développement et le déploiement d'un client AJAX (Asynchronous JavaScript And XML) sont nombreux. Ceux-ci peuvent utiliser différentes technologies côté serveur ainsi que des méthodes d'interactions différentes avec le client.

5.4.2.1 Avec JSF et Ajax4jsf

Google House (http://www.javaworld.com/javaworld/jw-09-2006/jw-0911-jsf.html[2])

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | HTML, CSS | JavaScript | Contenu HTML via http | |
| **Palier Web** | JSP, MyFaces, ajax4jsf | Servlet, Tags JSF pour logique présentation | | |
| **Serveur d'applications** | | Java | | |
| **Services dorsaux** | | | | Base de données |

**Tableau 5.2** Architecture de l'application Google House

Cette application basée sur le Framework JSF/MyFaces est structurée de manière très similaire à l'application HTML pure. Seules des fonctionnalités AJAX pour l'exécution de commandes de validation côté serveur ont été ajoutées. Les fonctionnalités AJAX proviennent de ajax4jsf, maintenant JBossRichFaces. Pour l'exécution de commandes côté serveur, des tags spéciaux sont insérés dans le JSP et le code JavaScript pour l'exécution de cette communication est généré dynamiquement. Les exemples présents dans l'application « Google House » utilisent du contenu HTML comme format d'échange entre le client et le

[2] Dernière consultation, août 2007

serveur pour la mise à jour de certaines sections de l'écran. Le contrôleur prend la forme d'un ManagedBean comprenant l'état de la session et exposant ses méthodes aux composants JSF.

5.4.2.2 Avec ZK

ZK Pet Shop (http://www.zkoss.org/demo/[3])

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | **HTML, CSS** | **JavaScript** | **Contenu XML (réduit) via HTTP** | |
| **Palier Web** | ZKML | Code Java dans le ZKML, Extension du Framework ZK | | |
| **Serveur d'applications** | | Java, JPA | JDBC | |
| **Services dorsaux** | | | | Base de données |

**Tableau 5.3** Architecture de l'application ZK Pet Shop

L'application basée sur le Framework ZK prend la forme d'une application AJAX exécutée par le navigateur avec les technologies HTML, CSS et JavaScript. Le Framework permet la définition d'une application à l'aide d'un fichier de définition combinant des tags XML permettant l'insertion de code Java pour la définition du comportement. Le servlet interprète l'application ZK et génère le code HTML initial. Les mises à jour à l'interface du client sont ensuite transmises à l'application à l'aide de fragments XML encodés de manière à réduire la taille des transferts.

[3] Dernière consultation, août 2007

5.4.2.3 Avec GWT

GWTBook (Démo chapitre 10, http://www.manning.com/hanson/)

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | JavaScript, HTML, CSS | JavaScript | GWT-RPC | |
| **Palier Web** | Java (Transformation JavaScript) | Java, Framework GWT | | |
| **Serveur d'applications** | | Java / EJB / Spring ... | | |
| **Services dorsaux** | | | | ? |

**Tableau 5.4** Architecture de l'application GWTBook

Du côté de GWT, l'application client est, dans un premier temps, écrite en Java pour ensuite être transformée en client JavaScript. Tout ce qui a trait à la logique de présentation est donc exécutée par le client. Dans l'exemple étudié, le mode de communication utilisé est GWT-RPC. Il s'agit d'un protocole de sérialisation d'objets Java / JavaScript transigeant dans un format binaire via http (McCarthy, 2006). La définition des interfaces utilisateurs incluant l'initialisation des composants (Widgets) et la génération de fragments HTML est effectuée directement en Java pour être ramenée en JavaScript. Des CSS et des modèles HTML permettent la personnalisation des interfaces utilisateurs de l'application. Le code java, côté serveur reçoit les requêtes RPC et agit comme contrôleur permettant ainsi l'accès à une couche applicative Java.

5.4.3 XUL

Le Framework Mozilla offre de nombreuses possibilités quant au développement d'application riche, déployé via internet. Ils peuvent être distribués sous forme d'extension à un navigateur Mozilla, être distribués sous forme d'application locale ou être exécutés en ligne, comme une application AJAX. L'application XUL étudiée est développée comme une application autonome, pouvant être chargée à l'intérieur d'un navigateur de famille Mozilla.

Mozilla Amazon Browser (http://mab.mozdev.org/[4])

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | XUL, HTML, CSS | JavaScript<br><br>Couche contrôleur | Contenu XML via http | Mozilla.org Preferences-Service |
| **Palier Web** | | | Contenu XML via http | |
| **Serveur d'applications** | | Java / .Net / PHP / Pearl / … | | |
| **Services dorsaux** | | | | ? |

**Tableau 5.5** Architecture de l'application Mozilla Amazon Browser

Pour l'exécution de requête de recherche auprès du site Amazon, une requête HTTP standard est formulée et le serveur génère une réponse en format XML. Certaines pages consultées depuis le client sont chargées du site principal d'Amazon en format HTML. La totalité de la logique d'affaire relative au client est codée en JavaScript et exécutée par le navigateur. Du côté du client, on retrouve une séparation des fichiers de présentation (XUL / CSS) et les fichiers contenant la logique (JavaScript). Ceux-ci prennent la forme de contrôleurs permettant de gérer les événements provenant de l'interface utilisateur.

L'application peut être installée localement et permet aussi l'enregistrement de paramètres au niveau du profil de l'utilisateur à l'aide du service de préférences Mozilla. L'architecture de l'application côté serveur est inconnue. Le rôle du serveur dans le cas présent est uniquement de retourner des résultats de recherche dans un format XML, n'importe quelle technologie (PHP, Pearl, .Net, Java,…) pourrait être utilisée. Le mode de communication est le même que pour AJAX.

---

[4] Dernière consultation, août 2007

### 5.4.4 Laszlo

OpenLaszlo est un Framework pour la définition et le déploiement de clients riches basés sur le lecteur Flash. Le format de définition couvre à la fois le code côté client et serveur.

RIA Amazon Store (http://www.openlaszlo.org/demos[5]).

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | Flash | Flash (ByteCode) | XML Data Requests | |
| **Palier Web** | LZX | Langage dérivé ECMAScript<br><br>Servlet Laszlo | Contenu XML via HTTP | |
| **Serveur d'applications** | | Java / .Net / PHP / Pearl / … | | |
| **Services dorsaux** | | | | ? |

**Tableau 5.6** Architecture de l'appliction RIA Amazon Store

Ici, la partie client est transmise en tant que document Flash et interprétée par le connecteur du même nom à l'intérieur du navigateur. Le client et la logique de présentation sont codés à l'intérieur du format LZX couvert précédemment. Le fichier LZX, exécuté côté serveur, sert donc à la génération du client ainsi qu'au support d'une part de la logique côté serveur. L'accès aux données est, entre autres, pris en charge par la partie serveur de l'application. Les requêtes sont transmises du client Laszlo vers le serveur Laszlo qui, par la suite, interroge l'application (le service XML d'accès aux données d'Amazon dans le cas présent). Le format d'échange entre les deux paliers est un format spécifique au Framework

[5] Dernière consultation, août 2007

(XML Data Requests). Le serveur Laszlo est un Servlet Java mais la couche application peut être sur n'importe quelle technologie et accédée via différents protocoles (SOAP, XML-RPC, ...).

### 5.4.5 Java

L'utilisation de Java du côté du client permet d'exploiter l'ensemble des fonctionnalités de la plateforme pour bâtir l'application. Cela offre, entre autres, le choix d'exécuter un même traitement sur un palier ou l'autre ainsi que la possibilité d'utiliser des fonctionnalités propres à Java comme RMI.

OpenSwing, Démo # 18 (http://oswing.sourceforge.net/samples.html[6])

| | **Présentation** | **Logique** | **Interaction** | **Gestion des données** |
|---|---|---|---|---|
| **Palier Client** | Swing | Java (UI + DataRetrieval) | Sérialisation via HTTP | |
| **Palier Web** | | Servlet Java | Sérialisation via HTTP | |
| **Serveur d'applications** | | Java, Spring | JDBC | |
| **Services dorsaux** | | | | Base de données |

**Tableau 5.7** Architecture, démonstration #18, projet OpenSwing

Le produit OpenSwing offre une architecture générique pour la construction d'un client riche Java exécuté sous forme d'applet. Le Framework offre une série de composants génériques pouvant être alimentés par une couche d'accès aux données côté client. Les écrans héritent de classes d'OpenSwing offrant des mécanismes génériques de gestion des fenêtres et de cycle de vie de l'application (lancement, connexion, fermeture, ...). Un modèle de

[6] Dernière consultation, août 2007

données est commun aux parties client et serveur sous forme d'objets sérialisables.
L'application client comporte une couche contrôleur faisant appel aux bibliothèques de OpenSwing pour l'envoi de requêtes au serveur. Ce dernier fait appel à la couche application, qui retourne les objets demandés. Ces objets sont ensuite sérialisés à l'aide des fonctionnalités livrées avec Java (ObjectOutputStream) pour ensuite être décodées côté client. Dans cet exemple, le Framework Spring est aussi utilisé pour la gestion des accès aux données.

## 5.5 Conclusion

Cette section aura permis d'exposer les différences entre des architectures types pour plusieurs plateformes de client Web riche. Première constatation : les paliers client et Web sont tous fortement différents. On note quelques éléments communs tels que l'utilisation de XML pour l'interaction ou JavaScript pour la logique, mais la structure des données et le code JavaScript générés dépendront des technologies utilisées.

Le fait que les technologies client interagissent de façon différente avec le palier Web de l'application limite les possibilités quant à l'utilisation d'une couche serveur générique. Chaque technologie de présentation met de l'avant leurs propres techniques qui leur sont adaptées. Par exemple, pour les clients Laszlo ou ZK, des protocoles de communication XML pour la mise à jour des composants de l'interface utilisateur ont été développés très étroitement en lien avec le reste de la plateforme. L'utilisation d'AJAX pour la mise à jour de sections de l'écran tel qu'utilisé avec ajax4jsf limite à une seule technologie de présentation la partie Web côté serveur. En somme, en fonction des exemples étudiés, nous pouvons affirmer que le palier client ne peut être dissocié du palier Web dans le cas d'un client Web riche.

Les paliers Client et Web sont ceux qui contiennent l'ensemble de la logique de présentation. La méthode de définition de cette logique est aussi différente d'une technologie à une autre. Pour les applications basées sur HTML ainsi que pour Laszlo, le code côté client est généré par le serveur en fonction d'une définition dans un format propre à la technologie (Tags JSF, code Java dépendant du framework, fichier LZX, ...). Si on ajoute XUL et Java pour lesquels la logique de présentation est codée dans leurs langages respectifs, nous avons

aussi des manières différentes de définir la logique de présentation pour chacune des technologies couvertes. La logique de présentation doit donc être codée spécifiquement pour la technologie utilisée.

Du côté de la couche application, le problème est différent. Bien que des technologies variées aient été utilisées dans les exemples couverts (EJB, Java pure, Spring, ou autre), ces dernières n'ont pas de dépendance avec la technologie client. En effet, dans tous les cas, c'est la couche Web, via un contrôleur qui fait appel à l'application, selon les méthodes disponibles. Dans plusieurs cas, l'application est appelée via service Web, ce qui ouvre la voie à pratiquement toutes les technologies disponibles (.Net, Java, PHP, Pearl, Python, ...). Toutes les technologies client devraient donc être en mesure de communiquer avec n'importe quel serveur d'applications. L'exploitation des technologies utilisées par la couche application (comme les EJB) au niveau des contrôleurs de la couche Web introduit toutefois une dépendance supplémentaire. C'est le cas ici pour les exemples utilisés pour HTML, ZK et GWT. Il s'agit toutefois d'une décision de conception qui aurait pu être évitée avec l'utilisation d'une architecture orientée service.

En conclusion, une technologie de présentation englobe un palier client et un palier Web, côté serveur. Ils couvrent ensemble tous les aspects de la présentation et de la logique de présentation. Ceux-ci sont généralement interdépendants et non interchangeables. Par contre, toute technologie client a le potentiel de s'intégrer avec toute technologie côté serveur d'applications.

# CHAPITRE VI

# CHOIX CONCEPTUELS

## 6.1 Mise en contexte

Parmi les approches étudiées, celles utilisant un modèle abstrait (UIML, UsiXML et XIML) ont les capacités de supporter plusieurs plateformes à l'aide d'une définition unique. Ces dernières pourraient être utilisées pour générer le code de différents types d'interfaces et, dans certains cas, générer les accès à la logique pouvant s'appliquer à plusieurs plateformes.

Il y a toutefois quelques inconvénients quant à l'utilisation de ces technologies, particulièrement pour un projet de développement conventionnel. Premièrement, ces solutions sont encore marginales et principalement utilisées à des fins de recherches. La licence d'utilisation de XIML restreint d'ailleurs son usage à cet effet (ximl.org). UIML est en processus de standardisation depuis plusieurs années par l'organisme OasisOpen. Les spécifications ne sont pas encore finales et aucune implémentation n'est offerte pour la dernière version du dialecte. L'entreprise Harmonia offre des outils pour la définition et la génération d'applications à l'aide d'UIML mais ces derniers sont propriétaires (harmonia.com). UsiXML offre davantage d'outils et d'implémentations ouvertes que les deux dialectes précédents. Ces derniers sont toutefois centrés sur l'aspect interface utilisateur et n'offrent pas de solution intégrée pour couvrir l'aspect logique, nécessaire lorsqu'on veut porter une application sur plusieurs plateformes. En effet, les derniers travaux réalisés par le groupe dans le domaine des applications Web riches ne couvraient que la présentation et nécessitaient toujours la réécriture de la logique au niveau de chacune des technologies de présentation utilisées (Ruiz, Arteaga et Vanderdonckt).

L'effort requis pour le développement à l'aide des approches étudiées permettant le support de plusieurs plateformes semble aussi plus imposant. En effet, le processus utilisé par UsiXML, basé sur MDA, exploite plusieurs modèles différents et peut nécessiter des interventions pour compléter chacun des modèles après la transformation. Du côté d'UIML, le support d'une nouvelle plateforme peut entrainer l'ajout de nouvelles propriétés au modèle contenant des éléments concrets comme le positionnement ou le style devant être utilisé. Ces particularités apportent le niveau de flexibilité nécessaire pour la définition de tout type d'interface utilisateur, mais pourraient entrainer des coûts supplémentaires.

La technologie d'OpenLaszlo permet aussi le support de plusieurs plateformes à l'aide d'une définition d'application unique. Une application Web riche définie à l'aide du dialecte LZX, initialement destiné à s'exécuter sous la plateforme de présentation Flash, peut aussi depuis la version 3.1 (LaszloSystems, 2006), être utilisé pour générer un client HTML/AJAX identique à la version Flash. Le fichier de définition reste un modèle concret englobant tous les détails reliés au client riche. La taille des écrans, les polices de caractères, la logique de présentation, les animations et les appels aux méthodes de communications restent constants d'une plateforme à l'autre. C'est donc le même client Web riche qui est exécuté sur deux plateformes de présentation différentes en y exploitant les mêmes fonctionnalités. Cette approche permet donc de satisfaire les besoins d'un client unique devant s'exécuter via deux plateformes et offrant des fonctionnalités et une présentation identiques. Par contre, le support d'une plateforme offrant des fonctionnalités réduites (HTML simple, WML, VoiceXML, ...) ne s'avère pas possible et les plateformes se limitent actuellement qu'à Flash et AJAX.

## 6.2 Choix conceptuels pour la nouvelle approche

### 6.2.1 Support de plusieurs plateformes

Pour le support multiplateforme, il a été montré dans la section précédente que les volets de la logique et des méthodes de communication dépendent aussi de la technologie de présentation. Le problème est le suivant : avec les Framework d'application Web conventionnels, les aspects relatifs au contrôleur, à la communication et à la présentation sont

dépendants de la technologie utilisée. La définition de l'application (données, écrans propres au modèle, logique de l'application...) est donc dépendante de cette dernière. Il faut donc trouver une technique permettant de rendre la définition de l'application indépendante de la méthode d'implémentation.

Une définition concrète d'une interface utilisateur, avec un langage comme XUL, n'est pas suffisante pour le support de plusieurs plateformes dont le mode d'interaction est différent (ex : Flash et WML). Les approches de définition concrètes dont fait partie XUL comportent beaucoup de détails d'implémentations qui sont propres à une plateforme. Par exemple, les widgets (composants) utilisés, le style ou les informations sur le positionnement font partie de la définition de l'interface. Ces données peuvent toutefois varier d'une plateforme à l'autre. Par exemple, chaque plateforme possède un jeu de widgets ainsi qu'un guide de style qui lui est propre. Il peut donc arriver qu'un widget donné ne soit pas disponible pour une plateforme et qu'une version alternative de l'interface doive être présentée pour une plateforme donnée. C'est le cas, par exemple, avec l'utilisation d'un menu contextuel pouvant être utilisé dans une interface riche, mais qui ne peut être disponible avec une page HTML conventionnelle. L'interface HTML devra donc présenter les fonctions d'une manière différente, ce qui fait en sorte que l'interface ne sera pas exactement la même.

Une définition d'application indépendante de plateforme devrait donc appliquer le choix des widgets, le style et la présentation au moment de la transformation vers l'interface finale, c'est ce qui est fait dans la section « structure » d'un document UIML (Abrams et Helms, 2000) ou dans le modèle abstrait avec UsiXML (Vanderdonckt, 2005). En somme, afin de respecter les exigences reliées au support de plusieurs plateformes, l'approche expérimentée devra comprendre un modèle abstrait de haut niveau décrivant le « quoi » de l'application et non le « comment », retrouvé dans les interfaces concrètes.

### 6.2.2 Effort de développement

Afin d'améliorer le temps de développement, des concepts issus du RAD (Rapid Application Developement, voir section 2) seront employés. Le but de RAD est, comme son nom l'indique, de minimiser le temps de développement. Pour y parvenir, les outils de développement issus de cette école mettent de l'avant des composants réutilisables et un

Framework riche en fonctionnalités. L'approche par composant aura une place de choix dans le cadre d'expérimentation. Ce concept est d'ailleurs utilisé pour d'autres définitions multiplateformes comme UIML (Abrams et Helms, 2000). Un jeu de composants très riche en fonctionnalités et dont le comportement est paramétrable devrait permettre un maximum de réutilisation.

Un point important pour la réduction des efforts dans le contexte d'une application multiplateforme est d'éviter le développement de définitions spécifiques à chacune des plateformes supportées. À ce niveau, des technologies comme UIML ne répondent pas toujours aux attentes, car, dans certaines situations, il est nécessaire de produire des définitions propres à chacune des plateformes supportées (Menkhaus et Fischmeister, 2003). Dans notre cas, le fait d'avoir un modèle de haut niveau, accompagné d'une étape de transformation conçue pour produire des résultats conformes aux attentes pour une plateforme donnée devrait pouvoir palier à ces problèmes.

L'utilisation d'un modèle de haut niveau aura aussi pour effet de réduire le nombre de lignes de code puisque tous les détails d'implémentations devraient être exclus de la définition. L'aspect présentation devant être généré depuis ce modèle, une grande partie du code qui aurait normalement été écrit avec une approche conventionnelle ne sera plus nécessaire. Il s'agit donc d'un autre facteur qui permettra de réduire le temps de développement.

### 6.2.3 Définition du modèle

Concernant l'approche utilisée pour la définition du modèle, il a été choisi d'expérimenter une approche de métaprogrammation plutôt qu'un processus MDA. La métaprogrammation consiste en l'écriture d'un programme ayant pour objectif de générer un autre programme (Tate et Hibbs, 2006). Avec cette approche, seul le métaprogramme est édité et maintenu pour produire les résultats souhaités. Ce n'est pas nécessairement le cas avec les autres approches de génération, souvent exécuté une seule fois pour augmenter la productivité au début du développement et qui permettent ensuite d'inclure des éléments de personnalisation à certains endroits spécifiques. La métaprogrammation est un concept

largement utilisé avec la technologie Ruby On Rails pour laquelle plusieurs études ont montré un gain en efficacité au niveau du développement (Tate et Hibbs, 2006).

À l'intérieur du Framework Rails, certaines fonctionnalités, comme la gestion de la persistance, utilisent la métaprogrammation pour définir des comportements qui sont alors spécifiés de manière déclarative. La spécification des comportements à utiliser se fait à l'aide d'un langage spécifique au domaine ou « Domain Specific Language ». Cela prend la forme de lignes de code décrivant le comportement désiré (ex : ajout d'un accesseur ou définition d'une référence avec une autre classe) qui seront ensuite interprétées au moment de l'exécution pour ajouter à la classe les méthodes et le comportement souhaité (Vanderburg, 2005). La métaprogrammation semble donc être une technique intéressante ayant pour effet de réduire le nombre de lignes de code. Comparativement à d'autres approches de génération de code depuis un modèle, elle devrait aussi permettre davantage de maintenabilité puisqu'elle élimine les manipulations relatives au code généré.

### 6.2.4 Transformation du modèle

Il a été choisi d'exécuter la transformation du modèle au moment de l'exécution, soit, de manière interprétée. L'utilisation de ce mode vient confirmer la décision de ne pas utiliser de mécanisme de génération de code qui aurait permis des modifications après la transformation, ce qui ne peut se faire dans un mode interprété. Ce mode facilite la maintenance et le développement du modèle puisque, lors d'une modification, il suffit de redémarrer l'application pour que le tout soit repris en compte. Si du code devait être généré afin d'améliorer les performances ou pour le support d'un certain type de plateforme, l'opération devrait être effectuée dynamiquement, sans permettre l'intervention de l'utilisateur, afin de préserver les avantages d'un modèle pouvant être exécuté sans intervention. C'est ce qui se produit avec le format JSP dont le contenu est transformé en classe Java et compilé au moment de l'exécution (Chopra, 2005). Le fait de permettre la modification du code généré entrainerait des problèmes importants au niveau de la maintenance : soit le fichier original ne pourrait plus être modifié ou les modifications devraient être constamment réappliquées sur le code généré.

Nous retrouvons aussi plusieurs exemples d'intervention au moment de l'exécution qui permettent d'améliorer la maintenabilité d'une application. Le Framework de persistance Hibernate génére des sous-classes pour chacune des entités affectées par le « mapping » relationnel-objet au moment de l'exécution (Sperko, 2003). Ces sous-classes héritent de l'objet du modèle et servent à ajouter les comportements qui auront été paramétrés à l'intérieur de la configuration du Framework. Ainsi, lorsqu'on demande d'utiliser du « Lazy-Loading » pour le chargement des données de certaines tables, les requêtes d'accès aux données seront ajoutées au moment de l'exécution, réduisant ainsi la complexité des classes du modèle. C'est pour cette raison que Hibernate peut effectuer un « mapping » relationnel-objet complet avec des POJO alors que des technologies de génération précédentes, comme Torque (db.apache.org, 2007), devaient générer des classes complexes à l'intérieur desquelles on retrouvait une panoplie d'attributs enregistrant l'état de l'objet et de chacune des colonnes (nouveau, modifié, colonnes modifiées, ...) en plus du code nécessaire au support des fonctionnalités comme le « Lazy-Loading ». En tenant aussi compte du fait que les modifications au modèle sous Torque doivent être appliquées à l'aide d'un générateur, on peut affirmer que les efforts nécessaires à la maintenance d'une couche de persistance générée par Torque sont beaucoup plus importants que pour les POJO utilisés par Hibernate. Avec Hibernate, toutes les manipulations effectuées par le Framework sont transparentes pour les développeurs et aucunement intrusives au niveau de l'application.

La transformation du modèle au moment de l'exécution facilite aussi les mises à jour de la partie interpréteur de l'application. En effet, les travaux de développement et l'optimisation du code de l'interpréteur sont totalement indépendants des modèles et vice versa. L'interpréteur peut être amélioré séparément et, lorsque son niveau de maturité est jugé acceptable, être mis à jour au niveau de chaque application qui l'utilise. Une application basée sur un modèle pourrait être mise à jour avec une nouvelle génération d'interpréteur. Advenant un mauvais fonctionnement, il ne suffirait qu'à retourner à une version précédente de l'interpréteur. Il s'agit d'un autre facteur pouvant faciliter le développement et la maintenance et qui serait beaucoup plus difficile à exploiter en utilisant une approche basée sur la génération de code.

Cette façon de faire devrait aussi pouvoir faciliter l'enrichissement des fonctionnalités existantes pour une application client. Par exemple, l'ajout de nouveaux mécanismes, comme la saisie semi-automatique devrait être mis en place uniquement au niveau de l'interpréteur et être effectif pour l'ensemble de l'application. L'utilisation du mode interprété devrait alors permettre de réduire considérablement le temps nécessaire au développement et aux tests lorsqu'il est question d'enrichir les fonctionnalités de bases présentes au niveau du client.

Un autre avantage à l'exécution des transformations au moment de l'exécution est qu'il rend possible l'adaptation au contexte d'exécution actuel. Par exemple, la taille de l'écran ou la langue d'exécution sont des paramètres qui peuvent varier d'un endroit à l'autre, la transformation peut alors être adaptée dynamiquement en fonction de ces paramètres. En somme, nous croyons que la transformation du modèle au moment de l'exécution devrait procurer certains avantages au niveau de la maintenabilité, du temps de développement et de l'adaptation à l'utilisateur.

### 6.2.5 Réutilisation de composants

La réutilisation de composants devrait aussi offrir certains avantages au niveau du développement. Pour les tests, le fait de ré-exécuter constamment le même code permettra de réduire de façon considérable le nombre de classes devant faire l'objet de tests. Au niveau de la maintenance et de l'évolution de l'application, le fait de réutiliser des composants existants, qui auront déjà été testés, devrait accélérer de façon importante les efforts nécessaires à l'ajout de nouvelles fonctionnalités ou de nouveaux écrans. Nous supposons donc qu'un gain appréciable devrait se produire à ce niveau.

### 6.2.6 Synthèse

En résumé, l'approche proposée sera axée sur l'utilisation de composants pour lesquels le code d'implémentation pourra être réutilisé à plusieurs reprises. Pour éviter d'avoir à gérer plusieurs versions de l'application, un modèle unique devra être utilisé pour l'ensemble des plateformes supportées. Le volet présentation/style d'affichage devra être pris en charge au moment de la transformation et le tout devra rester indépendant du modèle. Le modèle sera

transformé ou interprété au moment de l'exécution et nous éviterons d'avoir à maintenir du code généré.

## 6.3 Choix technologiques

Pour réaliser la preuve de concept qui permettra d'expérimenter l'approche proposée, plusieurs technologies ont été sélectionnées. Tout d'abord, pour respecter l'environnement technologique imposé par les différents projets qui seront impliqués dans les études de cas, mais aussi pour bénéficier du large éventail de bibliothèques et d'outils de développement ouverts existant, Java a été sélectionné comme langage de programmation. Cette technologie permet, entre autres de bénéficier d'outils de développements parmi les plus avancés au niveau des fonctionnalités d'édition et de débogage des applications. La partie serveur pourra être confinée à la technologie Java. Pour le support éventuel de technologies tierces du côté du client (ex : Flash ou .Net), l'interopérabilité pourra être assurée par l'utilisation d'une technologie de communication commune aux différentes technologies.

Le modèle de haut niveau représentant l'application sera conçu sous forme de classes Java. Des classes seront utilisées pour représenter des pages, des grilles de données, des formulaires, des assistants (wizard) ou tout autre composant. Ce choix technologique comporte de nombreux avantages. Il sera, entre autres possible d'utiliser les outils de modélisation existants, tels que des éditeurs UML, pour la conception du modèle. De plus, la documentation sur les modèles pourra être générée depuis des commentaires inscrits à l'aide du format JavaDoc.

Le fait de lier l'interpréteur à un modèle objet plutôt qu'à un format XML permet aussi d'améliorer l'intégrité de l'application dès la compilation puisque le modèle est compilé. Avec la lecture du modèle XML au moment de l'exécution, sans passer par des objets, le support du modèle par l'interpréteur ne pourrait être validé qu'au moment de l'exécution. Aussi, les inter-références entre entités se représentent mieux avec un modèle objet qu'avec un format XML puisqu'un modèle objet bénéficie de toute la flexibilité des pointeurs alors que des références entre éléments XML doivent être définis à l'aide du concept id/ref ou être représentés sous forme d'éléments enfants. Finalement, même si le modèle est dans un

premier temps implémenté sous forme de classes Java, rien n'empêche de générer aussi un schéma XML à l'aide d'outils existants (Laudati, 2003).

Ce n'est toutefois pas un dialecte XML propre au modèle qui sera utilisé. Il a plutôt été décidé d'utiliser un dialecte existant, indépendant du schéma pour l'instanciation du modèle. La technologie retenue est le Framework Spring qui sera détaillé dans le chapitre suivant. L'utilisation de Spring est justifiée par le fait qu'il offre un dialecte standard pour l'instanciation d'objets couvrant à la fois les propriétés, les types de données et les références entre les objets. Le principal avantage quant à l'utilisation d'un dialecte générique dans le cas présent est que le modèle pourra être très facilement étendu puisque les classes à utiliser sont spécifiées à l'intérieur du fichier de définition. Il sera donc possible d'enrichir le modèle sans toucher au dialecte. Spring pourra aussi être utilisé pour la configuration de l'ensemble des autres composants de l'application, y compris la partie interpréteur, qui pourra être adaptée au type d'application à l'aide du même fichier de définition.

Le dialecte XML sera accompagné d'un autre langage pour la définition des éléments dynamiques comme la liaison entre l'interface et le modèle ainsi que toutes les propriétés pouvant être modifiées dynamiquement (ex : composant actif ou visible selon le contexte). Le langage retenu est Expression Language (EL), définit par le standard JSTL. EL offre un sous-ensemble des fonctionnalités retrouvées dans JavaScript ou XPath. Il permet d'accéder aux propriétés des objets via des accesseurs et d'effectuer des opérations booléennes ou mathématiques. Des opérations de logiques peuvent aussi être codées en EL à l'aide d'une expression conditionnelle « ? ». Les EL pourront être intégrés aux définitions XML et évalués au moment de l'exécution par l'interpréteur.

# CHAPITRE VII

# SPRING

## 7.1 Introduction

Spring est, dans un premier temps, un conteneur Java permettant de gérer le cycle de vie des « beans » (Johnson et Hoeller, 2004) en couvrant l'initialisation, la configuration et offrant une méthode d'accès à ceux-ci. Spring est aussi un Framework offrant des méthodes pour la configuration, le support de l'internationalisation ou la possibilité d'étendre les fonctionnalités offerte. Une caractéristique qui le distingue est son support pour l'intégration et la définition de composants rattachés à une panoplie de technologies tierces. Il supporte, entre autres, les Framework Web, les systèmes de Messaging, les technologies de transaction, les technologies pour l'AOP, des couches de persistances, des tâches planifiées, etc.

Cette technologie est issue de la mouvance des « conteneurs légers », en réaction aux conteneurs d'EJB (Enterprise Java Beans), jugés trop intrusifs (Tate et Gehtland, 2005). En effet, les développements basés sur les EJB sont dépendants de plusieurs services spécifiques au conteneur, tel que les détails de transaction, la persistance, la sécurité, les invocations à distance ou les services de messaging. Les EJB doivent importer les services dont ils ont besoin, implémenter des interfaces et hériter de plusieurs classes de bases fournies par le conteneur (Tate et Gehtland, 2004). Cela fait donc en sorte de restreindre l'exécution des EJB au conteneur auquel ils sont rattachés. Puisque les conteneurs supportant les EJB sont généralement trop lourds pour être démarrés à l'intérieur d'un script de test (Tate et Gehtland, 2005), le niveau de difficulté pour tester les EJB lors du développement est aussi augmenté.

Le développement d'une application sous forme d'EJB impose aussi des limites au niveau du design. En effet, les EJB prennent la forme d'objets représentants des valeurs données. Les modèles objet riches exploitant l'héritage et les interrelations entre les entités du domaine sont beaucoup plus complexes à implémenter via cette technologie (Tate et Gehtland, 2005).

Bref, les conteneurs dit légers offrent, à l'opposé des conteneurs EJB, une méthode simple et non intrusive pour l'instanciation et la gestion des composants de l'application par l'utilisation d'objets conventionnels, souvent appelés POJO (Plain Old Java Object). Leur caractéristique est de limiter les dépendances avec des composants externes et de pouvoir profiter d'une très grande flexibilité puisqu'ils n'ont pas à être codés à l'intérieur d'un cadre strict imposé par le Framework. L'utilisation des POJO dans les conteneurs légers est aussi généralement associée à au concept d'IoC (Inversion Of Control) qui sera détaillé plus tard. Spring, comme conteneur, est aussi beaucoup moins volumineux que des conteneurs d'EJB, faisant moins de 100ko.

Le conteneur Spring assure l'initialisation, la configuration et l'assemblage des composants. Il permet de centraliser tous ces aspects à l'intérieur d'un mode de définition générique uniforme pour l'ensemble des composants de l'application. Le conteneur fait usage des fonctionnalités de réflexion offertes avec Java pour mener à bien ces tâches au moment de l'exécution. La principale utilisation se trouve à être l'assemblage de composants issus de Framework tiers (Tate et Gehtland, 2004). Par exemple, la définition de « pools » de connexion, le « mapping » relationnel / objet, la gestion des transactions et les contrôleurs Web peuvent chacun être assurés par des Frameworks différents. L'assemblage de chacun de ces composants sera alors assuré par Spring.

## 7.2 IOC

Un concept important utilisé dans les conteneurs « légers » est l'IoC, aussi connu sous le terme d'injection de dépendances. Prenons l'exemple d'une classe ayant besoin d'accéder à un service externe, l'approche conventionnelle veut qu'une référence au service direct, à un singleton ou à une « Factory » soit utilisée pour accéder au service en question. La méthode d'accès au service ainsi que la référence au service seront donc incluses dans la classe

appelante, créant ainsi une dépendance. Dans le cas de l'utilisation de Framework particulier (ex : gestion des connexions ou des accès aux données), la classe aura donc une dépendance avec celui-ci, empêchant du même coup un changement de technologie. L'IoC vient corriger ce problème en inversant les rôles. Au lieu que ce soit la classe qui appelle le service requis, c'est le conteneur qui appelle la classe pour lui remettre une instance du service, déjà initialisé. Résultat : des POJO sans dépendances peuvent être utilisés pour le développement de l'application. L'injection se fait en utilisant la réflexion en appelant les méthodes du POJO spécifié dans la configuration du conteneur. On peut résumer le concept par la phrase « Don't call us, we'll call you » (Johnson et Hoeller, 2004).

L'IoC et les conteneurs légers mènent à un nouveau paradigme de développement : la programmation par assemblage d'objets. L'assemblage est ici effectué au moment de l'exécution et peut être reconfiguré sans avoir à recompiler l'application. Lorsqu'on désire modifier le comportement d'une application existante, il est possible de sousclasser le service à redéfinir et de changer uniquement sa référence lors de son initialisation, évitant ainsi toute modification au code de l'application.

Un grand avantage se situe aussi au niveau des tests unitaires. Le fait que les classes n'aient qu'un minimum de dépendances envers des outils externes et qu'elles ne nécessitent pas absolument l'intervention d'un conteneur pour leur initialisation facilite grandement les tests (Tate et Gehtland, 2005). Il est possible d'écrire des tests unitaires effectuant l'initialisation d'une classe spécifique avec les paramètres nécessaires aux tests. Le fait qu'un conteneur léger consomme peu de ressource et s'exécute rapidement rend aussi possible l'utilisation de ce dernier à l'intérieur des tests unitaires. Une configuration d'assemblage différente, exploitant des classes mieux adaptées aux tests, peut alors être utilisée.

## 7.3 AOP

Un autre paradigme souvent présent dans les conteneurs légers est AOP. La programmation par aspect permet de redéfinir le comportement d'une classe au moment de l'exécution, ayant aussi pour effet de réduire les dépendances avec les bibliothèques externes ainsi que la taille du code (Pawlak, Seinturier et Retaillé, 2005). Un aspect représente un comportement donné, défini séparément et pouvant être appliqué à une classe ou méthode

après la compilation. Par exemple, il peut s'agir du code utilisé pour la journalisation (logs) à exécuter au début et à la fin de chaque méthode. Au lieu de réécrire les appels au logger, précisant, à chaque fois, le nom de la méthode et le texte mentionnant que l'application débute et termine de celle-ci, un aspect couvrant ce comportement pourrait être écrit et appliquer dynamiquement sur les méthodes spécifiées.

AOP repose, dans un premier temps, sur un format de définitions permettant d'identifier et spécifier les aspects à appliquer sur les différentes classes et instances. Il est possible de spécifier les méthodes concernées par l'AOP et le moment de l'exécution des aspects (avant, après ou au moment d'un appel des méthodes). L'application des aspects, communément appelés « wive » est effectué par le Framework AOP utilisé au moment de l'exécution ou en mode postcompilation. Lorsqu'utilisée au moment de l'exécution, une classe « Proxy » est alors générée dynamiquement par le Framework. Cette dernière étendra la classe sujette à l'AOP et prendra la place de cette dernière. Les méthodes générées par la classe proxy effectueront l'exécution du traitement rattaché à l'aspect et appelleront la classe originale au moment opportun. Lorsqu'appliqué après la compilation, l'outil AOP ajoutera le code à exécuter au niveau du bytecode produit par le compilateur Java. Ce mode offre une moins grande flexibilité, car il retire la possibilité de reconfigurer les aspects sur un code existant. Par contre, l'ajout des aspects au niveau du bytecode permet d'obtenir de meilleures performances (Pawlak, Seinturier et Retaillé, 2005).

Voici un exemple montrant l'état de la pile d'exécution nécessaire à l'appel de « HibSeqActivityTreeDao.create » faisant l'objet d'un traitement AOP depuis la méthode « RunState.processActionPopup »

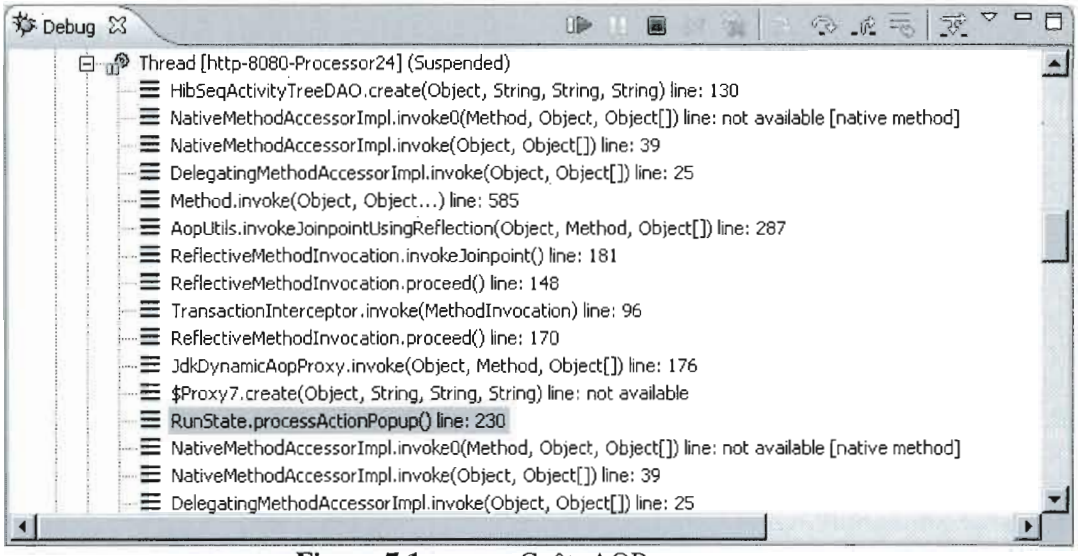


**Figure 7.1** Coût, AOP

Les exemples d'utilisation d'AOP sont nombreux. Il peut être utilisé pour des fins de validation de sécurité, de support des transactions ou pour la gestion des exceptions. L'implémentation de certaines interfaces peut aussi être effectuée à l'aide d'AOP. Par exemple, java.beans.PropertyChangeSupport, imposant un mécanisme d'enregistrement d'observateurs et de notifications de ces derniers en cas de changement, peut être ajouté entièrement à l'aide d'AOP.

Spring offre son propre mécanisme AOP dont la spécification est effectuée à l'intérieur du format XML du Framework. Seul le mode d'exécution dynamique est supporté. Les aspects, les classes proxy ainsi que les beans visés par l'AOP sont initialisés et paramétrés au même endroit. Il est aussi possible d'utiliser le Framework AspectJ à l'intérieur d'une application Spring (Pawlak, Seinturier et Retaillé, 2005). Les Framework AOP ne sont qu'un exemple d'outils externes supportés par Spring.

## 7.4 Outils disponibles

Spring offre une couche d'abstraction pour la plupart des Framework utilisés couramment dans les applications Java, et ce, pour une panoplie de services différents. En effet, en plus du conteneur de beans, Spring est livré avec des couches d'abstractions génériques couvrant, entre autres, les couches de persistances, les accès JDBC, les systèmes

de Messaging (pour l'intégration d'application d'entreprise), les transactions, les « pools » de connexions, les contrôleurs Web, les tâches planifiées, etc.

La logique est la suivante : lorsqu'on développe une application nécessitant une dépendance avec un outil particulier (ex : pool de connexions), au lieu de traiter directement avec l'outil en question, il est ici possible d'utiliser une interface offerte avec Spring. Au lieu d'avoir une dépendance avec l'outil utilisé, nous aurons alors une dépendance avec l'interface. Si l'outil est changé (ex : utilisation d'une source JNDI au lieu de DBCP), seule l'implémentation de l'interface Spring doit être changée. Une stratégie similaire est employée pour la gestion des exceptions par l'utilisation de classes d'exceptions fournies par Spring. Les erreurs du Framework externe sont encapsulées, ce qui fait que l'application ne doit connaître que les exceptions Spring. Il s'agit ici d'une autre technique permettant de limiter les dépendances entre une application et ses bibliothèques externes. Dans les prochains paragraphes, nous effectuerons un survol des principaux types de Framework supportés.

La *gestion des transactions* permet d'exécuter une série d'opérations qui seront appliquées ou annulées de façon atomique. Il peut y avoir des transactions locales, se limitant à une base de données par exemple, ou des transactions globales gérées au niveau application qui peuvent impliquer un plus grand nombre de ressources. Spring offre le concept de « stratégie de transaction » pouvant être implémentés par différents framework de gestion des transactions existants comme Hibernate ou JTA.

Pour l'*accès aux données*, Spring offre le concept des DAO (Data Access Object) qui permet de changer la technique de persistance avec un minimum d'impact. Spring rend entre autre disponible une interface ainsi que plusieurs exceptions devant être utilisées par les technologies de persistance. Des implémentations sont entre autres livrées pour les technologies JDBC, Hibernate, JDO, TopLink, JPA et iBatis; Elles offrent, entre autres, le support pour la conversion des exceptions propres à ces Framework vers des exceptions reconnues par Spring ainsi que, pour les Framework de « mapping » relationnel-objet, la gestion des sessions.

Pour le développement d'*application Web* basée sur les Servlets ou les Portlels, Spring est livré avec son propre Framework implémentant une architecture de type MVC. Spring Webmvc offre des mécanismes de validation, de gestion des formulaires et de gestion des actions côté contrôleur. Côté JSP, des « tags » peuvent être utilisés pour la liaison entre les éléments du modèle et les champs de la page et pour le support de certaines fonctionnalités comme l'internationalisation.

Des frameworks Web externes peuvent aussi être intégrés à Spring. Il est entre autre possible d'utiliser d'autres méthodes de définition que JSP pour la génération de la partie vue, comme Velocity ou XSL. Aussi, des frameworks Web complets comme Struts ou JSF peuvent s'intégrer à différents niveaux avec Spring, allant de l'accès aux ressources jusqu'à la définition des « beans » à même le dialecte de Spring. La définition des pages et des contrôleurs et du flux de navigation peut rester selon la configuration propre au framework, comme c'est le cas pour JSF.

Plusieurs autres catégories de framework peuvent être configurées via Spring. C'est entre autres le cas pour JMS, une norme Java pour l'échange de messages entre applications ou Quartz pour l'exécution de tâches planifiées. Dans tous les cas, Spring fournit un niveau d'abstraction supplémentaire afin de simplifier l'intégration et la configuration à l'aide du format XML de Spring.

## 7.5 Dialecte XML

Le format XML accompagnant Spring couvre l'initialisation et la configuration des composants (beans) de l'application. Il offre une méthode générique pour l'écriture des propriétés et l'appel des constructeurs. La principale valeur ajoutée par rapport à d'autres méthodes d'initialisation d'objets, comme Digester, (commons.apache.org, 2007) se trouve au niveau de la définition des références entre les objets ainsi que des collections. Cette fonctionnalité n'est couverte par la norme des schémas XML (wikibooks.org, 2006) que pour la validation de la contrainte. Son support dépend donc de la bibliothèque de lecture du XML. À l'intérieur du dialecte de Spring, pour chaque propriété, il est possible de définir une référence avec un objet existant ou de placer le code XML définissant les données à y placer

lors de l'instanciation. Ces fonctionnalités constituent le nécessaire à la composition d'applications.

Voici, de façon plus détaillée, la liste des fonctionnalités pouvant être utilisées dans le dialecte XML. Le contenu présenté ici est issu de la documentation officielle de Spring (SpringFramework, 2007).

```
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

C'est un exemple de base pour l'instanciation d'une classe Java. Il suffit de spécifier le nom complet de la classe et de lui donner un nom qui sera utilisé à l'intérieur de Spring. Le nom du « bean » est utilisé dans le dialecte pour effectuer des références ainsi que dans le code de l'application, pour l'identification du « bean » à récupérer. Le tag <bean> peut contenir des éléments pour l'accès au constructeur ainsi que pour la définition des propriétés. Une propriété peut, soit contenir une référence vers un autre « bean » ou inclure une nouvelle définition de « bean » sous forme d'un élément enfant.

```
<bean name="foo" class="x.y.Foo">
  <constructor-arg>
    <bean class="x.y.Bar"/>
  </constructor-arg>
  <constructor-arg>
    <bean class="x.y.Baz"/>
  </constructor-arg>
</bean>
<bean id="exampleBean" class="examples.ExampleBean">
  <property name="bean1"><ref bean="anotherExampleBean"/></property>
  <property name="bean2">
    <bean class="com.mycompany.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

Les collections java sont aussi supportées dans le dialecte de Spring. Une propriété de type Map, Set ou List peut donc être initialisé directement à l'intérieur des éléments de la définition du bean.

Exemple:

```
<property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
        <entry>
            <key>
                <value>yup an entry</value>
            </key>
            <value>just some string</value>
        </entry>
        <entry>
            <key>
                <value>yup a ref</value>
            </key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
```

D'autres éléments du langage permettent l'importation de ressources définies à l'intérieur de fichiers externes, avec le tag <import>. Le tag <alias> permet la définition d'un deuxième nom d'accès à un « bean » déjà défini. Un autre concept utilisé est celui d'héritage entre la définition des différents « beans ». Par exemple, si plusieurs « beans » partagent un même sous-ensemble de propriétés, il est possible de définir un « bean » parent et d'ajouter l'attribut « parent » avec référence à celui-ci pour que les propriétés soient héritées.

Les « beans » peuvent aussi avoir, à l'aide d'un attribut « scope », un contexte d'initialisation géré par le conteneur. Un « bean » peut être de type singleton, pour faire en sorte que l'instance retournée par le conteneur soit toujours la même ou de type prototype générant une nouvelle instance à chaque accès. D'autres modes sont offerts pour une utilisation à l'intérieur d'une application Web. L'attribut « lazy-init » permet de spécifier, dans le cas d'un singleton, si le « bean » sera créé au démarrage de l'application ou lors de son premier accès.

Le Framework Spring et son mode de définition peuvent aussi être étendus par l'utilisation de classes de type « Configurer », « EditorSupport » ou « Processor » redéfinissant un comportement. Les « Configurer » permettent de spécifier les méthodes d'accès aux configurations, les classes « EditorSupport » permettent de spécifier de quelle manière initialiser le contenu texte des propriétés en fonction des types des attributs des objets. Les interfaces de type « Processor » permettent d'appliquer un comportement personnalisé à différents moments pendant le processus d'initialisation.

Même si Spring favorise les pratiques permettant d'éviter l'introduction de dépendances, certaines fonctionnalités permettent de simplifier la gestion des « beans » en implémentant des interfaces fournies par le Framework. Par exemple, l'interface NameAware fait en sorte que le nom du « bean » sera injecté au moment de l'initialisation. Situation similaire pour ApplicationContextAware assurant l'injection du contexte de l'application.

D'autres schémas peuvent être ajoutés au dialecte. La version 2 du Framework a justement introduit de nouveaux espaces de noms, soit, « util », offrant des raccourcis pour la lecture de constantes ou l'initialisation de collections comme un « bean », « jee » pour l'accès à des ressources par JNDI et « lang » pour l'exécution de scripts dans des langages interprétés autres que Java. Des « NamespaceHandler » peuvent être écrits pour supporter d'autres schémas.

L'initialisation du conteneur Spring peut se faire directement depuis une classe Java à l'aide de l'appel suivant :

```
BeanFactory factory = new XmlBeanFactory(fichierXML);
```

## 7.6 Conclusion

En somme, le dialecte offert avec Spring offre des outils et formats génériques permettant l'assemblage de composants pour la construction de tout type d'application Java. Le Framework a réussi à s'imposer de façon considérable, en faisant l'un des conteneurs d'application Java les plus utilisés. Les statistiques tirées des offres d'emplois aux États-Unis confirment la progression que connait cette technologie.


**Figure 7.2** Popularité Spring vs EJB

Le fait d'utiliser un schéma standard et répandu offre plusieurs avantages par rapport à des techniques conventionnelles comme Digester. En effet, l'utilisation d'un format unique encourage la création d'outils d'édition plus efficaces. Certains éditeurs, comme Spring-IDE (springide.org) peuvent s'intégrer à l'environnement de développement Eclipse et être reliés directement au « classpath » de l'application utilisant Spring. Cela permet d'offrir les fonctionnalités de validation et de complétion automatique rattachée aux classes Java, et ce, de façon complètement dynamique. Par exemple, si une propriété utilisée dans un fichier Spring est renommée dans une classe Java, l'éditeur Spring, lors de la validation, détectera que le nom de la propriété ne concorde pas et lancera une erreur de compilation. Cela signifie donc que, même si l'initialisation et l'assemblage de l'application s'effectuent au moment de l'exécution, le compilateur associé à l'éditeur est en mesure de détecter les erreurs au moment du développement.

Spring est largement utilisé dans le domaine des applications Java à code source libre. Par exemple, dans le système de gestion documentaire Alfresco (alfresco.com), c'est ce dialecte qui est utilisé pour la définition de tous les composants et services utilisés par le référentiel. La sécurité et la gestion des transactions exploitent la technologie AOP livrée avec Spring. Le projet uPortal a commencé, depuis quelques années, à remplacer des configurations XML, utilisées par certains de ses services, par le dialecte Spring. Un autre

projet de la communauté JA-SIG, CAS, repose entièrement sur Spring pour la configuration de ses services et de ses méthodes de connexions.

# CHAPITRE VIII

# ÉTUDE DE CAS

## 8.1 Introduction

Cette section présentera trois projets de développement réels exécutés au cours de l'année 2007 au Centre de Recherche Informatique de Montréal (CRIM). À l'intérieur de ces projets, l'approche présentée dans les chapitres précédents a fait l'objet de deux implémentations distinctes pour couvrir des besoins différents.

L'approche conceptuelle définie précédemment propose une solution répondant aux objectifs de ce projet de recherche. Le fait de réaliser avec succès des prototypes basés sur cette approche permettra alors de répondre aux objectifs fixés en matière de définition d'interface utilisateur indépendante d'une technologie de présentation. D'autre part, pour répondre aux autres objectifs visés en matière de productivité et de qualité logicielle, un cadre d'analyse sera utilisé pour valider les résultats obtenus. Les critères suivants seront donc évalués et mesurés :

1. Le niveau de productivité pour les développements réalisés,
2. La maintenabilité du code de la partie client des applications,
3. La réutilisabilité du code de la partie client des applications,
4. La testabilité des applications développées selon l'approche présentée,
5. Le niveau de portabilité des prototypes vers les différentes plateformes d'exécutions.

Dans un premier temps, les différentes études de cas seront présentées et détaillées. Pour chaque projet, les étapes du développement, le modèle utilisé, l'interpréteur développé ainsi que l'application, basée sur le modèle, seront couverts. Par la suite, les résultats obtenus dans l'ensemble des projets seront évalués en fonction des critères mentionnés. Des comparaisons avec d'autres projets similaires seront utilisées afin de situer les cas étudiés et de constater s'il y a amélioration par rapport aux approches conventionnelles. Toutes les informations nécessaires à l'étude ont été recueillies lors de la réalisation des projets.

## 8.2 Outils ETL / Tableau de bord du gestionnaire

### 8.2.1 Description du projet

L'application ETLTools a été développée dans le cadre d'un projet d'intelligence d'affaire (Business Intelligence) visant la mise en place d'un entrepôt de données et d'un tableau de bord pour les gestionnaires de commissions scolaires. L'objectif du projet était de présenter des indicateurs de gestion couvrant divers domaines propres au domaine scolaire, tels que, les effectifs, les caractéristiques de la population étudiante, la réussite et l'absentéisme. Les technologies sélectionnées pour ce projet sont Mondrian, JPivot et OpenI; trois produits s'exécutant sous forme d'application Web Java. La base de données utilisée pour héberger l'entrepôt de données est MySQL.

Un entrepôt de données est une base de données accueillant une copie des données provenant de différentes sources, généralement des bases de données opérationnelles d'une organisation. Le rôle de cette BD est de conserver un grand volume de données historiques pouvant être utilisées pour la prise de décision (Godin, 2003). Cela permet aussi d'éviter d'encombrer les bases de données de production avec des requêtes d'analyse qui peuvent parfois être lourdes. Un autre avantage est de permettre le stockage de données sous une forme plus appropriée pour l'exécution de requête d'analyse (Godin, 2003).

Mondrian est une implémentation libre d'un serveur OLAP (*Online analytical processing*) permettant l'analyse de données multidimensionnelles. La technique d'accès aux données utilisée est plus spécifiquement ROLAP, qui se trouve à être une couche

d'abstraction OLAP sur une base de données relationnelle. Des fichiers de définition XML définissent les schémas multidimensionnels utilisés, communément appelés cubes de données. Ces derniers effectuent la correspondance entre les dimensions du schéma et les colonnes des tables de l'entrepôt de données.

L'interface utilisateur utilisée pour la navigation dans l'entrepôt de données est JPivot. L'outil offre les fonctionnalités nécessaires pour la configuration et l'affichage de graphiques et de tableaux croisés sur les données. En fonction des critères définis par l'utilisateur, des requêtes MDX sont générées et exécutées par le serveur OLAP.

Le produit final utilisé pour la formation du tableau de bord est OpenI. Il s'agit d'une application Web basée sur le Framework Spring et qui permet la définition d'indicateurs de gestion, basés sur JPivot. Un indicateur se compose d'un graphique et d'un tableau de données. L'administrateur du tableau de bord peut définir des indicateurs via l'interface Web pour ensuite les rendre accessibles à d'autres utilisateurs. Ceux-ci ont aussi la possibilité de créer leurs propres indicateurs et de consulter des tableaux effectuant l'agrégation d'un certain nombre d'indicateurs sous forme d'aperçus.

Le principal obstacle à la mise en place d'une telle solution est généralement le fait que les sources de données ne sont pas homogènes. Celles-ci doivent passer par une série de transformations dirigées par des règles ainsi que des tables de correspondances avant d'êtres transférées à l'entrepôt de données. Le cas des commissions scolaires québécoises est particulier puisqu'elles utilisent pratiquement toutes un même système de gestion pédagogique : GPI, développé par la société GRICS. Les schémas relationnels des bases de données opérationnelles des écoles sont, à quelques détails prêts identiques. Par contre, la solution GPI a comme caractéristique de laisser aux écoles le soin de définir ses propres listes de valeurs pour la majeure partie des données qu'elles contiennent. Ces listes de valeurs sont donc différentes d'une école à l'autre et même d'une année à l'autre pour une même école. Ces valeurs sont, entre autres, les codes donnés aux absences, les codes de cours, les codes d'objectifs spécifiques aux cours, etc. L'entrepôt de données doit contenir les données recueillies des différentes sources de données de manière uniforme afin de pouvoir effectuer

des comparaisons entre les écoles. Des interventions manuelles importantes sont donc nécessaires afin de paramétrer les règles d'importation des données.

Le projet comporte plus de soixante tables sur les dimensions et les faits à étudier. Le modèle a été pensé de manière à ce qu'il puisse être déployé facilement dans n'importe quelle commission scolaire. Puisque les données changent d'un endroit à l'autre, il est nécessaire d'offrir un éditeur permettant de modifier les données des dimensions et d'effectuer le paramétrage nécessaire à l'importation des données provenant de chacune des écoles.

Environ 45 tables contiennent des données devant faire l'objet d'édition ou de validation manuelle. Les besoins se rapprochaient davantage à ceux d'une application Web pour l'édition de données qu'à ce qu'un client SQL générique à la Microsoft Access pouvait offrir. En effet, les données éditées peuvent être sujettes à des politiques de sécurité et des règles de validation personnalisées. De plus, des tâches autres que l'édition, telle que le lancement manuel d'une tâche d'importation s'exécutant du côté du serveur, devaient pouvoir y être gérées. Le tout devait être convivial et offrir les outils permettant de naviguer à l'intérieur des différents ensembles de données.

C'est dans ce contexte qu'une première expérimentation de l'approche de développement par modèle a été effectuée. L'objectif étant de définir une application d'édition de données complète offrant un mécanisme d'authentification et un grand nombre de pages d'édition de données supportant soit les opérations conventionnelles (CRUD - Create, Read, Update et Delete) ou simplement la possibilité de lire des données provenant d'une ou de plusieurs tables. L'application devait se distinguer d'un simple client d'accès à une base de données en offrant une structure de navigation entre les pages, de l'aide contextuelle et beaucoup d'options pour le paramétrage des valeurs éditables. En effet, chaque colonne peut présenter un nom significatif, être visible ou non, avoir une liste de valeurs possibles, comporter des règles de validation complexes ou comporter une valeur non éditable déterminée par l'application. D'autres fonctionnalités, comme le lancement manuel d'une importation de données ou la consultation d'un journal d'événement furent nécessaires à l'application. L'internationalisation devait aussi pouvoir être supportée.

Un autre volet rattaché au même projet est celui d'une application de transformation ETL basée sur Spring. La décision de construire un outil pour l'ETL est venue à la suite de l'étude des technologies Open Source existantes. Les solutions les plus complètes, comme « Pentaho Data Integration » offrait des performances de bases décevantes pour les transformations simples. Les transformations plus complexes devaient, elles, être codées en JavaScript et offrent des performances désastreuses pouvant prendre près d'une seconde par ligne traitée. Dans le contexte d'un entrepôt de données comptant plusieurs millions d'enregistrements, une solution plus performante était recherchée.

Les outils ETL développés offrent quelques algorithmes d'importation, de synchronisation et de transformation simples, mais performants. Le langage utilisé pour les transformations plus complexes, impliquant de la logique, est EL (*Expression Language*). Le système s'adapte aussi parfaitement au contexte d'utilisation qui est constitué de plusieurs bases de données similaires comportant chacune des listes de valeurs distinctes et, donc, des tables de correspondances qui leur sont propres. Les outils développés permettent la réutilisation de règles d'importation en exploitant des tables de correspondances dépendantes des sources de données impliquées. Finalement, le fait de baser l'outil d'ETL sur le Framework Spring aura eu pour effet de faciliter l'intégration de la partie ETL à l'application éditeur, les deux étant basés sur les mêmes technologies.

### 8.2.2 Technologies utilisées

Les principaux Frameworks sur lesquels se basent l'application sont iBatis, Abator, Spring et Spring WebMVC. D'autres outils, tels que Quartz et les composants « Commons » d'Apache ont aussi été utilisés. Quartz (opensymphony.com/quartz) permet la gestion de tâches planifiées ainsi que des mécanismes avancés de contrôle pour l'exécution de tâches. Commons (commons.apache.org) est composé d'un grand nombre de composants réutilisables et comportant un minimum de dépendances avec des bibliothèques externes. Pour le présent projet, les bibliothèques commons-el, commons-pool et commons-logging ont été utilisés.

iBatis est une bibliothèque pour la gestion de la persistance similaire à Hibernate mais offrant le plein contrôle sur les requêtes exécutées sur la base de données. Il ne s'agit donc

pas d'un « mapping » relationnel objet conventionnel. Il se définit comme un outil de « Data Mapper » faisant le lien entre un modèle objet côté application et des requêtes SQL exécutées pour la matérialisation / dématérialisation des objets.

Le principe retenu pour l'accès aux tables SQL, utilisé à la fois pour l'éditeur et les outils ETL, est le suivant : Les tables sont définies à un seul endroit, à l'intérieur d'un fichier de définition Spring, pour ensuite être réutilisées par les autres composants de l'application. L'application ne connait donc pas les données sur lesquelles elle sera exécutée. Le code rattaché à l'éditeur et aux outils ETL est donc indépendant des données qu'il aura à traiter.

Une autre technologie reliée à iBatis a été intégrée à l'application. Il s'agit de Abator, un générateur de couche de persistance pour iBatis. Abator est généralement utilisé lors du développement pour générer la configuration iBatis ainsi que les classes Java qui correspondent aux tables de l'application. Il peut être appelé depuis un script de compilation Ant après avoir effectué la configuration de base comprenant les accès à la base de données et les modèles à utiliser pour la génération. Dans notre application, nous avons effectué l'intégration de la bibliothèque Abator, de manière à pouvoir l'utiliser au moment de l'exécution. Lorsque l'accès à une table doit être effectué, si le fichier de configuration iBatis associé à la table est inexistant, Abator est alors appelé pour que celui-ci soit généré en fonction des paramètres de configuration présents dans Spring. Il s'agit d'un endroit où les concepts de la méta-programmation ont été utilisés.

iBatis permet d'utiliser des objets ou des collections de types « Map » pour l'interrogation et la récupération des données. L'utilisation de classes spécifiques au domaine est fortement recommandée puisqu'elles permettent d'assurer l'intégrité de l'application au moment de la compilation. Ces classes peuvent aussi être réutilisées à l'intérieur de la logique d'affaire pour l'exécution du traitement. Dans le cas présent, oùu les tables et les données utilisées peuvent être modifiées après la compilation et prises en compte au moment de l'exécution, l'utilisation de classe Java était moins appropriée. Il a donc été décidé d'utiliser des objets de type « Map » pour la représentation de l'ensemble des données échangées à l'intérieur de l'application. Les classes effectuant un traitement sur des données ont donc été

développées autour de ce type de données, favorisant du même coup le développement de code générique et la réutilisation de code.

Spring, tel que décrit dans le chapitre précédent, a été utilisé pour centraliser la configuration, mais aussi la définition complète des applications développées sous ETL Tools. WebMVC a, de son côté, servi de base au développement de la couche Web de l'éditeur. L'internationalisation, la configuration des contrôleurs et le « binding » ont, entre autres, été assurés par l'utilisation du Framework WebMVC de Spring.

### 8.2.3 Architecture de la solution

L'application se divise en 4 grandes catégories. Le « package » « editor », comprenant le modèle générique et une implémentation Web pour l'exécution de l'application, les packages « table » et « connections » pour l'accès aux données, le package « ETL » comprenant le code générique utilisé pour la transformation, et les packages utilitaires (« util », « el » et « Abator ») permettant de contrôler l'accès et la configuration de certaines bibliothèques ou d'offrir des fonctionnalités réutilisables pour l'ensemble de l'application.

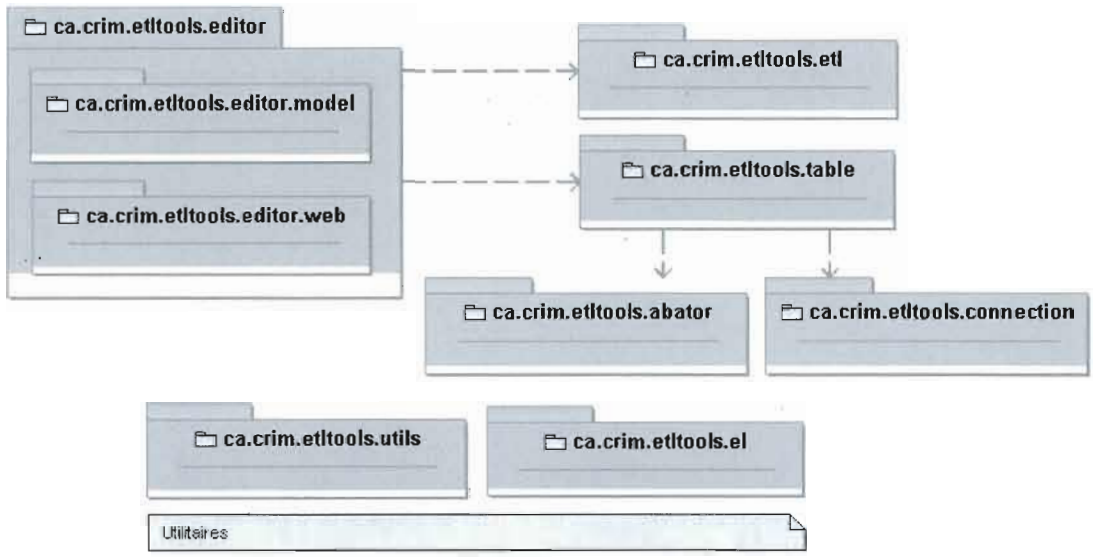


**Figure 8.1** « Packages » ETLTools

L'éditeur constitue la partie de l'application visée par cette preuve de concept. Il s'agit d'une application d'édition de données dont le contenu, l'affichage et les fonctionnalités disponibles à l'intérieur de chacun des écrans sont définis à l'aide d'un modèle objet, instancié à l'aide de Spring. Il couvre la structure de navigation ainsi que le contenu des pages impliquées dans l'application. Une approche par composants est utilisée pour représenter le contenu à afficher dans chacune des pages.

Le modèle comporte aussi certains liens externes. Par exemple, les composants permettant d'exécuter directement une transformation ETL ou d'interagir avec des données auront des attributs pointant sur des classes issues d'autres packages mais toujours instanciées via Spring. En utilisant les éditeurs appropriés, la validation de ces liens peut alors être effectuée au moment du développement.

Après le chargement du modèle, au moment de l'exécution, la partie « interpréteur » de l'application entre en jeu. À l'heure actuelle, un seul interpréteur, dédié aux applications Web, a été développé. Il comporte un modèle qui lui est propre, destiné à contenir l'état de l'application au moment de l'exécution. Un contrôleur ainsi qu'une couche de présentation générique font aussi partis de l'interpréteur.

Le package « table » offre une interface d'abstraction sur les sources de données utilisées par l'interface Web et par les outils ETL. L'interface impose la définition de méthodes de bases (select, selectAll, insert, detele, update) et permet aussi de spécifier des paramètres de sécurité au niveau de la table, comme la restriction « lecture seule » qui bloque systématiquement l'exécution de modifications sur les données. L'application est livrée avec deux types de tables : « Dynamic » et « Custom ». Pour les tables dynamiques, toutes les requêtes nécessaires sont générées au moment de l'exécution via l'outil Abator. Certaines données supplémentaires, comme les clés primaires utilisées ou la méthode d'obtention de la prochaine valeur d'une séquence, doivent être spécifiées lorsque la table est définie à l'aide de Spring. Une table de type « Custom » doit être rattachée à un fichier de configuration et un espace de nom iBatis dans lequel les méthodes de bases énoncées précédemment (select, insert, ...) sont définies. Dans tous les cas, la liste des colonnes n'a pas à être connue de la

table au moment du développement. C'est au moment de l'exécution que celles-ci seront chargées par l'objet table.

Les tables sont étroitement reliées avec un gestionnaire de connexion ayant pour objectif de permettre l'accès aux sources de données de façon transparente. Celui-ci effectue une mise en cache des connexions utilisées au niveau du « thread » courant et, si nécessaire, sélectionne la base de données à utiliser en fonction de l'environnement d'exécution.

Le volet sécurité et authentification est configuré directement au niveau des plateformes utilisées, via un écran d'authentification spécifique. Dans le cas de l'interpréteur Web, générant du HTML simple, la sécurité a été implémentée conformément à la norme J2EE, via une configuration au niveau de web.xml. Une validation des droits d'accès sur chacun des écrans de l'application pourra être sous la responsabilité de l'interpréteur.

Un élément important utilisé au niveau des interpréteurs, mais qui influence aussi la définition du modèle est l'utilisation d'un environnement d'exécution accessible depuis les composants au moment de l'exécution. Ce concept s'apparente à celui de la session côté serveur dans une application Web. Il contient, par exemple, les données rattachées à la page éditée. L'interpréteur doit maintenir l'environnement d'exécution et y donner accès depuis le langage EL (Expression Language). Les expressions EL couvrant des conditions et des valeurs calculées peuvent donc être spécifiées à l'intérieur du modèle donc, dans le fichier de définition Spring.

L'internationalisation est prise en compte par la solution. Chacune des instances du modèle comporte un identifiant unique, un « id » géré par Spring. La méthode d'internationalisation utilisée consiste simplement à permettre la spécification de certains attributs du modèle, comportant du texte spécifique à la langue, à l'intérieur d'un fichier de ressource. La clé utilisée dans le fichier doit alors suivre la nomenclature « id » + attribut. La sélection de la langue et le chargement des ressources associées au modèle doivent aussi être assurés par l'interpréteur. Les « id » des composants définis à l'intérieur de Spring, nécessaires pour l'accès aux ressources, sont récupérés par le modèle via l'utilisation de

l'interface « BeanNameAware », ajoutant une méthode appelée par le Framework pour y spécifier le nom de l'instance lors du chargement.

### 8.2.4 Le modèle

Pour la définition de l'application éditeur, voici le modèle ayant été utilisé :

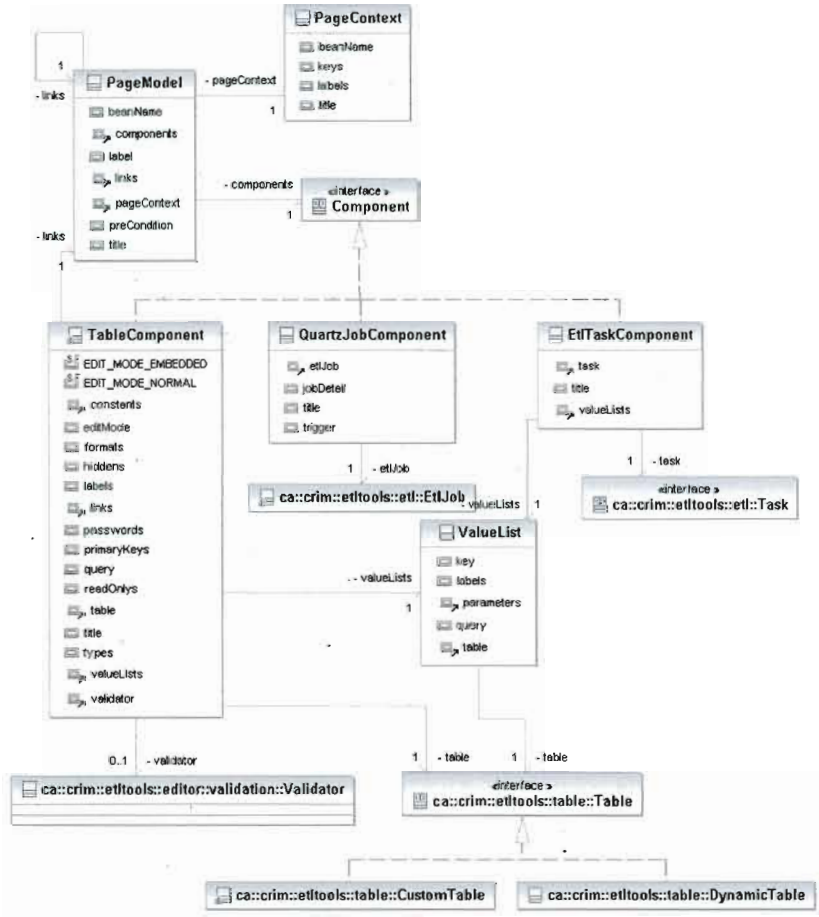


**Figure 8.2** Modèle ETLTools

Le point de départ pour l'application est une page du modèle. L'application recherchera une page nommée « page.root » qui sera utilisée comme page d'accueil. Une page contient un certain nombre d'attributs, comme une étiquette (utilisé pour identifier la

page lorsqu'un celle-ci fait l'objet d'un lien) et un titre (texte en entête de la page).
Une page peut être disponible ou non selon le contexte d'exécution. Si une expression EL est spécifiée dans l'attribut « preCondition », celle-ci doit être vraie pour que la page soit visible. Une page peut donner accès à d'autres pages via l'attribut « link ». L'interpréteur a alors la charge de présenter la liste des sous-pages disponibles ainsi que d'afficher les éléments de navigation permettant à l'utilisateur de connaitre l'emplacement de l'application dans lequel il se trouve et de revenir à un niveau supérieur. Un composant spécifique, greffé à la page, permet de définir les données du contexte devant être présentées à l'utilisateur. Par exemple, lorsque les données propres à une école sont éditées, les pages concernées seront rattachées à une instance « PageContext » définissant les éléments de l'environnement d'exécution à afficher ainsi que leur format d'affichage.

Un deuxième concept important est celui des composants qui sont placés à l'intérieur des pages. Un composant définit une section de l'écran contenant un ensemble de fonctionnalités. Les composants ont accès au contexte de la page, mais restent autonomes et ne comportent pas de liens entre eux.

Le principal composant utilisé est « TableComponent ». Celui-ci comporte l'ensemble des fonctionnalités pour l'affichage, la suppression et la modification de données provenant d'une table. La définition du composant est d'ailleurs rattachée à une table (section distincte de l'application) et à une liste d'attributs permettant de personnaliser la liste des fonctionnalités qui seront disponibles au moment de l'exécution. Une particularité de l'application ETLTools est que le contenu des tables est connu au moment de l'exécution. Pour ajouter une nouvelle colonne à éditer, il suffit d'effectuer l'ajout au niveau de la base de données, l'interface utilisateur sera mise à jour au prochain redémarrage de l'application. Cette façon de faire permet de réduire grandement le nombre de lignes de code puisque les détails des colonnes n'ont pas à être spécifiés. Par contre, la méthode utilisée pour définir les attributs propres aux colonnes doit être revue en conséquence. Au lieu d'utiliser une liste de colonnes accompagnées de leurs attributs, le modèle contient une liste de colonnes étant « readOnly », une liste de colonne ayant l'attribut « hidden », une liste de colonnes ayant l'attribut « password » etc. Les types utilisés pour l'édition peuvent aussi être spécifiés au niveau du modèle. Cela prend la forme d'un dictionnaire (clé, valeur) dans lequel le nom de

la colonne est associé au type utilisé. Cela permet, par exemple, qu'une colonne X, même si elle est détectée comme un champ texte, soit éditée comme un booléen. Les tables supportent aussi des constantes, définies sous forme d'un dictionnaire, permettant de spécifier des valeurs fixes à placer dans les colonnes lors de l'insertion de données. Cet attribut permet l'utilisation d'expressions EL pour l'accès à des valeurs provenant du contexte d'exécution. L'attribut « links » permet de spécifier des pages pouvant être accédées via une ligne de la table, ce qui offre la possibilité d'afficher des détails relatifs à un enregistrement. « Query » permet de spécifier les paramètres à utiliser lors de la requête. Le contenu du champ « Query » sera converti en Map pour l'exécution de la commande « SelectByExample » d'iBatis. Les expressions EL peuvent toujours y être utilisées pour accéder à des éléments de l'environnement d'exécution. « valueList » offre la possibilité de spécifier, pour une colonne, une liste de valeurs possibles, provenant d'une autre table de l'application. Finalement, il est possible de spécifier une implémentation à utiliser comme « validator » pour la validation des données saisies. Ce concept s'additionne à la validation de base pour les champs obligatoires et les types de données en offrant la possibilité d'avoir une logique plus poussée, comme, par exemple, le test d'une connexion JDBC. Les classes du package « validation » sont destinées à être exécutées du côté du serveur et peuvent donc faire appel à tout type de bibliothèques.

Le composant « EtlTask » permet de lancer l'exécution d'une tâche de transformation qui aura été définie à l'aide d'objets du package ETL. Le fait que les tâches soient présentes dans la même configuration Spring permet d'assurer l'intégrité des liens entre ceux-ci et les composants « EtlTask » au moment de la compilation via l'outil SpringIDE. Ce composant permet donc de spécifier la tâche à exécuter et offre aussi la possibilité de demander à l'utilisateur la sélection de valeurs, provenant d'une liste de type « ValueList » avant le lancement de la tâche. L'environnement d'exécution initial pour l'exécution de la tâche sera le même pour que la page affichée. Cet environnement contient les données de chacune des tables par lesquelles l'utilisateur est passé pour se rendre à la page courante. Par exemple, si l'utilisateur clique sur la ligne « École ABC » de la table « Écoles », l'environnement de la page suivante contiendra l'ensemble des colonnes de la table « Écoles » pour la ligne de l' « École ABC ». Ces données, que l'on appelle « environnement » sont accessibles par les composants d'une page et peuvent être accédées via des expressions EL.

Un autre composant développé permet la gestion de tâches planifiées basées sur le framework Quartz. Il offre la possibilité de télécharger le journal d'exécution des tâches de la partie « ETL », de vérifier et de modifier le statut d'exécution de la tâche (actif / inactif). Tout comme pour le composant tables et etl, le composant quartz comporte une référence à une instance d'une « job » Quartz spécifiée à l'intérieur de la configuration Spring de l'application.

Il est possible d'ajouter autant de nouveaux composants que souhaité. L'ajout, au niveau du modèle, demande uniquement d'implémenter l'interface « Component ». Des mécanismes d'ajouts non intrusifs ont aussi été prévus au niveau des interpréteurs et seront discutés dans les prochaines sections.

### 8.2.5 Exemples

Pour la définition de l'application, la structure adoptée est la suivante : le fichier d'entrée Spring se trouve à la racine du répertoire « classes » de l'application Java, sous context/applicationContext.xml. Les configurations propres au framework Spring, comme l'accès à des fichiers de configurations (.properties) externes, sont effectuées dans ce fichier. Aussi, pour simplifier l'édition de certaines propriétés contenant des correspondances clé-valeur, un nouvel éditeur a été ajouté, permettant ainsi une syntaxe plus compacte et tout aussi lisible ("key1=value1,key2=value2" au lieu de "<map><entry key="key1" value="value1" /><entry key="key2" value="value2" /></map>").

```
<bean id="customEditorConfigurer"
     class="org.springframework.beans.factory.config.CustomEditorConfigurer"
     >
  <property name="customEditors">
    <map>
      <entry key="ca.crim.etltools.table.GenericRow">
        <bean class="ca.crim.etltools.utils.GenericRowEditor" />
      </entry>
      <entry key="java.util.Map">
        <bean class="ca.crim.etltools.utils.SerialMapEditor" />
      </entry>
    </map>
  </property>
</bean>
```

Le fichier d'entrée pour Spring (applicationContext.xml) effectue aussi l'importation des composants de l'application, soit les fichiers quartz.xml, connections.xml, tables.xml, pages.xml et etl.xml. Si la solution ETLTools devait être réutilisée comme base pour des applications différentes, la configuration Spring de base resterait la même et seuls ces fichiers devraient être modifiés.

Le fichier de configuration des tables contient, dans un premier temps, des configurations abstraites comportant la configuration de base, destinés à être étendues par la configuration des tables réelles.

```
<bean id="DynamicTableBase" abstract="true"
     class="ca.crim.etltools.table.DynamicTable">
  <property name="dynamicSqlMapFactory">
    <bean class="ca.crim.etltools.abator.DynamicSqlMapFactory" />
  </property>
  <property name="connnectionManager" ref="connectionManager" />
</bean>

<bean id="CustomTableBase" abstract="true"
     class="ca.crim.etltools.table.CustomTable">
  <property name="sqlMapClient">
    <bean class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
      <property name="configLocation"
     value="classpath:/context/application/sqlmap/SqlMapConfig.xml" />
    </bean>
  </property>
  <property name="connnectionManager" ref="connectionManager" />
</bean>
```

Une définition de table ressemble à ceci :

```
<bean id="table.tbgest.ecole" parent="DynamicTableBase">
    <property name="connectionKey" value="tbgest" />
    <property name="tableName" value="ecole" />
    <property name="generatedKey">
      <bean class="ca.crim.etltools.table.GeneratedKey">
        <constructor-arg value="id_ecole" />
        <constructor-arg value="MySQL" />
        <constructor-arg value="true" />
      </bean>
    </property>
 </bean>
```

La lecture des colonnes s'effectue au moment de l'exécution en fonction du nom de la table et de la source de données utilisée. Les tables de type « Custom » sont rattachées à un fichier de configuration iBatis distinct. Le fichier « SqlMapConfig.xml » centralise la

configuration iBatis pouvant être utilisée pour chacune des tables. Chaque table utilisera un espace de nom distinct qui devra être spécifié au niveau de la définition de la table dans Spring, de la manière suivante :

```
<bean id="table.jade.matiere_secondaire" parent="CustomTableBase">
  <property name="connectionKey" value="jade" />
  <property name="namespace" value="matiere_secondaire" />
  <property name="readOnly" value="true" />
</bean>
```

La définition de l'application client a comme point d'entrée le fichier pages.xml. Le « bean » dont l'identifiant est « page.root » est chargé comme racine de l'application, donnant l'accès aux autres pages. Voici un exemple pour la définition de pages :

```
<bean id="page.root" class="ca.crim.etltools.editor.model.PageModel">
  <property name="links">
    <list>
      <ref bean="page.connection_source" />
      <ref bean="page.ecoles" />
      <ref bean="page.annee_ecoles" />
      <ref bean="page.dimension_eleve" />
      <ref bean="page.dimension_eleve_annee_ecole" />
      <ref bean="page.dimension_eleve_reussite" />
      <ref bean="page.dimension_eleve_absence" />
      <ref bean="page.importation" />
    </list>
  </property>
</bean>
```

Il s'agit de la page d'accueil dont la principale vocation est de donner accès à d'autres pages, définies dans d'autres « beans » du fichier XML et liés par l'élément "<ref...". Les étiquettes et autres éléments d'affichages sont spécifiés au niveau de la définition des pages référencées. Tout le comportement relatif à la disposition et à la navigation est délégué à l'interpréteur. Le résultat affiché, avec l'interpréteur Web sera le suivant :


**Figure 8.3** Liste de pages, ETLTools

```
<bean id="page.importation"
     class="ca.crim.etltools.editor.model.PageModel">
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.QuartzJobComponent">
        <property name="title" value="ETL MAIN JOB" />
        <property name="trigger" ref="etlJobTrigger" />
        <property name="jobDetail" ref="etlJobTrigger.jobDetail" />
        <property name="etlJob" ref="job.etl" />
      </bean>
    </list>
  </property>
</bean>
```

Ce deuxième exemple montre une page dont l'objectif est d'activer une tâche Quartz. Le contenu de la tâche et les paramètres d'exécution avancés sont définis dans des sections différentes de la configuration Spring.

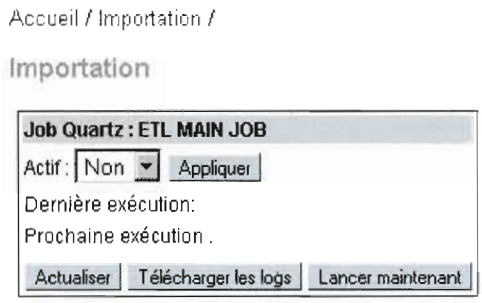


**Figure 8.4** Composant "Quartz", ETLTools

```
<bean id="page.ecoles" class="ca.crim.etltools.editor.model.PageModel">
  <property name="links">
    <list>
      <ref bean="page.clientele_ecole" />
      <ref bean="page.horaire_ecole" />
      <ref bean="page.vocation_speciale" />
      <ref bean="page.aire_desserte" />
      <ref bean="page.secteur_ecole" />
      <ref bean="page.ordre_enseignement" />
      <ref bean="page.cycle_enseignement" />
      <ref bean="page.degre_enseignement" />
      <ref bean="page.mapping" />
    </list>
  </property>
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.EtlTaskComponent">
        <property name="task" ref="task.import.fait.ecoles" />
      </bean>
      <bean class="ca.crim.etltools.editor.model.TableComponent">
        <property name="table" ref="table.tbgest.ecole" />
        <property name="editMode" value="embedded" />
        <property name="valueLists">
          <map>
            <entry key="id_secteur_ecole">
              <bean class="ca.crim.etltools.editor.model.ValueList">
                <property name="table" ref="table.tbgest.secteur_ecole" />
                <property name="key" value="id_secteur_ecole" />
                <property name="labels" value="code,etiquette" />
              </bean>
            </entry>
            <entry key="id_clientele_ecole">
              <bean class="ca.crim.etltools.editor.model.ValueList">
                <property name="table" ref="table.tbgest.clientele_ecole"
 />
                <property name="key" value="id_clientele_ecole" />
```

```
            <property name="labels" value="code,etiquette" />
          </bean>
        </entry>
        <entry key="id_connection">
          <bean class="ca.crim.etltools.editor.model.ValueList">
            <property name="table" ref="table.tbgest.connection_source"
/>
            <property name="key" value="id_connection" />
            <property name="labels" value="conn_name" />
          </bean>
        </entry>
      </map>
    </property>
    <property name="links">
      <list>
        <ref bean="page.ecole_annee" />
      </list>
    </property>
  </bean>
 </list>
</property>
</bean>
```

Cet autre exemple, plus complet, montre une page comportant des liens vers d'autres pages, un composant permettant l'exécution d'une tâche et une table complexe permettant l'édition de données directement dans la table (editmode=embedded). On y configure les listes de valeurs possibles, rattachées à une étiquette, pour certains champs qui seront alors transposés en menu déroulant du côté de la page Web. Le lien vers des pages rattachées à des enregistrements précis dans la table est aussi utilisé via la propriété « links » du « TableComponent ». Le résultat est le suivant :

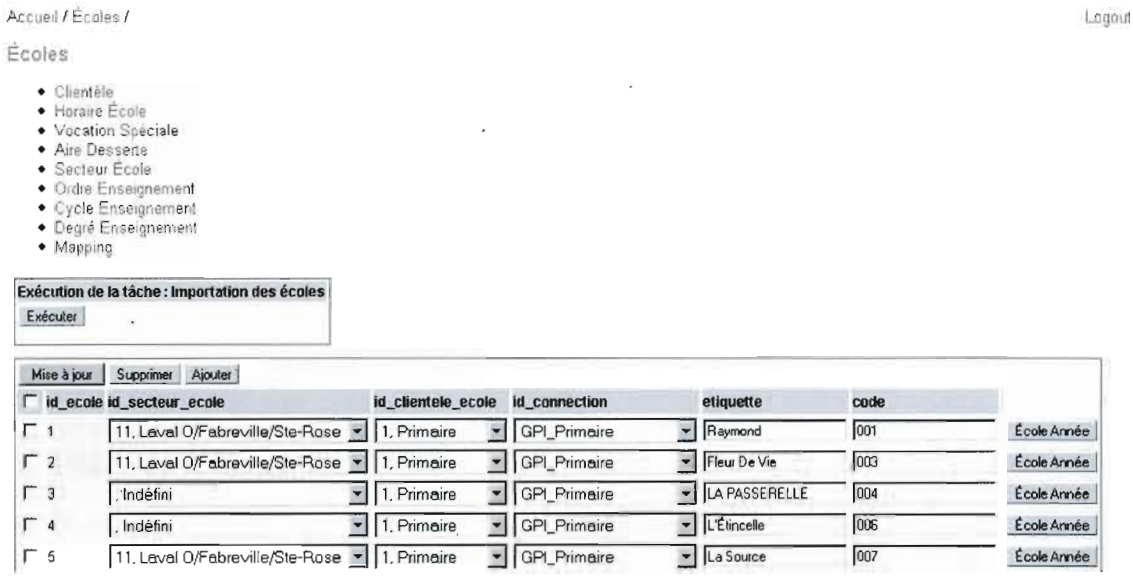


**Figure 8.5** Page ETLTools, mode édition

D'autres modes d'affichages sont aussi possibles pour l'édition du contenu des enregistrements. Voici un formulaire d'édition régulier :

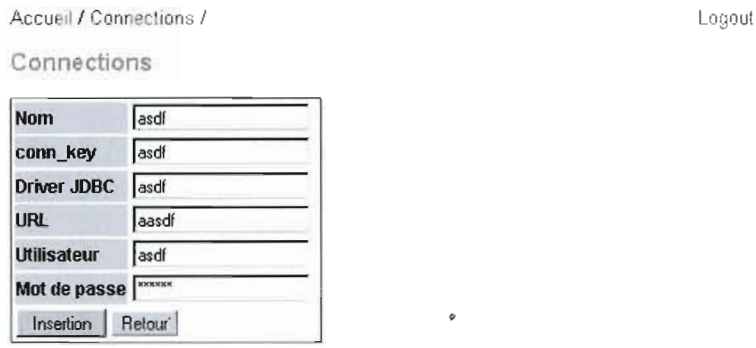


**Figure 8.6** Formulaire d'édition, ETLTools

Une nomenclature rigoureuse est utilisée pour l'identification des composants. Puisqu'un très grand nombre de « beans » Spring sont utilisés, l'absence de règles aurait pu entrainer des collisions ou rendre incompréhensible la définition de l'application. Le principe retenu consiste à faire commencer le nom du « bean » par un identifiant correspondant à son

type (« table », « page », ...). Pour les tables, le nom de la base de données suit, pour se terminer avec le nom de la table. Ex : « table.tbgest.ecole », constituant un identifiant unique pour l'application. Pour une page, la catégorie à laquelle elle appartient précède le nom de la table éditée. Ex : « page.dimension_eleve.lieu_naissance ».

Les identifiants des « beans » Spring sont ensuite utilisés pour l'internationalisation de certaines propriétés texte. Les fichiers de ressources de langue se basent alors sur les identifiants uniques pour spécifier les clés à utiliser. Ex :

```
page.root.title=Accueil
page.connection_source.title=Connections
page.connection_source.0.labels.id_connection=Id
page.connection_source.0.labels.conn_name=Nom
page.connection_source.0.labels.conn_driver=Driver JDBC
page.connection_source.0.labels.conn_url=URL
page.connection_source.0.labels.conn_username=Utilisateur
page.connection_source.0.labels.conn_password=Mot de passe
```

### 8.2.6 L'exécution

L'interpréteur Web développé pour ce projet se base sur le framework Spring WebMVC. Sa configuration fait l'objet d'un point d'entrée distinct pour sa configuration. Le fichier dispatcher-servlet.xml, spécifié dans le web.xml contient la liste des contrôleurs. Le contrôleur « PageController » génère l'ensemble des pages de l'application. Il est accessible via l'url « /editor/pages.htm ».

```
<bean name="/editor/page.html"
     class="ca.crim.etltools.editor.web.controller.PageController">
  <property name="connectionManager" ref="connectionManager" />
  <property name="componentManager" ref="componentManager" />
  <property name="methodNameResolver">
    <bean
     class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodN
     ameResolver">
      <property name="paramName" value="_pageAction" />
      <property name="defaultMethodName" value="doMain" />
    </bean>
  </property>
</bean>
```

Tel qu'indiqué dans cette configuration, la méthode utilisée pour l'exécution d'une action côté contrôleur est passée en paramètre et sera récupérée par réflexion. Il y a donc un seul contrôleur pour la gestion des pages, ce qui permet de partager des ressources communes pour l'exécution des différentes actions. Celui-ci est responsable de la gestion de la navigation et maintient le contexte associé aux pages (ex : lorsqu'une page est accédée depuis un enregistrement d'une table, le contexte contiendra les colonnes de cette table qui seront accessibles par les composants de la page suivante). Il gère aussi la navigation, en affichant les liens disponibles ainsi que le chemin d'accès de la page en entête. Il est donc responsable de la gestion de ce qui est défini dans un « PageModel ».

Les composants (Table, Tâche, Configuration Quartz) sont aussi gérés avec des contrôleurs qui leur sont propres. Ceux-ci se chargent de l'interprétation de la classe du modèle associé. Ces derniers reçoivent et traitent les requêtes HTTP qui sont reliées au composant et mettent à jour le modèle. La requête est ensuite redirigée au contrôleur de page, « page.html », pour le rendu de l'application. Une classe « componentManager » permet de référencer les implémentations à utiliser pour le support de chacun des composants :

```
<bean id="component.etltask" name="/editor/etltask.html"
  class="ca.crim.etltools.editor.web.controller.EtlTaskController">
  <property name="connectionManager" ref="connectionManager" />
  <property name="methodNameResolver">
    <bean
     class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodN
     ameResolver">
      <property name="paramName" value="_action" />
      <property name="defaultMethodName" value="doMain" />
    </bean>
  </property>
</bean>

<bean id="componentManager"
     class="ca.crim.etltools.editor.web.controller.ComponentManager">
  <property name="componentControllerList">
    <list>
      <ref bean="component.table" />
      <ref bean="component.etltask" />
      <ref bean="component.quartzJob" />
    </list>
  </property>
</bean>
```

Les composants sont aussi rattachés à des pages JSP qui en assurent le rendu. Ils sont consignés dans un emplacement précis, soit WEB-INF/jsp/editor/component, et sont ensuite incorporés par le JSP principal de l'application à l'aide d'un « tag » jsp:include. Certains éléments HTML se répétant à l'intérieur de plusieurs composants ont été implémentés sous forme de « tag » jsp personnalisé, favorisant ainsi la réutilisation de code.

En somme, cette mécanique fait en sorte que l'ajout de composants à l'application ne nécessite pas de modification au niveau des structures existantes. Il n'y a donc pas d'interdépendance entre les composants de l'application et l'application générique. Un ajout devra donc se faire en deux temps, premièrement, une classe doit être ajoutée au modèle, implémentant l'interface d'un composant. Elle pourra, à ce moment, être utilisée au niveau des fichiers de définition Spring. Pour que le rendu soit effectué, le comportement du composant, spécifique aux plateformes, devra donc être ajouté à chacunes d'elles.

La lecture des composants du modèle s'effectue à l'intérieur des contrôleurs et avec l'aide de classes utilitaires. Le chargement des pages et la gestion de la navigation est délégué au « PageController ». Lorsque vient le temps de lire les composants (liste « components » de « PageModel »), le « PageController » interroge le « componentManager » décrit plus haut. Ce dernier est responsable de trouver le contrôleur approprié, capable d'effectuer le rendu du composant. Les données accédées par les pages JSP, sous forme d'attributs à la requête, sont des objets spécifiques orientés sur les données de présentations. Ils contiennent, entre autres, l'ensemble des informations qui sont connues au moment de l'exécution, comme la liste des colonnes d'une table dans le cas du « TableComponent » ou les listes de valeurs provenant des bases de données. Ces objets implémentent le patron « View Bean » mis de l'avant dans certaines architectures pour donner accès aux données de l'application (Wahli *et al.*). Chaque composant a donc une classe qui lui est propre (« TableView », « EtlTaskView », ..) instanciée par le contrôleur associé et rendu accessible au code JSP.

Certaines difficultés ont été éprouvées lors du développement de l'interpréteur. Par exemple, au niveau de la gestion des tables, la liste des colonnes n'est connue qu'au moment de l'exécution et avec des tables de type dynamique, qu'au moment de l'exécution de la première requête. Un mécanisme d'instanciation paresseuse, « Lazy-Loading » a donc du être

mis en place pour que toutes les informations nécessaires à l'affichage des tables puissent être calculées au dernier moment, lorsque la page doit être affichée.

### 8.2.7 Conclusion

Ce cas constitue une première démonstration de la faisabilité de l'approche présentée au chapitre 5. La portée de l'application se limite par contre actuellement aux applications simples d'édition de données. Elle offre des écrans génériques sans permettre la mise en forme avancée. L'architecture de l'application permet toutefois l'extension de l'application et le support de nouvelles fonctionnalités sans avoir à modifier le code existant. L'ajout de nouveaux écrans s'effectue très rapidement avec peu de lignes de code XML, ce qui constitue le principal avantage de l'approche.

Il avait été prévu d'offrir un deuxième interpréteur supportant une technologie de client Web riche, comme GWT ou XUL mais, par manque de temps, cette deuxième phase n'a pu être réalisée. Le modèle, les tâches et les tables contenus dans les fichiers XML auraient alors pu être réutilisés intégralement puisqu'aucune dépendance à l'implémentation HTML ne se trouve du côté de la définition de l'application. La logique utilisée pour l'affichage et l'édition des données, ainsi que les écrans génériques utilisés pour les composants d'exécution de tâches auraient alors dû être réécrits pour la nouvelle plateforme.

## 8.3 Outil d'importation de données pour le Portail du projet Mille

### 8.3.1 Description du projet

Le projet Mille, pour Modèle d'Infrastructure de Logiciel Libre en Éducation, a débuté en 2002 avec l'objectif de réaliser une preuve de concept pour l'exploitation, à l'intérieur d'une commission scolaire, d'une infrastructure informatique entièrement basée sur du logiciel libre. Les considérations premières étaient économiques : faire face à une décroissance démographique (diminution des budgets), à la désuétude des parcs informatiques et une croissance des coûts de licence des applications propriétaires. L'objectif du projet était donc de démontrer que l'utilisation de logiciel libre était plus économique que le logiciel propriétaire. Le projet a été divisé en 5 sous-projets : l'infrastructure, pour les services réseaux de bases comme les serveurs de fichiers, le courriel, les bases de données,

l'annuaire LDAP, les serveurs DHCP ; les applications libres pour Windows ou Macintosh distribués à l'intérieur d'une distribution nommée Colibris (COntenu LIBRe pour Institutions Scolaires) ; le client léger mettant de l'avant une solution de mise à l'échelle pour le déploiement de terminaux X basés sous Linux ; le portail, nommé bureau virtuel, proposant un point d'accès unique aux applications en lignes ainsi que des outils de communications et, finalement, une étude économique sur l'impact de ces solutions a été réalisée (Mille.ca, 2007).

Le projet de Bureau Virtuel, basé sur uPortal, est livré avec plusieurs sous-projets tel qu'un agenda, des signets personnels, l'affichage des relevés de notes, une interface pour la consultation du courriel et un gestionnaire de communautés offrant des outils de partage entre les utilisateurs. Les services d'agenda et d'affichage des résultats scolaires nécessitaient l'accès aux données scolaires, plus précisément au calendrier scolaire (jours d'école de 1 à 9), aux événements (examens, activités), à l'horaire des cours, à la liste des élèves pour chacune des écoles, les enseignants, la composition des groupes et les heures des cours. La décision prise quant à l'accès aux données fut la suivante : maintenir une base de données locale, normalisée et ayant des tables de valeurs standardisées. Les sources de données étant hétérogènes, multiples et pouvant se trouver éloignées sur le réseau (ex : dans une école ayant un lien peu performant), cette solution permet de simplifier l'accès aux données par les outils du portail et en améliore aussi les performances. L'outil CSAdmin a, à l'époque, été développé (Rancourt, 2005).

L'application originale, développée en 2003 était constituée de classes Java spécifiques à chaque tâche d'importation possible. Les opérations les plus utilisées étaient la synchronisation, via un mécanisme similaire à ce qui fait partie de EtlTools. Une application graphique était aussi nécessaire pour la gestion des connexions aux écoles, des règles d'importations à exécuter ainsi que des tables de correspondances entre les données présentes dans les bases de données des écoles et le schéma normalisé distribué avec le Portail Mille. Cette application graphique était aussi développée en Java et comportait une classe pour générer le contenu de chaque écran de l'éditeur. Des classes utilitaires permettaient de simplifier le traitement et de générer le contenu pour l'affichage dont le rendu était assuré par un XSL.

Après la première phase du projet Mille, un seul type de base de données était supporté, il s'agit des banques GPI de la société GRICS. Suite à des demandes de la part de commissions scolaires, le support pour de nouveaux types de données devait être ajouté. Au lieu d'ajouter de nouveaux écrans et de nouvelles classes d'importation dans l'application originale, il a été décidé de migrer la totalité de l'application CSAdmin vers la plateforme ETLTools.

L'opération de migration s'est déroulée sur une période d'environ une semaine. L'ensemble des fonctionnalités présentes dans CSAdmin comportent des équivalents au niveau de la plateforme ETLTools. L'application Java a donc pu être remplacée par un ensemble de fichiers de configuration XML, suivant la même nomenclature et les mêmes standards que pour le projet de tableau de bord. Le modèle et l'interpréteur Web sont restés les mêmes que pour l'application précédente.

Ce cas aura permis de montrer la réutilisation possible d'une application générique dont le contenu est défini par un modèle. Cela a pour effet d'augmenter la réutilisation de code et, du même coup, de mutualiser les efforts de maintenance et de test du code Java utilisé. Il est toutefois important de noter que les deux applications ainsi que les besoins qu'elles couvraient étaient, à la base, très similaires, ce qui a permis d'effectuer la migration sans avoir à ajouter de nouveaux composants.

## 8.4 Générateur d'horaire scolaire

### 8.4.1 Description du projet

La Commission scolaire de Laval en partenariat avec le CRIM a investi dans un projet de recherche visant l'automatisation du processus de création des horaires des enseignants et des élèves des écoles secondaires québécoises. Le problème de la confection des horaires scolaires a déjà été abordé par plusieurs produits commerciaux, mais aucun produit commercial existant n'était en mesure de répondre aux besoins des écoles à cause de plusieurs façons de faire distinguant les écoles publiques du Québec. Notons, entre autres, l'utilisation d'une grille de 9 jours et l'utilisation d'arrangements devant être respectés pour la disposition de l'horaire d'un groupe sur la grille. Un arrangement définit un « pattern » à

suivre, par exemple, si un cours est donné à la position Jour1/Période1, la deuxième position du même groupe devra être Jour3/Période2 et ainsi de suite. Cette façon de faire permet de répartir les cours de façon uniforme sur la grille, faisant en sorte que chacun des cours soit donné à chacune des périodes de la grille.

Une deuxième spécificité importante, n'étant pas abordée dans les produits commerciaux existants, est le fait que, pour les polyvalentes de deuxième cycle (sec. 3 à 5), l'assignation des élèves aux groupes est effectuée au même moment ou après la formation de l'horaire. Cela est dû au fait que le nombre de combinaison de choix de cours possible est extrêmement élevé. Par exemple, pour une école de 2500 étudiants de deuxième cycle au secondaire, nous pouvons retrouver plus de 800 combinaisons différentes de cours possibles qui auront été choisis par les élèves. La méthode classique utilisée pour la génération d'emploi du temps aborde le problème à l'inverse, c'est-à-dire que, selon les cours choisis par les élèves, des contraintes sont placées entre les groupes et l'objectif de l'algorithme est de minimiser le nombre de contraintes violées. Dans une école secondaire de grande taille, on retrouve aussi un très grand nombre de groupes possibles pour chacun des cours, ce qui fait exploser le nombre de combinaisons d'horaire possible pour chaque élève. L'approche voulant former les groupes préalablement à la génération de l'horaire n'était donc pas exploitable dans le contexte des écoles visées par le projet.

La solution devait aussi gérer une panoplie de contraintes couvrant, entre autres, le respect de la vocation des locaux, les tâches des enseignants (définies au préalable), l'optimisation de l'horaire des enseignants, le fait de minimiser le nombre de locaux différents pour un enseignant, l'option de prioriser certains types d'élèves, etc. Bref, le projet de recherche s'est attaqué à cette problématique et a permis de mettre au point un solutionneur capable de résoudre le problème de la génération automatisé et répondant aux besoins spécifiques des écoles secondaires québécoises.

Un modèle de classes Java a été créé pour accueillir les données du problème. Le modèle est initialisé depuis une couche de persistance chargée de l'extraction depuis les bases de données utilisées pour la gestion pédagogique. Ce modèle contient un certain nombre de règles d'affaires et supporte quelques fonctionnalités comme la gestion d'un état

« supprimé » au niveau des instances et l'exécution de requêtes pour la récupération d'objets ou d'informations selon certains critères. Des règles d'intégrité sont aussi validées lors de la manipulation des objets. Ce modèle est utilisé à la fois comme source de données pour les algorithmes et pour l'interface utilisateur.

Un autre volet de ce projet couvrait le développement d'une application permettant à l'utilisateur devant créer l'horaire d'effectuer la saisie des contraintes ainsi que la configuration des paramètres liés au solutionneur. D'autres fonctionnalités, comme des assistants pour l'importation de données devaient aussi être développées. Puisque le solutionneur est développé en Java et afin de permettre l'exécution de l'application à la fois sous les systèmes d'exploitation Windows et Linux, il a été décidé de développer l'application à l'aide de Swing. Il n'était toutefois pas exclu d'avoir un jour à porter l'application sur une autre plateforme comme Eclipse RCP (Rich Client Platform) ou de l'exploiter à l'aide d'un client Web riche.

Une première version du client graphique a été développée en Swing à l'aide de l'approche traditionnelle. L'outil « Visual Editor », qui s'installe à l'intérieur de l'environnement de développement Eclipse a été utilisé pour le design des écrans. L'approche de navigation utilisée est celle des onglets et des séparateurs d'écrans, formant une application de type TDI (Tabbed document interface). Les onglets peuvent être divisés en plusieurs sections à l'aide de séparateurs d'écran. Chaque section d'écran est développée sous forme d'un panneau ou « pannel ». Pour cette première version basée sur Swing, l'ensemble de la logique de présentation est codé dans les interfaces. Chaque panneau est défini à l'intérieur de sa propre classe Java contenant les méthodes d'accès aux données du modèle, de formatage des données, de création des panneaux et de mise à jour des écrans en cas de modification. Le formatage et les dimensions faisaient aussi partie des mêmes écrans.

Beaucoup de code était nécessaire pour l'initialisation du contenu des écrans et, surtout, pour l'ajout des données dans les grilles de données. Sans parler de duplication de code, nous pouvons affirmer que nous retrouvions plusieurs passages exhaustifs similaires, mais s'appliquant sur des données quelque peu différentes. Pour favoriser la réutilisation de code, cette première version de l'application comportait, dans une superclasse utilisée pour

les grilles de données, plusieurs méthodes pour la création de contrôles présents à plusieurs endroits, comme par exemple, des listes déroulantes pour un jeu de valeurs communes. Une des principales difficultés se trouvait au niveau du rafraichissement des écrans de l'application. Il était en effet nécessaire, lors de chaque action impliquant des modifications aux données, de connaître les autres panneaux de l'application pouvant être influencés pour y lancer une commande de rafraichissement.

Cette première version du client graphique, réalisée en 12 jours, comportait une vingtaine d'écrans fonctionnels pour 10 000 lignes de code Java. Le développement a dû être arrêté à ce point suite à une réorganisation du personnel affecté au projet. Moins de la moitié du développement de l'application avait alors été effectué. Il restait une vingtaine de nouveaux panneaux à ajouter pour le support des fonctionnalités restantes. Le développement aurait pu continuer en utilisant la même technique, mais nous présumons que le nombre de lignes de code aurait presque doublé et le temps de développement requis pour certains aspects, comme l'extraction, le formatage et la mise à jour des données serait resté élevé.

Pour faire suite à ce projet de mémoire, il a été décidé, à ce moment du projet, d'appliquer l'approche de développement basée sur un modèle pour définir l'ensemble des écrans de l'application.

ETLTools aura servi de premier prototype pour l'expérimentation de l'approche. Certains éléments se seront avérés plus difficiles à gérer que prévu, comme le mécanisme de détection des colonnes au moment de l'exécution. De plus, aucun mécanisme n'avait été pensé pour effectuer la communication entre les composants. Ceux-ci s'exécutant de façon complètement autonome en exploitant leurs propres modèles et leurs propres vues. L'utilisation de type « Map » comme support pour les données se prêtait bien au contexte de traitement sur des données connues au moment de l'exécution. Cela n'est toutefois pas recommandable à l'intérieur d'une application contenant de la logique d'affaire dépendante du modèle.

Contrairement à ETLTools ou la totalité des traitements pouvaient être exprimés dans le modèle XML et géré entièrement par les contrôleurs génériques, l'interface utilisateur

devait ici communiquer avec une application existante avec ses objets d'affaires et sa logique. Tous les traitements se trouvaient donc au niveau des classes Java. Cette nouvelle génération de client devait donc offrir les mécanismes nécessaires pour s'intégrer à l'application.

Il a donc été décidé de développer une deuxième génération de solution mettant en pratique l'approche de développement basée sur un modèle indépendant de plateforme. Pour ce projet, le modèle et les interpréteurs ont été développés sous forme de bibliothèques externes à l'application de génération d'horaires.

### 8.4.2 Technologies utilisées

Les technologies utilisées sont similaires à celles d'ETLTools. Notons l'utilisation des bibliothèques « Common » d'Apache pour la journalisation et les expressions EL. Spring sert aussi de base pour la définition et l'instanciation de l'application. Cette technologie est aussi utilisée pour la configuration des interpréteurs disponibles. L'interpréteur développé supporte la plateforme de présentation Swing. L'exécution se fait actuellement comme une application autonome, mais elle pourrait faire l'objet d'une distribution via Java Web Start.

L'application de génération des horaires, qui est un projet distinct de celui présenté ici pour la définition de l'aspect présentation, utilise d'autres bibliothèques à l'interne, comme iBatis pour l'importation et l'exportation des données du modèle depuis les bases de données des écoles et XStream combiné aux utilitaires de compression du framework Java (java.util.zip) pour l'enregistrement des données de l'application. En plus d'un modèle objet propre au domaine, l'application comporte aussi plusieurs contrôleurs génériques, indépendants de toute plateforme de présentation. Ces contrôleurs sont en fait des classes java conventionnelles, ne nécessitant pas l'implémentation d'interfaces. Elles sont instanciées par Spring rendu disponible au niveau du contexte de l'application. La définition de l'application peut ensuite y faire référence en y spécifiant la méthode à exécuter et les paramètres associés.

Un nouveau modèle d'application, beaucoup plus riche que le précédent, a été conçu. Cette nouvelle génération se base davantage sur les mécanismes de réflexion pour l'accès aux

données et pour l'exécution de méthodes présentes dans le modèle et les contrôleurs. Ce modèle permet aussi une grande utilisation des mécanismes de liaisons, ou « binding », entre les composants ou entre les composants ou le modèle.

Un gestionnaire de liaison, ou « binding », entre l'interface utilisateur et l'application, a été développé. La mécanique implémentée permet d'adresser les composants du client riche, les contrôleurs et des variables de sessions à l'aide du langage d'expressions (EL) de JSTL. L'interpréteur a la responsabilité d'enregistrer chacune de ses liaisons (ex : texte d'un bouton, valeur d'un champ texte, contenu d'une liste) auprès du gestionnaire. Ils deviennent donc observateurs quant à un état donné. Les composants sont par la suite responsables de notifier ce gestionnaire lorsque tout changement est effectué. Les observateurs sont alors avisés que le modèle ou l'état de l'application a changé, ce qui provoquera leur rafraichissement.

Le contexte d'exécution, qui comporte les composants contrôleurs et variables, agit comme la session de l'application. Elle est gérée par le gestionnaire de liaison et peut être interrogée uniquement à l'aide d'expressions EL (Expression Language). Cette technologie utilise la réflexion pour l'accès aux attributs des objets. Par exemple, une instance « obj » comportant un attribut « att » et ayant une méthode d'accès « getAtt() » sera accessible via EL avec la syntaxe « obj.att ». L'accès en lecture aux attributs est donc supporté nativement. Pour la liaison de propriétés éditables par l'interface utilisateur, comme le champ « text » d'un « TextBox », le gestionnaire de liaison est en mesure de détecter, par réflexion, les méthodes « set » associées aux mêmes attributs. La liaison à « obj.att » effectuée sur un champ « text » d'un « TextBox » entrainera donc l'appel à « setAtt » de la part du gestionnaire de liaison lorsque l'utilisateur modifiera le contenu du champ texte.

Le diagramme de séquence suivant montre la méthode utilisée pour la gestion des liaisons entre les composants gérés par les interpréteurs et le gestionnaire de liaison inclus dans la classe « AppContext ». Lors de l'initialisation de l'application, l'interpréteur appelle « registerBinding » de « AppContext » pour chaque propriété du composant contenant une expression EL. Ces informations sont enregistrées dans une liste, nommée « BindingList ». Lorsqu'un événement susceptible d'influencer le modèle se produit, l'interpréteur doit

appeler la méthode « fireChange » de « AppContext » en spécifiant l'élément du contexte modifié. Le gestionnaire de liaison effectuera alors une recherche de toutes les liaisons susceptibles d'être influencées par ce changement et leur enverra une notification. Lorsqu'ils reçoivent une telle notification, les composants rechargent leurs propriétés en demandant à « AppContext » de réévaluer leurs expressions EL. Dans le cas d'une modification, du modèle par l'application, la méthode « updateBindingValue » effectue la modification au niveau du contexte et enclenche la même mécanique de notification.

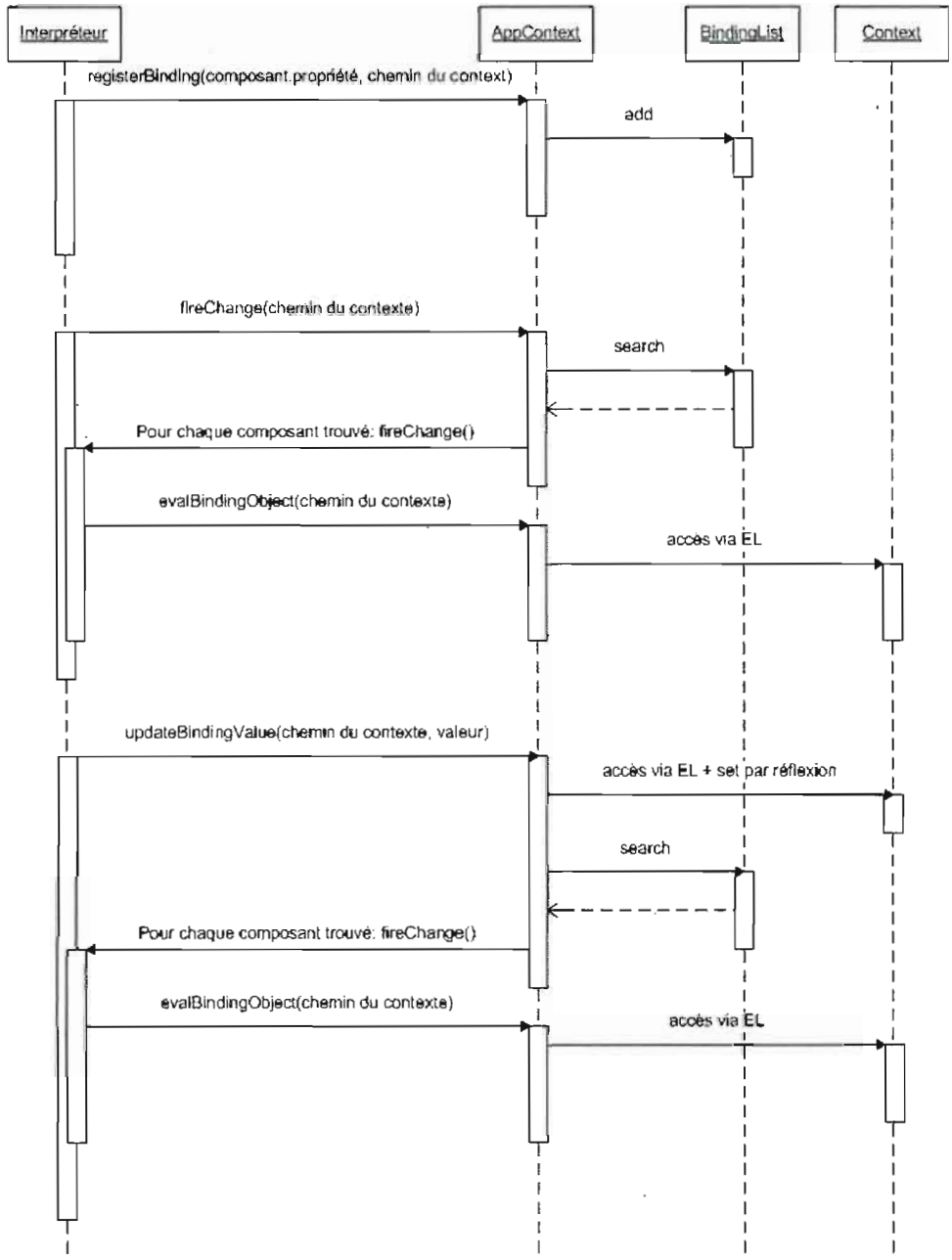


**Figure 8.7** « Binding », prototype 2

### 8.4.3 Le modèle

Ce modèle d'application, beaucoup plus riche que le précédent, sera découpé pour être présenté selon les différents aspects qu'il couvre.

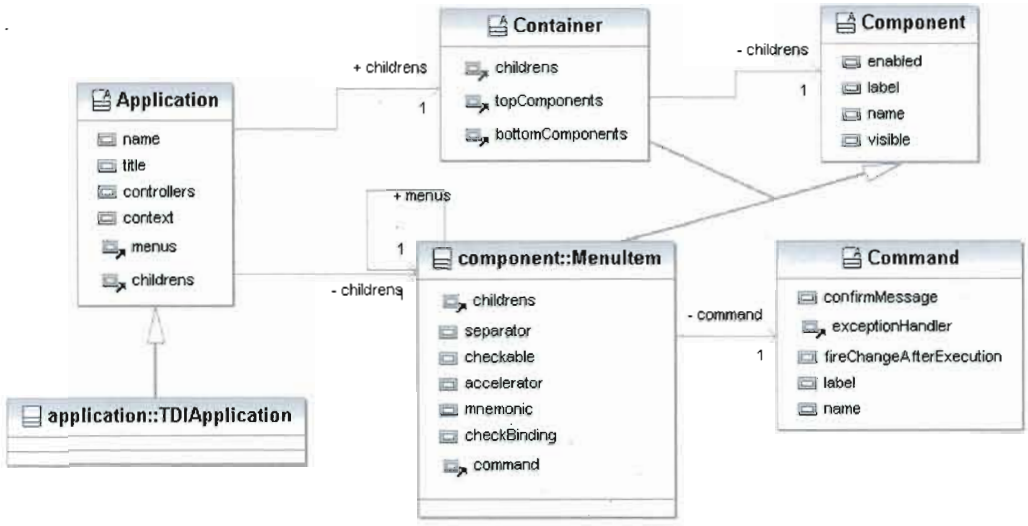


**Figure 8.8** Modèle de haut niveau, prototype 2

Le point d'entrée est une instance de type « Application », défini sous forme de classe abstraite. À l'heure actuelle, seul le sous-type « TDIApplication » est disponible, offrant une interface utilisateur centrée sur les onglets. D'autres modes possibles auraient pu être MDI (pour Multiple Document Interface, composé d'un espace de travail à l'intérieur duquel on retrouve des pages enfants) ou SDI (Single Document Interface ou l'ensemble de l'application tient dans un seul écran). Une application est rattachée à plusieurs conteneurs, spécifiés dans l'attribut « childrens ». Avec le mode TDI, chacun des conteneurs enfants représentera un onglet de l'application. La gestion des onglets et les mécanismes d'accès à ces derniers sont laissés au choix de l'interpréteur. La classe « Application » du modèle comporte aussi une liste de « MenuItem ». Un menu peut avoir des sous-menus, spécifiés dans « childrens », s'afficher sous forme de séparateur ou être accompagné d'une case à cocher. La classe « MenuItem », tout comme « Container » et l'ensemble des composants graphiques de l'application, hérite de la classe « Component ». Cette dernière définit une série d'attributs communs, comme le nom du composant, l'étiquette associée, ou l'état

d'activation du composant. Tous ces attributs supportent les expressions EL, permettant ainsi de lier un état, par exemple, « visible », à une expression liée au contexte de l'application. Donc, avec ces éléments, nous avons la possibilité de rendre la visibilité d'un menu ou sous-menu de conditionnel à un état externe, pouvant être modifiés au moment de l'exécution.

Les objets de type « Container » héritent aussi des attributs de bases, définis dans « Component ». Un conteneur comporte une liste de composants enfants, dont la disposition est effectuée selon le type du conteneur. Les attributs « bottomComponents » et « topComponents » contiennent des listes de composants (ex : boutons) à placer de façon linéaire au dessous ou au dessus de conteneur. Cinq types distincts sont actuellement supportés. Un « FormContainer » assure une disposition classique des composants pour un écran de type formulaire en y plaçant l'étiquette rattachée à gauche et le contrôle à droite. Pour ce composant, une commande peut être appelée avant affichage pour permettre de placer des instances des données à éditer au niveau du contexte. Pour un type « TabContainer », les éléments enfants seront affichés sous forme d'onglets. Un « SplitterContainer » affichera les composants enfants séparés par des séparateurs dont la dimension pourra être ajustée par l'utilisateur. Finalement, les conteneurs HBox et VBox permettent d'afficher séquentiellement des composants de manière horizontale ou verticale.

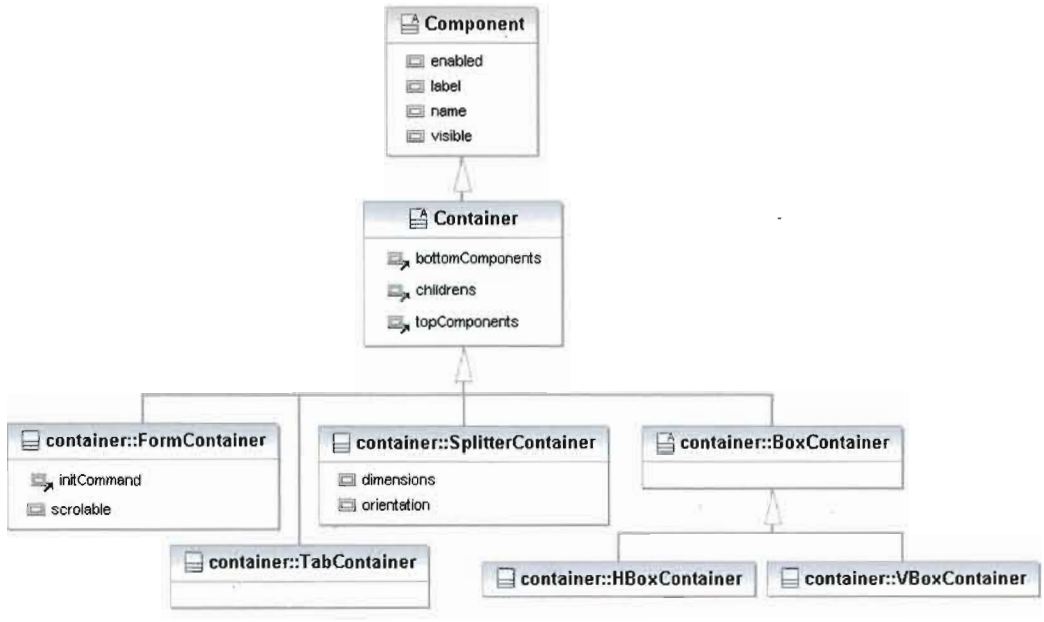


**Figure 8.9** Conteneurs, prototype 2

Le modèle offre actuellement 9 composants distincts. Ils comportent tous des attributs « enabled », « label » et « visible » hérités de la classe abstraite « Component ». Lorsque des composants sont placés à l'intérieur d'un conteneur, il appartient à ce dernier d'appliquer une disposition pour l'étiquette rattachée au composant. TextBox, CheckBox, ComboBox et ListBox permettent l'édition de données présentes dans le contexte via les attributs « binding » ou « text » dans le cas d'un TextBox. Ce dernier contrôle offre d'autres attributs comme « password » précisant qu'il s'agit d'une zone de mot de passe ou « multiLine » rendant possible l'édition sur plusieurs lignes.

ListBox et ComboBox affichent une liste de valeur, spécifiée par « valueListBinding » ou « valueListCallBinding » référant à une collection reliée au contexte ou à une méthode à appeler pour récupérer une collection. Un autre attribut optionnel, « textAttribute » permet de définir l'attribut à utiliser pour l'affichage. Par exemple, si le champ « binding » pointe sur une classe de type « Eleve » et que la liste de valeur contient aussi des élèves, « textAttribute » pourrait référer à l'attribut « nom » pour que ce soit le nom de l'élève qui soit affiché. L'attribut « fireChangeOnAction », présent pour ComboBox et CheckBox,

permet de spécifier manuellement quels éléments du contexte sont affectés lorsque les valeurs de ces composants sont changées.

Le composant DataGrid permet l'affichage d'une grille de données pouvant être éditée. Le contenu de la grille est définit par l'attribut « itemsBinding » contenant une expression EL référant à une collection du contexte. Il est aussi possible de définir l'attribut « itemsCallBinding » ayant la même fonction, mais permettant l'exécution d'une méthode d'un contrôleur pour récupérer une collection de valeurs. Par la suite, chacun des items de la collection peut être adressé par le nom de variable « item » à l'intérieur des autres expressions EL de la table. Des attributs de permettent, entre autres, d'appliquer un comportement sur chacune des lignes de la grille. La couleur ou le fait de barrer, de pouvoir éditer ou d'afficher une ligne peut être spécifié sous forme d'expression EL. La variable « item » est alors disponible à l'intérieur de l'expression, ce qui donne la possibilité de composer des conditions sur le contenu de la ligne (ex : « rowStriked={item.deleted == true} »).

L'attribut « columns » contient plusieurs instances de « DataGridColumn ». On y définit l'étiquette affichée en entête de la colonne et autres attributs comme triable, null autorisé, multivaluée, la couleur ou le format du texte. La variable « item », spécifique à chacune des lignes, est aussi disponible lors de l'évaluation de chacun de ces attributs. Le contenu des cellules de la colonne est aussi lié par une expression EL via l'attribut « binding ». Il est aussi possible de spécifier une liste de valeurs à utiliser avec les attributs « valueListBinding » ou « valueListCallBinding » ; « textAttribute » peut y être utilisé.

Il est aussi possible de définir des commandes pouvant être exécutées sur les données de la grille. Avec Swing, cette fonctionnalité est implémentée à l'aide d'un sous-menu, mais ce choix est laissé à l'interpréteur. Les « DataGridCommand » permettent d'exécuter des opérations sur les lignes sélectionnées comme, par exemple, une suppression. En plus d'une référence à la commande, « DataGridCommand » contient l'attribut « enabled » permettant de rendre accessible ou non la commande en fonction des lignes sélectionnées. À l'intérieur des commandes, des références aux variables « selectedItem », pour le premier item ou

« selectedItems », pour atteindre la collection des lignes sélectionnées, peuvent être utilisées au niveau des expressions EL.

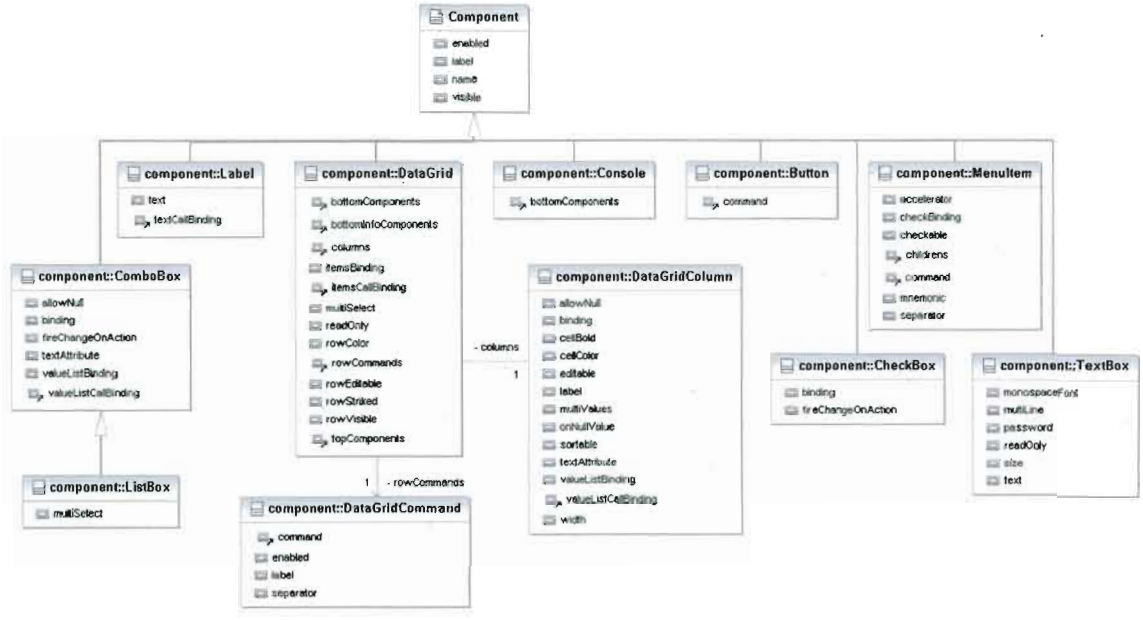


**Figure 8.10** Composants, prototype 2

Le modèle est livré avec une diversité de commandes. Les plus simples, « ExitCommand » et « CloseDialogCommand » permettent respectivement de quitter l'application et de fermer la fenêtre courante. Chaque commande comporte un attribut « confirmMessage » qui, si spécifié, commandera à l'interpréteur de demander une confirmation à l'utilisateur avant exécution. Une « ExceptionHandler » peut spécifier un comportement à appliquer si une exception est lancée par la commande exécutée. Par défaut, l'interpréteur pourra simplement journaliser l'erreur mais, si, par exemple, « AlertExceptionHandler » est spécifié, un message d'erreur sera affiché à l'utilisateur. L'attribut « fireChangeAfterExecution » permet de spécifier les éléments du contexte susceptibles d'avoir été modifiés lors de l'exécution de la commande. Le mécanisme de gestion des liaisons décrit précédemment se chargera ensuite de propager les mises à jour.

La commande de base, la plus utilisée, est « CallCommand ». On y spécifie, dans l'attribut « binding », la référence à l'objet du contexte ou au contrôleur contenant la méthode

à exécuter. L'attribut « method » contient le nom de la méthode à appeler et « args » est un tableau d'arguments. Les arguments peuvent être des expressions EL référant au contexte, aux variables disponibles (ex : « item » lorsque la commande est placée dans un DataGrid ) ou aux contrôles qui sont aussi disponibles au niveau du contexte sous le nom de variable « components ». Par exemple, une commande pourrait passer en argument la valeur texte d'un TextBox nommé « textbox1 » avec l'expression suivante : {components['textbox1'].text}. Une sous-classe nommée « BackgroundCallCommand » fait en sorte d'exécuter la commande en arrière-plan, sans bloquer l'interface utilisateur.

Certaines commandes sont composées de commandes existantes. C'est le cas de « CommandAggregation » et de « IterativeCallCommand » qui font respectivement l'exécution d'une liste de commandes ou l'exécution d'une même commande plusieurs fois, soit un appel pour chaque élément se trouvant dans l'attribut « itemsBinding » contenant une expression EL pointant sur un élément du contexte.

DialogCommand affiche une nouvelle fenêtre contenant un objet de type « container ». Il pourra être fermé lorsqu'une commande « CloseDialogCommand » sera exécutée. L'attribut « startupCommand » permet de référencer une autre commande à appeler pouvant être utilisée pour effectuer un traitement et placer de nouveaux objets dans le contexte. WizzardCommand affiche aussi une nouvelle fenêtre, mais est conçue pour générer un assistant. Les étapes de l'assistant sont alors définies dans une liste de WizardCommandItem. Un certain contrôle peut être effectué par les attributs « condition », « nextEnabled » et « validationCommand ». La condition, si spécifiée, doit être vraie pour que l'écran s'affiche. Cela permet de gérer un flux d'affichage personnalisé selon les sélections de l'utilisateur. « nextEnabled » définit, à l'aide d'une expression EL, si l'utilisateur a l'option ou pas de passer à l'étape suivante. Par exemple, si la condition est qu'au moins une case à cocher doit être sélectionnée, l'expression référera aux composants CheckBox (ex : {components['checkbox1'].value || components['checkbox2'].value}) et le mécanisme de liaison et de notification fera en sorte que le bouton « suivant » ne soit disponible que lorsque l'une des deux cases à cocher est cochée. Finalement, « validationCommand », si spécifié, doit retourner vrai pour permettre à l'utilisateur de passer à l'étape suivante.

Une dernière série de commandes permettent l'ouverture ou l'enregistrement de fichiers. Il est possible de conserver le dernier emplacement utilisé au niveau du contexte à l'aide de l'attribut « currentDirectoryBinding ». « fileNameBinding » permet de stocker le nom complet du fichier sélectionné de la même manière. Ces commandes définissent uniquement la gestion du dialogue. Une commande doit être spécifiée pour l'exécution de l'opération. L'interpréteur, exécutant le dialogue, est ensuite chargé de placer dans une variable « filename » le chemin du fichier disponible. Cette variable est alors disponible pour être utilisée comme argument lors de l'exécution de la commande.

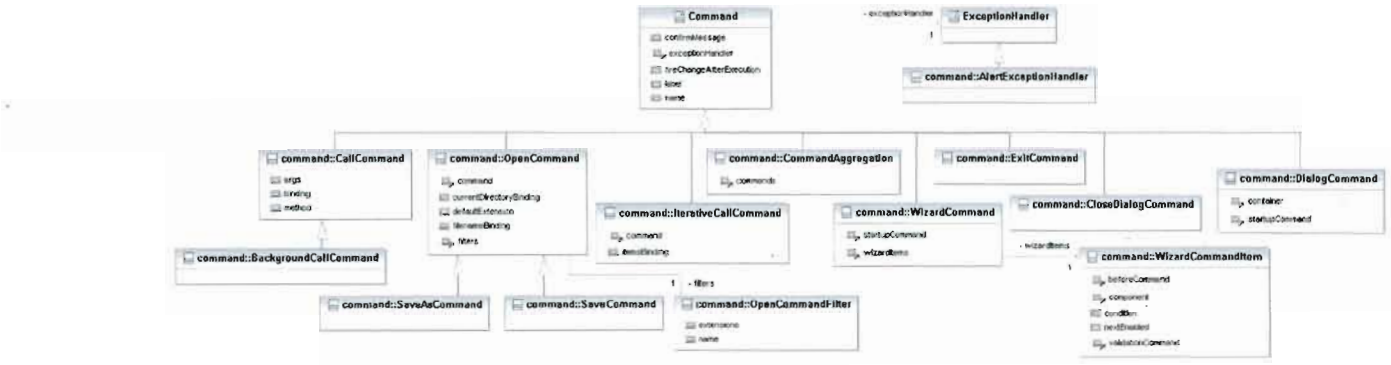


**Figure 8.11** Commandes, prototype 2

### 8.4.4 Exemples

Voici quelques exemples d'utilisation du modèle pour une application réelle. Le point d'entrée est l'instance de type application. Le fichier est définit en plusieurs fichiers, par souci de lisibilité, un fichier XML pour chacun des onglets principal a été créé.

Ce premier exemple montre une application de type TDI. La propriété « childrens » réfère aux onglets qui seront disponibles. L'attribut « title », comme pour la plupart des attributs du modèle, supporte les expressions EL. La seule chose à faire pour que le titre de la fenêtre contienne les informations sur le fichier ouvert est d'y insérer des expressions EL référant au contexte. Lorsqu'une modification est effectuée au niveau du contexte et que les composants liés sont notifiés, le titre sera rechargé pour prendre en compte les nouvelles données. L'attribut « context » de l'application contient les valeurs initiales du contexte pour l'application. Les contrôleurs disponibles peuvent ensuite être définis ou référés à cet endroit.

Ces derniers seront ensuite disponibles, toujours via EL, pour être accédés lors de l'appel de méthode.

```
<bean id="application"
       class="ca.crim.cdmpf.model.application.TDIApplication">
    <property name="title"
      value="Horaires {context['grille'].eco}{context['grille'] != null ? ' -
       ' : ''}{context['grille'].annee}{context['grilleFileName'] != null ? '
       - ' : ''}{context['grilleFileName']}" />
    <property name="menus" ref="application.menus" />
    <property name="context">
      <map>
        <entry key="show_deleted" value="false" />
      </map>
    </property>
    <property name="controllers">
      <map>
        <entry key="main">
          <bean class="ca.crim.horcs.app.controller.MainController">
            <property name="filePersistence">
              <bean class="ca.crim.horcs.persistence.xml.XMLFilePersistence"
       />
            </property>
            <property name="faisabilityCheckSolverFactory">
              <bean class="ca.crim.horcs.solver.LocalSolverFactory">
                <property name="className"

       value="ca.crim.horcs.solver.impl.SolverEngineFaisabilityCheck" />
              </bean>
            </property>
          </bean>
        </entry>
[[ ... ]]
      </map>
    </property>
    <property name="childrens">
      <list>
        <ref bean="tab.eleves" />
        <ref bean="tab.groupes" />
        <ref bean="tab.cours" />
        <ref bean="tab.interv" />
        <ref bean="tab.locaux" />
        <ref bean="tab.patrons" />
      </list>
    </property>
  </bean>
```

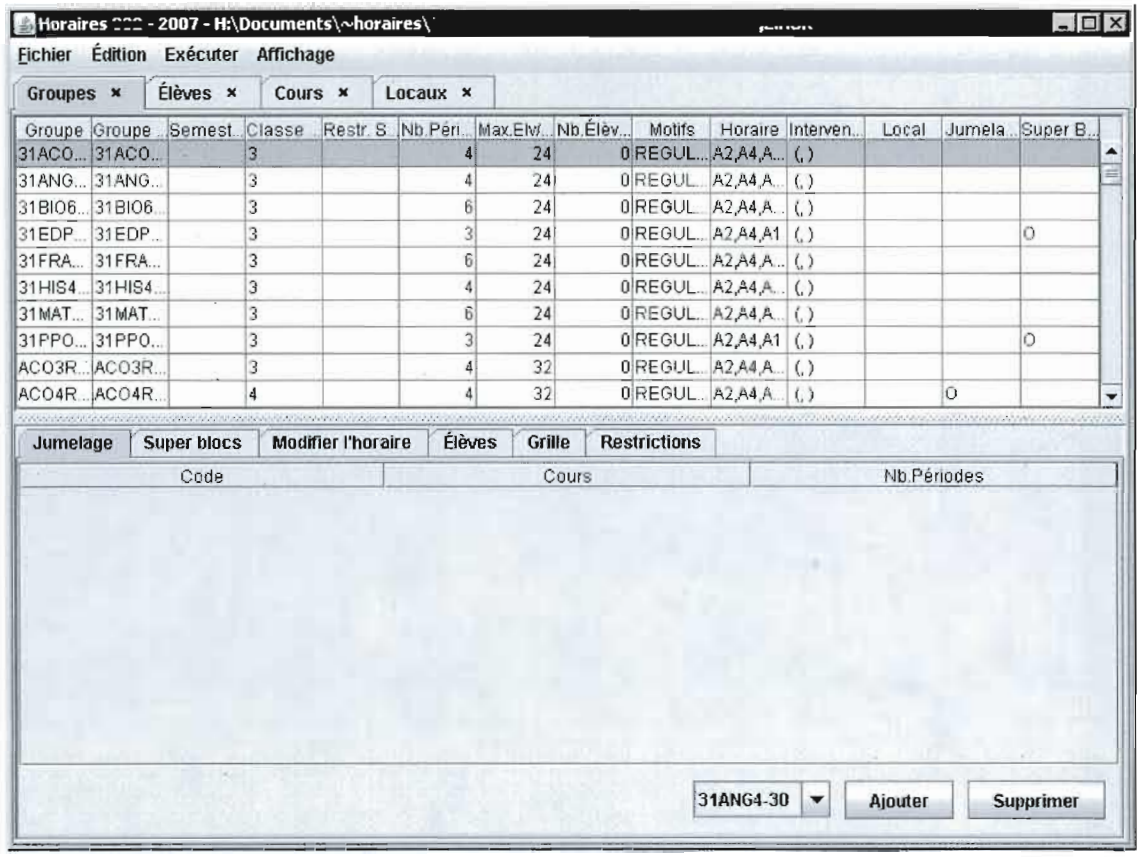


**Figure 8.12** Exemple d'application, prototype 2

Pour appeler un contrôleur, une commande de type « CallCommand » doit être utilisée :

```
<bean class="ca.crim.cdmpf.model.command.CallCommand">
  <property name="binding" value="{controllers['solver']}" />
  <property name="method" value="updateToResult" />
  <property name="args"
       value="{context['solverBean']},{components['solverresult.grid'].selecte
       dItem}" />
  <property name="fireChangeAfterExecution" value="context['grille']" />
  <property name="exceptionHandler">
    <bean class="ca.crim.cdmpf.model.command.AlertExceptionHandler" />
  </property>
</bean>
```

Dans cet exemple, « binding » référence l'objet contrôleur, « method » contient le nom de la méthode à utiliser et « args » contient les arguments à passer en paramètres, séparés par

des virgules. « ExceptionHandler » précise le comportement à appliquer si la méthode exécutée lance une exception.

Les expressions EL pour la liaison peuvent aussi pointer vers d'autres composants. Cet exemple montre un jeu complexe d'interrelations entre cases à cocher faisant en sorte que plusieurs cases à cocher du bas ne sont actives que si « import.wizard.config.horaire » est cochée. La propriété « nextEnabled » référence aussi plusieurs composants du panneau. Celle-ci est réévaluée dès que la valeur de l'un des composants identifiés est mise à jour.

```
<bean id="import.wizard.config"
      class="ca.crim.cdmpf.model.command.WizardCommandItem">
    <property name="component">
      <bean class="ca.crim.cdmpf.model.container.FormContainer">
        <property name="childrens">
          <list>
            <bean id="import.wizard.config.all"
      class="ca.crim.cdmpf.model.component.CheckBox">
              <property name="label" value="Importer TOUT (nouvelle grille)"
      />
              <property name="binding"
      value="{context['importBean'].configImportAll}" />
              <property name="enabled" value="{context['grille'] != null}" />
            </bean>
            <bean id="import.wizard.config.horaire"
      class="ca.crim.cdmpf.model.component.CheckBox">
              <property name="label" value="Lire l'horaire existant (pour
      détection de contraintes)" />
              <property name="binding"
      value="{context['importBean'].configReadHoraire}" />
            </bean>
            <bean class="ca.crim.cdmpf.model.component.CheckBox">
              <property name="enabled"
                value="{components['import.wizard.config.horaire'].value ==
      true and (components['import.wizard.config.all'].value ||
      components['import.wizard.config.groupes'].value)}" />
              <property name="label" value="Marquer comme FIXE les groupes
      ayant déjà un horaire" />
              <property name="binding"
      value="{context['importBean'].configFixeSpecHoraire}" />
            </bean>
            <bean id="import.wizard.config.eleves"
      class="ca.crim.cdmpf.model.component.CheckBox">
              <property name="enabled" value="{not
      components['import.wizard.config.all'].value}" />
              <property name="label" value="MÀJ Élèves / Choix de cours" />
              <property name="binding"
      value="{context['importBean'].configUpdateEleves}" />
            </bean>
            <bean id="import.wizard.config.intervs"
      class="ca.crim.cdmpf.model.component.CheckBox">
```

```
            <property name="enabled" value="{not
      components['import.wizard.config.all'].value}" />
            <property name="label" value="MÀJ Intervenants / Tâches" />
            <property name="binding"
      value="{context['importBean'].configUpdateIntervs}" />
          </bean>
          <bean id="import.wizard.config.locals"
      class="ca.crim.cdmpf.model.component.CheckBox">
            <property name="enabled" value="{not
      components['import.wizard.config.all'].value}" />
            <property name="label" value="MÀJ Locaux / Vocation" />
            <property name="binding"
      value="{context['importBean'].configUpdateLocals}" />
          </bean>
          <bean id="import.wizard.config.groupes"
      class="ca.crim.cdmpf.model.component.CheckBox">
            <property name="enabled" value="{not
      components['import.wizard.config.all'].value}" />
            <property name="label" value="MÀJ Cours / Groupes" />
            <property name="binding"
      value="{context['importBean'].configUpdateGroupes}" />
          </bean>
        </list>
      </property>
    </bean>
  </property>
  <property name="nextEnabled"
    value="{components['import.wizard.config.all'].value or
     components['import.wizard.config.groupes'].value or
     components['import.wizard.config.eleves'].value or
     components['import.wizard.config.intervs'].value or
     components['import.wizard.config.locals'].value}" />
</bean>
```

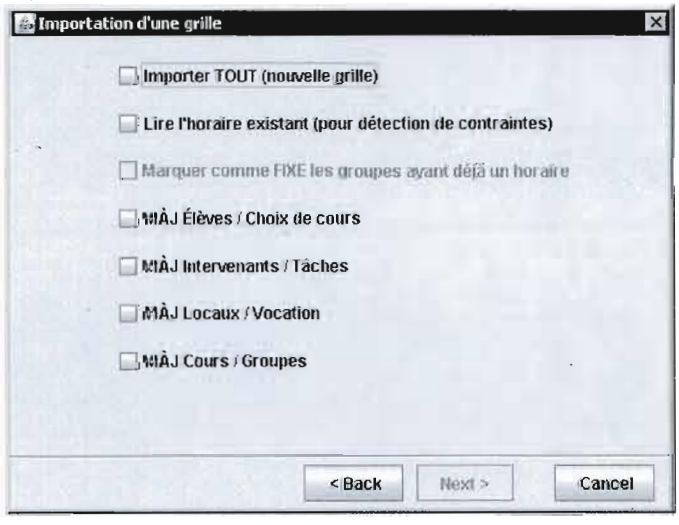


**Figure 8.13** Liaison entre composants, prototype 2

Au niveau de la disposition des contrôles dans les écrans, il est possible d'appliquer certains formats, comme une disposition en colonne. Par exemple :

```
<bean id="tab.groupes.horaire"
      class="ca.crim.cdmpf.model.container.HBoxContainer">
    <property name="label" value="Modifier l'horaire" />
    <property name="childrens">
      <list>
        <bean class="ca.crim.cdmpf.model.container.FormContainer">
          <property name="childrens">
            <list>
              <bean class="ca.crim.cdmpf.model.component.ListBox">
                <property name="label" value="Arrangements" />
                <property name="valueListBinding"
                  value="{context['grille'].grilleLayout.codeArrangList}" />
                <property name="multiSelect" value="true" />
                <property name="binding"

      value="{components['tab.groupes.grid'].selectedItem.codesArrang}" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.CheckBox">
                <property name="label" value="Horaire FIXE" />
                <property name="binding"

      value="{components['tab.groupes.grid'].selectedItem.horaireFixe}" />
              </bean>
            </list>
          </property>
        </bean>
        <bean class="ca.crim.cdmpf.model.container.FormContainer">
          <property name="childrens">
            <list>
              <bean class="ca.crim.cdmpf.model.component.ComboBox">
                <property name="label" value="Intervenant" />
                <property name="allowNull" value="true" />
                <property name="binding"
      value="{components['tab.groupes.grid'].selectedItem.interv}" />
                <property name="valueListBinding"
      value="{context['grille'].allIntervList}" />
                <property name="textAttribute" value="{item.codeInterv}
      ({item.nom}, {item.prenom})" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.CheckBox">
                <property name="label" value="Intervenant FIXE" />
                <property name="binding"

      value="{components['tab.groupes.grid'].selectedItem.intervFixe}" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.ComboBox">
                <property name="label" value="Local" />
                <property name="allowNull" value="true" />
                <property name="binding"
      value="{components['tab.groupes.grid'].selectedItem.local}" />
                <property name="valueListBinding"
      value="{context['grille'].allLocalList}" />
```

```
            <property name="textAttribute"
    value="{item.identifiant} ({item.descr})" />
          </bean>
          <bean class="ca.crim.cdmpf.model.component.CheckBox">
            <property name="label" value="Local FIXE" />
            <property name="binding"

    value="{components['tab.groupes.grid'].selectedItem.localFixe}" />
          </bean>
          <bean class="ca.crim.cdmpf.model.component.Button">
            <property name="label" value="Sélectionner un local
    disponible..." />
            <property name="enabled"
    value="{components['tab.groupes.grid'].selectedItem != null}" />
            <property name="command"
    ref="tab.groupes.horaire.chooselocal" />
          </bean>
        </list>
      </property>
    </bean>
  </list>
 </property>
 <property name="bottomComponents">
  <list>
    <bean class="ca.crim.cdmpf.model.component.Button">
      <property name="label" value="Actualiser la grille" />
      <property name="command">
        <bean class="ca.crim.cdmpf.model.command.CommandAggregation">
          <property name="fireChangeAfterExecution"
            value="components['tab.groupes.grid'].selectedItem" />
        </bean>
      </property>
    </bean>
  </list>
 </property>
</bean>
```

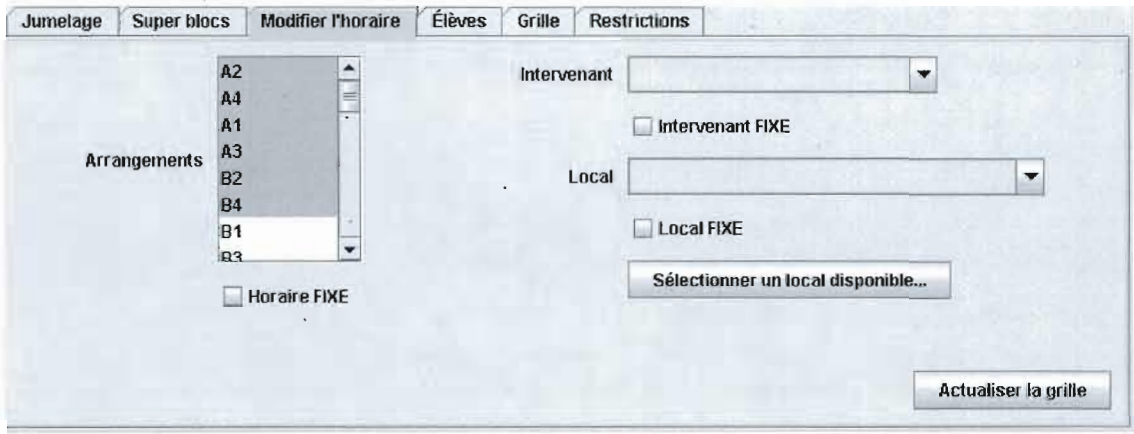


**Figure 8.14** Formulaire d'édition, prototype 2

L'application de génération des horaires permet d'exclure des données du processus de traitement. Cette opération s'effectue concrètement en assignant la valeur « vrai » à l'attribut « deleted » des objets à exclure. Les objets exclus peuvent, au besoin, être restaurés. Une case à cocher dans un menu permet de préciser si on veut ou non afficher les informations exclues. Cette mécanique a pu être totalement implémentée à l'intérieur du modèle d'application. Le composant DataGrid et ses attributs « rowVisible » et « rowStriked », ont permis la simulation de nouveaux comportements.

Le menu « Afficher les lignes exclues » est, dans un premier temps, lié à une variable du contexte.

```
<bean class="ca.crim.cdmpf.model.component.MenuItem">
<property name="label" value="Afficher lignes exclues" />
      <property name="checkable" value="true" />
      <property name="checkBinding" value="{context['show_deleted']}" />
</bean>
```

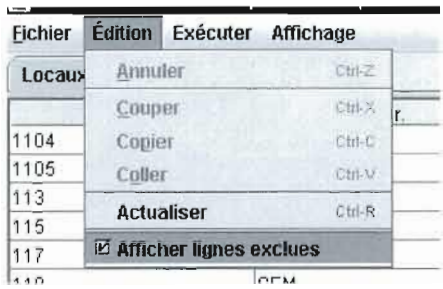


**Figure 8.15** Filtre sur les lignes, prototype 2

Cette variable influencera ensuite le comportement à appliquer à l'intérieur des DataGrid. Dès que la variable est modifiée, AppContext s'occupera de notifier tous les contrôles ayant une référence à « context['show_deleted'] », comme c'est le cas avec l'exemple suivant. Celui-ci montre que les lignes avec la propriété « deleted » ne seront visibles que si la variable « show_deleted » du contexte est vraie. Ces lignes, lorsqu'affichées, seront par contre rayées.

Cet exemple montre aussi l'utilisation d'une commande de type « DataGridCommand ». Il s'agit de commandes s'appliquant aux lignes du « DataGrid ».

Dans l'implémentation Swing, elles se présentent sous forme d'un menu contextuel. La commande est accessible (enabled) uniquement lorsque la ligne n'a pas la propriété « deleted » (enabled ="{not item.deleted}"). Cet exemple montre aussi l'utilisation d'une commande de type « IterativeCallCommand » pour laquelle la commande contenu (une « CallCommand » sera appelée pour chacun des éléments d'une liste, en occurrence, « {selectedItems} » contenant la collection de lignes actuellement sélectionnées dans le « DataGrid ». La commande appelle ici la méthode « excludeLocal » de l'objet du contexte « grille » avec, en paramètre, le « local » placé dans la variable « item » par l'itérateur « IterativeCallCommand ».

```
<bean id="tab.locaux.grid" class="ca.crim.cdmpf.model.component.DataGrid">
  <property name="readOnly" value="false" />
  <property name="itemsBinding" value="{context['grille'].allLocalList}" />
  <property name="rowVisible" value="{context['show_deleted'] == true or
     (not item.deleted)}" />
  <property name="rowStriked" value="{item.deleted}" />
  <property name="multiSelect" value="true" />
  <property name="rowCommands">
    <list>
      <bean class="ca.crim.cdmpf.model.component.DataGridCommand">
        <property name="label" value="Exclure" />
        <property name="enabled" value="{not item.deleted}" />
        <property name="command">
          <bean class="ca.crim.cdmpf.model.command.IterativeCallCommand">
            <property name="itemsBinding" value="{selectedItems}" />
            <property name="command">
              <bean class="ca.crim.cdmpf.model.command.CallCommand">
                <property name="binding" value="{context['grille']}" />
                <property name="method" value="excludeLocal" />
                <property name="args" value="{item}" />
              </bean>
            </property>
            <property name="fireChangeAfterExecution"
   value="context['grille'].allLocalList" />
          </bean>
        </property>
      </bean>
      ...
    </list>
  </property>
  <property name="columns">
    <list>
      <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
        <property name="label" value="Local" />
        <property name="binding" value="{item.identifiant}" />
        <property name="editable" value="{false}" />
        <property name="sortable" value="true" />
      </bean>
      ...
      <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
```

```
        <property name="label" value="Restr.champ.ens." />
        <property name="binding" value="{item.champEns}" />
        <property name="editable" value="{true}" />
        <property name="multiValues" value="true" />
        <property name="valueListBinding"
    value="{context['grille'].champEns}" />
        <property name="textAttribute" value="code" />
        <property name="sortable" value="true" />
      </bean>
      <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
        <property name="label" value="Capacité" />
        <property name="binding" value="{item.capacite}" />
        <property name="editable" value="{true}" />
        <property name="sortable" value="true" />
      </bean>
      <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
        <property name="label" value="Pér.Non Dispo." />
        <property name="binding" value="{item.codesArrangExclusionText}" />
        <property name="editable" value="{false}" />
        <property name="sortable" value="true" />
      </bean>
    </list>
  </property>
</bean>
```

Locaux ×

| Local | Descr. | Contenu | Restr.classes | FORCER restr. | Restr.champ. | Capacité | Pér.Non Dispo. |
|---|---|---|---|---|---|---|---|
| 1104 | Palestre | | | ☐ | 09 | 32 | |
| 1105 | Palestre | | | ☐ | 09 | 32 | |
| ~~113~~ | ~~SEM~~ | | | ☐ | | ~~32~~ | |
| ~~115~~ | ~~SEM~~ | | | ☐ | | ~~32~~ | |
| ~~117~~ | ~~SEM~~ | | | ☐ | | ~~32~~ | |
| ~~119~~ | ~~SEM~~ | | | ☐ | | ~~32~~ | |
| 1200 | Gymnase | | | ☐ | 09 | 32 | |
| 1201 | Gymnase | | | ☐ | 09 | 32 | |
| ~~1205~~ | ~~Musculation~~ | | | ☐ | ~~09~~ | ~~32~~ | |
| ~~1210~~ | ~~Atelier Scienc...~~ | | | ☐ | | ~~32~~ | |
| ~~1222~~ | ~~Atelier Scienc...~~ | | | ☐ | | ~~32~~ | |
| ~~1232~~ | ~~Musique studio~~ | | | ☐ | ~~10~~ | ~~32~~ | |
| 1233 | Danse | | | ☐ | 19C,19D | 32 | |
| 1239 | Musique studio | | | ☐ | 10 | 28 | |
| 1303 | Anglais | | 3 | ☐ | 08 | 32 | |
| 1304 | Anglais | | 3 | ☐ | 08 | 32 | |
| 1305 | Arts plastiques | | | ☐ | 01F | 32 | |
| 1306 | Anglais | | 3 | ☐ | 08 | 32 | |
| 1307 | Petit foyer | | | ☐ | | 0 | |
| 1309 | IMPRIMERIE | | | ☐ | | 20 | |

**Figure 8.16** Grille de données, prototype 2

Les « DataGrid » possèdent aussi des attributs pouvant contenir des listes de composants à disposer de différentes manières. Voici un exemple exploitant les propriétés « topComponent », « bottomInfoComponent » et « bottomComponent ».

```
<bean id="tab.eleves.choixcours"
     class="ca.crim.cdmpf.model.component.DataGrid">
  <property name="label" value="Choix de cours" />
  <property name="rowVisible"
    value="{item.selectionOrig == true or item.selection == true or
     components['tab.eleves.choixcours.showall'].value == true}" />
   ...
  <property name="topComponents">
    <list>
      <bean name="tab.eleves.choixcours.showall"
     class="ca.crim.cdmpf.model.component.CheckBox">
        <property name="label" value="Afficher tous les cours" />
      </bean>
    </list>
  </property>
  <property name="bottomComponents">
    <list>
      <bean id="tab.eleves.choixcours.add"
     class="ca.crim.cdmpf.model.component.ComboBox">
        <property name="valueListBinding"
     value="{context['grille'].allCoursTypeEleveList}" />
        <property name="textAttribute"
          value="{item.sigle} {item.fictif ? ' [ FICTIF ] ' : ''}
     ({item.desc})" />
      </bean>
      <bean class="ca.crim.cdmpf.model.component.Button">
        <property name="label" value="Ajouter" />
        <property name="command">
        ...
        </property>
      </bean>
    </list>
  </property>
  <property name="bottomInfoComponents">
    <list>
      <bean class="ca.crim.cdmpf.model.component.Label">
        <property name="label" value="Nb.pér.prévues:" />
        <property name="text"
     value="{components['tab.eleves.grid'].selectedItem.nbPeriodesPrevues}"
     />
      </bean>
    </list>
  </property>
</bean>
```

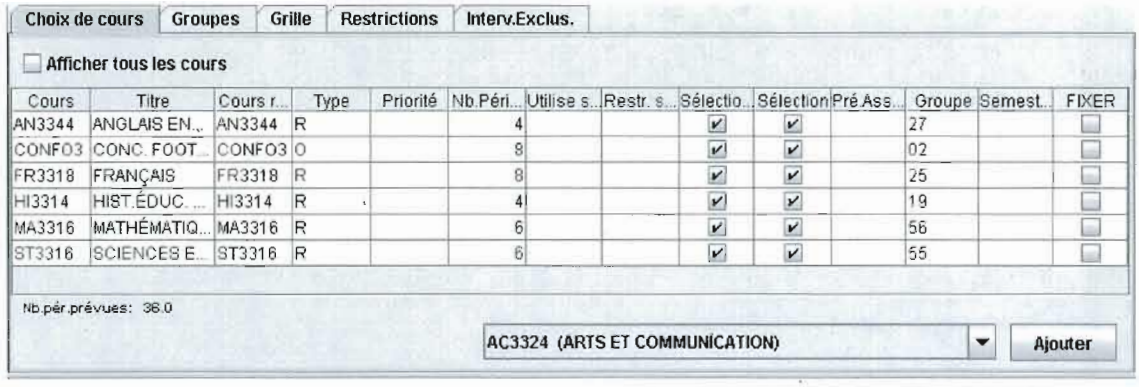


**Figure 8.17** Grille avancée, prototype 2

Cet exemple montre aussi l'utilisation d'une case à cocher influençant la propriété « rowVisible » de la table. Lors d'un changement de valeur, le gestionnaire de liaison fera en sorte que les composants liés soient mis à jour.

La gestion de certains dialogues peut s'effectuer avec l'aide d'un contrôleur spécifique. Cet exemple montre un dialogue de type Assistant pour lequel, à l'ouverture, la méthode « startImport » du contrôleur « retroLocaux » est appelé. Dans ce cas, sa responsabilité est d'initialiser un objet « retroLocauxBean », utilisé pour contenir les informations saisis par les différents écrans du flux d'exécution. Notons aussi l'utilisation des attributs de « WizardCommandItem » spécifiant que l'affichage de la page « retrolocaux.wizard.connexion » dépend de la valeur de « retroLocauxBean.importExternal ». Une commande de validation, « validateConnection » du contrôleur « retroLocaux », doit retourner vrai avant de pouvoir passer à l'étape suivante.

```
<bean id="retrolocaux.wizard"
    class="ca.crim.cdmpf.model.command.WizardCommand">
  <property name="label" value="Rétro-ingénierie des locaux..." />
  <property name="startupCommand">
    <bean class="ca.crim.cdmpf.model.command.CallCommand">
      <property name="binding" value="{controllers['retroLocaux']}" />
      <property name="method" value="startImport" />
      <property name="args" value="{context}" />
      <property name="fireChangeAfterExecution"
     value="context['retroLocauxBean']" />
    </bean>
  </property>
  <property name="wizardItems">
    <list>
```

```
      <ref bean="retrolocaux.wizard.type" />
      <ref bean="retrolocaux.wizard.connexion" />
      <ref bean="retrolocaux.wizard.ecole" />
      <ref bean="retrolocaux.wizard.config" />
      <ref bean="retrolocaux.wizard.finish" />
    </list>
  </property>
  <property name="fireChangeAfterExecution" value="context['grille']" />
</bean>

<bean id="retrolocaux.wizard.connexion"
     class="ca.crim.cdmpf.model.command.WizardCommandItem">
  <property name="condition"
     value="{context['retroLocauxBean'].importExternal == true}" />
  <property name="component">
    <bean class="ca.crim.cdmpf.model.container.FormContainer">
      <property name="childrens">
        <list>
          <bean class="ca.crim.cdmpf.model.component.TextBox">
            <property name="label" value="Driver" />
            <property name="size" value="18" />
            <property name="text"
    value="{context['retroLocauxBean'].connectionParameters['driver']}" />
          </bean>
          ...
          <bean class="ca.crim.cdmpf.model.component.TextBox">
            <property name="label" value="Mot de passe" />
            <property name="size" value="18" />
            <property name="password" value="true" />
            <property name="text"
    value="{context['retroLocauxBean'].connectionParameters['password']}"
    />
          </bean>
        </list>
      </property>
    </bean>
  </property>
  <property name="validationCommand">
    <bean class="ca.crim.cdmpf.model.command.CallCommand">
      <property name="binding" value="{controllers['retroLocaux']}" />
      <property name="method" value="validateConnection" />
      <property name="args" value="{context}" />
      <property name="exceptionHandler">
        <bean class="ca.crim.cdmpf.model.command.AlertExceptionHandler" />
      </property>
      <property name="fireChangeAfterExecution"
     value="context['retroLocauxBean']" />
    </bean>
  </property>
</bean>
```


**Figure 8.18** Assiatant, prototype 2

### 8.4.5 L'interpréteur et l'exécution

Un seul interpréteur a été développé pour ce modèle. Il s'agit d'un interpréteur Swing dédié à l'exécution d'application en mode autonome. Il se définit aussi à l'aide de Spring. Pour chaque composant ou groupe de composant, une classe de type « builder » est associée. La détection de la classe à utiliser s'effectue via la méthode « canHandle ». Par exemple :

```
public boolean canHandle(Object obj) {
        return obj.getClass() == Button.class;
}
```

Pour l'interpréteur Swing, une instance nommée « swingBuilder » sert de point d'entrée pour l'exécution de l'application. Le « ApplicationBuilder » correspondant au type d'application (TDI seulement pour l'instant) est alors lancé. Tout au long du parcours de l'arborescence, « swingBuilder » sera appelé afin de récupérer les classes permettant d'effectuer la conversion des objets du modèle en interface utilisateur Swing.

```
<bean id="swingBuilder"
     class="ca.crim.cdmpf.platform.swing.SwingBuilder">
  <property name="applicationBuilders">
    <list>
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingTDIApplicationBuilder"
     />
    </list>
  </property>
  <property name="componentBuilders">
    <list>
      <bean class="ca.crim.cdmpf.platform.swing.builder.SwingMenuBuilder"
     />
      <bean class="ca.crim.cdmpf.platform.swing.builder.SwingBoxBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingDataGridBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingTabContainerBuilder"
     />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingListBoxBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingComboBoxBuilder" />
      <bean class="ca.crim.cdmpf.platform.swing.builder.SwingButtonBuilder"
     />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingSplitterBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingFormContainerBuilder"
     />
      <bean class="ca.crim.cdmpf.platform.swing.builder.SwingLabelBuilder"
     />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingTextBoxBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingCheckBoxBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingConsoleBuilder" />
      <bean class="ca.crim.horcs.app.cdriven.swing.SwingHoraireBuilder" />
    </list>
  </property>
  <property name="commandBuilders">
    <list>
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingCommandBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingOpenSaveCommandBuilder
     " />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingExitCommandBuilder" />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingDialogCommandBuilder"
     />
      <bean
     class="ca.crim.cdmpf.platform.swing.builder.SwingWizardCommandBuilder"
     />
      <bean class="ca.crim.horcs.app.cdriven.swing.SwingCommandExecBuilder"
```

```
      />
        <bean
      class="ca.crim.horcs.app.cdriven.swing.SwingSolverCommandBuilder" />
      </list>
    </property>
    <property name="exceptionHandlerBuilders">
      <list>
        <bean
      class="ca.crim.cdmpf.platform.swing.builder.SwingAlertExceptionHandler"
      />
      </list>
    </property>
  </bean>
```

L'objectif de l'interpréteur dans cette situation est en fait de créer une application Swing qui s'exécutera par la suite indépendament. Les composants du modèle sont transformés et objets Swing et disposés au bon endroit. Lorsque des attributs du contrôle contiennent des expressions EL, des observateurs sont ensuite ajoutés. Par exemple, pour le composant « TextBox », la modification du texte lié est propagée au niveau du contexte uniquement lorsque le focus est retiré de la zone de texte. Le « builder » ajoutera donc un observateur à la zone de texte en utilisant la méthode « addFocusListener » du « JTextField ». Pour les liaisons en lecture seule (ex : attribut visible qui dépend d'une donnée du contexte), le « builder » créé une instance implémentant l'interface PropertyBinding, livré avec AppContext, et l'enregistre dans AppContext. L'instance créée doit implémenter la méthode « notifyChange » qui lance un rafraichissement de l'attribut contenant l'expression EL. Lorsqu'une modification au contexte est signalée, via un « fireChange », AppContexte appelle ensuite les « notifyChange » des liaisons affectées pour compléter le rafraichissement. Cela assure que seuls les composants affectés par une modification seront rafraichis lors d'un événement.

Pour les commandes, les objets du modèle, qui ne contiennent que des attributs, sont transférés vers des objets de l'interpréteur englobant la logique qui y est rattachée. Une « CallCommand » définie au niveau du modèle sera donc transférée dans une classe « SwingReflectiveCommand ». Cette dernière accède à un utilitaire pour l'opération de réflexion et est responsable d'afficher le message de confirmation (le cas échéant), d'appliquer une gestion des exceptions conformément à l'« exceptionHandler » (le cas échéant) et de notifier le contexte d'exécution via la méthode « fireChange » de la classe

« AppContext ». D'autres classes de l'interpréteur, soit SwingCommandAggregation, SwingBackgroundCommand et SwingIterativeCallCommand, implémentent le comportement de chacune des commandes décrites dans la section précédente.

Un conteneur qui référence des composants enfants est responsable d'appeler le « builder » pour chacun d'eux. Il récupère ensuite une référence au composant Swing ayant été créé et peut donc en disposer selon le format qu'il propose. Par exemple, pour un conteneur de type formulaire, « FormContainer », le « builder » récupérera l'étiquette associée pour la placer dans un « JLabel » et se chargera ensuite d'effectuer l'alignement avec le composant Swing associé.

Un concept aussi géré par l'interpréteur au moment de la construction de l'interface est celui d'une arborescence, prenant la forme d'instance de « UINode ». Lorsqu'un composant est traité par le « builder » qui lui est associé, ce dernier retournera un objet de type UINode qui sera ensuite ajouté au UINode du parent par le « builder » appelant. Une hiérarchie parallèle est donc ainsi formée. Celle-ci sert principalement à gérer les variables disponibles lors des expressions EL. Par exemple, lorsqu'une expression EL est exécutée à l'intérieur d'un DataGrid, les variables "selectedItem", "selectedItems" et "selectedItemsCount" pourront être accessibles directement à l'intérieur des expressions (ex : "{selectedItem .name}"). Chaque évaluation EL s'effectue au niveau d'un composant, donc, d'un UINode du côté de l'interpréteur. Lorsque l'expression est évaluée, les variables disponibles sont lues au niveau du UINode qui compile aussi les variables de son parent, récupérant ainsi toutes les variables de la hiérarchie. Ce sont aussi les UINode qui sont enregistrés au niveau du contexte sous « components ». Combiné au fait que plusieurs composants définissent des sous-classes au UINode pour y ajouter leurs attributs, nous pouvons donc accéder aux valeurs de la manière suivante : "{components['text2'].text}". L'arborescence de UINode est aussi utilisée pour le support de fonctionnalités spécifiques nécessitant la connaissance de leur emplacement hiérarchique. C'est le cas de « CloseDialogCommand » qui parcoure cet arborescence afin de déterminer quel est son premier parent de type « Window » pour en effectuer la fermeture.

Certaines fonctionnalités sont définies à l'extérieur de la structure de « builder », sous forme de composant Swing conventionnel. C'est le cas des écrans de Wizard qui pourraient être réutilisés dans une application Swing conventionnelle. Le package « wizard » comporte une classe principale dérivée de « WindowAdapter » à laquelle on ajoute des étapes à l'aide d'instances de « WizardModel ». Chaque étape comporte quelques paramètres de configuration, comme les boutons disponibles et les composants Swing qu'il contient. Le tout est instancié par le « builder » et peut être lancé sous la forme d'une commande.

### 8.4.6 Conclusions

Cette deuxième version de l'approche de développement par modèle présente une solution beaucoup plus complète et adaptée au contexte d'un client riche. Elle permet d'assembler rapidement des applications complexes englobant de nombreux liens entre ses composants. La mécanique de liaison utilisée, gérant l'enregistrement et la notification des composants de l'interface, permet d'alléger la mise en place de dynamisme à l'intérieur de l'application. Il s'agit d'une valeur ajoutée par rapport à d'autres dialectes comme UsiXML, ne gérant pas cet aspect, ou à ceux dont la liaison entre composants est essentiellement gérée au niveau des événements, comme UIML, XUL ou LZX.

Une autre force de cette implémentation est sa capacité à s'intégrer à toute application Java existante. Les classes contrôleurs utilisées ici sont des classes Java simples, sans dépendances à aucune technologie. Il serait possible d'ajouter, avec un minimum d'efforts, une couche graphique à une application qui rend accessible ses fonctionnalités via une couche service.

Cette solution, en étant basée sur un modèle n'ayant pas de lien avec une technologie de présentation, seul le développement d'une nouvelle couche interpréteur serait nécessaire à l'exécution des applications, définies selon ce modèle, sous SWT ou une technologie d'application Web riche comme ZK ou XUL.

La solution a toutefois pour effet de limiter les possibilités des développeurs quant au design des écrans et des formulaires. Advenant qu'une disposition particulière doive être

appliquée, la modification ou l'ajout de composant à l'interpréteur est nécessaire, ce qui est plus long et complexe que pour une application conventionnelle. Des ajouts sous forme de composant générique, pour répondre à ces besoins, pourraient venir enrichir le modèle et la part de développement nécessaire pour de nouveaux développements ne cesserait alors de diminuer.

Certains points devraient encore être améliorés, notamment au niveau de la liaison avec le modèle. Au lieu de spécifier les éléments du modèle de l'application susceptible d'êtres modifiés lors de l'exécution d'une commande, avec « fireChangeOnAction », il serait préférable d'utiliser la mécanique d'observateur, standard à JEE (java.beans.PropertyChangeListener) pour surveiller les objets contenant des données affichées. Imposer l'implémentation de cette interface serait toutefois contraire à la philosophie adoptée favorisant l'utilisation d'objets simples au niveau de l'application. Ce problème pourrait toutefois être corrigé via l'utilisation d'AOP pour ajouter cette implémentation sous forme d'aspect.

## 8.5 Évaluation des résultats

Ces trois projets auront permis de prouver la faisabilité et le bon fonctionnement de la méthode proposée. Les deux implémentations distinctes de l'approche ont toutes deux présenté un modèle et des composants de haut niveau indépendant d'une technologie de présentation. La preuve de concept quant à l'exécution de la transformation au moment de l'exécution a aussi été effectuée par le développement de ces applications.

La prochaine étape, qui permettra de valider l'hypothèse, est maintenant l'évaluation des résultats obtenus. Nous avons émis l'hypothèse que l'approche utilisée permettait de réduire l'effort de développement, d'améliorer la maintenabilité, de favoriser la réutilisation de code, de faciliter les tests tout en assurant la portabilité pour plusieurs technologies de présentation. Ces points, devant être évalués, sont en fait des critères de qualité logicielle.

L'ouvrage de David Budgen, Software Design (Budgen, 2003), présente un cadre d'évaluation de la qualité logicielle. Il est essentiel de retenir que l'évaluation de propriétés logicielles doit s'appuyer sur des éléments mesurables et quantifiables afin d'en premettre la

comparaison. Pour l'évaluation d'un concept abstrait, comme, par exemple, la complexité ou la maintenabilité d'une application, la première étape est d'identifier des métriques qui y sont associées. Une métrique est une caractéristique identifiable permettant de faire le lien entre un concept abstrait à des éléments pouvant être comptés (Budgen, 2003). Les résultats d'une métrique sont ensuite obtenus en dénombrant un ou plusieurs éléments propres au logiciel. L'interprétation de la métrique et, par conséquent, l'évaluation de la caractéristique souhaitée se font ensuite en comparant les résultats obtenus avec ceux d'autres applications.

Pour chacune des qualités devant être évaluées pour confirmer l'hypothèse, un certain nombre de métriques seront donc sélectionnées. Les résultats obtenus feront ensuite l'objet d'une évaluation basée sur d'autres données récoltées à l'intérieur de projets similaires, mais utilisant une approche de développement conventionnelle.

### 8.5.1 Productivité

La productivité lors du développement met en perspective la taille du logiciel par rapport à l'effort de développement (Galorath et Evans, 2006). L'estimation de la productivité est généralement utilisée lors de la planification d'un projet pour fin d'estimation de l'effort requis. Dans le cas présent, la productivité pourra être mesurée précisément en fonction des travaux déjà réalisés et du temps réel ayant été requis pour la réalisation.

Puisque la taille du logiciel est une donnée de base pour l'évaluation de la productivité d'un développement, il faut, dans un premier temps, déterminer les unités de mesure utilisées. Les mesures de taille les plus répandues aujourd'hui sont le nombre de lignes de code et le nombre de points de fonction (Galorath et Evans, 2006). Le nombre de lignes de codes apparait comme une métrique simple, mais plusieurs problèmes se pausent quant à son évaluation. En effet, les règles de formatage utilisées (ex : placer un saut de ligne avant chaque accolade ou pas) peuvent grandement influencer les résultats. Certains utiliseront un compte physique, soit, le nombre de lignes dans un fichier, et d'autres favoriseront un compte logique, soit compter le nombre d'expressions présente dans un fichier. Les résultats peuvent aussi être biaisés lorsque certaines classes sont générées à l'aide d'outils. Il est alors important de faire une distinction et évaluer de façon différente ce type de code. Cette

séparation peut difficilement être mise en pratique lorsque certaines classes contiennent à la fois du code généré et du code écrit à la main, comme c'est le cas avec des outils visuels de définition d'interface utilisateur tel que NetBeans (netbeans.org, 2007). Il est aussi difficile de comparer des résultats obtenus à l'aide de différentes technologies de développement.

La métrique des points de fonctions permet d'obtenir la taille d'une application de manière indépendante d'une technologie de développement. L'évaluation s'effectue du point de vue de l'utilisateur en dénombrant les composants fonctionnels (entrées, sorties, interfaces, fichiers, requêtes...). Il met l'accent sur ce qu'offre le système plutôt que sur ce qu'il fait. Cette approche à l'avantage d'être plus encadrée que le nombre de lignes de code et d'être près de l'utilisateur. Par contre, elle est plus complexe à calculer et, lorsqu'évaluée avant l'étape de développement, peut être bloquée par le manque d'information disponible (Galorath et Evans, 2006).

Dans notre situation, pour comparer la productivité de projets de développement déjà complétés et dont les différences sont essentiellement au niveau de technologies utilisées, l'utilisation de la taille fonctionnelle semble l'approche la plus appropriée. La méthode de mesure employée est COSMIC-FFP version 3.0 (Abran, 2007). Celle-ci se concentre sur les mouvements appliqués sur les groupes de données pour des applications de gestion et de type temps réel. Chaque point de fonction représente un mouvement (Entry, Exit, Read, Write) appliqué sur les données d'un processus fonctionnel.

La démarche retenue pour la mesure de productivité est la suivante. Nous utiliserons les deux versions du client développé pour le projet de génération d'horaires scolaires : la première étant une application Swing conventionnelle et la deuxième exploitant le prototype de définition d'application basée sur un modèle. L'évaluation de la taille fonctionnelle permettra de comparer les deux projets sous une base commune, soit, le nombre d'heures par point de fonction. Cette étape est nécessaire puisque ceux-ci ne comportent pas exactement les mêmes fonctionnalités.

Un point important à mentionner est que seuls les efforts rattachés au développement des interfaces utilisateurs et de la couche contrôleur seront mesurés. En fait, les deux clients exploitent, de la même manière, un modèle objet instancié par l'application contenant la logique d'affaire et toutes les règles associées au domaine. Les temps de rencontres et d'analyses rattachés au projet de génération automatisé des horaires n'ont également pas été comptabilisés dans le temps de développement des clients graphiques. Cette situation permet d'effectuer une comparaison juste entre les deux approches de développements d'interfaces utilisateurs mais fait en sorte de limiter les possibilités de comparaisons avec d'autres projets pour lesquels les mesures de productivité comprendraient du temps pour l'analyse et les accès aux données par exemple.

| | Temps (heures) |
|---|---|
| Développement du client original | 93 |
| Développement du modèle et de l'interpréteur | 107 |
| Définition de l'application, XML + Controlleur | 19 |

**Tableau 8.1** Temps de développement, client pour le générateur d'horaires

Le tableau 7.1 présente le nombre d'heures consacrées aux différents projets. Le développement du client original a été réalisé en 93 heures. Cela inclus le design des interfaces utilisateurs la logique de présentation et la couche contrôleur permettant la lecture et la mise à jour du modèle. Tous les écrans développés à l'intérieur de cette version sont spécifiques aux données qu'elles présentent. L'interpréteur effectuant la conversion du modèle en interface utilisateur Swing a été réalisé en 107 heures. Les composants développés sous ce point sont entièrement génériques et peuvent être réutilisés dans n'importe quel projet. Finalement, le temps consommé pour l'écriture des fichiers XML, décrivant le contenu du client, et pour l'écriture classes Java, servant de contrôleurs est de 19 heures.

| | Temps (heures) | COSMIC-FP | Heure(s) / CFP |
|---|---|---|---|
| 1. Client Swing original | 93 | 89 | 1,04 |
| 2.1 Interpréteur + Définition nouveau client | 126 | 297 | 0,42 |
| 2.2 Définition nouveau client seul | 19 | 297 | 0,06 |

**Tableau 8.2** Taille fonctionnelle, clients pour le générateur d'horaires

Ce tableau montre les résultats obtenus en termes d'heures par point de fonction. Le tableau de calcul détaillé est disponible en annexe. La ligne 2.1 tient compte du temps de

développement total, incluant la conception du modèle et de l'interpréteur Swing, ce qui montre que l'approche par modèle, dans le cas présent, aura offert une productivité de l'ordre de 2.5 fois plus élevée que pour le développement Swing original. Les objectifs recherchés ont donc été atteints. De plus, l'interpréteur développé étant générique et réutilisable, il est logique de croire que, pour le développement d'un client similaire pouvant être réalisé uniquement avec les composants déjà développés, seul du temps pour l'écriture de la définition XML serait alors requis. Advenant cette hypothèse confirmée, le nombre d'heures par point de fonction COSMIC serait alors grandement réduit, augmentant la productivité par un facteur de 17 par rapport à un développement Swing traditionnel.

### 8.5.2 Maintenabilité

La maintenabilité implique la facilité avec laquelle on peut modifier ou appliquer des corrections à un logiciel. Elle est reliée à la facilité de compréhension, de changement, de test et à la flexibilité (Wiegers, 2003).

On peut mesurer la flexibilité de différentes manières. La facilité de modification est liée directement au temps requis pour une modification (Wiegers, 2003). Un exemple de modification à l'application ayant été documenté est celui de l'ajout d'un sous-menu pour l'application d'un traitement à des données sélectionnées depuis une grille de données. Suite à une demande du client, il fut nécessaire d'ajouter un mécanisme permettant de modifier simultanément une sélection de groupes depuis l'onglet « Groupes » de l'application afin de leur ajouter ou de leur retirer les attributs « fixe » pour l'horaire. Trois attributs (horaire fixe, enseignant fixe et local fixe) devaient être modifiés pour chacune des lignes sélectionnées. Le développement couvrait donc, l'ajout d'une méthode Java a un contrôleur, utilisé pour la modification du domaine et l'ajout de ces commandes au niveau d'un sous-menu dans le composant « DataGrid ». Le temps de développement total, calculé lors de cette opération, a été de 6 minutes, incluant 4 minutes pour les tests.

Pour mesurer le niveau de maintenabilité d'une application, l'indice de maintenabilité MI, de Welker, peut être utilisé (VanDoren, 2002). Plus le résultat est élevé, plus le niveau de maintenabilité est élevé.

$$MI = 171 - 5.2\ln(aV) - 0.23aV(g') - 16.2\ln(aLOC) + 50\sin[(2.4 * perCM)^{1/2})]$$

aV = la moyenne des mesures d'Halstead par module

aV(g') = la complexité cyclomatique moyenne par module

aLOC = le nombre moyen de lignes de code par module

La mesure d'Halsted (VanDoren, 1997) correspond à une mesure basée sur les opérandes et opérateurs présents dans un module: V= N * (LOG2 n) ou V = le nombre total d'opérandes et d'opérateurs et n = le nombre total d'opérandes et d'opérateurs distincts.

La complexité cyclomatique (VanDoren, 2000) se base sur la structure de contrôles d'un programme (conditions, boucles, ..) ramenée sous forme d'un graph. L'équation est la suivante : M = E – N + 2P ou E = le nombre d'arrêtes, N le nombre de sommets et P, le nombre de composants connectés au graph.

Ces mesures s'appliquent difficilement aux définitions XML utilisées puisque nous ne sommes pas en présence d'éléments de contrôle ou de flux d'exécution. Le fait s'avoir un modèle descriptif ne contenant pas de flux d'exécution, tel que retrouvé dans du code conventionnel, devrait ramener la mesure de la complexité cyclomatique à un niveau très peu élevé. Le nombre d'opérateurs et d'opérandes pourrait englober les noms des variables référencés par le modèle ainsi que les opérations contenues dans les expressions EL. Ce nombre est évidemment moins grand que pour les mêmes développements réalisés à l'aide de l'approche conventionnelle.

Puisque, pour la fonction de calcul MI, les valeurs des mesures d'Halstead et de la complexité cyclomatique seront inférieures à ce qu'on retrouve pour un code conventionnel équivalent. Des valeurs faibles à ce niveau font augmenter la valeur de l'indice et font ressortir, du même coup, une meilleure maintenabilité.

Concernant des lignes de codes, la variable « aLoc » de la formule, l'écart est aussi remarquable. L'application originale des horaires, développée en Swing, comportait 26

classes dédiées à l'affichage des panneaux (sections d'écran), pour un total de 5612 lignes de code et une moyenne de 215 lignes par panneau. La version XML, pour la définition de panneaux identiques (ayant même des fonctionnalités en plus) comporte 3099 lignes de XML pour 41 panneaux, donnant une moyenne de 75 lignes XML par panneaux.

Du côté de ETLTools, l'utilisation d'une approche classique utilisant, pour chaque table de données, 6 fichiers (ex : itemsList.jsp, item.jsp, itemForm.jsp, AddItemController, EditItemController, FindItemController ~ 300 lignes de code) tel qu'utilisée dans le projet PetClinic (http://static.springframework.org/spring/docs/2.0.x/reference/testing.html#testing-examples-petclinic[7]) aurait porté à 270 le nombre de fichiers et 13 500 lignes de code nécessaires pour les 45 tables couvertes. En comparaison, les 135 écrans distincts actuellement générés par ETLTools dans le projet de tableau de bord sont entièrement définis à l'intérieur d'un seul fichier XML de 1150 lignes.

Les évaluations précédentes tiennent compte uniquement du code XML pour la version de l'application basée sur un modèle. La partie interpréteur a été écartée en considérant que la part de l'application devant être maintenue, pouvant évoluer, se limite à son volet descriptif, soit la définition XML. Pour les versions Java conventionnelles, la description de l'application et la logique est contenue dans le code Java de l'application, ce qui nous pousse à inclure l'ensemble de celui-ci dans les calculs reliés à la maintenabilité. La ligne n'est cependant pas clairement tracée entre ce qui devrait entrer ou pas dans l'évaluation de la maintenabilité d'une application basée sur un modèle.

### 8.5.3 Réutilisabilité

La réutilisation dans le contexte d'un développement logiciel se traduit par la récupération de code développé antérieurement sans devoir y appliquer de modification (Futrell, Shafer et Shafer, 2002). La principale métrique pouvant être utilisée pour cette qualité est le nombre de lignes de code réutilisées pour un composant, faisant ici ressortir le pourcentage de réutilisation de code.

---

[7] Dernière consultation, août 2007

Certaines fonctions sont réutilisées un grand nombre de fois. Par exemple, « CallCommand » se retrouve 110 fois dans l'ensemble de l'application des horaires. On retrouve aussi 304 instances de « DataGridColumn », définissant, pour chacune des colonnes des grilles de données, la méthode d'édition à utiliser et les listes de valeur. Du côté d'ETLTools, le composant « table » et toutes les fonctionnalités rattachées (ajout, suppression, édition) est repris à 47 endroits.

Le fait que les interpréteurs se basent sur du code générique et que les efforts de développement consacrés aux interfaces utilisateurs ne font qu'assembler des composants génériques ramène à 100% le niveau de réutilisation du code Java. Dans le cas de l'application des horaires, il faut toutefois ajouter les contrôleurs qui sont appelés par l'application depuis la définition XML. Ces contrôleurs permettent d'effectuer des traitements sur le modèle et de gérer certains processus, comme la saisie de plusieurs informations au niveau de la session avant d'appliquer un changement au modèle. Si on considère ce code ajouté (1151 lignes) par rapport à celui de l'interpréteur (7227 lignes), qui constitue le code réutilisé, on obtient un pourcentage de réutilisation du code Java de 86 %. Il est important de noter que les traitements effectués ici auraient été intégrés au code des interfaces utilisateurs dans la version développée directement en Swing. Les classes développées ici sont indépendantes d'une technologie de présentation.

Comme preuve de réutilisabilité d'une application complète, nous avons le cas de CSAdmin qui a pu être redéveloppé en bénéficiant de 100% de code réutilisé. Cela est dû au fait que l'ensemble des besoins de l'application CSAdmin était déjà couvert par le modèle existant. Un tel niveau de réutilisabilité apporte certains avantages au niveau de la maintenance et des tests.

### 8.5.4 Testabilité

La testabilité représente les efforts qui sont nécessaires pour assurer qu'un programme fonctionne tel que ce qui est attendu. Cette qualité peut, dans un premier temps, être liée à la réutilisabilité. Dans l'ouvrage « Quality Software Project Management », les auteurs adressent les efforts nécessaires pour les tests en fonction du degré de réutilisabilité. Une application constituée à 100% de nouveau code impliquera que 100% des efforts de tests

devront être appliqués. Une application constituée de code modifié pourra impliquer 70% des efforts de tests. Finalement, pour une réutilisation pure de code, les efforts de tests tomberont à 0% (Futrell, Shafer et Shafer, 2002). Dans nos implémentations, puisque le code de présentation fait l'objet d'une réutilisation pure, celui-ci peut être entièrement ignoré lors de la phase de tests d'une application.

D'autres types de tests doivent par contre être menés. Lors du développement d'une application via le dialecte de Spring, les références aux composants et les relations entre ceux-ci sont validées au moment de l'édition, par l'éditeur Spring intégré à l'environnement de développement Eclipse. Les erreurs à ce niveau sont donc peu probables. Par contre, les références basées sur les expressions EL sont susceptibles de contenir des erreurs. C'est le cas pour l'ensemble des liaisons entre les composants graphiques et les données du modèle. Il n'est pas possible ici d'appliquer une vérification automatique avant l'exécution du client puisque le contenu du contexte d'exécution, sur lequel les composants sont liés, n'est connu qu'au moment de l'exécution. Lorsqu'une erreur de liaison est présente au niveau de la définition de l'application, c'est l'interpréteur qui se charge de présenter une erreur détaillée permettant d'identifier le composant et la propriété affectée. Pour faire ressortir l'ensemble des erreurs de liaison, un plan de tests doit être exécuté manuellement afin que chacun des écrans ait été affiché au moins une fois et que chaque commande ait aussi été exécutée au moins une fois.

À l'intérieur des études de cas présentés, le temps précis pour le développement de certaines parties de l'application a été consigné. Pour le développement des 7 onglets pour 16 panneaux permettant la consultation des résultats, le temps requis pour l'écriture des fichiers de définition XML s'est élevé à 35 minutes. 25 minutes supplémentaires ont ensuite été requises pour tester l'application, s'assurer que chacune des liaisons soit appelée au moins une fois, effectuer les correctifs nécessaires et reprendre les tests du début. Nous ajoutons donc ici un 70% de l'effort de développement. Un deuxième cas visant à ajouter un sous-menu à une grille de données pour l'exécution d'une transformation sur les éléments sélectionnés a été chronométré à 2 minutes pour les modifications et 4 minutes pour les tests et correctifs, soit 200% du temps de développement. Le temps minimum requis pour la

compilation et l'exécution de l'application vient toutefois jouer un rôle déterminant pour ce calcul.

Les contrôleurs appelés par l'application de génération des horaires sont des classes Java simples sans dépendance à des bibliothèques ou à un conteneur d'application. Ils peuvent donc être repris et instanciés à l'extérieur du client graphique pour lequel ils sont destinés, ce qui constitue une pratique reconnue pour améliorer la testabilité d'une application (Johnson et Hoeller, 2004). Les tests sur les classes contrôleurs peuvent alors se faire sans avoir à initialiser l'ensemble des interfaces utilisateurs et l'environnement d'exécution complet de l'application, ce qui n'est pas toujours le cas lorsque la couche contrôleur est intégrée à une technologie de présentation.

### 8.5.5 Portabilité

La portabilité est la qualité permettant à un développement logiciel d'être porté d'un environnement l'exécution à un autre (Wiegers, 2003). La mesure pouvant confirmer cette qualité serait simplement la confirmation que le code peut être exploité dans les environnements requis. Les exigences au niveau de la portabilité, dans le cadre de ces travaux, se situent uniquement au niveau de la technologie de présentation. La couche applicative devant être exploitée par les clients graphiques étant déjà en Java, la portabilité de la couche interpréteur n'est pas un critère.

L'utilisation d'un modèle pouvant être transformé automatiquement vers une implémentation pour une plateforme concrète est déjà un mécanisme reconnu pour améliorer la portabilité (Mellor, 2004). Les études de cas présentés ici ont pu démontrer que les modèles utilisés ici peuvent faire l'objet de transformation automatisée, puisque cette opération est effectuée à chaque exécution.

Faute de temps, une seule version d'interpréteur aura été réalisée pour chacun des projets. L'objectif du support de plusieurs plateformes a toutefois été pris en considération lors de la conception et nous supposons que l'ajout de nouvelles plateformes à supporter pourrait se faire facilement. Dans le cas d'ETLTools, ou le modèle ne contient que des composants de haut niveau, le support de plateformes plus riches de fera sans problèmes.

L'ensemble des fonctionnalités (présentation, ajout, modification, suppression, formulaires d'édition, ..) est laissé à la discrétion de l'interpréteur, ce qui laisse la marge de manœuvre pour mettre en place tout type de technologie pour les parties serveur et client d'une application Web riche.

Du côté du modèle utilisé pour le développement de l'application des horaires, le support d'une nouvelle technologie de client Web riche pourrait s'avérer plus complexe. Le problème est que la liaison entre l'interface utilisateur, le modèle et les contrôleurs doit obligatoirement être exécutée sur le même palier que l'application Java pour permettre l'évaluation des expressions EL et l'exécution des commandes par réflexion sur les contrôleurs Java. Pour développer une implémentation dédiée à une technologie de présentation Web riche, l'interpréteur devrait maintenir, du côté du serveur, le contexte et le contenu des composants. La technologie client devrait alors inclure ses propres implémentations des composants pouvant recevoir les données préparées par l'interpréteur. Un mécanisme de communication par service Web, AJAX ou autre devrait ensuite être utilisé par le client pour soumettre les commandes exécutées par le client. La commande reçue par le serveur sera alors en mesure d'appeler le modèle d'application et les contrôleurs pour mettre à jour le contexte et, par le mécanisme de liaison, le contenu des composants. Les données modifiées seront alors retournées au client qui pourra rafraichir ses éléments d'affichage. Cette stratégie permettrait l'exploitation de toute technologie de présentation (.Net, Flash, AJAX, ...) puisque les seuls liens avec l'interpréteur Java seraient au niveau du protocole de communication sélectionné.

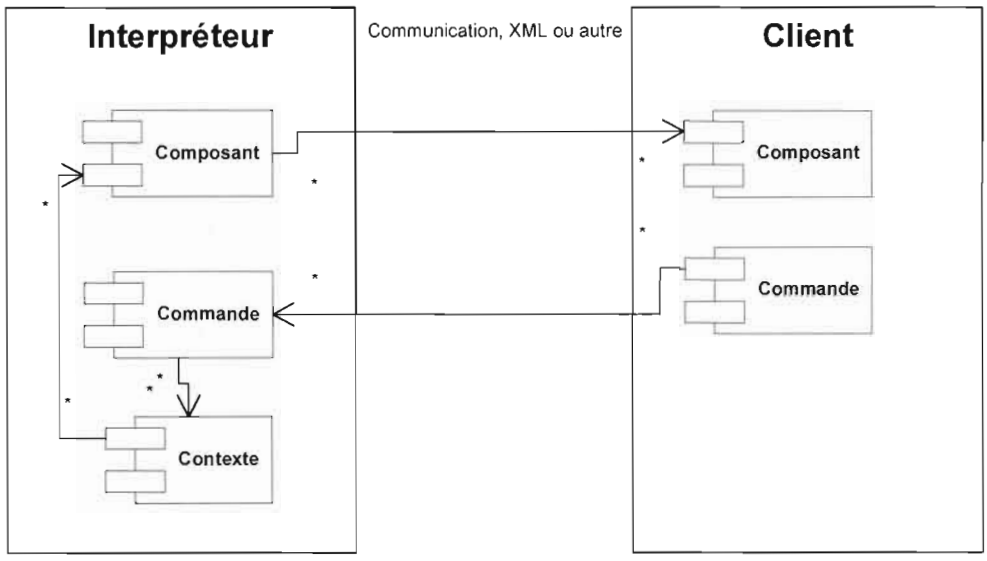


**Figure 8.19** Exécution sous forme de client Web riche, prototype 2

C'est cette stratégie qui est utilisée par le Framework AJAX ZK (Potix, 2007). Une autre possibilité serait d'utiliser directement une technologie comme ZK, pouvant s'exécuter sur le serveur Java, afin de bénéficier d'un mécanisme de communication existant pour les interactions avec le client riche.

# CONCLUSION

## Évolution par rapport à l'idée de base

Alors que les objectifs initiaux étaient principalement de rendre indépendant d'une plateforme les investissements rattachés à la définition des interfaces utilisateurs d'une application, les résultats les plus intéressants ont été obtenus au niveau des méthodes de développement exploitant un modèle de haut niveau. Tel que démontré dans la section 5, les plateformes d'exécutions dépendent d'une multitude de technologies distinctes pour le support des différents aspects du client. Dans le cadre d'un projet destiné à être exploité sous plusieurs plateformes, une simple définition d'interface utilisateur portable pourrait ne pas suffire, nécessitant toujours des développements supplémentaires au niveau de la logique d'affaires et des mécanismes de communications spécifiques aux plateformes de destinations. Pour les preuves de concepts, le temps de développement est aussi devenu un des points les plus importants pour justifier des travaux dans ce domaine. Une solution alliant une définition indépendante de plateforme, couvrant autant les liens avec l'application que la définition de l'interface et pouvant procurer des gains en productivité a donc dû être mise de l'avant.

## Retour sur les résultats

Dans l'ensemble, les objectifs fixés ont été comblés avec l'approche expérimentée. La première partie du mémoire aura permis d'approfondir nos connaissances quant aux plateformes d'exécution contemporaines, aux méthodes de définition d'interface utilisateur, aux dialectes existants exploitant un modèle XML ainsi qu'aux architectures généralement

utilisées pour le déploiement d'applications Web riches. Cette partie aura permis de concevoir une approche originale inspirée des travaux et technologies déjà reliés au domaine.

Les prototypes, réalisés à l'intérieur des études de cas, auront permis d'acquérir de l'expertise poussée en matière de développement d'interpréteur pour l'exécution d'applications client basées sur un modèle de haut niveau. Les progrès, par rapport à la situation de départ, sont importants du fait que nous avons maintenant des prototypes fonctionnels permettant une définition des clients, indépendants d'une technologie de présentation. Ces prototypes pourraient êtres réutilisés dans le développement de clients graphiques, à condition que ces derniers se limitent au sous-ensemble de fonctionnalités ayant été couvert.

Les résultats obtenus s'avèrent très concluants pour l'ensemble des qualités logicielles étudiées. Ces résultats ne s'appliquent toutefois qu'aux applications répondant à des critères particuliers. Les deux prototypes développés ne permettent en effet que la définition d'applications orientées sur la gestion de données via des tables, des actions et des formulaires d'édition. Il est possible de sortir du cadre imposé par les modèles, mais cela nécessite des efforts importants, probablement plus élevés que ce qui serait requis pour une même fonctionnalité à l'aide d'une approche de développement conventionnel.

Les modèles indépendants de plateformes présentés peuvent êtres vu comme des langages neutres, offrant leurs propres vocabulaires pour la représentation du contenu à supporter dans les plateformes spécifiques. La définition du langage neutre s'avère critique puisque c'est lui qui délimitera l'ensemble des fonctionnalités qui pourront êtres utilisées dans l'application. Pour le support de plusieurs plateformes, le langage neutre devra être aussi riche que ce que peut contenir la plateforme la plus pauvre. Dans les études de cas présentés, les langages neutres ont été créés spécifiquement pour la couverture des domaines des applications développés, mettant ainsi l'accent sur la couverture du sous-ensemble de fonctionnalités requis pour la définition de celles-ci.

Pour l'utilisation de l'approche par modèle décrite dans ces travaux, une limite pourrait être tracée entre les projets qui en bénéficieraient et ceux pour lesquels les économies

ne seraient pas au rendez-vous. Par exemple, pour une application comprenant un très grand nombre de composants similaires (comme pour les cas présentés), les avantages peuvent être importants. Par contre, advenant une application composée uniquement d'interfaces spécifiques, comme pour un outil de courrier électronique, nous pouvons supposer que la réutilisation serait beaucoup plus difficile, réduisant ainsi les avantages de la solution présentée.

Certains éléments ne sont pas encore au point en ce qui concerne l'approche de développement. Premièrement, le fait que le modèle soit interprété fait en sorte qu'un grand nombre d'erreurs, généralement identifiées au moment de la compilation, ne ressortent qu'au moment de l'exécution. Cela peut être problématique dû au fait qu'il n'y a pas non plus d'outils d'édition visuelle pour les écrans (comme NetBeans ou VisualBasic) et que la qualité de la définition dépend du programmeur codant la définition XML.

Un autre point n'ayant pas été abordé dans ces travaux est le temps d'apprentissage. Comme une même personne s'est chargée de la conception du modèle et de l'instanciation de ce même modèle, aucun temps ne fut requis à ce niveau. Une excellente compréhension du modèle est toutefois requise avant de pouvoir développer selon cette approche, ce qui pourrait aussi influencer les résultats.

Apport à la recherche

L'ensemble des développements présentés constitue des travaux de recherches et développements originaux, réalisés par l'auteur en lien avec ce projet de maîtrise. La réalisation de deux prototypes exploitant l'approche de développement d'application basée sur un modèle aura permis de recueillir bon nombre de données sur la qualité des résultats pouvant être obtenus ainsi que d'identifier les points forts et les lacunes des concepts mis de l'avant. L'expertise acquise dans le cadre de ces projets pourra être utilisée au bénéfice de nouvelles générations de produits de ce type.

L'approche expérimentée est originale et vient enrichir les connaissances du domaine du fait qu'elle reprend des concepts existants, agencés d'une nouvelle manière. Par exemple, nous utilisons ici un interpréteur, comme le font déjà des technologies telles que ZK ou

OpenLaszlo. Le modèle est toutefois indépendant d'une plateforme spécifique, comme avec MDA. Il se distingue par contre de cette approche du fait qu'un seul modèle est créé et qu'aucune intervention ne peut être effectuée lors des transformations. Les travaux les plus près de ce qui a été réalisé sont ceux autour d'UIML, pouvant aussi s'exécuter en mode interprété. Notre approche utilise un niveau d'abstraction plus élevé qu'UIML, limitant la flexibilité, mais réduisant la taille de la définition d'une application. L'exploitation de Spring et des expressions JSTL pour la liaison du modèle avec l'application distingue aussi l'approche des autres technologies existantes.

D'autres apports, non reliés à la problématique, mais s'inscrivant dans les projets présentés doivent aussi être soulignés. Dans le cas d'ETLTools, une version dynamique du Framework de persistance iBatis a été mise en place, permettant la génération « au besoin » de couches de persistances au moment de l'exécution. Un outil ETL entièrement basé sur Spring et permettant la définition et l'exécution de tâches de transformation sur des bases de données a aussi été développé. Dans le projet de génération d'horaires scolaires, des avancées importantes ont aussi été réalisées au niveau des méthodes de recherche de solutions, adaptées au contexte des polyvalentes québécoises.

Travaux futurs

Un point important n'ayant pas été couvert directement par les prototypes présentés est le support de plusieurs plateformes depuis un même modèle. Le fait que les modèles présentés ne comportent aucun lien avec des plateformes de présentation implique qu'ils pourraient être repris par de nouveaux interpréteurs pour générer tout type de présentation. Une preuve de concept concrète permettrait toutefois de confirmer avec certitude cette affirmation.

Dans l'étude de cas, deux modèles différents, couvrant des besoins différents, ont été présentés. Une fusion des deux modèles pourrait être envisagée afin d'obtenir une solution plus générique et de réduire les efforts de maintenance des modèles. Puisque le modèle utilisé pour les horaires englobe déjà, les composants retrouvés dans ETLTools, seuls le support des données propres à la navigation entre pages ainsi que les concepts de générations de tables dont la structure n'est connue qu'au moment de l'exécution devraient être ajouté.

Un autre créneau pouvant faire l'objet de recherches approfondies est celui des interpréteurs développés. Une meilleure architecture pourrait être définie pour permettre l'ajout facile de nouveaux composants ainsi que le support de nouvelles fonctionnalités propre à de nouveaux types d'applications. La liaison d'interfaces riches avec un modèle objet pourrait aussi être améliorée par l'utilisation de technologies comme AOP pour l'ajout de mécanismes de surveillance en toute transparence sur les objets du modèle, permettant ainsi une diminution de la quantité de données rechargées.

# BIBLIOGRAPHIE


Abrams, M., et J. Helms (2000). User Interface Markup Language UIML Specification. 2

Abran, A., Descharnais, J-M., Oligny, S., St. Pierre, D., Symons, C.R. (2007). The COSMIC Functional Size Measurement Method Version 3.0 Method Overview

Alliance, Open XUL. 2004. «XUL Grand Coding Challenge 2004». En ligne. <http://xul.sourceforge.net/challenge.html>.

Almeida, J. P. A., M. E. Iacob, H. Jonkers et D. Quartel (2006). Model-Driven Development of Context-Aware Services: 213-227 p

Anderson, Tim. 2005. «Hands on with Macromedia Flex 2.0». En ligne. <http://www.regdeveloper.co.uk/2005/12/04/anderson/>.

Arhelger, A., A. Hanson et A. Erwin (2004). New alphaWorks toolkit speeds your Java GUI development. IBM, July 2004

Boshernitsan, M., et M. Downes (2004). Visual Programming Languages: A Survey

Brown, Millward. 2007. «Millward Brown survey, conducted December 2007». En ligne. <http://www.adobe.com/products/player_census/flashplayer/>.

Budgen, D. (2003). Software Design, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

Chen, N., et K. K. Ma (2004). Java's future: Challenge and opportunity, IEEE Educational Activities Department Piscataway, NJ, USA. 6: 19-27 p

Chopra, V. 2005. *Beginning JavaServer Pages*: Wiley.

code.google.com. 2007. En ligne. <http://code.google.com/webtoolkit/>.

commons.apache.org. 2007. En ligne. <http://commons.apache.org/digester>. Consulté le 2 août 2007.

Courtaud, Didier. 2000. «La langage UIML». En ligne. <http://www.aristote.asso.fr/Presentations/UIML/Cours_UIML/all.htm>.

Cover, Robin. 2005. «XML Markup Languages for User Interface Definition». En ligne. <http://xml.coverpages.org/userInterfaceXML.html>.

David, Ryan, DeSerranno, Young. 2005. *Professional WinFX Beta covers "Avalon" Windows presentation foundation and "Indigo" Windows communication foundation*. Indianapolis, Ind.: Wiley. En ligne. <http://www.books24x7.com/marc.asp?bookid=11781 >.

db.apache.org. 2007. En ligne. <http://db.apache.org/torque/>. Consulté le 2 août 2007.

Deakin, Neil (2006). XUL Tutorial En ligne. <http://www.xulplanet.com/tutorials/xultu/>.

Domenig, Marc. 2006. «Rich Internet Applications and AJAX - Selecting the best product». En ligne. <http://www.javalobby.org/articles/ajax-ria-overview/>.

Eich, Brendan. 2006. «JavaScript 2 and the Future of the Web». En ligne. <http://developer.mozilla.org/presentations/xtech2006/javascript/>.

Eisenstein, J., J. Vanderdonckt et A. Puerta (2001). Applying model-based techniques to the development of UIs for mobile computers, ACM Press New York, NY, USA: 69-76 p

Encarta, Microsoft. 2008. «Model definition». En ligne. <http://encarta.msn.com/dictionary_1861630702/model.html>.

Forbrig, P., A. Dittmar, D. Reichart et D. Sinnig (2004). From Models to Interactive Systems Tool Support and XIML

Futrell, R. T., D. F. Shafer et L. Shafer. 2002. *Quality Software Project Management*: Prentice Hall PTR.

Galorath, D. D., et M. W. Evans. 2006. *Software Sizing, Estimation, and Risk Management: When Performance Is Measured Performance Improves*: Auerbach Pub.

Garrett, J. J. (2005). Ajax: A New Approach to Web Applications. 18

Gibbons, P. 2002. . *NET development for Java programmers*: Apress.

Godin, R. 2003. *Systèmes de gestion de bases de données par l'exemple*: Loze-Dion.

Guojie, Jackwind Li. 2005. *Professional Java native interfaces with SWT/JFace*. Indianapolis, IN: Wiley Pub., xx, 508 p. p. En ligne. <http://www.loc.gov/catdir/toc/ecip0421/2004018715.html >.

Helmes, J. (2003). The Relationship of the UIML 3.0 Spec. to other Standards

Horrocks, I. 1999. *Constructing the User Interface with Statecharts*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Hosea, B. 2006. *The Focal Easy Guide to Macromedia Flash 8: For New Users and Professionals:* Focal Press.

ISO13407 (1999). ISO 13407:1999 Human-centred design processes for interactive systems. En ligne. <http://www.userfocus.co.uk/resources/iso9241/iso13407.html>.

Jablonski, S. 2004. *Guide to Web Application and Platform Architectures*: Springer.

Johnson, R., et J. Hoeller. 2004. *Expert One-on-One J2EE Development without EJB*: John Wiley & Sons.

LaszloSystems. 2006. «Laszlo Systems Announces Extension of OpenLaszlo Platform to Support Delivery of Web 2.0 Applications in Browsers Without Flash». En ligne. <http://www.laszlosystems.com/news/pressreleases/77>.

Laudati, P. 2003. *Application Interoperability: Microsoft. NET and J2EE*: Microsoft.

Limbourg, Q., J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins et D. Trevisan (2004). USIXML: A User Interface Description Language for Context-Sensitive User Interfaces: 55-62 p

McCarthy, Philip. 2006. «Ajax for Java developers: Exploring the Google Web Toolkit». En ligne. <http://www.ibm.com/developerworks/opensource/library/j-ajax4/>.

McFarlane, Nigel. 2004. *Rapid application development with Mozilla*. Coll. «Bruce Perens' Open source series». Upper Saddle River, NJ: Prentice Hall/ PTR, xxvii, 770 p. p. En ligne. <Online access http://proquest.safaribooksonline.com/0131423436 >.

Mellor, S. J. 2004. *MDA Distilled: Principles of Model-Driven Architecture*: Addison-Wesley Professional.

Menkhaus, G., et S. Fischmeister (2003). Evaluation of User Interface Transcoding Systems

Mille.ca. 2007. «Historique du projet». En ligne. <http://www.mille.ca/Description-du-projet,27>.

Myers, B., S. E. Hudson et R. Pausch (2000). Past, present, and future of user interface software tools, ACM Press New York, NY, USA. 7: 3-28 p

netbeans.org. 2007. «Designing a Swing GUI in NetBeans IDE». En ligne. <http://www.netbeans.org/kb/60/java/quickstart-gui.html>.

Noda, T., et S. Helwig (2005). Rich Internet Applications

O'Rourke, C. (2004). A Look at Rich Internet Applications

Paulson, L. D. (2005). Building rich web applications with Ajax. 38: 14-17 p

Pawlak, R., L. Seinturier et J. P. Retaillé. 2005. *Foundations of Aop for J2ee Development*: Apress.

Potix, Corp. 2007. «ZK Architecture Overview». En ligne. <http://www.zkoss.org/doc/architecture.dsp>.

Puerta, A., et J. Eisenstein (2002). XIML: A Universal Language for User Interfaces

Puerta, A., et J. Eisenstein (2003). Developing a Multiple User Interface Representation Framework for Industry

Rancourt, René. 2005. «Interface de gestion csadmin». En ligne. <http://aide.mille.ca/index.php/Interface_de_gestion_csadmin>.

Richardson, W. Clay. 2007. «Professional Java, JDK 6 edition». Wiley Pub. En ligne. <http://www.books24x7.com/marc.asp?bookid=17074 >.

Ruiz, F. J. M., J. M. Arteaga et J. Vanderdonckt Transformation of XAML schema for RIA using XSLT & UsiXML

Sottet, J. S., G. Calvary et J. M. Favre (2005). Ingénierie de l'Interaction Homme-Machine Dirigée par les Modèles

Souchon, N., et J. Vanderdonckt. 2003. «A review of xml-compliant user interface description languages». Springer.

Sperko, R. 2003. *Java Persistence for Relational Databases*: Apress Berkeley, Calif.

SpringFramework. 2007. «The Spring Framework - Reference Documentation». En ligne. <http://static.springframework.org/spring/docs/2.0.x/reference/index.html>. Consulté le 2 août 2007.

Stanciulescu, A., Q. Limbourg, J. Vanderdonckt, B. Michotte et F. Montero (2005). A transformational approach for multimodal web user interfaces based on UsiXML, ACM Press New York, NY, USA: 259-266 p

Tate, B. A., et J. Gehtland. 2004. *Better, Faster, Lighter Java*: O'Reilly.

Tate, B. A., et C. Hibbs. 2006. *Ruby on Rails: Up and Running*: O'Reilly Media, Inc.

Tate, B., et J. Gehtland (2005). Spring:: A Developer's Notebook (Developer's Notebook), O'Reilly & Associates, Inc. Sebastopol, CA, USA

tibco.com. 2007. En ligne. <http://www.tibco.com/>. Consulté le 2 août 2007.

UsabilityNet. 2006. «Human centred design processes for interactive systems». En ligne. <http://www.usabilitynet.org/tools/13407stds.htm>.

Vanderburg, Glenn. 2005. «Metaprogramming Ruby, Domain-Specific Languages for Programmers». In *O'Reilly Open Source Convention* (August 1 - 5, 2005). En ligne. <http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf>.

Vanderdonckt, J. (2005). A MDA-Compliant Environment for Developing User Interfaces of Information Systems, Springer

Vanderdonckt, J., Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan et M. Florins (2004). USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces: 19-20 p

VanDoren, Edmond. 1997. «Halstead Complexity Measures». En ligne. <http://www.sei.cmu.edu/activities/str/descriptions/halstead.html>.

VanDoren, Edmond. 2000. «Cyclomatic Complexity». En ligne. <http://www.sei.cmu.edu/activities/str/descriptions/cyclomatic.html>.

VanDoren, Edmond. 2002. «Maintainability Index Technique for Measuring Program Maintainability». En ligne. <http://www.sei.cmu.edu/activities/str/descriptions/mitmpm.html>.

Wahli, U., M. Fielding, G. Mackown, D. Shaddon et G. Hekkenberg „Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java ", IBM Corporation

Web Application Formats Working Group, WAF. 2006. En ligne. <http://www.w3.org/2006/appformats/>. Consulté le 2 août 2007.

Wiegers, K. E. 2003. *Software Requirements*: Microsoft Press Redmond, WA, USA.

wikibooks.org. 2006. «XML-Managing Data Exchange/ The many-to-many relationship». En ligne. <http://en.wikibooks.org/wiki/XML_-_Managing_Data_Exchange/The_many-to-many_relationship>.

Wroblewski, Luke, et Frank Ramirez. 2006. «Web Application Solutions: A Designer's Guide». En ligne. <http://www.lukew.com/resources/WebApplicationSolutions.pdf>.

Zakas, N. C., J. McPeak et J. Fawcett. 2007. *Professional Ajax*: Wrox Press Ltd. Birmingham, UK, UK.

zkoss.org. 2007. En ligne. <http://www.zkoss.org/>.

# ANNEXE 1

# TAILLE FONCTIONNELLE DES PROJETS

| # | Module | Functional Process | Data Group | Movement types | Reuse type | 91<br>CFP Entry | 114<br>CFP Exit | 68<br>CFP Read | 24<br>CFP Write | 297<br>CFP Total | 0<br>Reuse Impact | 297<br>Weighted size |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | Système d'horaire | Importer données | Commande importer | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **2** | Système d'horaire | Importer données | Paramètre d'import | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **3** | Système d'horaire | Importer données | Écoles-grilles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **4** | Système d'horaire | Importer données | Connexion | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **5** | Système d'horaire | Importer données | Classifications | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **6** | Système d'horaire | Importer données | Contrôle d'import | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **7** | Système d'horaire | Importer données | Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **8** | Système d'horaire | Afficher les élèves | Menu Aff. Élèves | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **9** | Système d'horaire | Afficher les élèves | Filtre, élèves | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **10** | Système d'horaire | Afficher les élèves | Affichage des élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **11** | Système d'horaire | Afficher les élèves | Modifier élèves | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **12** | Système d'horaire | Afficher les élèves | Choix de cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **13** | Système d'horaire | Afficher les élèves | Filtre, choix de cours | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **14** | Système d'horaire | Afficher les élèves | Lister périodes possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **15** | Système d'horaire | Afficher les élèves | Lister groupes de l'école | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **16** | Système d'horaire | Afficher les élèves | Modifier choix de cours | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **17** | Système d'horaire | Afficher les élèves | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **18** | Système d'horaire | Afficher les élèves | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **19** | Système d'horaire | Afficher les élèves | Restrictions | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **20** | Système d'horaire | Afficher les élèves | Intervenants | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **21** | Système d'horaire | Afficher les élèves | Compteur nb. élèves | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **22** | Système d'horaire | Afficher les élèves | Compteur nb. Périodes | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **23** | Système d'horaire | Afficher les groupes | Menu Aff. Groupes | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **24** | Système d'horaire | Afficher les groupes | Filtre, groupes | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **25** | Système d'horaire | Afficher les groupes | Affichage des groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **26** | Système d'horaire | Afficher les groupes | Modifier groupes | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **27** | Système d'horaire | Afficher les groupes | Jumelage / Défaire jumelage | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **28** | Système d'horaire | Afficher les groupes | Superbloc / Défaire superbloc | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **29** | Système d'horaire | Afficher les groupes | Fixer / Dé-fixer l'horaire | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **30** | Système d'horaire | Afficher les groupes | Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **31** | Système d'horaire | Afficher les groupes | Jumelage | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **32** | Système d'horaire | Afficher les groupes | Lister groupes de l'école | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **33** | Système d'horaire | Afficher les groupes | Modifier jumelage | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **34** | Système d'horaire | Afficher les groupes | Superblocs | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **35** | Système | Afficher les | Lister groupes | Table dynamique : Read & | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | d'horaire | groupes | de l'école | Exit | | | | | | | | |
| **36** | Système d'horaire | Afficher les groupes | Modifier superblocs | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **37** | Système d'horaire | Afficher les groupes | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **38** | Système d'horaire | Afficher les groupes | Modifier l'horaire | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **39** | Système d'horaire | Afficher les groupes | Modifier l'horaire / afficher formulaire | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **40** | Système d'horaire | Afficher les groupes | Modifier l'horaire / lister locaux disponibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **41** | Système d'horaire | Afficher les groupes | Modifier l'horaire / appliquer local sélectionné | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **42** | Système d'horaire | Afficher les groupes | Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **43** | Système d'horaire | Afficher les groupes | Grille horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **44** | Système d'horaire | Afficher les groupes | Restrictions | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **45** | Système d'horaire | Afficher les groupes | Modifier restrictions | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **46** | Système d'horaire | Afficher les cours | Menu Aff. Cours | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **47** | Système d'horaire | Afficher les cours | Filtre, cours | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **48** | Système d'horaire | Afficher les cours | Affichage des cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **49** | Système d'horaire | Afficher les cours | Modifier cours | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **50** | Système d'horaire | Afficher les cours | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **51** | Système d'horaire | Afficher les cours | Modifier groupe | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **52** | Système d'horaire | Afficher les cours | Ajouter groupe | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **53** | Système d'horaire | Afficher les cours | Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **54** | Système d'horaire | Afficher les cours | Élèves / autres groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **55** | Système d'horaire | Afficher les cours | Locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **56** | Système d'horaire | Afficher les cours | Locaux / Paramètre, type de sélection | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **57** | Système d'horaire | Afficher les cours | Modifier locaux | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **58** | Système d'horaire | Afficher les cours | Locaux / Lister locaux école | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **59** | Système d'horaire | Afficher les cours | Locaux / Charger valeurs défaut | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **60** | Système d'horaire | Afficher les cours | Locaux / Charger préférences profs | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **61** | Système d'horaire | Afficher les cours | Locaux / Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **62** | Système d'horaire | Afficher les cours | Intervenants possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **63** | Système d'horaire | Afficher les intervenants | Menu Aff. Intervenants | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **64** | Système d'horaire | Afficher les intervenants | Filtre, intervenants | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **65** | Système d'horaire | Afficher les intervenants | Affichage des intervenants | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **66** | Système d'horaire | Afficher les intervenants | Tâches | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **67** | Système d'horaire | Afficher les intervenants | Modifier tâche | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **68** | Système d'horaire | Afficher les intervenants | Lister tâches possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **69** | Système d'horaire | Afficher les intervenants | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **70** | Système d'horaire | Afficher les intervenants | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **71** | Système d'horaire | Afficher les intervenants | Groupes / Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **72** | Système d'horaire | Afficher les intervenants | Préférences locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **73** | Système d'horaire | Afficher les intervenants | Préférences locaux / cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **74** | Système d'horaire | Afficher les intervenants | Lister cours possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **75** | Système d'horaire | Afficher les intervenants | Lister locaux possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **76** | Système d'horaire | Afficher les intervenants | Modifier locaux | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **77** | Système d'horaire | Afficher les intervenants | Changer ordre locaux | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **78** | Système d'horaire | Afficher les intervenants | Périodes possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **79** | Système d'horaire | Afficher les intervenants | Modifier périodes possibles | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **80** | Système d'horaire | Afficher les intervenants | Actualiser la grille | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **81** | Système d'horaire | Afficher les locaux | Menu Aff. Locaux | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **82** | Système d'horaire | Afficher les locaux | Filtre, locaux | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **83** | Système d'horaire | Afficher les locaux | Affichage des locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **84** | Système d'horaire | Afficher les locaux | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **85** | Système d'horaire | Afficher les locaux | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **86** | Système d'horaire | Afficher les locaux | Groupes / élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **87** | Système d'horaire | Afficher les locaux | Cours possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **88** | Système d'horaire | Afficher les locaux | Périodes non disponibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **89** | Système d'horaire | Afficher les locaux | Modifier périodes non | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | disponibles | | | | | | | | | |
| **90** | Système d'horaire | Afficher les locaux | Actualiser la grille | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **91** | Système d'horaire | Afficher les patrons | Menu Aff. Patrons | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **92** | Système d'horaire | Afficher les patrons | Filtre, patrons | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **93** | Système d'horaire | Afficher les patrons | Affichage des patrons | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **94** | Système d'horaire | Afficher les patrons | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **95** | Système d'horaire | Afficher les patrons | Modifier groupes | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **96** | Système d'horaire | Afficher les patrons | Lister groupes possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **97** | Système d'horaire | Super blocs | Menu Aff. Super blocs | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **98** | Système d'horaire | Super blocs | Affichage des superblocs | Table dynamique : Tout | New | 1 | 1 | 1 | 1 | 4 | 0 | 4 |
| **99** | Système d'horaire | Super blocs | Affichage des groupes | Table dynamique : Tout | New | 1 | 1 | 1 | 1 | 4 | 0 | 4 |
| **100** | Système d'horaire | Contraintes à ignorer | Menu Aff. Contraintes | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **101** | Système d'horaire | Contraintes à ignorer | Affichage des contraintes locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **102** | Système d'horaire | Contraintes à ignorer | Éditer contraintes locaux | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **103** | Système d'horaire | Contraintes à ignorer | Affichage des contraintes jointures | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **104** | Système d'horaire | Contraintes à ignorer | Éditer contraintes jointures | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **105** | Système d'horaire | Rétro-ingénierie, tâches des intervenants | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **106** | Système d'horaire | Rétro-ingénierie, tâches des | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | intervenants | | | | | | | | | | |
| **107** | Système d'horaire | Rétro-ingénierie, tâches des intervenants | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **108** | Système d'horaire | Rétro-ingénierie, vocation locaux | Menu Aff. Wizard | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **109** | Système d'horaire | Rétro-ingénierie, vocation locaux | Paramètre d'import | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **110** | Système d'horaire | Rétro-ingénierie, vocation locaux | Connexion | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **111** | Système d'horaire | Rétro-ingénierie, vocation locaux | Paramètre rétro-ingénierie | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **112** | Système d'horaire | Rétro-ingénierie, vocation locaux | Contrôle rétro-ingénierie | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **113** | Système d'horaire | Rétro-ingénierie, vocation locaux | Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **114** | Système d'horaire | Rétro-ingénierie, préférences locaux / intervenants | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **115** | Système d'horaire | Rétro-ingénierie, préférences locaux / intervenants | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **116** | Système d'horaire | Rétro-ingénierie, préférences locaux / intervenants | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **117** | Système d'horaire | Rétro-ingénierie, restrictions des élèves | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **118** | Système d'horaire | Rétro-ingénierie, restrictions des élèves | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **119** | Système d'horaire | Rétro-ingénierie, restrictions des élèves | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **120** | Système d'horaire | Rapport / validation des contraintes | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **121** | Système d'horaire | Rapport / validation des contraintes | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **122** | Système d'horaire | Rapport / validation des contraintes | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **123** | Système d'horaire | Rapport / validation des résultats | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **124** | Système d'horaire | Rapport / validation des résultats | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **125** | Système d'horaire | Rapport / validation des résultats | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **126** | Système d'horaire | Rapport / balancement groupes et tâches interv. | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **127** | Système d'horaire | Rapport / balancement groupes et tâches interv. | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **128** | Système d'horaire | Rapport / balancement groupes et tâches interv. | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **129** | Système d'horaire | Rapport / faisabilité | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **130** | Système d'horaire | Rapport / faisabilité | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **131** | Système d'horaire | Rapport / faisabilité | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **132** | Système d'horaire | Initialisation des choix de cours des élèves | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **133** | Système d'horaire | Initialisation des choix de cours des élèves | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **134** | Système d'horaire | Initialisation des choix de cours des élèves | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **135** | Système d'horaire | Initialisation de la grille | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **136** | Système d'horaire | Initialisation de la grille | Lancer commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **137** | Système d'horaire | Générer horaire maître | Menu Aff. commande | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **138** | Système d'horaire | Générer horaire maître | Saisie des paramètres | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **139** | Système d'horaire | Générer horaire maître | Lancement de la génération | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **140** | Système d'horaire | Générer horaire maître | Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **141** | Système d'horaire | Solutionneur | Afficher rapport | Affichage du curseur | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **142** | Système d'horaire | Solutionneur | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **143** | Système d'horaire | Solutionneur | Groupes / Analyse | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **144** | Système d'horaire | Solutionneur | Intervenants | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **145** | Système d'horaire | Solutionneur | Intervenants / Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **146** | Système d'horaire | Solutionneur | Cours / analyse croisée 1 | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **147** | Système d'horaire | Solutionneur | Cours / analyse croisée 1 / Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **148** | Système d'horaire | Solutionneur | Cours / analyse croisée 1 / Élèves / cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **149** | Système d'horaire | Solutionneur | Cours / analyse croisée 2 | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **150** | Système d'horaire | Solutionneur | Cours / analyse croisée 2 / Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **151** | Système d'horaire | Solutionneur | Cours / analyse croisée 2 / Élèves / cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **152** | Système d'horaire | Solutionneur | Cours / analyse croisée 3 | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **153** | Système d'horaire | Solutionneur | Cours / analyse croisée 3 / Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **154** | Système d'horaire | Solutionneur | Cours / analyse croisée 3 / Élèves / cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **155** | Système d'horaire | Solutionneur | Élèves en rejet | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **156** | Système d'horaire | Solutionneur | Élèves en rejet / cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **157** | Système d'horaire | Solutionneur | Charger les données | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **158** | Système d'horaire | Solutionneur | Messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **159** | Système d'horaire | Paramètres de la grille | Menu Aff. Paramètres | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **160** | Système d'horaire | Paramètres de la grille | Paramètres grille | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **161** | Système d'horaire | Paramètres de la grille | Paramètres jointures | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **162** | Système d'horaire | Paramètres de la grille | Paramètres codes difficultés | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **163** | Système d'horaire | Application | Ouvrir | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **164** | Système d'horaire | Application | Enregistrer | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **165** | Système d'horaire | Application | Enregistrer / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **166** | Système d'horaire | Application | Enregistrer / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **167** | Système d'horaire | Application | Enregistrer sous | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **168** | Système d'horaire | Application | Enregistrer sous / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **169** | Système d'horaire | Application | Enregistrer sous / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **170** | Système d'horaire | Application | Exporter horaire / TXT | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **171** | Système d'horaire | Application | Enregistrer sous / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **172** | Système d'horaire | Application | Enregistrer sous / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **173** | Système d'horaire | Application | Exporter horaire / SQL | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **174** | Système d'horaire | Application | Exporter horaire / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **175** | Système d'horaire | Application | Enregistrer sous / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **176** | Système d'horaire | Application | Exporter groupes / SQL | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **177** | Système d'horaire | Application | Exporter groupes / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **178** | Système d'horaire | Application | Exporter groupes / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **179** | Système d'horaire | Application | Quitter | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**Tableau 10.1** Taille fonctionne de l'application de génération des horaires, nouvelle version

| # | Module | Functional Process | Data Group | Movement types | Reuse type | 30<br>CFP Entry | 26<br>CFP Exit | 22<br>CFP Read | 11<br>CFP Write | 89<br>CFP Total | 0<br>Reuse Impact | 89<br>Weighted size |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Application orig. | Importer données | Commande importer | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 2 | Application orig. | Importer données | Paramètre d'import | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 3 | Application orig. | Afficher les élèves | Menu Aff. Élèves | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 4 | Application orig. | Afficher les élèves | Filtre, élèves | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 5 | Application orig. | Afficher les élèves | Affichage des élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| 6 | Application orig. | Afficher les élèves | Modifier élèves | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| 7 | Application orig. | Afficher les élèves | Choix de cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| 8 | Application orig. | Afficher les élèves | Modifier choix de cours | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| 9 | Application orig. | Afficher les élèves | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| 10 | Application orig. | Afficher les élèves | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| 11 | Application orig. | Afficher les élèves | Restrictions | Table dynamique : Tout | New | 1 | 1 | 1 | 1 | 4 | 0 | 4 |
| 12 | Application orig. | Afficher les groupes | Menu Aff. Groupes | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 13 | Application orig. | Afficher les groupes | Filtre, groupes | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 14 | Application orig. | Afficher les groupes | Affichage des groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| 15 | Application orig. | Afficher les groupes | Modifier groupes | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| 16 | Application orig. | Afficher les groupes | Jumelage | Table dynamique : Tout | New | 1 | 1 | 1 | 1 | 4 | 0 | 4 |
| 17 | Application orig. | Afficher les | Lister groupes | Table dynamique : | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | groupes | de l'école | Read & Exit | | | | | | | | |
| **18** | Application orig. | Afficher les groupes | Modifier jumelage | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **19** | Application orig. | Afficher les groupes | Superblocs | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **20** | Application orig. | Afficher les groupes | Lister groupes de l'école | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **21** | Application orig. | Afficher les groupes | Modifier superblocs | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **22** | Application orig. | Afficher les groupes | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **23** | Application orig. | Afficher les groupes | Élèves | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **24** | Application orig. | Afficher les groupes | Grille horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **25** | Application orig. | Afficher les cours | Menu Aff. Cours | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **26** | Application orig. | Afficher les cours | Filtre, cours | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **27** | Application orig. | Afficher les cours | Affichage des cours | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **28** | Application orig. | Afficher les intervenants | Menu Aff. Intervenants | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **29** | Application orig. | Afficher les intervenants | Filtre, intervenants | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **30** | Application orig. | Afficher les intervenants | Affichage des intervenants | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **31** | Application orig. | Afficher les intervenants | Modifier intervenants | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **32** | Application orig. | Afficher les intervenants | Tâches | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **33** | Application orig. | Afficher les intervenants | Modifier tâche | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **34** | Application orig. | Afficher les intervenants | Horaire | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **35** | Application orig. | Afficher les intervenants | Groupes | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **36** | Application orig. | Afficher les intervenants | Préférences locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **37** | Application orig. | Afficher les intervenants | Périodes possibles | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **38** | Application orig. | Afficher les locaux | Menu Aff. Locaux | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **39** | Application orig. | Afficher les locaux | Filtre, locaux | Saisie d'un filtre ou paramètre | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **40** | Application orig. | Afficher les locaux | Affichage des locaux | Table dynamique : Read & Exit | New | 0 | 1 | 1 | 0 | 2 | 0 | 2 |
| **41** | Application orig. | Afficher les locaux | Modifier locaux | Table dynamique : Input & Write | New | 1 | 0 | 0 | 1 | 2 | 0 | 2 |
| **42** | Application orig. | Paramètres de la grille | Menu Aff. Paramètres | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| | Application orig. | Paramètres de la grille | Édition, paramètres | Table dynamique : Tout | New | 1 | 1 | 1 | 1 | 4 | 0 | 4 |
| **44** | Application orig. | Application | Ouvrir | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **45** | Application orig. | Application | Enregistrer | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **46** | Application orig. | Application | Enregistrer / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **47** | Application orig. | Application | Enregistrer / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **48** | Application orig. | Application | Enregistrer sous | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **49** | Application orig. | Application | Enregistrer sous / commande | Appel paramétré et code retour | New | 1 | 1 | 0 | 0 | 2 | 0 | 2 |
| **50** | Application orig. | Application | Enregistrer sous / messages | Message(s) simple(s) | New | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **51** | Application orig. | Application | Quitter | Déclencheur de l'action | New | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**Tableau 10.2** Taille fonctionnelle, version originale, application de génération d'horaires scolaires

# ANNEXE 2

# DÉTAILS DE MISE EN ŒUVRE

## 1. Définition XML, Tableau de bord du gestionnaire

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
xmlns:lang="http://www.springframework.org/schema/lang"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd
http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

  <bean id="context.mapping" class="ca.crim.etltools.editor.model.PageContext">
    <property name="keys" value="id_mapping,etiquette" />
  </bean>

  <bean id="context.ecole" class="ca.crim.etltools.editor.model.PageContext">
    <property name="keys" value="id_ecole,id_ecole_annee,annee,etiquette" />
  </bean>

  <bean id="page.root" class="ca.crim.etltools.editor.model.PageModel">
    <property name="links">
      <list>
        <ref bean="page.connection_source" />
        <ref bean="page.ecoles" />
        <ref bean="page.annee_ecoles" />
        <ref bean="page.dimension_eleve" />
        <ref bean="page.dimension_eleve_annee_ecole" />
        <ref bean="page.dimension_eleve_reussite" />
        <ref bean="page.dimension_eleve_absence" />
        <ref bean="page.importation" />
        <ref bean="page.generation.tablefait" />
      </list>
    </property>
  </bean>

  <bean id="page.importation" class="ca.crim.etltools.editor.model.PageModel">
    <property name="components">
      <list>
        <bean class="ca.crim.etltools.editor.model.QuartzJobComponent">
          <property name="title" value="ETL MAIN JOB" />
          <property name="trigger" ref="etlJobTrigger" />
          <property name="jobDetail" ref="etlJobTrigger.jobDetail" />
```

```
        <property name="etlJob" ref="job.etl" />
      </bean>
    </list>
  </property>
</bean>

<bean id="page.connection_source" class="ca.crim.etltools.editor.model.PageModel">
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.TableComponent">
        <property name="table" ref="table.tbgest.connection_source" />
        <property name="passwords" value="conn_password=true" />
        <property name="query" value="id_connection &gt; 0 " />
      </bean>
    </list>
  </property>
</bean>

<bean id="page.horaire_ecole" class="ca.crim.etltools.editor.model.PageModel">
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.TableComponent">
        <property name="table" ref="table.tbgest.horaire_ecole" />
        <property name="query" value="id_horaire_ecole &gt; 0 " />
      </bean>
    </list>
  </property>
</bean>

<bean id="page.dimension_aire_desserte"
class="ca.crim.etltools.editor.model.PageModel">
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.EtlTaskComponent">
        <property name="task" ref="task.aggr.dimension_ecole.aire_desserte" />
        <property name="valueLists">
          <map>
            <entry key="id_connection">
              <bean class="ca.crim.etltools.editor.model.ValueList">
                <property name="query" value="id_connection &gt; 0,conn_key = gpi"
/>
                <property name="table" ref="table.tbgest.connection_source" />
                <property name="key" value="id_connection" />
                <property name="labels" value="conn_name" />
              </bean>
            </entry>
          </map>
        </property>
      </bean>
      <bean class="ca.crim.etltools.editor.model.TableComponent">
        <property name="table" ref="table.tbgest.aire_desserte" />
        <property name="query" value="id_aire_desserte &gt; 0 " />
      </bean>
    </list>
  </property>
</bean>

<bean id="page.vocation_speciale" class="ca.crim.etltools.editor.model.PageModel">
  <property name="components">
    <list>
      <bean class="ca.crim.etltools.editor.model.TableComponent">
        <property name="table" ref="table.tbgest.vocation_speciale" />
        <property name="query" value="id_vocation_speciale &gt; 0 " />
      </bean>
    </list>
```

```
    </property>
  </bean>

  <bean id="page.secteur_ecole" class="ca.crim.etltools.editor.model.PageModel">
    <property name="components">
      <list>
        <bean class="ca.crim.etltools.editor.model.EtlTaskComponent">
          <property name="task" ref="task.import.dimension_ecole.secteur_ecole" />
        </bean>
        <bean class="ca.crim.etltools.editor.model.TableComponent">
          <property name="table" ref="table.tbgest.secteur_ecole" />
          <property name="query" value="id_secteur_ecole &gt; 0 " />
        </bean>
      </list>
    </property>
  </bean>

...

</beans>
```

2. Contrôleur pour la génération des pages, Tableau de bord du gestionnaire

```
package ca.crim.etltools.editor.web.controller;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.multiaction.MultiActionController;

import ca.crim.etltools.connection.ConnectionManager;
import ca.crim.etltools.editor.model.Component;
import ca.crim.etltools.editor.model.PageModel;
import ca.crim.etltools.editor.web.beans.ComponentView;
import ca.crim.etltools.editor.web.beans.LinkInfo;
import ca.crim.etltools.editor.web.beans.PageView;
import ca.crim.etltools.table.GenericRow;

/**
 * Main controller for page rendering
 *
 * @author bondst
 *
 */
public class PageController extends MultiActionController implements Constants {

    protected final Log logger = LogFactory.getLog(getClass());
```

```
    private ComponentManager componentManager;

    private ConnectionManager connectionManager;

    /**
     * Change current page, go to pageId from page link
     *
     * @param request
     * @param response
     * @return
     * @throws Exception
     */
    public ModelAndView doGoto(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpSession session = request.getSession(true);
        String pageId = request.getParameter(PARAM_PAGE_ID);
        PageView currentPage =
ControllerHelper.getCurrentPage(getApplicationContext(), session);

        if (currentPage.getLinks() != null) {
            boolean validLink = false;
            for (LinkInfo linkInfo : currentPage.getLinks()) {
                if (linkInfo.getPageId().equals(pageId)) {
                    validLink = true;
                }
            }
            if (validLink) {
                PageModel pageModel =
ControllerHelper.getPageModel(getApplicationContext(), pageId);
                PageView gotoPage =
ControllerHelper.createPageView(currentPage.getEnv(), pageModel, null);
                List<PageView> navigation =
ControllerHelper.getNavigationList(getApplicationContext(), session);
                navigation.add(gotoPage);
                session.setAttribute(SESSION_CURRENT_PAGE, gotoPage);
            } else {
                throw new Exception("Page not found for goto: " + pageId);
            }
        }
        return doMain(request, response);
    }

    /**
     * Change current page to a previous page from navigation
     *
     * @param request
     * @param response
     * @return
     * @throws Exception
     */
    public ModelAndView doBack(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpSession session = request.getSession(true);
        int idx = Integer.parseInt(request.getParameter(PARAM_IDX));
        List navigation = ControllerHelper.getNavigationList(getApplicationContext(),
session);

        int remSize = navigation.size();
        for (int i = idx + 1; i < remSize; i++) {
            navigation.remove(navigation.size() - 1);
        }
        PageView pageView = (PageView) navigation.get(idx);
        session.setAttribute(SESSION_CURRENT_PAGE, pageView);

        return doMain(request, response);
```

```
    }

    /**
     * Show a Page view
     *
     * @param request
     * @param response
     * @return
     * @throws Exception
     */
    public ModelAndView doMain(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpSession session = request.getSession(true);

        PageView currentPage =
ControllerHelper.getCurrentPage(getApplicationContext(), session);
        List<PageView> navigation =
ControllerHelper.getNavigationList(getApplicationContext(), session);
        GenericRow env = ControllerHelper.compileEnv(navigation);

        // Setup ContextInfo
        ControllerHelper.getCurrentContextInfo(getApplicationContext(), session, env);

        Map<String, Object> model = new HashMap<String, Object>();
        String errorMsg = null;
        List<ComponentView> componentList = null;
        try {
            // Init for connections
            connectionManager.setCurrentEnv(env);

            // Init components
            PageModel pageModel =
ControllerHelper.getPageModel(getApplicationContext(), currentPage.getId());
            componentList = new ArrayList<ComponentView>();
            if (pageModel.getComponents() != null) {
                for (int i = 0; i < pageModel.getComponents().size(); i++) {
                    String compId = pageModel.getId() + "." + String.valueOf(i);
                    Component comp = (Component) pageModel.getComponents().get(i);
                    componentList.add(componentManager.initViewBean(session, comp,
compId, model, env));
                }
            }

        } catch (Exception e) {
            errorMsg = e.getMessage();
            logger.error(e.getMessage(), e);
        } finally {
            // Close connections
            connectionManager.releaseAllConnections();
        }

        if (errorMsg != null) {
            model.put(MODEL_ERROR, errorMsg);
        } else if (request.getAttribute(MODEL_ERROR) != null) {
            model.put(MODEL_ERROR, request.getAttribute(MODEL_ERROR));
        }
        model.put(MODEL_COMPONENTS, componentList);

        return new ModelAndView(PAGE_VIEW, model);
    }

    public ConnectionManager getConnectionManager() {
        return connectionManager;
    }
```

```
    public void setConnectionManager(ConnectionManager connectionManager) {
        this.connectionManager = connectionManager;
    }

    public ComponentManager getComponentManager() {
        return componentManager;
    }

    public void setComponentManager(ComponentManager componentManager) {
        this.componentManager = componentManager;
    }

}
```

## 3. Définition XML des contrôleurs, horaires scolaires

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd">

  <import resource="swingBuilder.xml" />
  <import resource="commands.xml" />
  <import resource="menus.xml" />
  <import resource="panel_eleves.xml" />
  <import resource="panel_groupes.xml" />
  <import resource="panel_cours.xml" />
  <import resource="panel_interv.xml" />
  <import resource="panel_locaux.xml" />
  <import resource="panel_patrons.xml" />
  <import resource="panel_grilleparams.xml" />
  <import resource="panel_import.xml" />
  <import resource="panel_retrolocaux.xml" />
  <import resource="panel_solverresult.xml" />
  <import resource="panel_jointuresets.xml" />
  <import resource="panel_solverparams.xml" />
  <!--
    <import resource="panel_contraintes.xml" />
  -->
  <import resource="panel_superblocs.xml" />

  <bean id="propertyConfigurerImpl"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:/context/application.properties" />
  </bean>

  <bean id="preprocessorFactoryImpl"
class="ca.crim.horcs.solver.factory.PreprocessorLocalFactory">
    <property name="className" value="ca.crim.horcs.solver.engine.PreprocessorImpl" />
  </bean>

  <bean id="faisabilityCheckFactoryImpl"
class="ca.crim.horcs.solver.factory.FaisabilityCheckLocalFactory">
```

```
    <property name="className"
value="ca.crim.horcs.solver.engine.FaisabilityCheckImpl" />
  </bean>

  <bean id="solverModelBuilderImpl" class="ca.crim.horcs.solver.SolverModelBuilder">
    <property name="preprocessorFactory" ref="preprocessorFactoryImpl" />
  </bean>


  <!--
    Application
  -->
  <bean id="application" class="ca.crim.cdmpf.model.application.TDIApplication">
    <property name="title"
      value="Horaires {context['grille'].eco}{context['grille'] != null ? ' - ' :
''}{context['grille'].annee}{context['grilleFileName'] != null ? ' - ' :
''}{context['grilleFileName']}" />
    <property name="menus" ref="application.menus" />
    <property name="closeCommand" ref="exitCommand" />
    <property name="context">
      <map>
        <entry key="show_deleted" value="false" />
      </map>
    </property>
    <property name="controllers">
      <map>
        <entry key="main">
          <bean class="ca.crim.horcs.app.controller.MainController">
            <property name="filePersistence">
              <bean class="ca.crim.horcs.persistence.xml.XMLFilePersistence" />
            </property>
            <property name="faisabilityCheckFactory" ref="faisabilityCheckFactoryImpl"
/>
            <property name="solverModelBuilder" ref="solverModelBuilderImpl" />
          </bean>
        </entry>
        <entry key="import">
          <bean class="ca.crim.horcs.app.controller.GPIImportController">
            <property name="importBeanName" value="importBean" />
            <property name="dbImport">
              <bean class="ca.crim.horcs.persistence.gpi.GPIDBImport" />
            </property>
          </bean>
        </entry>
        <entry key="retroLocaux">
          <bean class="ca.crim.horcs.app.controller.GPIRetroLocauxController">
            <property name="importBeanName" value="retroLocauxBean" />
            <property name="dbImport">
              <bean class="ca.crim.horcs.persistence.gpi.GPIDBImport" />
            </property>
          </bean>
        </entry>
        <entry key="params">
          <bean class="ca.crim.horcs.app.controller.ParamsController">
            <property name="paramsBeanName" value="paramsBean" />
          </bean>
        </entry>
        <entry key="locauxPrefs">
          <bean class="ca.crim.horcs.app.controller.LocauxPrefsController" />
        </entry>
        <entry key="solverResults">
          <bean class="ca.crim.horcs.app.controller.SolverResultsController" />
        </entry>
        <entry key="solver">
```

```
        ...

        </entry>
        <entry key="postAssign">
          <bean class="ca.crim.horcs.app.controller.PostAssignationController" />
        </entry>
      </map>
    </property>
    <property name="childrens">
      <list>
        <ref bean="tab.eleves" />
        <ref bean="tab.groupes" />
        <ref bean="tab.cours" />
        <ref bean="tab.interv" />
        <ref bean="tab.locaux" />
        <ref bean="tab.patrons" />
      </list>
    </property>
  </bean>


</beans>
```

## 4. Définition XML d'un écran, horaires scolaires

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd">

  <!--
    Tab Locaux
  -->
  <bean id="tab.locaux" class="ca.crim.cdmpf.model.container.SplitterContainer">
    <property name="label" value="Locaux" />
    <property name="orientation" value="vertical" />
    <property name="dimensions" value=",400" />
    <property name="childrens">
      <list>
        <ref bean="tab.locaux.grid" />
        <bean class="ca.crim.cdmpf.model.container.TabContainer">
          <property name="childrens">
            <list>
              <ref bean="tab.locaux.horaire" />
              <ref bean="tab.locaux.groupes" />
              <ref bean="tab.locaux.cours.grid" />
              <ref bean="tab.locaux.exclusions" />
            </list>
          </property>
        </bean>
      </list>
    </property>
  </bean>
```

```
  <!--
    Liste des locaux
  -->
  <bean id="tab.locaux.grid" class="ca.crim.cdmpf.model.component.DataGrid">
    <property name="readOnly" value="false" />
    <property name="itemsBinding" value="{context['grille'].allLocalList}" />
    <property name="rowVisible" value="{context['show_deleted'] == true or (not
item.deleted)}" />
    <property name="rowStriked" value="{item.deleted}" />
    <property name="multiSelect" value="true" />
    <property name="rowCommands">
      <list>
        <bean class="ca.crim.cdmpf.model.component.DataGridCommand">
          <property name="label" value="Exclure" />
          <property name="enabled" value="{not item.deleted}" />
          <property name="command">
            <bean class="ca.crim.cdmpf.model.command.IterativeCallCommand">
              <property name="itemsBinding" value="{selectedItems}" />
              <property name="command">
                <bean class="ca.crim.cdmpf.model.command.CallCommand">
                  <property name="binding" value="{context['grille']}" />
                  <property name="method" value="excludeLocal" />
                  <property name="args" value="{item}" />
                </bean>
              </property>
              <property name="fireChangeAfterExecution"
value="context['grille'].allLocalList" />
            </bean>
          </property>
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridCommand">
          <property name="label" value="Restaurer" />
          <property name="enabled" value="{item.deleted}" />
          <property name="command">
            <bean class="ca.crim.cdmpf.model.command.IterativeCallCommand">
              <property name="itemsBinding" value="{selectedItems}" />
              <property name="command">
                <bean class="ca.crim.cdmpf.model.command.CallCommand">
                  <property name="binding" value="{context['grille']}" />
                  <property name="method" value="restaureLocal" />
                  <property name="args" value="{item}" />
                </bean>
              </property>
              <property name="fireChangeAfterExecution"
value="context['grille'].allLocalList" />
            </bean>
          </property>
        </bean>
      </list>
    </property>
    <property name="columns">
      <list>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Local" />
          <property name="binding" value="{item.identifiant}" />
          <property name="editable" value="{false}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Descr." />
          <property name="binding" value="{item.descr}" />
          <property name="editable" value="{false}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
```

```
          <property name="label" value="Contenu" />
          <property name="binding" value="{item.contenu}" />
          <property name="editable" value="{false}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Restr.classes" />
          <property name="binding" value="{item.restrictionClasses}" />
          <property name="editable" value="{true}" />
          <property name="multiValues" value="true" />
          <property name="valueListBinding" value="{context['grille'].classeList}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="FORCER restr.classes" />
          <property name="binding" value="{item.forceRestrictions}" />
          <property name="editable" value="{true}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Restr.champ.ens." />
          <property name="binding" value="{item.champEns}" />
          <property name="editable" value="{true}" />
          <property name="multiValues" value="true" />
          <property name="valueListBinding" value="{context['grille'].champEns}" />
          <property name="textAttribute" value="code" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Capacité" />
          <property name="binding" value="{item.capacite}" />
          <property name="editable" value="{true}" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Pér.Non Dispo." />
          <property name="binding" value="{item.codesArrangExclusionText}" />
          <property name="editable" value="{false}" />
          <property name="sortable" value="true" />
        </bean>
      </list>
    </property>
  </bean>


  <!--
    Horaire du groupe
  -->
  <bean id="tab.locaux.horaire" class="ca.crim.horcs.app.cdriven.model.Horaire">
    <property name="label" value="Grille" />
    <property name="layoutBinding" value="{context['grille'].grilleLayout}" />
    <property name="groupesBinding"
value="{components['tab.locaux.grid'].selectedItem.groupes}" />
  </bean>

  <!--
    Groupes
  -->
  <bean id="tab.locaux.groupes"
class="ca.crim.cdmpf.model.container.SplitterContainer">
    <property name="label" value="Groupes" />
    <property name="orientation" value="horizontal" />
    <property name="dimensions" value=",200" />
    <property name="childrens">
      <list>
```

```
        <bean id="tab.locaux.groupes.grid"
class="ca.crim.cdmpf.model.component.DataGrid">
          <property name="itemsBinding"
value="{components['tab.locaux.grid'].selectedItem.groupes}" />
          <property name="columns">
            <list>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Groupe" />
                <property name="binding" value="{item.key}" />
                <property name="editable" value="{false}" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Semestre" />
                <property name="binding" value="{item.semestre}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Restr. Sexe" />
                <property name="binding" value="{item.sexe}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Nb.Périodes" />
                <property name="binding" value="{item.nbPeriodes}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Nb.Élèves" />
                <property name="binding" value="{item.nbEleves}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Local" />
                <property name="binding" value="{item.local}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Intervenant" />
                <property name="binding"
                  value="{item.interv.codeInterv} ({item.interv.nom},
{item.interv.prenom})" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
              <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
                <property name="label" value="Horaire" />
                <property name="binding" value="{item.codesArrangText}" />
                <property name="editable" value="false" />
                <property name="sortable" value="true" />
              </bean>
            </list>
          </property>
        </bean>
      </list>
    </property>
  </bean>


  <!--
```

```
    Cours possibles
  -->
  <bean id="tab.locaux.cours.grid" class="ca.crim.cdmpf.model.component.DataGrid">
    <property name="label" value="Cours possibles" />
    <property name="itemsCallBinding">
      <bean class="ca.crim.cdmpf.model.command.CallCommand">
        <property name="binding" value="{controllers['main']}" />
        <property name="method" value="getPossibleCours" />
        <property name="args"
value="{context['grille']},{components['tab.locaux.grid'].selectedItem}" />
      </bean>
    </property>
    <property name="columns">
      <list>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Sigle" />
          <property name="binding" value="{item.sigle}" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Cours" />
          <property name="binding" value="{item.desc}" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Nb. Périodes" />
          <property name="binding" value="{item.nbPeriodes}" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Champ.Ens." />
          <property name="binding" value="{item.champEns}" />
          <property name="textAttribute" value="code" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Max.Elv/Grp" />
          <property name="binding" value="{item.overrideNbElevesGroupeMaxPrevu}" />
          <property name="onNullValue" value="{item.nbElevesGroupeMaxPrevu}" />
          <property name="cellBold" value="{item.overrideNbElevesGroupeMaxPrevu !=
null}" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
        <bean class="ca.crim.cdmpf.model.component.DataGridColumn">
          <property name="label" value="Nb.Groupes" />
          <property name="binding" value="{item.nbGroupes}" />
          <property name="editable" value="false" />
          <property name="sortable" value="true" />
        </bean>
      </list>
    </property>
  </bean>


  <!--
    Édition des périodes en exclusion
  -->
  <bean id="tab.locaux.exclusions"
class="ca.crim.cdmpf.model.container.FormContainer">
    <property name="label" value="Pér.Non Dispo." />
```

```
    <property name="childrens">
      <list>
        <bean class="ca.crim.cdmpf.model.component.ListBox">
          <property name="label" value="Arrangements" />
          <property name="valueListBinding"
value="{context['grille'].grilleLayout.codeArrangList}" />
          <property name="multiSelect" value="true" />
          <property name="binding"
value="{components['tab.locaux.grid'].selectedItem.codesArrangExclusion}" />
        </bean>
      </list>
    </property>
    <property name="bottomComponents">
      <list>
        <bean class="ca.crim.cdmpf.model.component.Button">
          <property name="label" value="Actualiser la grille" />
          <property name="command">
            <bean class="ca.crim.cdmpf.model.command.CommandAggregation">
              <property name="fireChangeAfterExecution"
                value="components['tab.locaux.grid'].selectedItem" />
            </bean>
          </property>
        </bean>
      </list>
    </property>
  </bean>


</beans>
```

## 5. Point d'entrée, interpréteur Swing, horaires scolaires

```
package ca.crim.cdmpf.platform.swing;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.util.List;

import ca.crim.cdmpf.interpreter.AppContext;
import ca.crim.cdmpf.model.Application;
import ca.crim.cdmpf.model.Command;
import ca.crim.cdmpf.model.Component;
import ca.crim.cdmpf.model.ExceptionHandler;
import ca.crim.cdmpf.platform.IPlatformBuilder;
import ca.crim.cdmpf.platform.swing.builder.ISwingApplicationBuilder;
import ca.crim.cdmpf.platform.swing.builder.ISwingCommandBuilder;
import ca.crim.cdmpf.platform.swing.builder.ISwingComponentBuilder;
import ca.crim.cdmpf.platform.swing.builder.ISwingExceptionHandlerBuilder;

public class SwingBuilder implements IPlatformBuilder {

  private List<ISwingApplicationBuilder> applicationBuilders;

  private List<ISwingComponentBuilder> componentBuilders;

  private List<ISwingCommandBuilder> commandBuilders;

  private List<ISwingExceptionHandlerBuilder> exceptionHandlerBuilders;

  private Dimension screenSize;
```

```
  /**
   * Constructor
   *
   */
  public SwingBuilder() {
    Toolkit kit = Toolkit.getDefaultToolkit();
    screenSize = kit.getScreenSize();
  }

  /**
   * Build a Swing application from an application model
   */
  public void buildApplication(AppContext appContext,
      Application applicationModel) {
    ISwingApplicationBuilder applicationBuilder =
getApplicationBuilder(applicationModel);
    applicationBuilder.buildApplication(this, appContext, applicationModel);
    appContext.fireChange("");
  }

  public Callable buildCommand(AppContext appContext, UINode parent,
      Command command) {
    ISwingCommandBuilder scb = this.getCommandBuilder(command);
    return scb.buildCommand(this, appContext, parent, command);
  }

  public UINode buildComponent(AppContext appContext, UINode parent,
      Component modelComponent) {
    ISwingComponentBuilder scb = this.getComponentBuilder(modelComponent);
    if (modelComponent.getName() != null
        && appContext.getComponent(modelComponent.getName()) != null) {
      return (UINode) appContext.getComponent(modelComponent.getName());
    } else {
      return scb.buildComponent(this, appContext, parent, modelComponent);
    }
  }

  public SwingExceptionHandler buildExceptionHandler(AppContext appContext,
      UINode parent, ExceptionHandler exceptionHandler) {
    ISwingExceptionHandlerBuilder sehb = this
        .getExceptionHandlerBuilder(exceptionHandler);
    return sehb.buildExceptionHandler(this, appContext, parent,
        exceptionHandler);
  }

  /**
   * Return an application builder
   *
   * @param object
   * @return
   */
  protected ISwingApplicationBuilder getApplicationBuilder(Object object) {
    for (ISwingApplicationBuilder builder : applicationBuilders) {
      if (builder.canHandle(object)) {
        return builder;
      }
    }
    throw new RuntimeException("Builder not found for object : "
        + object.getClass());
  }

  /**
   * Return a component builder
   *
```

```
 * @param object
 * @return
 */
protected ISwingComponentBuilder getComponentBuilder(Object object) {
  if (object == null) {
    throw new RuntimeException("Builder not found for null object");
  }
  for (ISwingComponentBuilder builder : componentBuilders) {
    if (builder.canHandle(object)) {
      return builder;
    }
  }
  throw new RuntimeException("Builder not found for object : "
      + object.getClass());
}

/**
 * Return a command builder
 *
 * @param object
 * @return
 */
protected ISwingCommandBuilder getCommandBuilder(Object object) {
  for (ISwingCommandBuilder builder : commandBuilders) {
    if (builder.canHandle(object)) {
      return builder;
    }
  }
  throw new RuntimeException("Builder not found for object : "
      + object.getClass());
}

/**
 * Return a exceptionHandler builder
 *
 * @param object
 * @return
 */
protected ISwingExceptionHandlerBuilder getExceptionHandlerBuilder(
    Object object) {
  for (ISwingExceptionHandlerBuilder builder : exceptionHandlerBuilders) {
    if (builder.canHandle(object)) {
      return builder;
    }
  }
  throw new RuntimeException("ExceptionHandler not found for object : "
      + object.getClass());
}

public Dimension getScreenSize() {
  return screenSize;
}

public List<ISwingComponentBuilder> getComponentBuilders() {
  return componentBuilders;
}

public void setComponentBuilders(
    List<ISwingComponentBuilder> componentBuilders) {
  this.componentBuilders = componentBuilders;
}

public List<ISwingApplicationBuilder> getApplicationBuilders() {
  return applicationBuilders;
}
```

```
  public void setApplicationBuilders(
      List<ISwingApplicationBuilder> applicationBuilders) {
    this.applicationBuilders = applicationBuilders;
  }

  public List<ISwingCommandBuilder> getCommandBuilders() {
    return commandBuilders;
  }

  public void setCommandBuilders(List<ISwingCommandBuilder> commandBuilders) {
    this.commandBuilders = commandBuilders;
  }

  public List<ISwingExceptionHandlerBuilder> getExceptionHandlerBuilders() {
    return exceptionHandlerBuilders;
  }

  public void setExceptionHandlerBuilders(
      List<ISwingExceptionHandlerBuilder> exceptionHandlerBuilders) {
    this.exceptionHandlerBuilders = exceptionHandlerBuilders;
  }

}
```

## 6. Interpréteur composant DataGrid, horaires scolaires

```
package ca.crim.cdmpf.platform.swing.builder;

import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.swing.DefaultCellEditor;
import javax.swing.JComboBox;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.JPopupMenu.Separator;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import ca.crim.cdmpf.interpreter.AppContext;
import ca.crim.cdmpf.interpreter.PropertyBinding;
import ca.crim.cdmpf.interpreter.ReflectiveCommand;
import ca.crim.cdmpf.model.Component;
import ca.crim.cdmpf.model.command.CallCommand;
import ca.crim.cdmpf.model.component.DataGrid;
```

```
import ca.crim.cdmpf.model.component.DataGridColumn;
import ca.crim.cdmpf.model.component.DataGridCommand;
import ca.crim.cdmpf.platform.swing.Callable;
import ca.crim.cdmpf.platform.swing.SwingBuilder;
import ca.crim.cdmpf.platform.swing.UINode;
import ca.crim.cdmpf.platform.swing.application.JMultiValuesComponent;
import ca.crim.cdmpf.platform.swing.application.MultiValuesCellEditor;
import ca.crim.cdmpf.platform.swing.application.TableModelImpl;

public class SwingDataGridBuilder implements ISwingComponentBuilder {

  public UINode buildComponent(SwingBuilder sBuilder,
      final AppContext appContext, UINode parent, Component modelComponent) {

    final DataGrid dataGrid = (DataGrid) modelComponent;

    UINode superNode = new UINode(null, parent);

    // Init the JTable
    JTable jTable = new JTable();
    jTable.setAutoCreateRowSorter(true);
    jTable.setRowHeight(18);

    if (dataGrid.isMultiSelect()) {
      jTable
          .setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    } else {
      jTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    }

    JScrollPane jScrollPane = new JScrollPane();
    jScrollPane.setViewportView(jTable);

    // Init TableModel
    DataGridColumn[] dataGridColumns = dataGrid.getColumns().toArray(
        new DataGridColumn[0]);
    TableModelImpl tableModel = new TableModelImpl(appContext, jTable);
    tableModel.setRowEditableExpression(dataGrid.getRowEditable());
    tableModel.setRowStrikedExpression(dataGrid.getRowStriked());
    tableModel.setColumns(dataGridColumns);

    jTable.setModel(tableModel);
    for (int i = 0; i < dataGridColumns.length; i++) {
      if (dataGridColumns[i].getValueListBinding() != null
          || dataGridColumns[i].getValueListCallBinding() != null) {
        if (dataGridColumns[i].isMultiValues()) {
          Window parentWindow = BuilderHelper.getParentWindow(parent);
          JMultiValuesComponent jMultiValuesComponent = new
JMultiValuesComponent(parentWindow);
          jTable.getColumnModel().getColumn(i).setCellEditor(
              new MultiValuesCellEditor(jMultiValuesComponent));
        } else {
          JComboBox jComboBox = new JComboBox();
          jTable.getColumnModel().getColumn(i).setCellEditor(
              new DefaultCellEditor(jComboBox));
        }
      }
    }

    // Init parent component
    final UIDataGridNode dataGridNode = new UIDataGridNode(dataGrid
        .getName(), superNode, jTable, tableModel);
    dataGridNode.setJComponent(jScrollPane);
    appContext.registerComponent(dataGridNode);
```

```
// Register binding
registerBinding(appContext, dataGridNode, dataGrid);

// Init row commands
if (dataGrid.getRowCommands() != null) {
  final JPopupMenu jPopupMenu = new JPopupMenu();
  final List<JMenuItem> menuItems = new ArrayList<JMenuItem>();
  for (DataGridCommand command : dataGrid.getRowCommands()) {
    if (command.isSeparator()) {
      jPopupMenu.add(new Separator());
      menuItems.add(new JMenuItem());
    } else {
      final Callable sCommand = sBuilder.buildCommand(appContext,
          dataGridNode, command.getCommand());
      JMenuItem jMenuItem = new JMenuItem();
      jMenuItem.setText(command.getLabel());
      jMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          sCommand.call();
        }
      });
      jPopupMenu.add(jMenuItem);
      menuItems.add(jMenuItem);
    }
  }
  jTable.addMouseListener(new MouseListener() {
    public void mousePressed(MouseEvent e) {
      if (e.getButton() == MouseEvent.BUTTON3) {
        Map<String, Object> variables = dataGridNode
            .getVariables();
        variables.put("item", dataGridNode.getSelectedItem());
        for (int i = 0; i < menuItems.size(); i++) {
          DataGridCommand command = dataGrid.getRowCommands()
              .get(i);
          boolean enabled = true;
          if (command.getEnabled() != null) {
            Boolean enabledValue = (Boolean) appContext
                .evalBindingObject(variables, command
                    .getEnabled(), Boolean.class);
            enabled = (enabledValue == null || enabledValue == true);
          }
          menuItems.get(i).setEnabled(enabled);
        }
        jPopupMenu.show(e.getComponent(), e.getX(), e.getY());
      }
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void mouseReleased(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }
  });
}

superNode.setJComponent(BuilderHelper.addTopBottomComponents(sBuilder,
    appContext, dataGridNode, dataGrid.getTopComponents(), dataGrid
        .getBottomInfoComponents(), dataGrid
        .getBottomComponents()));
```

```
  return superNode;
}

protected void reloadTableValues(final AppContext appContext,
    final UIDataGridNode dataGridNode, final DataGrid dataGrid) {
  JTable jTable = dataGridNode.getJTable();
  // Stop cell editing
  if (jTable.getCellEditor() != null) {
    jTable.getCellEditor().stopCellEditing();
  }

  int currentRow = jTable.getSelectedRow();
  Object currentRowObject = null;
  if (currentRow >= 0) {
    currentRowObject = dataGridNode.getTableModel().getRows()[jTable
        .convertRowIndexToModel(currentRow)];
  }

  String rowVisibleExpression = dataGrid.getRowVisible();

  // Load new data
  Collection c = null;
  if (dataGrid.getItemsBinding() != null) {
    c = BuilderHelper.asCollection(appContext.evalBindingObject(
        dataGridNode.getVariables(), dataGrid.getItemsBinding(),
        Object.class));
  } else if (dataGrid.getItemsCallBinding() != null) {
    CallCommand callCommand = dataGrid.getItemsCallBinding();
    c = BuilderHelper.asCollection(ReflectiveCommand.call(appContext,
        dataGridNode.getVariables(), callCommand.getBinding(),
        callCommand.getMethod(), callCommand.getArgs()));
  }
  if (c != null) {
    // Process Row visible information
    if (rowVisibleExpression != null) {
      List<Object> newList = new ArrayList<Object>();
      Map<String, Object> variables = dataGridNode.getVariables();
      for (Object row : c) {
        variables.put("item", row);
        Boolean visible = (Boolean) appContext.evalBindingObject(
            variables, rowVisibleExpression, Boolean.class);
        if (visible == null || visible == true) {
          newList.add(row);
        }
      }
      c = newList;
    }
    // Set row values
    dataGridNode.getTableModel().setVariables(
        dataGridNode.getVariables());
    dataGridNode.getTableModel().setRows(c.toArray());
  } else {
    // Set empty values
    dataGridNode.getTableModel().setRows(null);
    dataGridNode.getTableModel().setVariables(
        new HashMap<String, Object>());
  }
  dataGridNode.getTableModel().fireTableDataChanged();

  if (currentRowObject != null
      && dataGridNode.getTableModel().getRows() != null) {
    Object[] rows = dataGridNode.getTableModel().getRows();
    for (int i = 0; i < rows.length; i++) {
      if (rows[i].equals(currentRowObject)) {
        currentRow = jTable.convertRowIndexToView(i);
```

```
        break;
      }
    }
  }
  int rowCount = dataGridNode.getTableModel().getRowCount();
  if (currentRow >= 0 && rowCount > 0) {
    if (currentRow < rowCount) {
      jTable.setRowSelectionInterval(currentRow, currentRow);
    } else {
      jTable.setRowSelectionInterval(rowCount - 1, rowCount - 1);
    }
  }
}

/**
 * Register binding for component
 *
 * @param appContext
 * @param jComponent
 * @param component
 */
protected void registerBinding(final AppContext appContext,
    final UIDataGridNode dataGridNode, final DataGrid dataGrid) {
  BuilderHelper
      .registerEnabledBinding(appContext, dataGridNode, dataGrid);
  BuilderHelper
      .registerVisibleBinding(appContext, dataGridNode, dataGrid);
  if (dataGrid.getItemsBinding() != null) {
    appContext.registerBinding(new PropertyBinding() {
      public String getExpression() {
        return dataGrid.getItemsBinding();
      }

      public void notifyChange() {
        reloadTableValues(appContext, dataGridNode, dataGrid);
      }
    });
  }
  if (dataGrid.getItemsCallBinding() != null) {
    appContext.registerBinding(new PropertyBinding() {
      public String getExpression() {
        return dataGrid.getItemsCallBinding().getBinding()
            + (dataGrid.getItemsCallBinding().getArgs() != null ? Arrays
                .toString(dataGrid.getItemsCallBinding()
                    .getArgs())
                : "");
      }

      public void notifyChange() {
        reloadTableValues(appContext, dataGridNode, dataGrid);
      }
    });
  }

  if (dataGrid.getRowVisible() != null) {
    appContext.registerBinding(new PropertyBinding() {
      public String getExpression() {
        return dataGrid.getRowVisible();
      }

      public void notifyChange() {
        reloadTableValues(appContext, dataGridNode, dataGrid);
      }
    });
  }
```

```
  dataGridNode.getJTable().getSelectionModel().addListSelectionListener(
      new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
          if (!dataGridNode.isUpdateInProgress()) {
            dataGridNode.setUpdateInProgress(true);
            try {
              appContext.fireChange("components['"
                  + dataGridNode.getName() + "']");
            } finally {
              dataGridNode.setUpdateInProgress(false);
            }
          }
        }
      });

  // update de la ligne on firechange sur selectedItem
  appContext.registerBinding(new PropertyBinding() {
    public String getExpression() {
      return "components['" + dataGridNode.getName()
          + "'].selectedItem";
    }

    public void notifyChange() {
      if (!dataGridNode.isUpdateInProgress()) {
        dataGridNode.setUpdateInProgress(true);
        try {
          reloadTableValues(appContext, dataGridNode, dataGrid);
        } finally {
          dataGridNode.setUpdateInProgress(false);
        }
      }
    }
  });
}

public boolean canHandle(Object obj) {
  return obj.getClass() == DataGrid.class;
}

/**
 * DataGrid component node
 *
 * @author bondst
 *
 */
public class UIDataGridNode extends UINode {

  private TableModelImpl tableModel;

  private boolean updateInProgress = false;

  private JTable jTable;

  public UIDataGridNode(String name, UINode parent, JTable jTable,
      TableModelImpl tableModel) {
    super(name, parent);
    this.jTable = jTable;
    this.tableModel = tableModel;
  }

  public Object getSelectedItem() {
    if (tableModel.getRows() != null && tableModel.getRows().length > 0
        && jTable.getSelectedRow() >= 0) {
      // selectElementIfNot();
```

```
            int modelRow = jTable.convertRowIndexToModel(jTable
                .getSelectedRow());
            return tableModel.getRows()[modelRow];
          } else {
            return null;
          }
        }

        public List<Object> getSelectedItems() {
          if (tableModel.getRows() != null && tableModel.getRows().length > 0
              && jTable.getSelectedRow() >= 0) {
            // selectElementIfNot();
            List<Object> selectedItems = new ArrayList<Object>();
            for (int row : jTable.getSelectedRows()) {
              int modelRow = jTable.convertRowIndexToModel(row);
              selectedItems.add(tableModel.getRows()[modelRow]);
            }
            return selectedItems;
          } else {
            return null;
          }
        }

        public int getSelectedItemsCount() {
          if (jTable.getSelectedRows() != null) {
            return jTable.getSelectedRows().length;
          } else {
            return 0;
          }
        }

        @Override
        public Map<String, Object> getVariables() {
          Map<String, Object> variables = super.getVariables();
          variables.put("selectedItem", getSelectedItem());
          variables.put("selectedItems", getSelectedItems());
          variables.put("selectedItemsCount", getSelectedItemsCount());
          return variables;
        }

        public JTable getJTable() {
          return jTable;
        }

        public TableModelImpl getTableModel() {
          return tableModel;
        }

        public boolean isUpdateInProgress() {
          return updateInProgress;
        }

        public void setUpdateInProgress(boolean updateInProgress) {
          this.updateInProgress = updateInProgress;
        }

      }

    }
```