Universite catholique de Louvain
Faculté des Sciences Appliquées
Département d'ingénierie informatique

Etude et implémentation d'un générateur d'interfaces vectorielles à partir d'un langage de description d'interfaces utilisateur

Promoteur : Professeur Jean Vanderdonckt

Mémoire présenté en vue de l'obtention du grade de licencié en informatique par

Vanden Berghe Youri

Louvain-la-Neuve
Année académique 2003-2004

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé dans la réalisation de ce mémoire.

– Monsieur Jean Vanderdonckt, professeur à l'Université catholique de Louvain, pour son encadrement tout au long de l'évolution de ce mémoire.

– Messieurs Quentin Limbourg et Benjamin Michotte, pour leurs nombreuses explications et exemples concrets du langage usiXML.

– Mesdames Florence Wathelet et Martine Rolin ainsi que Monsieur Louis-Philippe Vanden Berghe, pour les précieux conseils dans la rédaction.

Table des matières

Table des figures

Liste des tableaux

Première partie

Introduction

Chapitre 1

Introduction générale

Face à l'évolution rapide des systèmes informatiques et aux changements de type de plate-forme, de résolution d'écran et de langage de programmation qui y sont liés, il est de plus en plus difficile de maintenir à jour les logiciels qui doivent, en outre, souvent être disponibles dans plusieurs langues.

Tous ces changements influencent, d'une manière ou d'une autre, le contexte d'utilisation, qui représente l'environnement global dans lequel l'utilisateur final exécute les tâches liées à l'interface.

Fort de ce constat, l'objectif de ce mémoire est de développer un générateur d'interfaces utilisateur vectorielles multilingues à partir d'une description d'interface indépendante de son contexte d'utilisation. Ceci devrait en effet permettre d'augmenter la portabilité des applications générées et de diminuer le nombre de versions à mettre à jour.

Pour ce faire, dans un premier temps, il conviendra d'analyser les avantages et inconvénients des solutions existantes en matière de génération d'interfaces vectorielles. Ensuite, il sera possible de déterminer l'architecture idéale pour notre générateur en nous inspirant des méthodes classiques en matière de génération d'interfaces, à savoir l'interprétation orientée serveur et la compilation. En effet, ces deux méthodes présentent à côté de leurs avantages respectifs des inconvénients que nous pensons pouvoir éviter.

Ainsi, le côté dynamique de l'interpréteur orienté serveur est, comme son nom l'indique, contrebalancé par la présence indispensable d'un serveur. La compilation, qui génère une interface indépendante, nécessite une intervention (recompilation) lors de chaque modification de l'interface. En outre, après chaque compilation, l'interface générée doit être redistribuée à tous les utilisateurs. L'architecture choisie dans le présent mémoire tentera donc d'allier les avantages de ces deux méthodes.

Dans un second temps, nous nous pencherons d'une part sur le choix d'un langage source permettant une description d'interface indépendante de son contexte d'exécution et, d'autre part, sur le choix d'un langage cible vectoriel et répondant aux objectifs de portabilité, dans lequel nous pourrons générer l'interface.

Après avoir déterminé ces deux langages, nous en établirons les règles de correspondances.

Enfin, nous décrirons l'implémentation d'une solution hybride qui tentera, sur base des analyses préliminaires, de répondre aux objectifs définis. Cette solution hybride implémentée dans le cadre du présent mémoire a été baptisée FlashiXML.

Pour terminer, nous testerons à travers plusieurs exemples, l'efficacité de FlashiXML.

Chapitre 2

Etat de l'art

2.1 Introduction

Dans le présent chapitre, nous nous pencherons sur les solutions existantes dans le domaine de la génération d'interfaces vectorielles à partir d'un langage de description. L'objectif sera de dégager les points forts et les points faibles de ces solutions ainsi que d'analyser les intérêts de l'implémentation d'une nouvelle solution.

A l'heure actuelle, il existe principalement deux solutions qui permettent de générer une interface vectorielle à partir d'un langage de description. Ces deux solutions présentent les points communs suivants :

- elles sont propriétaires ;
- le langage de description de leur interface est basé sur le standard XML ;
- leur langage cible, c'est-à-dire le langage vectoriel dans lequel l'interface est générée, est Macromedia Flash ;
- elles permettent d'exécuter des fonctions définies dans un langage de script au sein du fichier de description ;
- elles sont orientées serveur, c'est-à-dire que la génération de l'interface se fait dynamiquement en réponse à une requête HTTP à un serveur.

Les sections suivantes mettent en évidence les avantages et inconvénients de chaque solution. A titre d'illustration et de comparaison, elles présentent, pour chaque solution, le code d'une petite application permettant de convertir une température, donnée en degrés Fahrenheit, en degrés Celsius.

2.2 Macromedia Flex Presentation Server

Flex, proposé par Macromedia depuis avril 2004, vise à « combiner l'interactivité des applications *desktop* avec la facilité de pénétration et de dis-

tribution des applications web »[1].

Flex Presentation Server constitue le *tiers présentation* du modèle d'application N-tiers d'une organisation. Il ajoute une couche de code exécutable sur la *machine client* au dessus du HTML habituellement généré. Flex permet ainsi de soulager le serveur et d'augmenter le temps de réponse en déplaçant vers la *machine client* l'exécution de toute une série d'opérations telles que la validation de champs, le tri de données, ...

La figure 2.1 illustre l'architecture du *Flex Presentation Server* composée d'un framework contenant :
- un langage de description propre, MXML, basé sur XML ;
- le langage orienté objet ActionScript 2.0 qui n'est autre que le langage de programmation de Flash MX 2004. Il s'agit d'un langage de script qui, comme Javascript, est basé sur la norme ECMA262 ;
- une librairie de classes Flex.

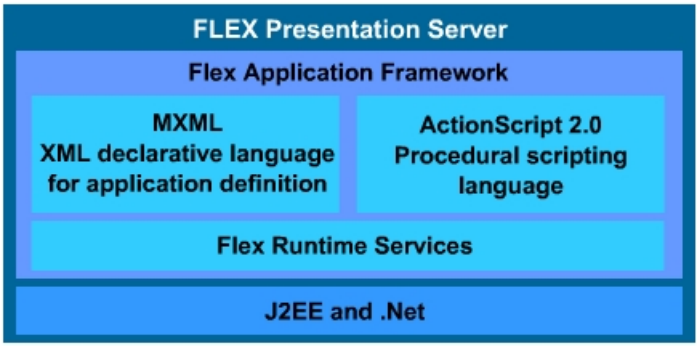

FIG. 2.1 – Illustration de l'architecture du *Flex Presentation Server*.

A titre d'illustration, la figure 2.2 présente le code Flex d'un convertisseur de température Fahrenheit-Celsius.

Les points forts de Flex identifiés sont :
- la compatibilité avec tous les standarts HTML, HTTP(S), XML, SOAP/web services, CSS, SVG, J2EE, .NET,... ;
- une intégration facile dans les modèles d'application N-tiers existants ;
- le fait d'être développé par la même société que Flash. Tous les composants graphiques de Flash MX 2004 sont donc disponibles ;
- le langage de script ActionScript est le même que celui de Flash, ce qui simplifie la tâche des programmeurs habitués à Flash ;

[1] voir [9]

```
<?xml version="1.0" encoding="utf8" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:NumberFormatter id="myFormat" precision="2"/>
    <mx:Label text="Temperature in Fahrenheit:"/>
    <mx:TextInput id="fahrenheit" width="100"/>
    <mx:Button label="Convert"
            click="celsius.text=myFormat((fahrenheit.text-32)/1.8)"/>
    <mx:Label text="Temperature in Celsius:"/>
    <mx:Label id="celsius" width="100"/>
<mx:Application>
```

FIG. 2.2 – Code Flex (*MXML* - *ActionScript*) d'un convertisseur de température Fahrenheit-Celsius.

– la distribution de nouvelles versions se fait automatiquement lors de l'exécution suivante.

Flex présente par contre les inconvénients suivants :
– il nécessite une architecture N-tiers basée sur un J2EE ou .Net même pour réaliser de petites interfaces ;
– ses applications sont lourdes à déployer ;
– il nécessite un navigateur internet sur la *machine client* ;
– il est une solution propriétaire.

2.3 Laszlo Systems Presentation Server

Bien que le nom porte à confusion, *Laszlo Presentation System* n'est pas, au même titre que Flex, destiné à être utilisé comme *tiers présentation* au sein d'une application N-tiers. L'architecture de Laszlo Presentation System comprend :
– un langage de description propre LZX basé sur XML ;
– des scripts Javascript intégrés dans le code LZX.

Laszlo Presentation server est une méthode orientée serveur dont le fonctionnement est comparable à celui de langages tels que PHP. En effet, le code Laslo, rédigé au format LZX et Javascript, est entreposé sur le serveur. De ce fait, lorsque l'utilisateur émet une requête HTTP au serveur, ce dernier compile dynamiquement le code et renvoie l'animation Flash résultante.

Le principal avantage de Laszlo par rapport à Flex réside dans le fait qu'il est moins lourd et ne nécessite pas une architecture N-tiers. En outre, le langage de javascript utilisé comme langage de script est très répandu.

En contrepartie, Laslo présente plusieurs inconvénients dont le fait que :

– le nombre de composants graphiques disponibles n'est pas aussi élevé que celui de Flex et les composants sont moins robustes ;
– le module serveur de compilation de Laszlo nécessaire n'est pas très répandu ;
– il nécessite, comme Flex, un navigateur sur la machine client pour être exécuté ;
– Laszlo est une solution propriétaire.

A titre d'illustration, la figure 2.3 présente le code Laszlo d'un convertisseur de température Fahrenheit-Celsius.

```
<canvas width="300">
    <view>
        <text>Temperature in Fahrenheit : </text>
        <edittext id="fahrenheight" width="100"><edittext>
        <button onclick="celsius.setText((fahrenheit.getText()-32/1.8))">
            Convert
        </button>
        <text>Temperature en Celsius :</text>
        <text id="celsius" width="100"></text>

        <simplelayout axis="y" spacing="2"/>
    </view>
</canvas>
```

FIG. 2.3 – Code Laszlo (*LZX - Javascript*) d'un convertisseur de température Fahrenheit-Celsius.

2.4 Conclusion

Les deux solutions étudiées dans les sections précédentes permettent de générer des interfaces :
– en Macomedia Flash ;
– à partir d'un langage de description d'interface basé sur XML ;
– et de fonctions codées dans un langage de scripts orientés objet

Toutefois, ces deux solutions possèdent certains inconvénients communs :
– elles nécessitent au minimum la présence d'un serveur pour pouvoir être exécutées ;
– elles requièrent la présence d'un navigateur internet sur la machine client ;
– elles sont propriétaires.

Il est donc intéressant d'envisager une nouvelle solution réalisant les mêmes objectifs mais palliant au moins à l'un des inconvénients cités ci-dessus.

Deuxième partie

Choix de base du générateur

Chapitre 3

Choix de l'architecture

Dans le chapitre 2 nous avons vu que les solutions existantes en matière de génération d'interface sont basées sur une interprétation dynamique du langage source par un serveur, avec pour conséquence directe la dépendance par rapport à ce serveur. En outre, les solutions existantes nécessitent également toutes de disposer d'un navigateur internet sur la machine client.

Une alternative à l'utilisation du serveur est celle d'un compilateur. En effet, celui-ci permet d'exécuter l'interface générée au format *SWF*[1] dans tout environnement possédant un lecteur Flash. Il n'est, dans ce cas, plus nécessaire de disposer d'un navigateur internet. Malheureusement, ce procédé présente également une série d'inconvénients dont les principaux sont :

- la nécessité de recompiler l'application chaque fois qu'une modification doit être apportée ;
- l'obligation d'installer le compilateur sur chaque machine susceptible de devoir apporter des modifications à l'application ;
- l'impossibilité de ré-utiliser les composants développés par Macromedia qui devront donc tous être ré-implémentés.

L'objectif de ce mémoire sera donc de tenter de développer une *solution hybride* alliant les avantages de l'application orientée serveur et du compilateur. Pour cela, elle devra, selon les exigences :

- permettre une compilation dynamique ;
- pouvoir bénéficier d'une facilité de mise à jour et de distribution de nouvelles versions ;
- pouvoir fonctionner de manière indépendante par rapport à un serveur.

A cette fin, l'application hybride mise en place dans le cadre de ce mémoire, baptisée FlashiXML, devra, tout comme les solutions existantes

[1] format compilé de Flash

se baser sur :

- un langage de description d'interface (langage source) ;
- un langage de génération (langage cible) ;
- un langage de compilation dynamique.

Chapitre 4

Choix du langage source

4.1 Introduction

La description de l'interface constitue le point de départ incontournable du processus de génération. Il convient, par conséquent, de bien choisir le langage dans lequel l'interface sera décrite (*langage source*).

Le chapitre 2 a mis en évidence le fait que les solutions existantes utilisent leur propre langage de description. Ce dernier est dès lors taillé sur mesure de manière à simplifier les règles de conversion *langage source - langage cible*.

A titre d'exemple, MXML (voir section 2.2), qui est le *langage source* de Flex, comporte les mêmes composants que Flash (*langage cible*).

En contrepartie de la facilité de génération qu'il apporte, le fait d'utiliser un langage de description propre présente l'énorme inconvénient que ce dernier n'est pas standard, trop *personnalisé* et peu répandu. Il est donc peu probable qu'une autre solution permette de générer une interface à partir du même langage source.

Dans le chapitre 2 nous avons également pu constater que les solutions existantes possèdent toutes un langage source basé sur le standard XML qui présente l'avantage de pouvoir organiser la description de l'interface sous forme de hiérarchie très intuitive.

Sur base de ces constats, afin de concilier les avantages évoqués et de ne pas être confronté aux inconvénients des autres solutions, la section 4.2 est consacrée à une étude des différents langages de description d'interfaces existants (*UIDL, User Interface Description Language*).

4.2 Analyse des UIDL existants et choix à proprement parler

Il existe un grand nombre d'*UIDL* (User Interface Description Language)qui diffèrent parfois très fortement les uns des autres de par leurs propriétés ou leur champ d'application. Les tableaux 4.1 et 4.2 [26] reprennent différentes caractéristiques de dix UIDL existans : UIML [17], AUIML [10], XIML [22], SeescoaXML [13], TeresaXML [21], WSXL [2], XUL [31], XISL [28], AAIML [6] et TADEUS-XML [3].

Les critères qui nous intéressent particulièrement, à court ou à long terme, dans le choix du langage source sont :

une couverture suffisante : Ce critère n'est pas rempli par les langages UIML, AUIML, Seescoa-XML, XUL, XISL et AAIML qui ne possèdent pas de modèle des tâches ou du domaine. Bien que ces modèles ne seront pas utilisé dans le cadre de ce mémoire, il est intéressant de les avoir à disposition pour d'éventuels travaux futurs ;

une sémantique bien définie : Or, la sémantique du langage XIML n'est pas bien définie ;

un objectif de spécification : Ce critère n'est pas rencontré par le langage XUL qui vise à décrire des interfaces en vue de la visualisation et non de la spécification ;

un support par les outils : Les langages XIML et AUIML ne sont pas accompagnés de bons outils de génération.

Face aux différents critères recherchés, un nouveau langage, usiXML (*USer Interface eXtensible Markup Language*)[29] a été choisi comme langage source de FlashiXML. Dans la section 4.3 nous analyserons la portée de usiXML. Ensuite, la section 4.4 approchera, plus en détail, les concepts de usiXML utilisés par FlashiXML.

	Models	Methodology	Tools	Supported languages	Supported Platforms	Target
UIML	Presentation and dialog	Specification of multiple UI presentations and factoring out/corrective decoration.	Liquid UI: rendering engine, code generator and editor	Java, HTML, WML, VoiceXML, C++, PalmOS	Handheld and desktop PC; Smart, standard and mobile phone; vocal UI.	Multi-platform
AUIML	Presentation and dialog.	Specification of a generic UI description. The decoration can be done either by the renderer or by the developer.	Rendering engine	HTML, DHTML, Java Swing, PalmOS WML	Handheld and desktop PC	Multi-platform (available interactors, displays)
XIML	Any model.	Specification of multiple UI descriptions or of a generic one.	Rendering engine, code editor	HTML, WML, Java	Handheld and desktop PC ; mobile phone ; Java Terminal	In theory, multi-platform, -user and -environment; in practice, multi-platform
Seescoa XML	Presentation and dialog	Specification of a generic UI description.	Rendering engine	Java Swing and AWT, HTML	Handheld and desktop PC	Multi-platform
Teresa XML	Presentation and dialog ; Task, domain and platform	Specification of a generic UI description.	TERESA	XHTML, Voice XML	Handheld and desktop PC, mobile phone	Multi-platform
WSXL	Presentation, dialog and data.	Specification of multiple UI descriptions and factoring out/corrective decoration	WSXL SDK	Markup languages: HTML, XUL, UIML ...	Desktop PC	Mono -platform, -user -environment
XUL	Presentation and dialog.	Specification of multiple UI descriptions and factoring ou/corrective decoration	Rendering engine Gecko, XPCOM/ XPConnect, XPInstall, Mozilla	XUL	Desktop PC (Web application using Mozilla)	Multi-platform
XISL	Dialog.	Specification of multiple UI descriptions and factoring out/corrective decoration	XISL Interpreter	XML-based languages	Mobile phone, Desktop PC, digital TV with multi-modal capabilities	Multi-platform
AAIML	Presentation and dialog.	Not yet defined	Not yet developed, Prototypical architecture of the URC.	Not yet defined	Simulation on Handheld PC, Smart TV	Multi-user on multiple platforms
TADEUS XML	Presentation (based on a user, a task and an object models)	Specification of a generic UI description	TADEUS, XML converter to be developed	Not specified	Not specified	Multi-platform

FIG. 4.1 – Comparaison de différents UIDL existants

	Level	Tags	Expressivity	Openness	Concepts
UIML	Model level	36 tags	Moderate	No	Interface, structure, style, content, behavior, part, peers, logic, presentation
AUIML	Model level	No clear information available, at least 55 tags	Moderate	No	Date-group, group, actions
XIML	Meta-model level	33 tags.	High: everything can be expressed, as the language is open	Yes	Component models, model element, relation_definition, feature_definition, attribute_definition
Seescoa XML	Model level	On the way to be completed, no stable DTD available	Low	No	Group, interactor, action
Teresa XML	Model level	32 tags for the UI description	High	No	Presentation, structure, AIO, interaction_AIO, application_AIO, AIO_composition and connection
WSXL	Instance and model level	No limit, XFORMS	Low: web-application, only graphical UI	No	
XUL	Instance and model level	At least 60 tags	Low: limited to windows-based graphical UI	No	Window, box, hbox, vbox
XISL	Model level	53 tags	High: multimodal UI on multiple platforms. Input Modalities supported: DTMF, speech, pointing, keyboard. Output Modalities: window, speech, video, audio, agent.	Yes	Dialog, exchange, operation, action, input, output
AAIML	Model level	Not yet defined	Moderate	Not specified	Not yet defined
TADEUS XML	Model level	Not specified	Low	No	Uio, input, output, trigger

FIG. 4.2 – Comparaison de différents UIDL existants

4.3 Portée et objectifs du langage usiXML

usiXML (*USer Interface eXtensible Markup Language*)[29] est un UIDL qui permet de décrire une interface à plusieurs niveaux d'abstraction. Il est ensuite possible de transformer la description d'un niveau d'abstraction à un autre. En outre, plusieurs outils permettent de générer des descriptions usiXML.

L'organisation des différents niveaux de description d'interface de usiXML est inspirée du *framework de Cameleon*[16] illustré à la figure 4.3. Les différentes couches du framework sont les suivantes :

Final UI (FUI) : constitue l'interface opérationnelle exécutée sur une plate-forme spécifique, soit par interprétation, soit après compilation.

Concrete UI (CUI) : représente l'abstraction du FUI qui décrit l'interface de manière indépendante de la plate-forme d'exécution. De manière symétrique, le CUI est une concrétisation de l'AUI en *CIO*'s (*Concrete Interaction Objects*) en fonction d'un contexte.

Abstract UI (AUI) : est une description abstraite de l'interface qui est indépendante de toute modalité d'interaction (interaction graphique, interaction vocale, synthèse et reconnaissance vocale,...).

Task & Concepts : constitue une desctiption des différentes tâches ainsi que des concepts du domaine requis pour l'exécution correcte de ces tâches.

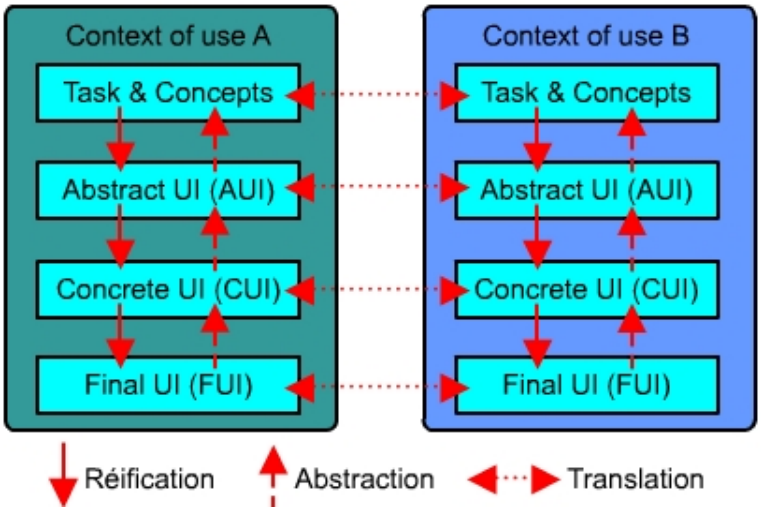

FIG. 4.3 – Illustration du *framework Cameleon*

En outre, le *framework Cameleon* définit trois types de transformations :

L'abstraction : transformation permettant le passage d'un niveau de description à un niveau plus abstrait.

La reification : transformation d'un niveau de description à un niveau plus concret.

La translation : processus de traduction d'artefacts destinés à un contexte d'utilisation particulier en artefacts du même niveau d'abstraction mais destinés à un context d'utilisation différent.

La possibilité de transformer une description d'un niveau d'abstraction à un autre permet de décrire l'interface à un niveau indépendant du contexte et/ou des modalités d'interaction et de la transformer ensuite en fonction des différents contextes dans lesquels elle devra pouvoir être exécutée. Il est dès lors possible de décrire des interfaces usiXML de manière totalement indépendante du langage cible utilisé pour générer l'interface. Ceci constitue un avantage énorme par rapport aux langages de description utilisés par les applications existantes (voir section 2).

Au delà des modèles *AUI* et *CUI*, usiXML est équipé d'une série de modèles de base (voir figure 4.4) tels que :

uiModel : superclasse contenant les caractéristiques communes à tous les modèles. Un uiModel peut-être composé d'une liste de modèles en tout nombre et dans tout ordre. La description d'une interface utilisateur ne doit pas nécessairement comporter un exemplaire de chaque modèle.

taskModel (hérite de uiModel) : décrit les tâches d'interactions telles qu'elles seront perçues par l'utilisateur final. Le taskModel est une décomposition de tâches en sous-tâches liées par des relations de tâches.

domainModel (hérite de uiModel) : description des classes d'objets manipulés par l'utilisateur lorsqu'il intéragit avec le système.

mappingModel (hérite de uiModel) : modèle contenant une série de conversions entre des modèles ou des éléments de modèle.

contextModel (hérite de uiModel) : modèle décrivant les trois aspects d'un contexte d'utilisation dans lequel l'utilisateur exécute une tâche interactive, sur une plate-forme spécifique, dans un environnnement donné. Par conséquent, un *contextModel* est composé d'un *modèle utilisateur*, d'un *modèle de plate-forme*, et d'un *modèle d'environement.*

4.4 Description des concepts de usiXML implémentés

Dans le cadre de ce mémoire, nous utiliserons les couches *FUI* et *CUI* du *framework Cameleon*. En effet, l'interface sera décrite dans la couche *CUI* et le générateur produira la couche *FUI* correspondante.

Dans la section suivante, nous étudierons, en détail, les concepts de usiXML utilisés dans la suite de ce document.

4.4.1 L'uiModel

Comme nous l'avons vu à la section 4.3 l'*uiModel* (User Interface Model) est la superclasse de tous les modèles. Il en contient les caractéristiques communes. Parmi celles-ci, les paramètres d'information (version, auteur, liens vers la spécification usiXML...) nous intéressent particulièrement. Ils peuvent, en effet, être mis à la disposition de l'utilisateur.

En outre, l'*uiModel* contient les trois modèles qui seront nécessaires pour générer l'interface :
- le *cuiModel* ;
- le *contextModel* ;
- le *resourceModel*.

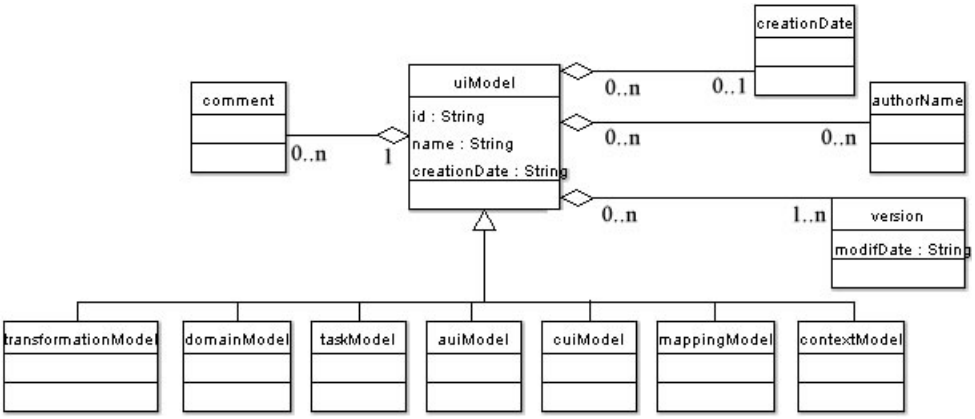

FIG. 4.4 – Illustration de l'uiModel et des modèles qui en héritent.

4.4.2 Le cuiModel

Le *cuiModel* (Concrete User Interface Model) est un modèle permettant de spécifier l'apparence et les comportements des éléments qui composent l'interface. A cette fin, le *cuiModel* est composé d'un agencement de *CIO* et de *CUIR*.

CIO : Concrete Interaction Object

Les *CIO* sont des entités que l'utilisateur peut percevoir et/ou manipuler (boutons, images, case texte, ...). Les *CIO* constituent donc une abstraction

d'un ensemble d'éléments graphiques extraits de *toolkits* fortement répandus (Java Swing, HTML, Flash MX 2004). La figure 4.5 illustre la structure des *CIO*'s.

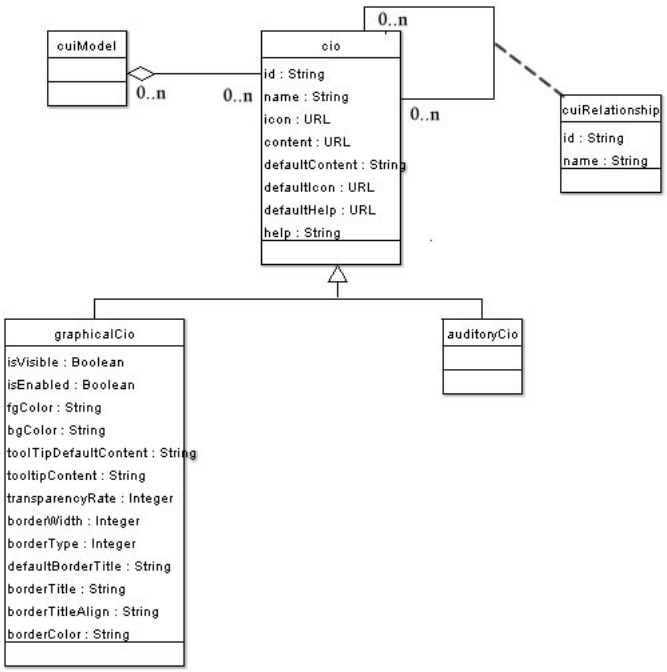

FIG. 4.5 – Illustration de la structure des *CIO*'s.

Dans le cadre de ce mémoire et donc pour la génération d'interfaces, les *Graphical CIO's* sont parmi les CIO ceux qui nous intéressent le plus. Ces derniers se divisent en deux catégories :

***Graphical Container* :** élément pouvant contenir d'autres éléments (fenêtre, dialogBox, menuBar,...) (voir figure 4.6) ;

***Graphical Individual Component* :** élément terminal (bouton, case texte, image, ...) (voir figure 4.7).

Il convient également de souligner que les *CIO* peuvent être associés à des comportements. Le *CIO*adoptera le comportement lors de la survenance d'un événement. Un comportement est dès lors décrit par :

- un événement déclencheur (clic avec la souris, ...) ;
- une ou plusieurs action(s) à effectuer lorsque l'événement déclencheur se produit.

A son tour, une action peut être composée de :

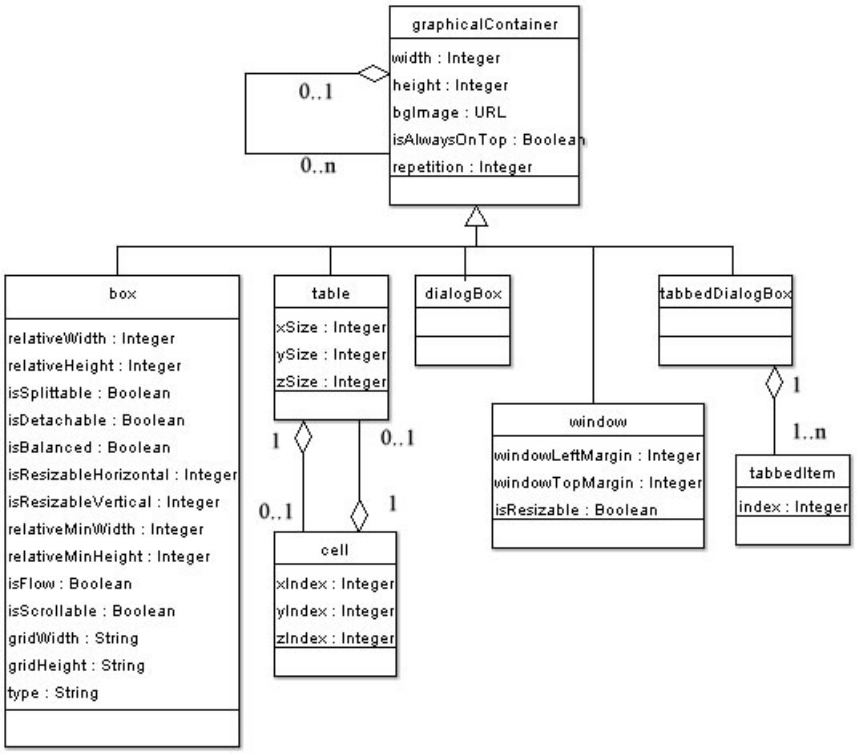

FIG. 4.6 – Illustration de la structure des *Graphical Container*.

– une ou plusieur(es) transition(s) graphique(s) ;
– un ou plusieur(s) appel(s) de fonctions à effectuer.

CUI : Concrete User Interface Relationschip

Les CUIR représentent différents types de relations possibles entre les *CIO*. Parmi celles-ci, nous nous intéresserons uniquement aux *graphical transitions* qui permettent de spécifier la navigation entre les *CIO* (par exemple, ouvrir, fermer ou minimiser une fenêtre...).

A titre d'illustration, la figure 4.8 contient un fragment de code usiXML décrivant une interface composée d'une fenêtre, d'un bouton et d'une transition permettant au bouton de fermer la fenêtre.

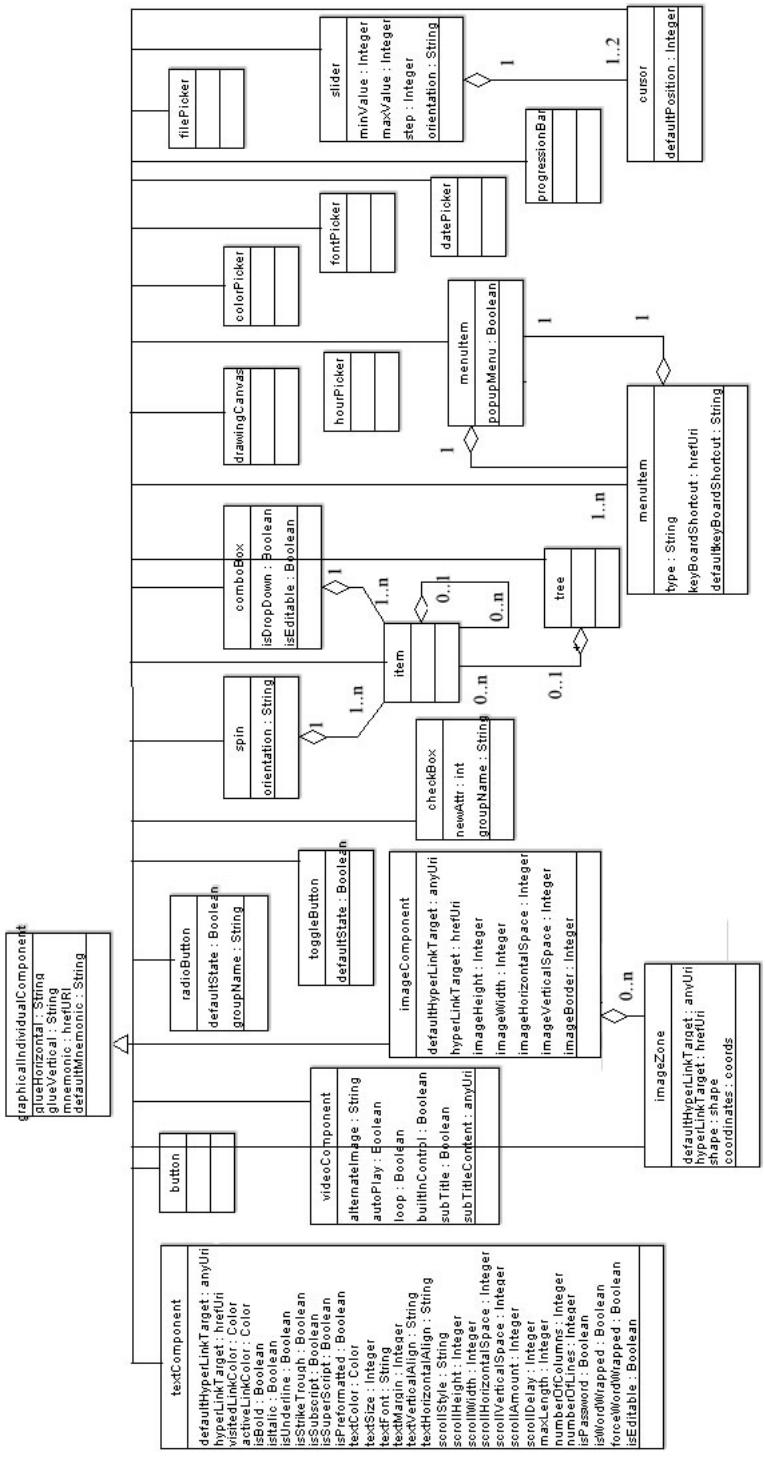

FIG. 4.7 – Illustration de la structure des *Graphical Individual Components*.

```
<window id="win1">
    <button id="btn1" content="bouton 1"/>
</window>
<graphicalTransition id="Tr1" transitionType="close">
    <source sourceId="btn1">
    <target targetId="win1">
</graphicalTransition>
```

FIG. 4.8 – Exemple de code usiXML associant une transition à un bouton.

4.4.3 Le contextModel et le resourceModel

Comme la section 4.3 l'a démontré, le *contextModel* décrit les trois aspects d'un contexte d'utilisation dans lequel l'utilisateur exécute une tâche intéractive, sur une plate-forme spécifique et dans un environnement donné. Dans le cadre de ce mémoire, le seul aspect que nous utiliserons est celui relatif à l'utilisateur (*userStereotype*). Ce dernier contient en effet la langue d'utilisation de l'interface.

Le *resourceModel* contient, quant à lui, des indications spécifiques concernant les *CIO* liés au contexte. Nous entendons par là les contenus à afficher en fonction d'un contexte et donc également en fonction de la langue de l'utilisateur.

Ensemble, le *contextModel* et le *resourceModel* vont permettre de récupérer, pour chaque élément graphique, le contenu à afficher en fonction de la langue de l'utilisateur.

Chapitre 5

Choix du langage cible

5.1 Introduction

Dans le chapire 4 nous avons déterminé le langage source : usiXML. Dans le présent chapitre, nous allons nous pencher sur le choix du langage *cible*, c'est-à-dire le langage dans lequel l'interface sera générée.

Pour une question de portabilité, l'objectif de ce mémoire est de générer l'interface dans un langage vectoriel permettant un redimensionnement sans dégradation et, par conséquent, l'éxécution dans des environnements de résolutions d'écrans différents. Le choix de générer une interface vectorielle limite fortement le nombre de langages cibles disponibles.

Comme nous l'avons vu dans le chapitre 2, les deux solutions existantes en matière de génération d'interfaces vectorielles utilisent Macromedia Flash comme langage cible.

Cependant, d'autres langages vectoriels existent tels que le standard W3C *SVG*. De ce fait, la section 5.2 du présent chapitre étudie dans quelle mesure *SVG* pourrait constituer une alternative à l'utilisation de Flash.

Dans la section 5.3, sur base de la conclusion obtenue au point précédent, nous analyserons en profondeur le langage choisi.

5.2 Analyse des langages vectoriels existants

SVG [1] est un standard ouvert, basé sur XML. Il s'agit dès lors d'un langage balisé dont le contenu est structuré hiérarchiquement. Les balises décrivant une animation SVG sont interprétées et affichées par un *Viewer* au sein d'un navigateur internet.

Le format *SVG* décrit une série de formes (rond, carré, polygones,...)

et une série de types prédéfinis (image, texte,...). Il est également possible d'utiliser des fonctions Javascript au sein des balises *SVG*, ce qui permet de créer des animations vectorielles interactives.

Le standard semble d'autant plus prometteur que de nombreux projets en cours étudient les possibilités de migration entre des animations *SWF* (format compilé de Flash qui est actuellement le plus répandu sur le marché) et *SVG*. Ainsi, [25] et [23] analysent et comparent la sémantique de *SVG* et celle de Flash ainsi que les relations qui existent entre elles. [24] étudie les possibilités de transformer une animation *SWF* en animation *SVG*.

Malheureusement, *SVG* présente plusieurs inconvénients majeurs qui le rendent peu adapté à la génération dynamique d'une interface. A titre d'exemple, nous pouvons citer que :

- *SVG* ne possède pas de *toolkit* complet (fenêtres,...). Il faudrait dès lors ré-implémenter presque entièrement ce dernier ;
- il existe plusieurs *Viewer SVG*, mais qui sont peu répandus et pas standardisés. Une animation affichée correctement par un *Viewer* peut être rendue de manière inappropriée par d'autres.

Face à ces inconvénients, notre choix de langage cible s'est porté sur Flash. Les avantages de celui-ci sont repris à la section suivante.

5.3 Description de Macromedia Flash

Flash permet la création d'animations au format éditable *FLA* qui doivent être compilées au format *SWF* afin de pouvoir être exécutées. Le format *SWF* est très léger et peut être visualisé :

- dans un navigateur internet possédant un *plug-in Flash*. Ce qui est le cas de la plupart des navigateurs ;
- dans le *Player Flash* (qui doit être installé sur la machine utilisateur) ;
- sous forme d'exécutable comprenant le *Player Flash*.

Il est intéressant de reprendre l'historique de Flash afin de bien comprendre les concepts qu'il englobe dans les versions courantes. Initialement, Flash a été conçu pour créer des animations graphiques. Une animation était alors composée uniquement de formes et de *MovieClips* (ensemble de formes). Flash était alors limité au fait de pouvoir créer des imbrications de *MovieClips* et de leur appliquer des déplacements dans le temps.

Ensuite, petit à petit, Flash a permis de rajouter un nombre croissant de propriétés aux MovieClips (couleur, taille,...) ainsi que de l'interactivité permettant de changer dynamiquement ces propriétés.

Enfin, dans les versions récentes de Flash, l'interactivité est contrôlée par un langage de script complet, nommé ActionScript, basé sur la même norme que Javascript. Depuis la dernière version de Flash (Flash MX 2004), disponible depuis octobre 2003, ActionScript est fortement typé et implémente les notions de classe, d'interface, d'héritage, d'encapsulation et de polymorphisme et est, dès lors, un langage orienté objet à part entière [20]. Bien que la syntaxe d'ActionScript soit proche de celle de Java, trois différences principales sont à noter :

- ActionScript ne permet pas l'héritage multiple ;
- l'interaction avec les éléments graphiques de l'interface est très différente, ce qui est bien entendu lié à l'évolution de Flash ;
- ActionScript n'a aucun accès au système de fichier de la machine client.

En outre, bien que les *MovieClip* constituent toujours la base de tout élément graphique de Flash, Macromedia a introduit le concept de *composant* qui les surplante de loin. Il s'agit d'ensembles de *MovieClips* et du code qui leur est associé. Les *composants* sont des objets (au sens de langage orienté objet) et possèdent donc des propriétés ainsi que les méthodes qui permettent de les modifier. A cela viennent s'ajouter quatre avantages des *composants* :

- ils peuvent être instanciés dynamiquement ;
- Macromedia a développé un *toolkit* complet de *composants* [8] dont l'apparence peut être personnalisée [11] ;
- il est possible de créer ses propres composants qui seront ensuite instanciable dynamiquement au même titre que les *composants* développés par Macromedia [7] ;
- il est possible de récupérer les composants développés par quelqu'un d'autre et de les intégrer dans ses propres projets.

Il existe deux catégories de composants :

- les *composants graphiques* (bouton, liste déroulante, ...) ;
- les *composants de données* qui servent à mettre en relation des composants graphiques et des données externes (base de données, fichiers xml,...).

Enfin, il convient de souligner que les composants disponibles dans la version normale et dans la version professionnelle de Flash ne sont pas identiques. Dans ce document, nous ne nous occuperons que des *composants graphiques* de la *version professionnelle* de Flash *fournis par Macromedia* (voir figure 5.1).

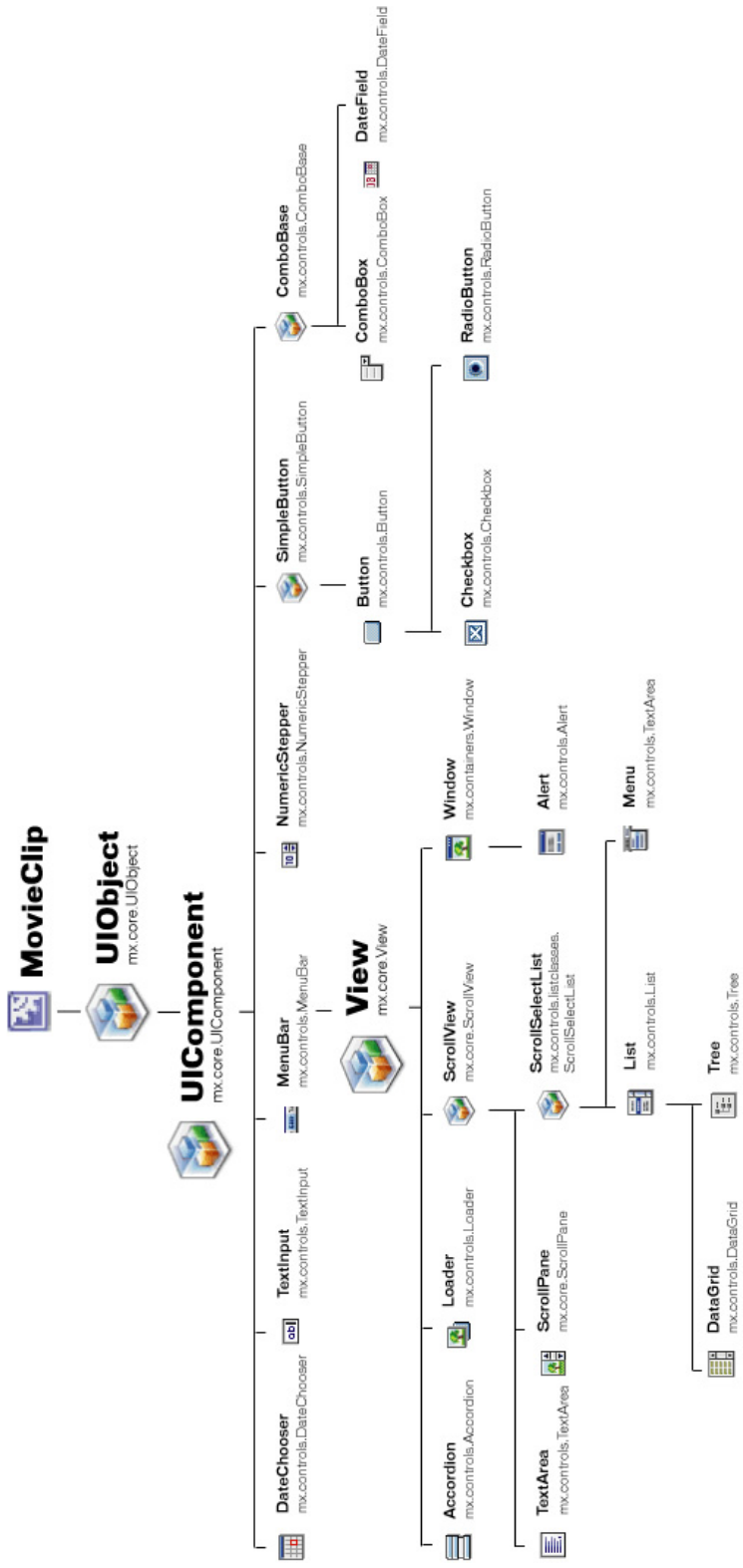

FIG. 5.1 – Hiérarchie des composants de Flash MX 2004 Professional.

5.4 Conclusion

L'étude des propriétés des langages vectoriels Flash et SVG nous a poussés à préférer Flash comme langage cible du générateur.

En effet, il semble que le seul reproche qui puisse être fait à Flash dans le cadre de la génération d'une interface vectorielle est le fait que, contrairement au standard *SVG*, Flash est un langage propriétaire. Toutefois, les spécifications du langage *SWF* sont publiques.

Chapitre 6

Choix du langage orienté serveur

6.1 Introduction

Afin de permettre à notre application d'imiter le fonctionnement des solutions orientées serveur, il sera nécessaire de disposer d'un langage orienté serveur.

PHP[1] s'est imposé directement dans ce rôle car il est le seul capable de générer des animations Flash à la volée. Initialement créé dans le but de générer des pages web dynamiques, PHP a ensuite été largement développé et dispose aujourd'hui de deux fonctionnalités qui nous intéressent particulièrement dans le cadre de notre recherche :

1. des fonctions d'accès au système de fichiers du serveur http ;
2. la librairie MING.

A ces fonctionnalités viennent s'ajouter les avantages suivants :
- PHP est gratuit ;
- PHP tourne sur le serveur web *Apache* qui est gratuit ;
- PHP peut interfacer des bases de données *MySQL* dont l'utilisation est gratuite.

La figure 6.1 montre le cheminement normal de l'exécution d'un script PHP sur le serveur depuis la requête du client jusqu'à la réception, par ce dernier, du résultat.

[1] Personal HyperText Preprocessor

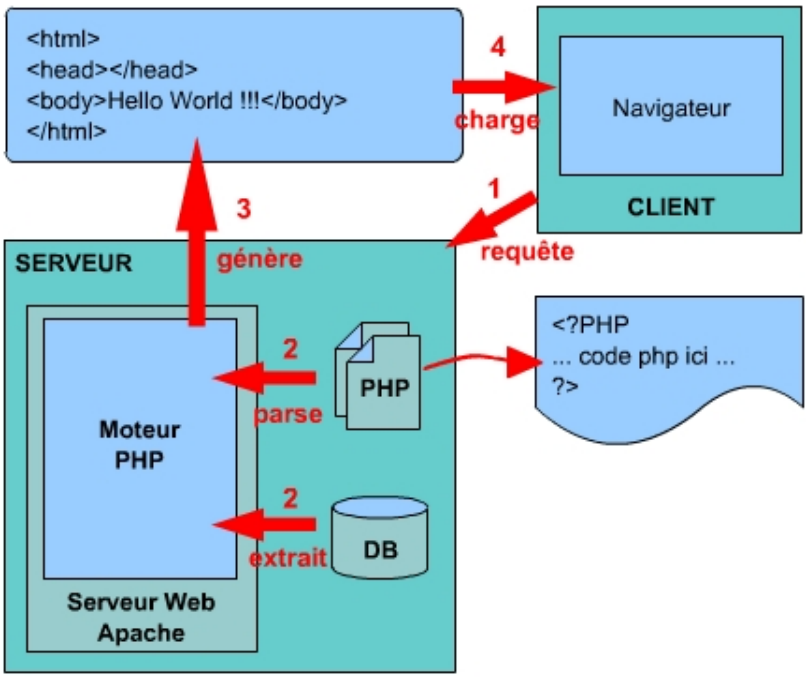

FIG. 6.1 – Génération d'une page web dynamique en PHP

6.2 La librairie Ming

Ming est une librairie open-source (LGPL) qui permet la génération dynamique d'animations Flash en Python, C ou PHP. Utilisée avec PHP, MING permet :

- de récupérer les animations qu'elle génère sous forme d'objet PHP ;
- d'enregistrer ces animations au format swf sur le serveur.

La librairie MING est cependant encore expérimentale, et donc susceptible d'évoluer aussi bien en ce qui concerne les fonctions que les comportements et ce, sans que l'utilisateur n'en soit averti.

Ming supporte toutes les fonctionnalités de Flash 4 : les formes, les gradients, les images JPEG et PNG, les transformations de forme (morphing), les textes, les actions, les sprites, le streaming MP3 et les transformations de couleurs. En outre, MING 0.3a, actuellement en version bêta, inclut un grand nombre de fonctionnalités d'ActionScript MX telles que :

- le chargement dynamique de fichiers XML ;
- l'API de dessin dynamique[2] ;

[2]MX Drawing API

– la duplication de MovieClips ;
– l'association dynamique d'un MovieClip à une classe.

Les figures 6.2 et 6.3 illustrent respectivement le code ActionScript et le code MING d'une animation Flash qui dessine un triangle de couleur rouge. La différence entre ces deux codes réside dans le fait que le premier doit être pré-compilé au format *SWF* pour être exécuté alors que le second génère dynamiquement l'animation Flash et la sauvegarde au format *SWF* sur le serveur.

```
_root.createEmptyMovieClip( 'triangle', 1 );
with ( _root.triangle ) {
    lineStyle( 5, 0xff000, 100 );
    moveTo( 200, 200 );
    lineTo( 300,300 );
    lineTo( 100, 300 );
    lineTo( 200, 200 );
}
```

FIG. 6.2 – Code ActionScript MX d'une animation utilisant l'API de dessin de Flash pour dessiner un triangle.

```
<?
    ming_setScale(20.00000000);
    ming_useswfversion(6);
    $movie = new SWFMovie();
    $movie->setDimension(550,400);
    $movie->setBackground(0xcc, 0xcc, 0xcc );
    $movie->setRate(31);

    $strAction = "
        _root.createEmptyMovieClip( 'triangle', 1 );
        with ( _root.triangle ) {
            lineStyle( 5, 0xff0000, 100 );
            moveTo( 200, 200 );
            lineTo( 300,300 );
            lineTo( 100, 300 );
            lineTo( 200, 200 );
        }
    ";

    $movie->add(new SWFAction(str_replace("\r", "", $strAction)));
    $movie->save("trianglemx.swf");
? >
```

FIG. 6.3 – Code PHP - MING pour créer *dynamiquement* une animation FLASH utilisant l'API de dessin de Flash MX pour dessiner un triangle.

Troisième partie

Conception du générateur d'interfaces

Chapitre 7

Règles de correspondances usiXML-Flash

7.1 Introduction

Sur base de l'étude des solutions existantes (voir chapitre 2), nous avons, dans le chapitre 3 défini l'architecture idéale du générateur. Ensuite, dans les chapitres 4 et 5 nous avons déterminé le langage source et le langage cible de l'interface. Toutefois, avant de pouvoir aborder l'implémentation du générateur, il nous reste à définir les règles de correspondances entre les éléments graphiques du langage source (usiXML) et ceux du langage cible (Macromedia Flash). Cette étape importante constitue l'objet de ce chapitre.

Dans cette analyse des correspondances, les situations suivantes peuvent être rencontrées :

1. l'élément usiXML dispose d'un équivalent parmi les composants de Flash ;
2. l'élément usiXML ne dispose pas d'un équivalent parmi les composants de Flash :
   (a) et l'élément usiXML est indispensable à toute description d'interface mais :
      i. l'élément usiXML peut être reproduit par une combinaison de composants Flash ;
      ii. l'élément usiXML ne peut être reproduit par une combinaison de composants Flash et devra dès lors être implémenté ;
   (b) mais l'élément usiXML est un élément particulier et/ou peu fréquent dont l'utilisation n'est pas capitale à la majorité des descriptions d'interfaces ;

3. un élément n'est pas disponible dans le langage usiXML, mais figure parmi les composants de Flash.

Vu le but du présent mémoire, soit la génération en Flash d'une interface décrite en usiXML, les éléments de Flash correspondants à la catégorie *3)* ci-dessus peuvent naturellement être écartés. En outre, dans un premier temps, les éléments entrant dans la catégorie *2)(b)* ne seront pas non plus pris en considértion.

Ci-dessous, nous décrirons en détail les règles de conversion des éléments correspondants :
- à la catégorie *1)*, et, qui seront probablement intégrés le plus aisément ;
- à une des sous-catégories de la catégorie *2(a)* et, qui devront probablement bénéficier d'une attention particulière.

Pour rappel, les figures 4.6 et 4.7 reprennent la structure des éléments graphiques de usiXML tandis que la figure 5.1 fait de même pour les composants de Flash.

7.2 Règles de correspondances usiXML-Flash des éléments usiXML retenus

7.2.1 Propriétés communes aux éléments usiXML

Comme nous l'avons vu à la section 4, tous les éléments graphiques de usiXML héritent des propriétés de l'élément *graphicalCIO* (et donc également de celles de *CIO*). De même, nous avons vu à la section 5 que tous les composants de Flash héritent des propriétés de *UIComponent*. Le tableau 7.1 reprend les propriétés de l'élément *graphicalCIO* ainsi que les propriétés Flash correspondantes (lorsque ces dernières existent).

propriété usiXML	type usiXML	use	propr. Flash	type Flash
content	String (hrefURI)	optional	label	String
defaultContent	String	optional	-	-
tooltip	String (hrefURI)	optional	-	-
defaultTooltip	String	optional	-	-
icon	String (anyURI)	optional	icon	String
defaultIcon	String (anyURI)	optional	icon	String
isVisible	Boolean	required	visible	Boolean
isEnabled	Boolean	required	enabled	Boolean
fgColor	Color	optional	-	-
bgColor	Color	optional	-	-
borderWidth	Integer	optional	-	-
borderType	String	optional	-	-
defaultBorderTitle	String	optional	-	-
borderTitle	String (hrefURI)	optional	-	-
borderTitleAlign	String	optional	-	-
borderColor	Color	optional	-	-
transparencyRate	Integer	optional	alpha	Number
glueVertical	String	optional	-	-
glueHorizontal	String	optional	-	-
mnemonic	String	optional	-	-
defaultMnemonic	String	optional	-	-

TAB. 7.1 – Règles de correspondances usiXML-Flash des propriétés communes à tous les éléments usiXML.

7.2.2 Correspondances bi-univoques

Certains éléments usiXML disposent d'un correspondant direct parmi les composants de Flash. Il s'agit des éléments entrant dans la catégorie 1) de la section 7.1. Dans la présente section, nous reprendrons, pour chacun d'entre eux, l'élément Flash correspondant ainsi que les règles de correspondances des propriétés spécifiques à l'élément usiXML.

usiXML button - Flash Button Component

L'élément *button* de usiXML ne possède pas de propriétés spécifiques.

usiXML checkBox - Flash CheckBox Component

propriété usiXML	type usiXML	use	propr. Flash	type Flash
defaultState	Boolean	optional	selected	Boolean
groupName	String	optional	-	-

TAB. 7.2 – Règles de correspondances usiXML-Flash de l'élément *checkBox* de usiXML.

usiXML datePicker - Flash DateChooser Component

L'élément *datePicker* de usiXML ne possède pas de propriétés spécifiques.

usiXML imageComponent - Flash Loader Component

propriété usiXML	type usiXML	use	propr. Flash	type Flash
defaultHyperLinkTarget	String (anyURI)	optional	-	-
hyperLinkTarget	String (hrefURI)	optional	-	-
imageHeight	Integer	optional	width	Number
imageWidth	Integer	optional	height	Number
imageHorizontalSpace	Integer	optional	-	-
imageVerticalSpace	Integer	optional	-	-
imageBorder	Boolean	Integer	-	-

TAB. 7.3 – Règles de correspondances usiXML-Flash de l'élément *imageComponent* de usiXML.

usiXML menu - Flash Menu Component

L'élément *menu* de usiXML ne possède pas de propriétés spécifiques.

usiXML radioButton - Flash RadioButton Component

propriété usiXML	type usiXML	use	propr. Flash	type Flash
defaultState	Boolean	optional	selected	Boolean
groupName	String	optional	groupName	String

TAB. 7.4 – Règles de correspondances usiXML-Flash de l'élément *radioButton* de usiXML.

usiXML tabbedDialogBox - Flash Accordion Component

Bien que l'affichage du composant Flash *Accordion* soit relativement différent de celui de tabbedDialogBox de usiXML, les fonctionnalités sont équivalentes.

propriété usiXML	type usiXML	use	propr. Flash	type Flash
bgImage	String (anyURI)	optional	-	-
width	Integer	optional	width	Number
height	Integer	optional	height	Number
repetition	Integer	optional	-	-
isAlwaysOnTop	Boolean	optional	-	-

TAB. 7.5 – Règles de correspondances usiXML-Flash de l'élément *tabbed-DialogBox* de usiXML.

tree

L'élément *tree* de usiXML ne possède pas de propriétés spécifiques.

window

propriété usiXML	type usiXML	use	propr. Flash	type Flash
bgImage	String (anyURI)	optional	-	-
width	Integer	optional	width	Number
height	Integer	optional	height	Number
repetition	Integer	optional	-	-
isAlwaysOnTop	Boolean	optional	-	-
windowLeftMargin	Integer	optional	-	-
windowTopMargin	Integer	optional	-	-
isResizable	Boolean	optional	-	-

TAB. 7.6 – Règles de correspondances usiXML-Flash de l'élément *window* de usiXML.

7.2.3 Combinaisons de composants Flash

Dans cette section, nous reprendrons les éléments de usiXML qui ne disposent pas d'un équivalent direct parmi les composants de Flash, mais qui peuvent être reproduit par une combinaison de composants Flash. Il s'agit des éléments usiXML de la catégorie *2)a)i.* de la section 7.1. Pour

chacun d'eux, nous reprendrons les éléments Flash à combiner ainsi que les règles de correspondances des propriétés spécifiques à l'élément usiXML.

comboBox

Selon la valeur de la propriété *isDropDown* de l'élément comboBox de usiXML, il convient d'utiliser un composant Flash différent. Ainsi, lorsque *isDropDown* :

- vaut false, c'est le composant *List* de Flash qui sera utilisé ;
- vaut true, c'est le composant *comboBox* de Flash qui sera utilisé.

Les deux composants Flash *List* et *comboBox* possèdent une propriété *editable* équivalente à la propriété *isEditable* de l'élément usiXML. Les règles de conversion peuvent donc être reprises en un seul tableau.

propriété usiXML	type usiXML	use	propr. Flash	type Flash
isDropDown	Boolean	optional	-	-
isEditable	Boolean	optional	editable	Boolean

TAB. 7.7 – Règles de correspondances de l'élément *comboBox* de usiXML vers Flash.

textComponent

L'élément textComponent de usiXML englobe toutes les formes possibles pour les cases texte. Il n'existe pas de composant aussi complet en Flash. Selon les propriétés de l'élément usiXML, il conviendra d'utiliser le composant TextInput (une ligne) ou l'élément TextArea (plusieurs lignes, barres de défilement,...).

En outre, en Flash, toutes les propriétés de mise en forme de texte se font par l'intermédiaire d'un formatage de texte (*TextFormat*). La *partie Flash* du tableau de correspondance 7.8 reprend donc des propriétés des trois objets *TextInput Component* (I), *TextArea Component* (A) et *TextFormat Object* (F). La dernière colonne indique, pour chaque propriété, l'objet Flash auquel elle appartient.

propriété usiXML	type usiXML	use	propriété Flash	type Flash	
defaultHyperLinkTarget	String	optional	-	-	-
hyperLinkTarget	String	optional	-	-	-
visitedLinkColor	Color	optional	-	-	-
activeLinkColor	Color	optional	-	-	-
isBold	Boolean	optional	bold	Boolean	F
isItalic	Boolean	optional	italic	Boolean	F
isUnderline	Boolean	optional	underline	Boolean	F
isStrikethrough	Boolean	optional	-	-	-
isSubscript	Boolean	optional	-	-	-
isSuperscript	Boolean	optional	-	-	-
isPreformatted	Boolean	optional	-	-	-
textColor	Color	optional	color	Number	F
textSize	Integer	optional	size	Number	F
textFont	String	optional	font	String	F
textMargin	Integer	optional	leftMargin	Number	F
textVerticalAlign	String	optional	-	-	-
textHorizontalAlign	String	optional	align	String	F
scrollStyle	String	optional	-	-	-
scrollDirection	String	optional	-	-	-
scrollHeight	Integer	optional	-	-	-
scrollWidth	Integer	optional	-	-	-
scrollHorizontalSpace	Integer	optional	-	-	-
scrollVerticalSpace	Integer	optional	-	-	-
scrollAmount	Integer	optional	-	-	-
scrollDelay	Integer	optional	-	-	-
maxLength	Integer	optional	-	-	-
numberOfColumns	Integer	optional	width	Number	I, A
numberOfLines	Integer	optional	height	Number	A
isPassword	Boolean	optional	password	Boolean	I, A
isWordWrapped	Boolean	optional	wordWrap	Boolean	A
forceWordWrapped	Boolean	optional	-	-	-
isEditable	Boolean	optional	editable	Boolean	I, A
filter	String	optional	-	-	-
defaultFilter	String	optional	restrict	String	I, A

TAB. 7.8 – Règles de correspondances du textComponent de usiXML.

7.2.4 Inexistence de composants Flash

Le *box* est le seul élément usiXML indispensable à toute génération d'interface qui ne soit pas présent parmi les composants de Flash.

Le *box* constitue l'un des éléments centraux de usiXML. Il s'agit d'un élément *réceptacle* pouvant contenir d'autre éléments.

Les propriétés du *box* qui permettent de définir l'alignement des éléments en son sein font de lui un élément pratique pour gérer la disposition des éléments de l'interface. Il sera dès lors indispensable d'implémenter un équivalent Flash du *box*.

Chapitre 8

Implémentation du générateur

8.1 Introduction

Nous avons analysé les méthodes existantes en matière de génération d'interfaces, à savoir : la compilation et les interpréteurs dynamiques orientés serveur. Afin de concilier les avantages de ces deux méthodes, nous avons opté pour une *architecture hybride*, baptisée FlashiXML. Celle-ci permet l'interprétation dynamique d'une description d'interface usiXML en Flash, soit :

- de manière indépendante, en se basant sur une précompilation des éventuelles fonctions externes utilisées par la description ;
- avec l'aide du langage orienté serveur PHP et de sa librairie MING afin de compiler dynamiquement les éventuelles fontions externes.

Enfin, dans le chapitre 7 nous avons décrit les règles de correspondances entre les balises du langage usiXML et les composants de Flash.

Ce chapitre décrit l'impplémentation de FlashiXML. Pour ce faire, les sections 8.2 et 8.3 décrivent la manière dont les données externes sont récoltées et gérées par FlashiXML. En effet, comme nous avons pu le constater dans le chapitre II, l'abstraction d'interface présente des avantages indéniables. En contrepartie, elle implique une étape préalable à la génération : la récupération des données externes. Ces données sont nombreuses et diverses. En effet, elles couvrent tous les aspects de l'interface :

- la configuration de FlashiXML ;
- les éléments graphiques de l'interface ainsi que les comportements qu'ils doivent adopter lors de la survenance d'un événement ;
- les fonctions externes utilisées par les éléments de l'interface ;
- la gestion du contenu des éléments graphiques en fonction de la langue

choisie par l'utilisateur.

Ensuite, la section 8.4 étudiera les propriétés des éléments graphiques de l'interface ainsi que la manière dont ils seront générés et positionnés dynamiquement.

En analysant la possibilité de décrire des interactions entre les différents composants, la section 8.5 apporte une dimension supplémentaire considérable à notre interpréteur.

Suite à cela et en fonction des desiderata, la solution hybride retenue pourra être plus ou moins *orientée serveur*. La section 8.6 est consacrée à une étude des différentes répartitions possibles ainsi qu'aux avantages et inconvénients de chacune d'entre elles.

La section 8.7, quant à elle, décrit la possibilité d'interpréter des interfaces multilingues et de proposer à l'utilisateur un choix de langue au lancement de l'application.

Enfin, les sections 8.8 et 8.9 présenteront le *gestionnaire des fenêtres* et le *gestionnaire des erreurs* qui augmentent l'ergonomie de FlashiXML.

8.2 Fichier de configuration

Le fichier de configuration est le premier fichier qu'il convient de récupérer et de parcourir lors du lancement de FlashiXML. Il contient en effet une série de paramètres décrivant le comportement de l'application par rapport à certaines fonctionnalités de cette dernière. Ce fichier de configuration constitue le seul point de coordination entre FlashiXML et le monde extérieur. De ce fait, il est primordial que les règles suivantes soient respectées :

- le nom du fichier et le chemin relatif depuis l'application FlashiXML doivent être prédéfinis ;
- le fichier de configuration doit être présent pour que l'application puisse s'exécuter ;
- les paramètres doivent être regroupés selon la syntaxe du squelette XML défini à la figure 8.1.

8.2.1 Paramètres du fichier de configuration

Paramètres système

Lors de son exécution, l'application doit être en mesure de localiser les différentes ressources nécessaires à une génération correcte de l'interface. Ce sont les *paramètres système* du fichier de configuration qui vont permettre

```
<config>
    <system ... paramètres systèmes ... />
    <languages ... paramètres de langues ... />
    <navigation ... paramètres de navigation ... />
    <items>
        <textComponent ... paramètres de l'item textComponent ... />
                                    .
                                    .
                                    .
        <imageComponent ... paramètres de l'item imageComponent ... />
    </items>
</config>
```

FIG. 8.1 – Squelette du fichier XML de configuration de FlashiXML

à FlashiXML de réaliser cette tâche. Le tableau 8.1 reprend la liste des *paramètres systèmes*, leur type et leur description.

paramètre	type	description
standalone	Boolean	indique si l'application fonctionne en mode serveur ("false") ou en mode *standalone* ("true")
usiXMLFiles	String	chemin complet du fichier reprenant la liste des fichiers usiXML exécutables
externalFunctionsFile	String	chemin complet du fichier contenant les scripts pouvant être associés aux éléments graphiques de l'interface

TAB. 8.1 – Paramètres systèmes du fichier de configuration.

standalone

Lorsque l'application est destinée à tourner sans serveur, par exemple, sur un CD-Rom de présentation, il convient d'attribuer la valeur *true* au paramètre *standalone* afin d'indiquer à FlashiXML qu'aucun serveur n'est disponible et qu'il faudra se passer de la librairie MING de PHP. Dans le cas contraire, le paramètre *standalone* se voit attribuer la valeur *false*. La section 8.5.4 étudie en détail les implications de la présence de la librairie MING de PHP.

usiXMLFiles

Ce paramètre constitue une alternative pemettant de pallier à l'un des

principaux inconvénients de Flash, à savoir l'absence de toute possibilité d'accès au système de fichiers. Flash ne permet en effet pas de proposer à l'utilisateur de choisir, lors du lancement de l'application, le fichier de description à partir duquel il désire générer l'interface (voir section 8.3, page 58).

Le paramètre usiXMLFiles contient le chemin complet d'un fichier XML que nous appellerons *files* dans la suite du document et dont la figure 8.2 illustre un exemple. Le fichier *files* reprend la liste des descriptions que l'on souhaite proposer à l'utilisateur avec pour chacune :

- le nom du fichier de description ;
- le chemin relatif du fichier de description ;
- le nom complet d'un fichier contenant les fonctions externes (*externalFunctionsFile*) utilisées par la description.

L'explication du paramètre *externalFunctionsFile* est la suivante. Le langage usiXML permet d'attribuer des scripts comme comportement d'un élément graphique déclenché à la suite d'un événement. Toutefois, vu l'objectif d'indépendance entre usiXML et le langage de génération (Flash), il convient de placer ces scripts dans un fichier séparé. Dès lors, lorsque l'interface générée fait appel à des fonctions externes, il convient de permettre à FlashiXML de trouver le fichier qui les contient. Pour ce faire, le paramètre *externalFunctionsFile* indique le chemin relatif du fichier contenant les scripts.

```
<usiXMLFiles>
    <file fileName="form_rw.xml" filePath="rw/" externalFunctionsFile="rw/fun.txt" />
    <file fileName="presentation.xml" filePath="" externalFunctionsFile="" />
    <file fileName="calculatrice.xml" filePath="" externalFunctionsFile="fun.php"/>
</usiXMLFiles>
```

FIG. 8.2 – Exemple de fichiers de références vers les descriptions usiXML proposées à l'utilisateur au lancement de FlashiXML

Lorsque le paramètre *standalone* a la valeur *false*, il est possible de demander au serveur de générer le fichier *files* dynamiquement, par exemple, en fonction des fichiers *.xml* trouvés dans le répertoire courant et/ou dans les sous-répertoires (voir figure 8.3). Dans le cas contraire, soit lorsque le paramètre *stanalone* indique *true*, le fichier *files* devra être édité manuellement pour chaque ajout ou suppression d'une description de la liste.

Configuration des langues

Dans l'hypothèse où une interface a été décrite en plusieurs langues dans un seul fichier usiXML, FlashiXML proposera, sauf contre-indication, à l'uti-

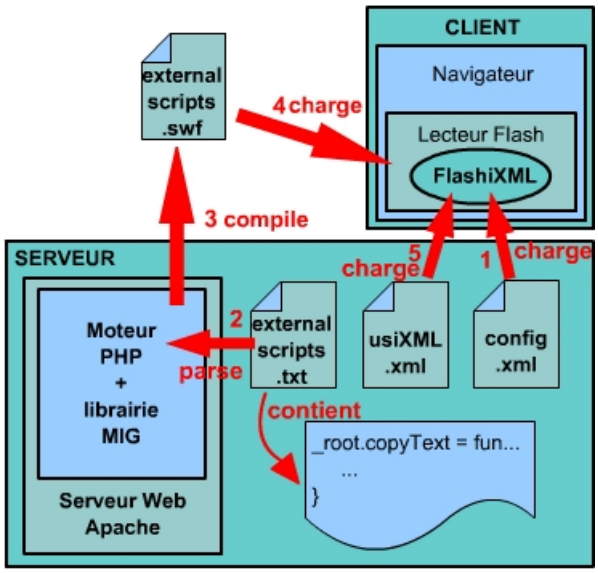

FIG. 8.3 – Génération dynamique d'un fichier de type usiXMLFiles

lisateur de choisir la langue dans laquelle il souhaite générer l'interface. Le paramètre *languageSelection* permet d'imposer à FlashiXML de ne pas proposer de choix de la langue à l'utilisateur. Le paramètre *languageNavigation* permet d'autoriser ou d'interdire à l'utilisateur de changer de langue en cours d'utilisation.

paramètre	type	description
languageSelection	Boolean	indique si le choix des langues doit (*true*) ou non (*false*) être proposé à l'utilisateur au lancement de l'application
languageNavigation	Boolean	indique si l'utilisateur est autorisé (*true*) ou non (*false*) à changer de langue en cours d'exécution de l'application

TAB. 8.2 – Paramètres de langue du fichier de configuration.

Gestion de la navigation

En fonction du type d'application générée et de ses objectifs, il sera parfois opportun de mettre certaines options d'affichage et de navigation à disposition de l'utilisateur. Certaines d'entre elles sont purement ergonomiques (telle que la possibilité ou non d'afficher l'application en *plein écran*),

d'autres sont informatives et peuvent servir à la détection d'erreurs ou au *versionning*. Le tableau 8.3 reprend les différents paramètres de la gestion de la navigation et leurs descriptions.

paramètre	type	description
displayWindowBar	Boolean	indique si l'on désire (*true*) ou non (*false*) afficher la barre des icônes des fenêtres réduites ou fermées
displayMenu	Boolean	indique si l'on désire (*true*) ou non (*false*) afficher le menu d'options
displayUsiXMLInformation	Boolean	indique si l'on désire (*true*) ou non (*false*) proposer, dans le menu, les informations contenues dans le fichier usiXML
displayFlashiXMLInformation	Boolean	indique si l'on désire (*true*) ou non (*false*) proposer, dans le menu, les informations concernant FlashiXML
displayErrorLog	Boolean	indique si l'on désire (*true*) ou non (*false*) proposer, dans le menu, l'affichage des erreurs de génération
allowFileReloading	Boolean	indique si l'on désire (*true*) ou non (*false*) proposer, dans le menu, de charger une autre description d'interface en cours d'exécution
displayFullScreen	Boolean	indique si l'on désire (*true*) ou non (*false*) que l'application soit exécutée en plein écran
allowFullScreen	Boolean	indique si l'on désire (*true*) ou non (*false*) proposer la possibilité de basculer du mode plein écran au mode normal et vice-versa.

Tab. 8.3 – Paramètres de navigation du fichier de configuration.

Paramètres des éléments graphiques de l'interface

La grande portabilité de usiXML réside, en partie, dans la sous-spécification de certaines propriétés telles les tailles et positions des éléments graphiques de l'interface. Toutefois, ceci implique que le générateur soit de temps à autre contraint de prendre des décisions afin de parvenir à un rendu graphique cohérent. Les paramètres de cette section du fichier de configuration visent à aider le générateur dans la prise de certaines décisions.

Il convient, afin de bien comprendre ceci, de s'arrêter aux deux exemples suivants :

Le composant de texte de usiXML (textComponent)
Théoriquement, ce dernier est décrit non pas par une largeur et une hauteur en pixels, mais bien en nombre de colonnes et de lignes. L'idée implicite qui réside derrière ce concept est d'attribuer à une colonne une largeur et, à une ligne, une hauteur proches de celles d'un caractère. Bien que très simple, cette solution n'est pas implémentable sans avoir préalablement défini une taille moyenne de caractère. La largeur et la hauteur d'un caractère dépendent de la police d'écriture, de la taille de cette police, de la résolution d'écran, du système d'exploitation, de la plate-forme, ... En outre, même lorsque tous ces paramètres sont définis, il reste des différences de taille entre les caractères eux-mêmes. Ainsi, des caractères tels que i et w n'ont pas du tout la même largeur. Calculer une taille moyenne des caractères est encore rendu plus difficile du fait que la fréquence de chacun d'entre eux peut varier fortement d'une langue à une autre.

Afin de ne pas être contraint de fixer la largeur moyenne d'un caractère, celle-ci sera entreposée dans un *paramètre des éléments graphiques*. Lors du lancement de l'application, FlashiXML chargera le paramètre et l'utilisera pour la génération des *textComponent*.

Le composant image de usiXML (imageComponent)
L'idéal, pour respecter la puissance de la portabilité de usiXML est que le générateur affiche l'image dans sa résolution en dehors du générateur. Cela permet de ne pas devoir changer les propriétés *hauteur* et *largeur* de sa balise, dans le fichier usiXML lorsque l'application est transposée d'une plateforme à une autre. En effet, il suffit alors de remplacer le fichier source de l'image. Malheureusement, Flash ne permet pas de détecter la taille d'une image chargée dynamiquement. Il est donc pratique de pouvoir indiquer au générateur une taille d'image par défaut afin de ne pas devoir modifier la source usiXML[1]. Toutefois, définir une taille par défaut peut être très contraignant lorsque cette taille par défaut n'est pas modifiable facilement. Une fois de plus, les *paramètres des éléments graphiques* apportent une solution au problème en permettant d'entreposer cette taille par défaut dans un paramètre extérieur à FlashiXML, plutôt que dans une constante interne.

Bien que les *paramètres des éléments graphiques* soient peu utilisés à ce stade du développement de FlashiXML, ils apportent une grande flexibilité qu'ils apportent. En outre, dans une multitude de situations, ces paramètres des éléments graphiques pourraient permettre de garder une plus grande

[1] voir section 8.4.3 page 65

souplesse par rapport à une constante programmée de manière immuable au sein du générateur.

8.2.2 Parsing du fichier de configuration

Etant donné l'importance des informations contenues dans le fichier de configuration, il est nécessaire que ce fichier :

- soit présent à un emplacement spécifique ;
- soit syntaxiquement correct ;
- reprenne les informations indispensables.

Lors du chargement de ce fichier, le générateur récupère les informations qu'il contient et les confie au ConfigManager. Le ConfigManager est accessible par tous les autres composants du générateur tout au long de la génération. Son rôle est de répondre à toutes les questions relatives aux paramètres de la configuration.

Par souci de robustesse, si certaines informations cruciales venaient à manquer, elles seraient complétées par l'application au moyen de paramètres par défaut. Cependant, le bon fonctionnement de l'interface pourrait être altéré.

8.3 Parsing du fichier usiXML

Comme nous l'avons analysé dans la section 4.4.1, la balise *uiModel* de usiXML comprend l'entièreté de l'interface. Ceci implique que tout élément qui lui serait externe ne sera pas pris en considération par le générateur.

Le chargement du fichier usiXML et l'extraction des données qu'il contient se décomposent de la manière suivante :

1. Chargement du fichier usiXML
2. Récupération de la balise uiModel sous forme d'objet XML
3. Récupération des paramètres de la balise uiModel
    - paramètre de création du fichier usiXML
    - paramètre de la version du schéma usiXML
    - liens vers les spécifications usiXML
4. Gestion des balises enfant de la balise uiModel
    - gestion des balises d'informations
        - balise indiquant la version du fichier usiXML
        - balise contenant l'auteur de la description
        - balise comprenant d'éventuels commentaires complémentaires
    - gestion des balises de type cuiModel
    - gestion des balises de type contextModel
    - gestion des balises de type resourceModel

Le pseudo-code de la figure 8.4 illustre l'extraction des différentes balises enfant de l'objet XML uiModel ainsi que la délégation à des fonctions spécifiques de l'analyse de leur contenu.

Les données collectées dans les différentes *balises d'informations* seront mises à la disposition de l'utilisateur par le biais du menu, selon les paramètres du fichier de configuration de l'application.

```
childNodes = uiModel.getChildNodes();

for(var i in childNodes){
      switch (childNodes[i].nodeName){
            case "cuiModel":
                  cuiModelParser(rootNodes[i]);
            case "contextModel":
                  contextModelParser(rootNodes[i]);
            case "resourceModel":
                  resourceModelParser(rootNodes[i]);
            case "version" :
                  informationParser(childNodes[i]);
            case "authorName" :
                  informationParser(childNodes[i]);
            case "comment" :
                  informationParser(childNodes[i]);
            default:
                  ErrorManager.addErrorMessage("unKnown node found", childNode[i]);
      }
}
```

FIG. 8.4 – Pseudo code de la gestion des balises enfant de l'objet XML uiModel

8.4 Génération des éléments graphiques de l'interface

8.4.1 Introduction

L'objectif du générateur étant de créer le rendu graphique d'une interface décrite en usiXML, il est naturel que la structure hiérarchique des objets de FlashiXML soit calquée sur celle de usiXML.

Comme nous l'avons vu à la section 7 page 42, les éléments graphiques les plus courants du langage usiXML disposent d'un équivalent parmi les composants de Flash[2]. Il convient donc principalement d'assurer la conversion des propriétés des éléments d'un langage à l'autre. Pour ce faire, FlashiXML ajoute, au dessus de chaque élément une couche servant de *convertisseur*. Cette couche s'intitule comme le composant mais son nome est précédé du préfixe *FUI*, abréviation de *Flash User Interface*. La figure 8.5 illustre la hiérarchie des éléments graphiques de FlashiXML.

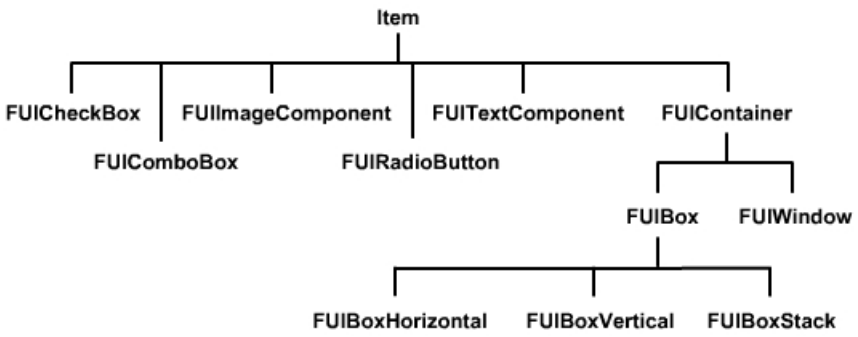

Fig. 8.5 – Hiérarchie des éléments graphiques de FlashiXML

La section 8.4.2 commence par reprendre les propriétés communes à tous les éléments graphiques implémentés dans FlashiXML. Ensuite, la section 8.4.3 détaille tour à tour les propriétés spécifiques de chaque élément.

Le *CIOManager*[3] est responsable de la gestion de tous les éléments graphiques. La section 8.4.4 analyse comment chaque référence à un élément de l'interface se fait par son biais.

[2]Pour plus de renseignements sur les composants de Flash, le lecteur peut se référer à la section 5.3

[3]Concrete Interaction Object Manager

8.4.2 Propriétés communes des éléments graphiques

Le tableau 8.4 distingue les propriétés communes à tous les éléments graphiques.

Afin d'assurer un rendu visuel correct de FlashiXML la propriété *id* doit impérativement exister et être unique pour chaque élément. Les autres propriétés ne sont par contre pas indispensables. Dans l'hypothèse où les propriétés *isEnabled* ou *isVisible* ne seraient pas indiquées, elles prendraient automatiquement la valeur *true*. Enfin, pour certains éléments, la largeur et/ou la hauteur pourront être calculées par l'application en fonction d'autres propriétés.

propriété usiXML	type	description
height	Integer	hauteur de l'élément en pixels
id	Boolean	identifiant unique de l'élément
isEnabled	Boolean	indique si l'élément est actif
isVisible	Boolean	indique si l'élément est visible
width	Integer	largeur de l'élément en pixels

TAB. 8.4 – Evénements du bouton implémentés.

Tous les éléments graphiques finaux, c'est-à-dire, les éléments qui n'héritent pas de *FUIContainer* dans la figure 8.5, possèdent également les propriétés *glueHorizontal* et *glueVertical* qui permettent de préciser leur alignement par rapport à la place dont ils disposent dans le *box*.

8.4.3 Propriétés spécifiques des éléments graphiques

Button

Propriétés implémentées

Les propriétés du bouton implémentées sont peu nombreuses par rapport aux possibilités offertes par usiXML. Elles comprennent bien entendu la propriété *defaultContent* qui correspond au texte à afficher sur le bouton.

Au stade actuel du développement de FlashiXML, la hauteur de tous les boutons est fixée à une même constante. La largeur, quant à elle, est calculée dynamiquement en fonction du nombre de caractères que comprend le texte du bouton.

La figure 8.6 illustre ce principe par deux boutons portant respectivement les textes *bouton* et *très long bouton*.

FIG. 8.6 – Illustration des propriétés de l'élément graphique *bouton*

Evénements implémentés

Le tableau 8.5 reprend les différents événements du bouton implémentés dans FlashiXML. Il s'agit des événements associés aux boutons dans la majorité des langages de programmation. La figure 8.7 illustre un exemple de code usiXML définissant des actions pour chaque événement possible d'un bouton.

nom usiXML	nom Flash	description
depress	onPress	le bouton est enfoncé
release	onRelease	le bouton est relâché
rollOver	onRollOver	la souris survole le bouton
rolOut	onRollOut	la souris sort de la zone du bouton

TAB. 8.5 – Evénements du bouton implémentés.

```
<button id="btn1" defaultContent="btn1">
    <behavior id="behav1">
        <event id="evt1" eventType="depress" eventContext="btn1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav2">
        <event id="evt2" eventType="release" eventContext="btn1">
        <action id="act2">  ...  </action>
    </behavior>
    <behavior id="behav3">
        <event id="evt1" eventType="rollOver" eventContext="btn1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav4">
        <event id="evt2" eventType="rollOut" eventContext="btn1">
        <action id="act2">  ...  </action>
    </behavior>
</button>
```

FIG. 8.7 – Code usiXML définissant des actions pour chaque événement possible d'un bouton.

CheckBox

Propriétés implémentées

Comme pour les boutons, la propriété *defaultContent* de l'élément *checkBox* fait référence au texte qui accompagne la case à cocher. Ici également, la largeur doit être calculée dynamiquement en fonction du nombre de caractères de *defaultContent*.

La propriété *defaultState* permet d'indiquer si la *checkBox* est initialement sélectionnée.

FIG. 8.8 – Illustration des propriétés de l'élément graphique *checkBox*

Evénements implémentés

Le tableau 8.5 reprend les deux événements associés aux *checkBox* que FlashiXML supporte. Flash ne prévoit toutefois qu'un seul événement. Il convient dès lors d'accompagner cet événement d'un test sur la valeur de la sélection. La figure 8.9 est un exemple de code usiXML définissant des actions pour chacun de ces événements.

Nom usiXML	Nom Flash	Description
select	click	le *checkBox* est sélectionné
unselect	click	le *checkBox* est désélectionné

TAB. 8.6 – Evénements implémentés du *checkBox*.

```
<checkBox id="cb1" defaultContent="yes">
    <behavior id="behav1">
        <event id="evt1" eventType="select" eventContext="cb1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav2">
        <event id="evt2" eventType="unselect" eventContext="cb1">
        <action id="act2">  ...  </action>
    </behavior>
</checkBox>
```

FIG. 8.9 – Code usiXML définissant des actions pour les deux événements possibles d'une *checkBox*.

ComboBox

Propriétés implémentées

L'élément *comboBox* peut être affiché soit sous forme de liste, soit sous forme de liste déroulante. La propriété *isDropDown* permet de définir l'affichage désiré. Dans les deux cas, la largeur du *comboBox* sera définie en nombre de caractères, comme c'est le cas pour le bouton. Lorsque le *mode liste déroulante* est préféré, la hauteur sera fixée par FlashiXML. Dans le cas contraire (*mode liste*), la hauteur sera définie en nombre de lignes. Le tableau 8.7 reprend les propriétés de l'élément *comboBox*. La figure 8.10 illustre les propriétés du *comboBox*.

propriété	type	description
isDropDown	Boolean	indique si l'élément doit être affiché sous forme de liste déroulante (*true*) ou sous forme de liste (*false*)
numberOfLines	Integer	précise la largeur d'affichage souhaitée de l'élément
numberOfColons	Integer	précise la hauteur d'affichage souhaitée lorsque la propriété isDropDown est mise à *false*

TAB. 8.7 – Propriétés propres au *comboBox* implémentées.

FIG. 8.10 – Illustration des propriétés de l'élément graphique *comboBox*

Bien que le *comboBox* soit considéré comme un élément graphique terminal[4], il contient des balises *item* permettant de préciser le contenu à afficher dans la liste ainsi que les valeurs cachées associées. La figure 8.11 montre le code d'une liste déroulante affichant les choix *Vrai* et *Faux* dont les valeurs associées sont, respectivement, *true* et *false*.

Evénements implémentés

Le seul événement implémenté dans le cadre du *comboBox* est le *onChange* qui se déclenche lorsque l'utilisateur sélectionne un *item* différent de la liste

[4] c'est-à-dire qu'il n'agit pas comme un réceptacle capable de contenir d'autres éléments graphiques

```
<comboBox id="combo1" isDropDown="false">
    <item id="item1" defaultContent="Vrai" value="true" />
    <item id="item2" defaultContent="Faut" value="false" />
</comboBox>
```

FIG. 8.11 – Code associant des valeurs à afficher dans une liste

(ou de la liste déroulante). De par la définition même de la liste déroulante, cette sélection entraînera la désélection de l'item précédemment sélectionné. Il est à noter que, théoriquement, la liste simple (non déroulante) devrait permettre une sélection multiple lorsque la propriété correspondante est active. Cette propriété n'étant pas, actuellement, implémentée dans FlashiXML, le *comboBox* réagira de la même manière pour les deux types de listes.

L'événement *onChange* est le seul à pouvoir déclencher des actions, il n'y a dès lors pas de raison de placer plusieurs balises *behavior* dans un *comboBox*. La figure 8.12 illustre le fait que l'unique balise *behavior* se trouve sur le même pied que les balises *item*.

```
<checkBox id="cb1" defaultContent="yes">
    <behavior id="behav1">
        <event id="evt1" eventType="select" eventContext="cb1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav2">
        <event id="evt2" eventType="unselect" eventContext="cb1">
        <action id="act2">  ...  </action>
    </behavior>
</checkBox>
```

FIG. 8.12 – Code usiXML définissant des actions pour l'événement *onChange* du *comboBox*.

ImageComponent

Propriétés implémentées

L'*imageComponent* est un réceptacle destiné à accueillir une image. Il est, avec l'élément *window*, le seul qui nécessite une hauteur et une largeur définies en pixels, bien que cela aille à l'encontre de la portabilité de usiXML. Flash ne permet, en effet, pas de dériver la taille d'une image à partir du fichier source. Le tableau 8.8 reprend les propriétés de l'*imageComponent*.

Evénements implémentés

Le tableau 8.9 reprend les événements de l'*imageComponent* implémentés

propriété	type	description
defaultContent	String	indique le nom et le chemin relatif du fichier source de l'image
defaultLink	String	si non vide, contient une référence vers une page web à afficher lorsque l'utilisateur clique sur l'image
height	Integer	précise la hauteur d'affichage souhaitée de l'image
width	Integer	précise la largeur d'affichage souhaitée de l'image

TAB. 8.8 – Propriétés propres à l'élément *imageComponent* implémentées.

dans FlashiXML. Il s'agit des mêmes événements que pour le *buttonComponent*. Il est important de noter que, si l'événement *release* est utilisé, la propriété *defaultLink* n'est plus prise en compte par FlashiXML. La figure 8.13 reprend un exemple de code usiXML définissant des actions pour chaque événement possible de l'*imageComponent*.

Nom usiXML	Nom Flash	Description
depress	onPress	le bouton de la souris est enfoncé sur l'image
release	onRelease	le bouton de la souris est relâché après avoir été enfoncé sur l'*imageComponent*
rollOver	onRollOver	la souris survole l'*imageComponent*
rolOut	onRollOut	la souris sort de la zone de l'*imageComponent*

TAB. 8.9 – Evénements de l'*imageComponent* implémentés.

```
<imageComponent id="I1" defaultContent="I1">
    <behavior id="behav1">
        <event id="evt1" eventType="depress" eventContext="I1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav2">
        <event id="evt2" eventType="release" eventContext="I1">
        <action id="act2">  ...  </action>
    </behavior>
    <behavior id="behav3">
        <event id="evt1" eventType="rollOver" eventContext="I1">
        <action id="act1">  ...  </action>
    </behavior>
    <behavior id="behav4">
        <event id="evt2" eventType="rollOut" eventContext="I1">
        <action id="act2">  ...  </action>
    </behavior>
</imageComponent>
```

FIG. 8.13 – Code usiXML définissant des actions pour chaque événement possible de l'*imageComponent*.

RadioButton

Propriétés implémentées

Le fonctionnement du *radioButton* est très proche de celui du *checkBox*. En effet, ici également, la propriété *defaultContent* fait référence au texte qui accompagne le bouton radio. Tout comme pour les *boutons* et les *checkBox*, la largeur de la zone de texte doit être calculée dynamiquement en fonction du nombre de caractères de la propriété *defaultContent*.

Le *radioButton* se distingue toutefois des autres éléments car il n'a de sens qu'accompagné d'au moins un autre *radioButton*. En effet, comme dans la majorité des langages, les *radioButton* de Flash doivent appartenir à un *radioButtonGroup*.

Bien qu'il existe aujourd'hui un attribut *groupName* pour le *radioButton* de usiXML, ce n'était pas le cas jusqu'il y a peu. C'était alors le nom du *box*[5] parent du *radioButton* qui était utilisé comme nom de groupe. FlashiXML fonctionne encore suivant cette méthode, ce qui entraine deux conséquences importantes :

1. tous les *radioButton* liés[6] doivent avoir le même *box* comme parent

[5] la section 8.4.3 page 72 étudie en détail les *box* qui sont des éléments graphiques *réceptacles*, pouvant contenir d'autres éléments

[6] c'est-à-dire, des *radioButton* que Flash doit considérer comme appartenant au même groupe

direct ;

2. deux *radioButtonGroup* ne peuvent avoir le même parent direct, c'est-à-dire que deux *radioButton* non liés ne peuvent avoir le même *box* comme parent direct.

La propriété *defaultState* permet d'indiquer si le *radioButton* doit être initialement sélectionné. Etant donné que, par définition, un seul *radioButton* par *radioButtonGroup* peut être sélectionné simultanément, lorsque l'attribut *defaultState* est mis à *selected* pour plusieurs *radioButton* du même *radioButtonGroup*, FlashiXML sélectionnera le dernier d'entre eux.

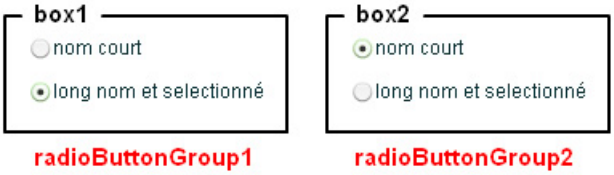

FIG. 8.14 – Illustration des propriétés de l'élément graphique *radioButton*

Evénements implémentés

Le seul événement implémenté pour les *radioButton* est le *select* qui se déclenche lorsque l'utilisateur sélectionne un *radioButton*. Rappelons toutefois que, par définition, ceci aura pour conséquence de désélectionner le *radioButton* du même *radioButtonGroup* précédemment sélectionné.

```
<radioButton id="rb1" defaultContent="yes">
    <behavior id="behav1">
        <event id="evt1" eventType="select" eventContext="rb1">
        <action id="act1">  ...  </action>
    </behavior>
</radioButton>
```

FIG. 8.15 – Code usiXML définissant des actions pour l'événement *select* du *radioButton*.

TextComponent

Propriétés implémentées

Le *textComponent* de usiXML est un élément assez complexe représentant toutes les formes possibles de cases textes. Comme nous l'avons signalé dans la section 7 (page 42), il n'existe pas d'équivalent en Flash du *textComponent* de usiXML. En fonction de ses propriétés, le *textComponent* sera représenté

par différents composants de Flash. Le FUITextComponent de la figure 8.5 est un composant, conçu spécialement pour FlashiXML, qui regroupe les différents composants de Flash qui devront être utilisés afin de couvrir toutes les propriétés possibles du textComponent. La figure 8.16 illustre différents rendus possibles du textComponent. Le tableau 8.11 reprend, quant à lui, toutes les propriétés possibles du textComponent.

FIG. 8.16 – Illustration de différents rendus possibles du *textComponent*.

Evénements implémentés

Le seul événement du *textComponent* implémenté dans FlashiXML est l'événement *onChange* qui se déclenche lorsque le contenu de la case texte est modifié.

propriété	type	description
activeLinkColor	String	uniquement utilisé lorsque linkTarget est défini. Indique la couleur du texte lorsque l'utilisateur clique dessus.
bgColor	String	couleur du fond de la case texte
borderColor	String	couleur de la bordure de la case texte
borderWidth	Integer	indique si la bordure est visible (*tout entier non nul*) ou non (*0*).
content	String	texte à afficher dans la case texte
fgColor	String	couleur du texte de la case texte
isBold	Boolean	indique si le texte doit être affiché en gras (*true*) ou non (*false*).
isEditable	Boolean	indique si l'utilisateur peut éditer le le texte de la case texte (*true*) ou non (*false*).
isItalic	Boolean	indique si le texte doit être affiché en italique (*true*) ou non (*false*).
isPassword	Boolean	indique si le texte doit être remplacé par des * (*true*) ou non (*false*).
isUnderline	Boolean	indique si le texte doit être souligné (*true*) ou non (*false*).
isWordWrapped	Boolean	indique si le texte doit passer à la ligne lorsqu'il dépasse la taille de la case texte (*true*) ou non (*false*). Cette propriété n'est prise en considération que lorsque la propriété *numberOfLines* a une valeur plus grande que *1*.
itemHeight	Integer	hauteur de la case texte en pixels
itemWidth	Integer	largeur de la case texte en pixels
linkTarget	String	lien vers un fichier distant à afficher lorsque l'utilisateur clique sur la case texte.
numberOfColumns	Integer	largeur en nombre de colonnes de caractères
numberOfLines	Integer	hauteur en nombre de lignes de caractères
textAlign	String	indique si le texte doit être aligné horizontalement à gauche (*left*), à droite (*right*) ou au centre (*center*)
textFont	String	indique la police de caractère à utiliser pour le texte
textSize	Integer	indique la taille de la police de caractère

TAB. 8.11 – Propriétés propres au *textComponent* implémentées.

Window

Propriétés implémentées

L'élément *window* est un composant particulier car :

- il est le seul à pouvoir être instancié dans la racine de l'application. De ce fait, un élément *window* au moins devra être présent dans chaque description usiXML ;
- il ne peut être instancié que dans la racine de l'application. Un élément *window* ne pourra dès lors pas contenir d'autres éléments *window* ;
- il ne peut avoir qu'un seul élément fils direct. En outre, cet élément fils doit impérativement être un box (voir section 8.4.3).

La propriété *defaultContent* est la seule propriété implémentée propre à l'élément *window*. Elle permet de définir le titre à afficher dans la barre de titre de la fenêtre.

A l'instar de l'élément *imageComponent*, FlashiXML requiert une valeur valide pour les propriétés *height* et *width* de l'élément *window*.

Comme nous l'avons vu dans la section 7 l'élément *window* de usiXML est concrétisé en FlashiXML par le *window Component* de Flash. Ce dernier est peu personnalisable, ce qui entraine les conséquences suivantes :

- bien que usiXML permette de préciser si l'élément *window* est redimensionnable par l'utilisateur, cette propriété n'a pas pu être implémentée ;
- tout élément *window* possède un bouton dans la *barre de titre* mettant la propriété *isVisible* de l'élément à *false* ;
- tout élément *window* de FlashiXML peut être déplacé en cliquant sur la *barre de titre* ;
- l'affichage des barres de défilement est géré automatiquement par Flash en fonction de la taille et de la position des éléments contenus dans la fenêtre. La figure 8.17 illustre une fenêtre avec et sans barres de défilement.

Fig. 8.17 – Illustration de l'élément graphique *window*

Evénements implémentés

Aucun événement particulier n'est spécifiable pour l'élément *window*.

Box

L'élément graphique *box* représente un réceptacle pouvant contenir d'autres éléments graphiques. Etant donné qu'il n'existe pas d'équivalent de l'élément *box* en Flash (voir section 7), il a fallu l'implémenter entièrement dans FlashiXML.

Les éléments *box* et *window* constituent les deux seuls éléments indispensables d'une interface décrite en usiXML. En effet :

- l'élément *window* est le seul à pouvoir être instancié dans la racine de l'application ;
- l'élément *box* est le seul fils possible de l'élément *window*.

Il en découle directement que le *box* est le seul élément à pouvoir contenir les autres types d'éléments. Toutefois, un élément *box* peut contenir d'autres éléments *box*.

Bordures des box

La zone couverte par un *box* peut être délimitée par une bordure et illustrée par un titre. La figure 8.18 illustre le fait que lorsqu'une bordure est affichée, la taille du *box* diffère de la taille disponible pour les éléments qu'il contient.

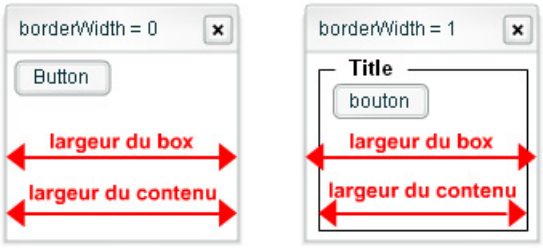

FIG. 8.18 – Illustration de la possibilité d'afficher une bordure autour d'un *box*.

Types de box

La manière dont les éléments contenus dans un box sont alignés est déterminée par le type du box. La figure 8.19 illustre les trois types de box possibles : *horizontal*, *vertical* et *stack*.

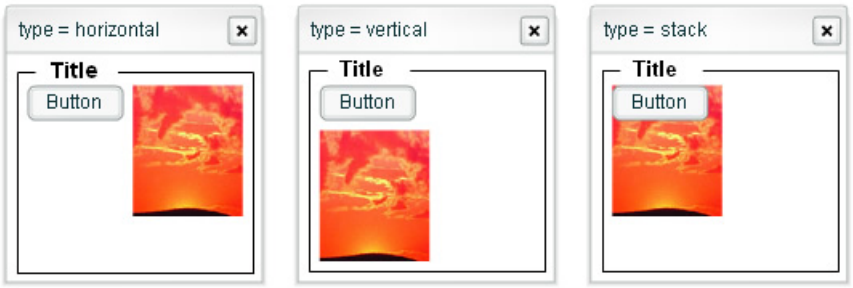

FIG. 8.19 – Illustration des types de *box* possibles : *horizontal*, *vertical* et *stack*.

Dimensionnement des box

Afin de bénéficier au maximum de la modularité offerte par usiXML, la taille d'un *box* sera généralement décrite en fonction de la taille de son parent (*relativeHeight* et *relativeWidth*). Les situations suivantes (illustrées par la figure 8.20) peuvent se présenter :

- Lorsqu'aucune taille n'est indiquée, le *box* prend la taille minimum dont il a besoin pour afficher correctement les éléments qu'il contient ;
- Lorsqu'une taille relative à celle du parent est indiquée et que cette dernière est suffisante pour afficher les éléments contenus dans le *box*, le *box* prend la taille indiquée ;
- Lorsqu' une taille relative à celle du parent est indiquée mais que celle-ci est insuffisante pour afficher les éléments contenus dans le *box*, elle ne sera pas respectée. En effet, FlashiXML donne priorité à un affichage correct des éléments. Il préfèrera donc agrandir le *box* plutôt que de respecter la taille relative.

La section 8.4.4 aborde, dans le détail, le procédé récursif de calcul de la taille et de la position des *box* en fonction des éléments qu'ils contiennent lorsque plusieurs *box* sont imbriqués.

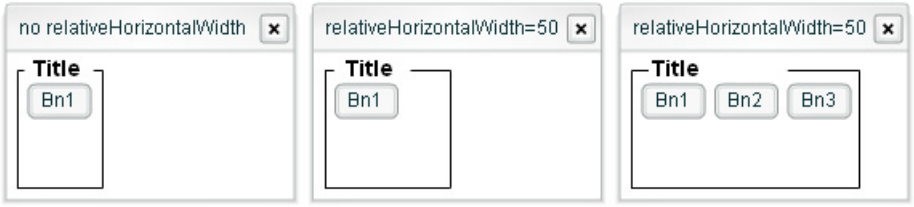

FIG. 8.20 – Illustration du dimensionnement d'un *box* en fonction d'une taille relative par rapport au parent et du contenu du *box*.

Imbrication de box

Comme l'illustr la figure 8.21, il est possible d'imbriquer des *box* l'un dans l'autre et ce, quel que soit le type des différents *box*. Toutefois, un *box* pourra contenir soit d'autres *box*, soit d'autres éléments terminaux[7] mais jamais une combinaison des deux.

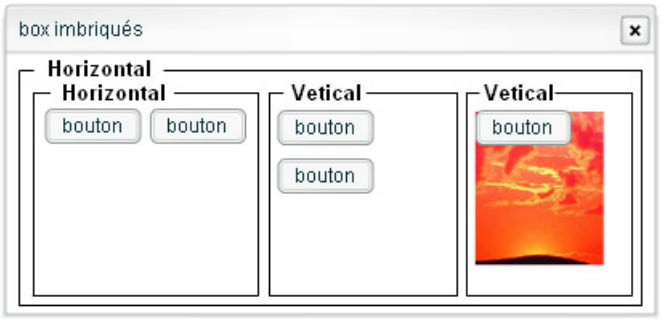

FIG. 8.21 – Illustration de l'imbrication de *box* de types différents.

Justification des éléments terminaux

Alors que le type du *box* définit l'alignement de ses fils, la propriété *isBalanced* permet d'indiquer si les éléments terminaux doivent être disposés :

- l'un à la suite de l'autre
- en fonctions de positions qui doivent être justifiées par rapport à place disponible dans le *box*.

La figure 8.22 illustre l'effet de la propriété *isBalanced* dans les différents types de *box*.

Le tableau 8.12 reprend toutes les propriétés propres aux *box* ainsi que leur type et leur description.

Evénements implémentés

Aucun événement particulier n'est spécifiable pour l'élément *window*.

[7] élément qui n'hérite pas de FUIContainer dans la figure 8.5

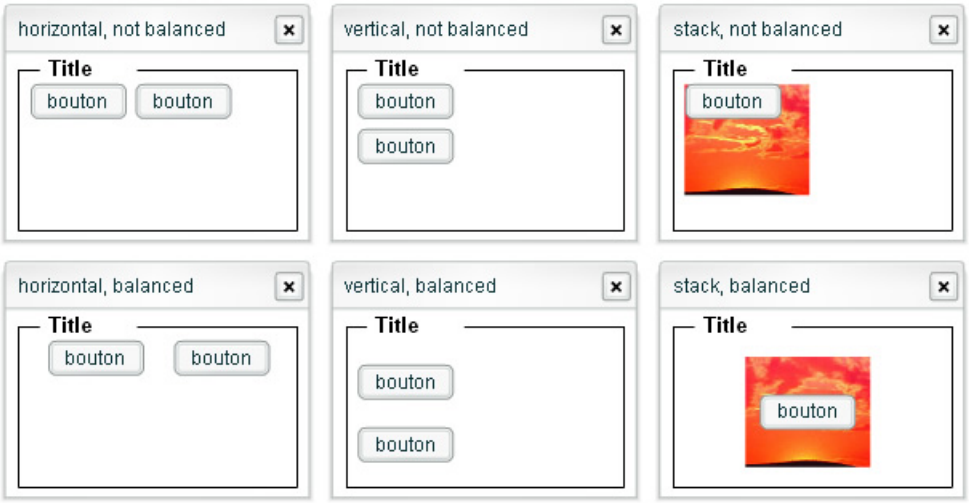

FIG. 8.22 – Illustration de l'effet de la propriété *isBalanced* dans différents types de *box*.

propriété	type	description
border	Integer	indique si la bordure doit être affichée (*tout entier positif*) ou pas *0* La largeur de la bordure ne dépend pas de la valeur indiquée.
borderTitle	String	titre à afficher lorsque la bordure du *box* est visible.
isBalanced	Boolean	indique si les positions des éléments fils sont justifiées (*true*) ou si les éléments sont placés l'un directement derrière l'autre (*false*)
relativeHeight	Integer	hauteur relative du *box* par rapport à la hauteur du contenu de son parent
relativeWidth	Integer	largeur relative du *box* par rapport à la largeur du contenu de son parent
type	String	indique si les éléments du *box* sont alignés horizontalement (*horizontal*), verticalement (*vertical*) ou l'un sur l'autre (*stack*).

TAB. 8.12 – Propriétés propres à l'élément *box*.

8.4.4 Instanciation et positionnement des éléments graphiques

Introduction

Comme nous l'avons souligné dans la section 8.4.3, il est fréquent que la taille des éléments graphiques ne soit pas précisée dans le fichier usiXML. Elle doit alors être calculée dynamiquement. C'est, par exemple, le cas de l'élément *buttonComponent* dont la largeur s'adapte à la longueur du texte qu'il contient.

En outre, la taille des *box* dépend, d'une part de la taille de tous les éléments qu'ils contiennent et, d'autre part, de la taille relative souhaitée par rapport au parent. Ceci complique considérablement le calcul de la taille et de la position des *box* et, par conséquent, le calcul de la position de tous les éléments contenus dans les *box*.

Cette section décrit en détail les trois étapes indispensables au positionnement des éléments graphiques :

1. l'instanciation de tous les éléments graphiques ;
2. le calcul de la taille des *box* ainsi que de leur position ;
3. le positionnement des éléments graphiques terminaux au sein de leur *box* parent.

Instanciation des éléments graphiques

Le CIOManager [8] est l'élément de FlashiXML responsable de toutes les instances graphiques de l'interface. Lui seul est autorisé à créer des éléments graphiques. Par conséquent, toute création d'élément graphique devra se faire par son intermédiaire.

Lorsque FlashiXML a terminé de charger le fichier usiXML dont il faut générer l'interface, il le décortique afin d'en extraire toutes les balises *window* et les accumule dans un tableau. Une fois toutes les balises récupérées, le tableau est envoyé au CIOManager.

Le CIOManager parcourt une à une les balises du tableau. Pour chacune d'entre elles, il instancie une fenêtre et lui transmet la balise qui la décrit. Le CIOManager garde en mémoire une référence vers la fenêtre instanciée.

Lorsqu'une fenêtre est instanciée, elle récupère la balise *box* qu'elle contient et la transmet au CIOManager afin que ce dernier instancie le *box* corres-

[8] Concrete Interaction Object Manager

pondant[9]. Le CIOManager récupère la référence vers le *box* et la garde en mémoire. Il envoie également une copie de cette référence à la fenêtre afin qu'elle puisse aussi la retenir.

Lorsqu'un *box* est instancié, il récupère toutes ses balises fils. Il envoie tour à tour chacune d'elle au CIOManager afin que ce dernier instancie les éléments correspondants. Une fois de plus le CIOManager garde en mémoire la référence de chaque élément instancié et en transmet une copie au *box* parent.

Lorsque tous les descendants d'une fenêtre ont été instanciés, le CIOManager récupère la balise fenêtre suivante dans son tableau et instancie la fenêtre correspondante.

C'est ainsi que l'interface se crée élément par élément. Chaque fois que le CIOManager crée un élément graphique, il lui transmet la balise qui décrit l'élément en question ainsi que tous ses éléments fils. Dès sa création, chaque élément graphique compulse la balise qu'il s'est vu attribuer. Pour chaque élément fils trouvé dans sa balise, l'élément émet une requête d'instanciation au CIOManager en lui transmettant la balise fils en question. Chaque élément garde une référence vers tous ses éléments fils. Le CIOManager, quant à lui, garde une référence vers tous les éléments de l'interface. De ce fait, il dispose d'une vue globale de toute l'interface.

La figure 8.24 illustre les étapes suivies lors de la génération des éléments graphiques d'une description usiXML contenant les deux balises *window* simplifiées resprises à la figure 8.23. Ci-dessous, une explication plus détaillée des douze premières étapes[10] :

1. lorsque la description usiXML de la figure 8.24 est entièrement chargée, toutes les balises *window* en sont extraites et sont entreposées dans un tableau ;
2. le tableau de balises *window* est envoyé au CIOManager ;
3. le CIOManager instancie la *fenêtre W0* et garde une référence vers l'objet ainsi créé ;
4. la *fenêtre W0* demande au CIOManager d'instancier le *box B0* dont elle a trouvé la description dans sa balise ;
5. le CIOManager instancie le *box B0* et garde une référence vers l'objet créé ;

[9]Comme étudié à la section 8.4.3 un élément *window* ne peut contenir qu'un, et un seul, fils direct qui doit être de type *box*

[10]les étapes 13 à 22 sont équivalentes aux étapes 3 à 12 où il suffit de remplacer W0 par W1, B0 par B1, T0 par T1 et BT0 par BT1

6. le CIOManager envoie à la *fenêtre W0* une référence vers le *box BO* ;
7. le *box B0* demande au CIOManager d'instancier le *textComponent T0* dont il a trouvé la description dans sa balise ;
8. le CIOManager instancie le *textComponent T0* et garde une référence vers l'objet créé ;
9. le CIOManager envoie au *box B0* une référence vers le *textComponent TO* ;
10. le *box B0* demande au CIOManager d'instancier le *buttonComponent BT0* dont elle a trouvé la description dans sa balise ;
11. le CIOManager instancie le *buttonComponent BT0* et garde une référence vers l'objet créé ;
12. le CIOManager envoie au *box B0* une référence vers le *buttonComponent BTO* ;

```
<window id="W0">
    <box id="B0">
        <textComponent id="T0"/>
        <button id="BT0"/>
    </box>
</window>
<window id="W1">
    <box id="B1">
        <textComponent id="T1"/>
        <button id="BT1"/>
    </box>
</window>
```

FIG. 8.23 – Code usiXML simplifié comprenant deux balises *windows*.

La figure 8.25 illustre l'ordre de création des éléments graphiques du code de la figure 8.23, par rapport à leur structure hiérarchique. Elle permet de remarquer que cet ordre est celui du parcours infixe d'un arbre. La figure 8.26 illustre, quant à elle, les références gardées par chaque élément graphique vers ses éléments fils ainsi que les références du CIOManager vers tous les éléments graphiques de l'interface.

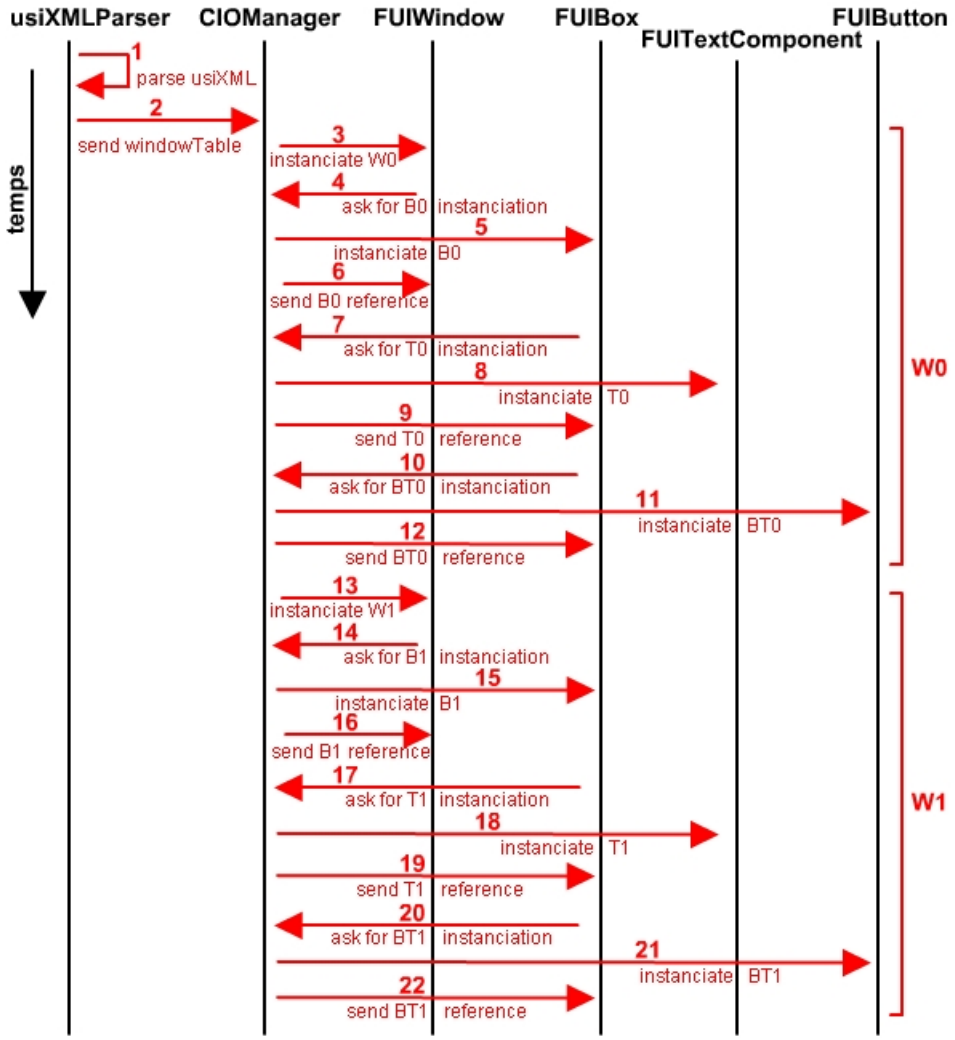

FIG. 8.24 – Diagramme de séquence de l'instanciation des deux balises *window* dont le code est repris à la figure 8.23.

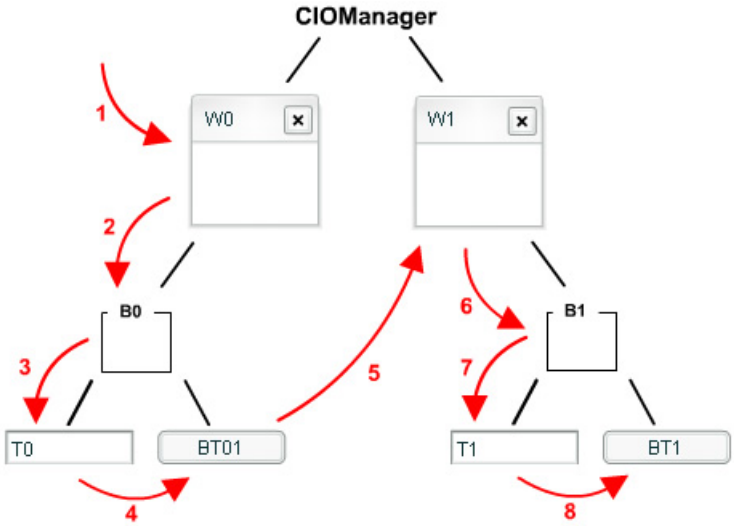

FIG. 8.25 – Illustration de l'ordre de création des éléments graphiques du code de la figure 8.23.

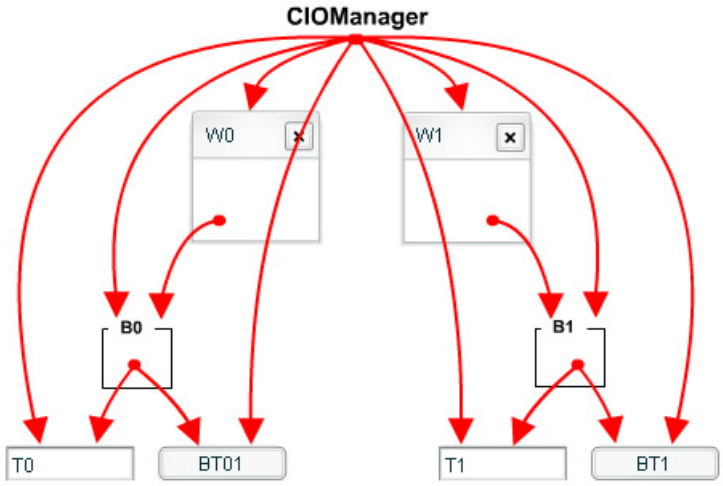

FIG. 8.26 – Illustration de la gestion des références vers les éléments graphiques de l'interface décrite par le code de la figure 8.23.

Calcul de la taille et de la position des *box*

Lorsque tous les éléments fils d'une fenêtre ont été instanciés pas le CIO-Manager, ils sont tous positionnés l'un sur l'autre.

Les éléments terminaux, c'est-à-dire ceux qui ne possèdent pas de fils, ont, par contre, directement été construits dans leur taille définitive. Pour rappel, cette taille est, en principe (voir 8.4.3) :
- soit définie par les paramètres *height* et *width* de la balise usiXML qui les décrit ;
- soit calculée dynamiquement en fonction de certains paramètres.

Toutefois, la section 8.4.3 a également mis en évidence une double dépendance de la taille des *box* en fonction :
- de la taille totale des éléments graphiques qu'ils contiennent ;
- de la taille relative souhaitée par rapport à la taille du contenu de leur parent[11].

Par conséquent, il convient de calculer la taille des *box* en deux étapes :
- calcul de la taille minimale de chaque *box* ;
- réajustement éventuel de la taille des *box* en fonction de la taille relative souhaitée par rapport à leur parent.

Calcul de la taille minimale des box.
Lors de la création des éléments de l'interface, chaque *box* calcule la hauteur et la largeur minimale dont il a besoin pour afficher tous ses fils. Dans le cas d'un *box horizontal*, la hauteur minimale correspond à la hauteur du fils le plus haut, alors que la largeur minimale correspond à la somme des largeurs de tous ses fils. Dans le cas d'un *box vertical*, le calcul est symétrique. Comme illustré à la figure 8.27, le calcul de la taille des éléments graphiques se fait suivant le parcours postfixe de l'arbre hiérarchique de l'interface.

Il peut paraître contradictoire que le calcul de la taille minimale se fasse durant la création des éléments graphiques, alors que la figure 8.25 de la section 8.4.4 indique que la création des éléments graphiques se fait selon un parcours infixe ! En réalité, les deux opérations s'entremêlent selon le parcours d'un *eulerTour*[12]. En effet :
- la création des éléments graphiques se fait lors des visites *à gauche* des noeuds ;

[11] le parent peut être soit une fenêtre, soit un autre *box*
[12] Le *patern eulerTour* est une extension des parcours préfixes, infixes, et postfixes. Pour plus de détails le lecteur peut se référer à [18]

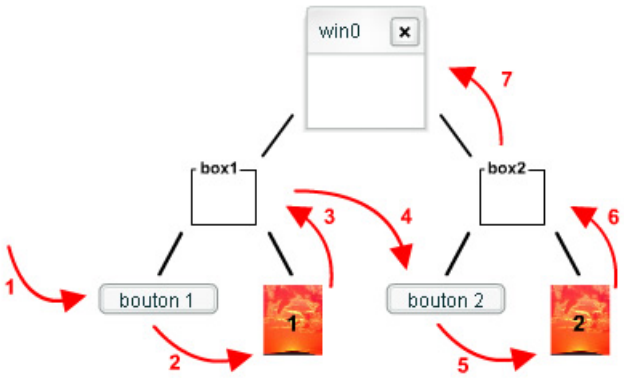

FIG. 8.27 – Illustration de l'ordre de calcul des tailles des éléments graphiques.

– le calcul de la taille minimale se fait lors des visites *à droite* des noeuds.

La particularité de notre cas par rapport à un parcours *eulerTour* classique réside dans le fait que l'arbre parcouru n'est pas complet au départ mais se construit au fil des visites *à gauche*. Finalement, le résultat est le même que si l'arbre [13] avait été construit entièrement[14] et qu'il avait ensuite été parcouru selon un parcours postfixe. La différence réside donc dans le fait que l'*eulerTour* permette de ne parourir qu'une seule fois l'arbre.

Ajustement de la taille des box.

Lorsque la taille minimale de tous les *box* d'une fenêtre a été calculée, la fenêtre relance un parcours de tous les *box* afin de voir si la taille du *box* peut être réajustée, selon les critères suivants :

– si la taille minimale nécessaire du *box* est plus grande que la taille relative souhaitée par rapport à la taille du parent, le *box* garde sa taille minimale ;
– si la taille minimale est plus petite que la taille relative souhaitée, le *box* est redimensionné à la taille désirée.

Il est malheureusement indispensable de re-parcourir l'arbre pour faire cet ajustement. En effet, comme le calcul de la taille minimale se fait selon un parcour postfixe, il n'est pas possible de connaître la taille du box parent au moment où l'on calcule la taille d'un *box*. Un second parcours est dès lors nécessaire. Afin de pouvoir directement positionner les *box*, le second parcours se fera également selon un ordre postfixe. Ceci permet en effet de

[13] et donc aussi les éléments graphiques

[14] ce qui doit obligatoirement se faire selon un parcours préfixe étant donné l'imbrication des éléments graphiques

positionner chaque *box* en fonction de la position et de la taille du précédent.

Positionnement des éléments graphiques au sein des *box*

Lorsque tous les *box* ont acquis leur taille définitive, il reste convient de positionner les éléments terminaux au sein de ces *box*. Ce positionnement est très simple du fait que :

- la taille du *box* est connue et est au moins aussi grande que la taille minimale nécessaire pour placer tous les éléments contenus dans le *box* ;
- la taille de tous les éléments fils est connue.

Le positionnement se fait alors comme décrit à la section 8.4.3, c'est-à-dire selon la valeur des propriétés *isBalanced* et *type* du *box*. Ainsi, lorsque *isBalanced* vaut *false*, les éléments sont placés soit :

- l'un à la suite de l'autre, de gauche à droite, si le *box* est de type *horizontal* ;
- l'un à la suite de l'autre, de haut en bas, si le *box* est de type *vertical* ;
- l'un sur l'autre dans le coin supérieur gauche si le *box* est de type *stack*.

Lorsque la propriété *isBalanced* vaut *true*, les éléments sont placés :

- l'un à côté de l'autre, de manière justifiée par rapport à la largeur ;
- l'un en dessous de l'autre, de manière justifiée par rapport à la hauteur ;
- l'un sur l'autre, centrés sur le centre du *box*.

Notons que l'effet de cet alignement ne sera perceptible que dans l'hypothèse où le *box* dispose d'une taille plus large que la taille minimale nécessaire à l'affichage des éléments qu'il contient.

Conclusion

L'instanciation des éléments graphiques est coordonnée par le CIOManager qui garde en mémoire une référence vers tous les éléments graphiques de l'interface. Il est le seul composant de FlashiXML à avoir une vue globale de tous les éléments. Par contre, chaque élément ayant des fils garde également une référence vers chacun d'entre eux.

Les éléments graphiques sont instanciés selon un parcours préfixe de leur arborescence, le calcul de la largeur minimale nécessaire des *box* doit se faire selon un parcours postfixe. Toutefois, le pattern *eulerTour* a permis de réunir ces deux étapes en un seul parcours.

Malgré tout, il sera nécessaire de reparcourir l'entièreté de l'arborescence afin de voir si la taille de certains *box* peut être réajustée en fonction de la

taille relative souhaitée par rapport à la taille de leur parent.

Une fois les *box* correctement dimensionnés, il est possible de positionner les éléments qu'ils contiennent en fonction des propriétés *isBalanced* et *type* du *box*.

8.5 Gestion des comportements : transitions et scripts

8.5.1 Introduction

Afin de conférer un intérêt pratique à l'interface générée, il est indispensable de pouvoir attribuer des comportements aux éléments graphiques qui la composent. Ceux-ci seront associés à un type d'événement déclencheur.

Dans le cadre de la présente application, les comportements (*behavior*) pourront, soit, être des transitions graphiques (*graphicalTransition*), soit l'exécution de scripts (*methodCall*), soit enfin une combinaison des deux.

8.5.2 La balise behavior de usiXML

Introduction

La balise *behavior* de usiXML, étudiée dans la section 4.4.2[15] permet d'associer, à un élément graphique, des actions à déclencher lors de la survenance d'un événement.

La spécification usiXML prévoit que la balise behavior puisse se retrouver à tout endroit de l'uiModel. Toutefois, pour une question de simplicité, FlashiXML exige que chaque balise behavior soit située à l'intérieur de l'objet source de l'événement déclencheur. Comme illustré à la figure 8.28, plusieurs comportements peuvent être associés à des événements différents du même élément graphique.

Implémentation des behaviors

Chaque comportement de l'interface est représenté par un objet spécifique (*Behavior*). La gestion des *behavior* et de leur exécution se fait par l'intermédiaire du *BehaviorManager*. Elle se décompose en plusieurs étapes :

1. l'utilisateur lance la génération de l'interface ;
2. lors de sa création, chaque élément graphique communique au BehaviorManager les balises behavior qu'il contient ;
3. le manager entrepose les données concernant le comportement dans un objet spécifique. Il renvoie ensuite à l'élément graphique une référence vers l'objet dédié au comportement ;
4. l'élément graphique associe le comportement et l'événement qui y correspond ;

[15] page 27

```
<button id="btn1">
    <behavior id="behav1">
        <event id="evt1" eventType="depress" eventContext="btn1">
        <action id="act1">
            ... liste des actions à effectuer ...
        </action>
    </behavior>
    <behavior id="behav2">
        <event id="evt2" eventType="release" eventContext="btn1">
        <action id="act2">
            ... liste des actions à effectuer ...
        </action>
    </behavior>
</button>
```

FIG. 8.28 – Balise behavior associée à un bouton conformément aux exigences de FlashiXML

5. l'utilisateur déclenche l'événement associé au comportement ;
6. le bouton repère l'événement et prévient le BehaviorManager que le comportement correspondant doit être exécuté ;
7. le behaviorManager exécute le comportement.

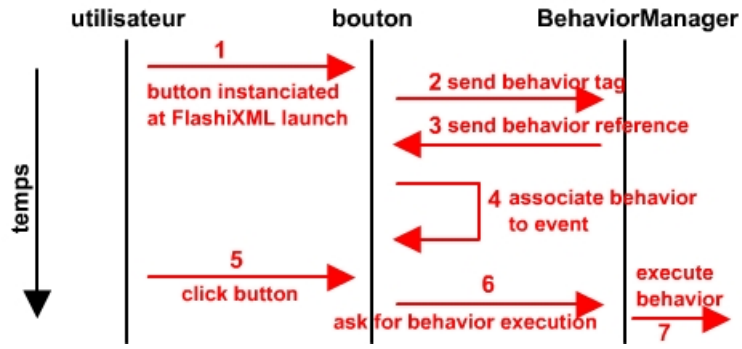

FIG. 8.29 – Diagramme de séquence de la récupération à l'exécution d'une balise behavior.

8.5.3 Gestion des transitions graphiques

Introduction

Le langage usiXML permet de définir des transitions graphiques associées à des éléments de l'interface. Il s'agit essentiellement de *passages* de l'état visible à l'état invisible ou vice-versa. Bien que usiXML prévoit d'autres

moyens, FlashiXML impose qu'une transition soit associée à une balise behavior. Elle devra donc être déclenchée par un événement associé à un élément graphique. FlashiXML exige également que les transitions graphiques soient déclarées comme fils directs de la balise cuiModel. La balise *action* du comportement devra, quant à elle, contenir une référence vers la transition à effectuer.

FlashiXML n'émet aucune contrainte quand au type de l'élément source et de l'élément cible d'une transition. De même, plusieurs transitions pourront être associées à un même événement d'un comportement.

```
<window id="win1">
    <box id="box1">
        <button id="btn1">
            <behavior id="behav1">
                <event id="evt1" eventType="release" eventContext="btn1">
                <action id="act1">
                    <transition transitionIdRef="Tr1">
                    <transition transitionIdRef="Tr2">
                </action>
            </behavior>
        </button>
    </box>
</window>

<graphicalTransition id="Tr1" transitionType="close">
    <source sourceId="but1">
    <target targetId="win1">
</graphicalTransition>

<graphicalTransition id="Tr2" transitionType="fadeOut">
    <source sourceId="but1">
    <target targetId="but1">
</graphicalTransition>
```

FIG. 8.30 – Exemple de transition respectant les exigences de FlashiXML

Implémentation des transitions

Chaque transition sera représentée par un objet spécifique (*CUIR*[16]). La gestion des *CUIRs* et de leur exécution se fait par l'intermédiaire du *CUIRManager* selon les opérations suivantes :

[16]Concrete User Interface Relationship

1. Lors du lancement de l'application, les balises *graphicalTransition* sont déléguées au CUIRManager ;
2. Les éléments graphiques de l'interface sont instanciés ;
3. Lors de sa création, chaque élément graphique communique au BehaviorManager les balises behavior qu'il contient ;
4. Le BehaviorManager crée un objet Behavior correspondant au comportement ;
5. L'objet behavior demande au CUIRManager une référence vers les transitions qui le concernent ;
6. Le CUIRManager répond à l'objet Behavior ;
7. Le BehaviorManager renvoie à l'élément graphique une référence vers le Behavior dédié au comportement ;
8. L'élément graphique associe le comportement à l'événement qui y correspond ;
9. L'utilisateur déclenche l'événement associé au comportement ;
10. Le bouton repère l'événement et prévient le BehaviorManager que le comportement correspondant doit être exécuté ;
11. Le behaviorManager transmet l'ordre d'exécution au Behavior ;
12. Le behavior informe le CUIRManager de toutes les transitions à effectuer ;
13. Le CUIRManager exécute les transitions indiquées.

Transitions implémentées

Les propriétés vectorielles de Flash permettent d'envisager un nombre illimité de transitions très différentes. Le tableau 8.13 reprend les 6 transitions actuellement implémentées dans FlashiXML ainsi que leur description. A elles seules, elles donnent un bon aperçu de ce qu'il est possible de réaliser.

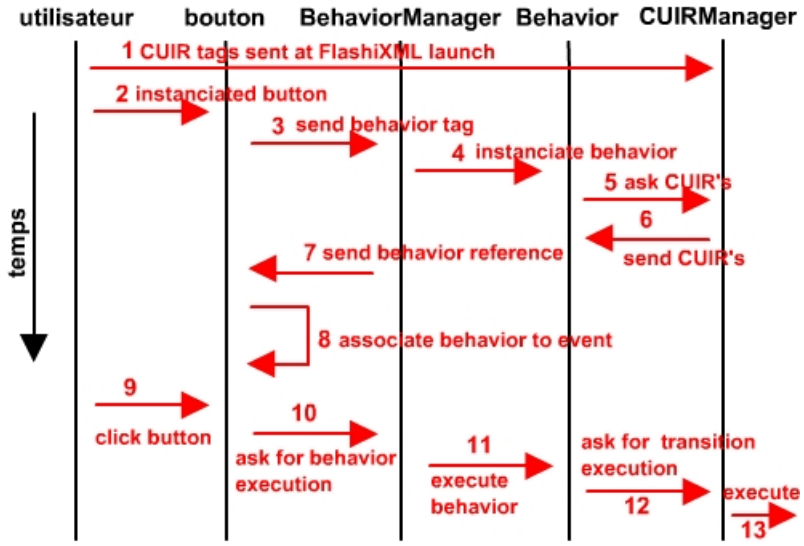

FIG. 8.31 – Scénario de la récupération à l'exécution d'une balise behavior avec des transitions.

FIG. 8.32 – Illustration d'une transition de type *fadeOut* appliquée à une fenêtre.

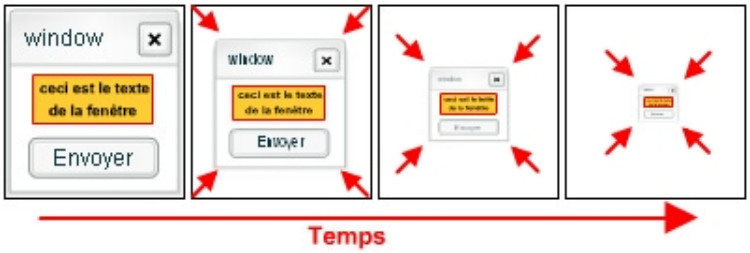

FIG. 8.33 – Illustration d'une transition de type *boxOut* appliquée à une fenêtre.

nom de la transition	description
open *invisible - visible*	Ramène l'élément graphique cible au premier plan de la scène avant de mettre sa propriété de visibilité à *true*.
close *visible - invisible*	Met la propriété de visibilité de l'élément graphique cible à *false*.
fadeIn *invisible - visible*	Ramène l'élément graphique cible au premier plan et initialise sa transparence au maximum[17]. Ensuite, applique une transparence décroissante jusqu'à opacité complète.
fadeOut *visible - invisible*	Ramène l'élément graphique cible au premier plan et lui applique une transparence croissante. Une fois la transparence nulle, la propriété de visibilité est mise à *false* et la transparence est réinitialisée au maximum.
boxIn *invisible - visible*	Ramène l'élément graphique au premier plan. Réduit sa hauteur et sa largeur avant de le rendre visible. Applique ensuite un re-dimensionnement progressif jusqu'à ce que la taille de départ soit atteinte.
boxOut *visible - invisible*	Ramène l'élément graphique cible au premier plan et lui applique une compression progressive tant en hauteur qu'en largeur. Une fois la taille suffisamment petite, la propriété de visibilité est mise à *false* et la taille est réinitialisée.

TAB. 8.13 – Transitions implémentées dans FlashiXML

8.5.4 Gestion des scripts externes

Introduction

Les transitions étudiées dans la section précédente constituent des effets visuels permettant de changer l'état de visibilité d'un élément graphique. En permettent d'accéder aux autres propriétés des éléments, les scripts externes apportent une dimension supplémentaire aux interactions entre les composants de l'interface.

Comme les transitions, les scripts sont associés à des comportements déclenchés par des événements. Un script pourra donc être déclenché par tout type d'élément. De même, il pourra affecter les propriétés de tout élément de l'interface.

Implémentation des scripts externes

Les balises *methodCall* et *methodCallParam* permettent de préciser la fonction externe à appeler ainsi que les paramètres à lui transmettre. Ainsi,

dans le code usiXML de la figure 8.34, l'événement *relâcher le bouton btn1* entraînera l'exécution consécutive des fonctions :

1. *copyText* avec comme paramètres les chaînes de caractères *tC1* et *tC2*, ce qui devra donc être traduit par le générateur en *copyText(tC1, tC2)*.
2. *addstringToText* avec comme paramètres les chaînes de caractères *!!!* et *tC2*, ce qui devra donc être traduit par le générateur *addStringToText(" !!!", tC2)*.

```
<window id="win1">
    <box id="box1" type="vertical">
        <textComponent id="tC1" defaultContent="" isEditable="true"/>
        <textComponent id="tC2" defaultContent="" isEditable="false"/>
        <button id="btn1">
            <behavior id="behav1">
                <event id="evt1" eventType="release" eventContext="btn1">
                <action id="act1">
                <methodCall methodName="copyText">
                    <methodCallParam name="p1" value="tC1"/>
                    <methodCallParam name="p2" value="tC2"/>
                </methodCall>
                <methodCall methodName="addStringToText">
                    <methodCallParam name="p1" value="!!!"/>
                    <methodCallParam name="p2" value="tC2"/>
                </methodCall>
                </action>
            </behavior>
        </button>
    </box>
</window>
```

FIG. 8.34 – Exemple de balise *methodCall* faisant appel aux fonctions externes *copyText()* et *addStringToText()*.

Bien entendu, il convient de respecter plusieurs conditions afin qu'un script externe puisse être exécuté :

- Le fichier de configuration[18] doit indiquer correctement le chemin du fichier contenant toutes les fonctions externes ;
- Le fichier contenant les fonctions externes doit avoir été préalablement chargé au format swf ;
- Le code des fonctions externes doit être écrit en ActionScript MX syntaxiquement correct ;
- Dans l'hypothèse où le code des fonctions externes est compilé dynamiquement, les règles de la section 8.5.4 doivent être respectées ;
- Les balises *methodCall* et *methodCallParam* doivent être placées dans une balise *behavior* et attachées à un événement.

[18] voir section 8.2, page 51

Comme nous venons de le voir, il convient de charger au formet *SWF* (soit le format de compression de Flash)le fichier contenant les fonctions externes. La figure 8.35 illustre un exemple de code des fonctions *copyText* et *addStringToText* conformes à l'appel défini dans le fichier usiXML.

```
_root.copyText = function(_tC1, _tC2){
      temp = _root.getItemGUIById(_tC1);
      tempText = temp.getText();
      temp = _root.getItemGUIById(_tC2);
      temp.setText(tempText);
}

_root.addStringToText = function(_str, _tC){
      temp = _root.getItemGUIById(_tC);
      tempText = temp.getText();
      temp.setText(tempText + _str);
}
```

FIG. 8.35 – Code ActionScript des fonctions externes *copyText()* et *addStringToText()*

Afin d'être utilisables par FlashiXML en tant que fichier de script externe, les fonctions de la figure 8.35 doivent simplement être copiées dans l'éditeur de Flash et être compilées. Les scripts ainsi pré-compilés peuvent être chargés et exécutés entièrement sans serveur. La figure 8.36 décompose le procédé de chargement des scripts externes.

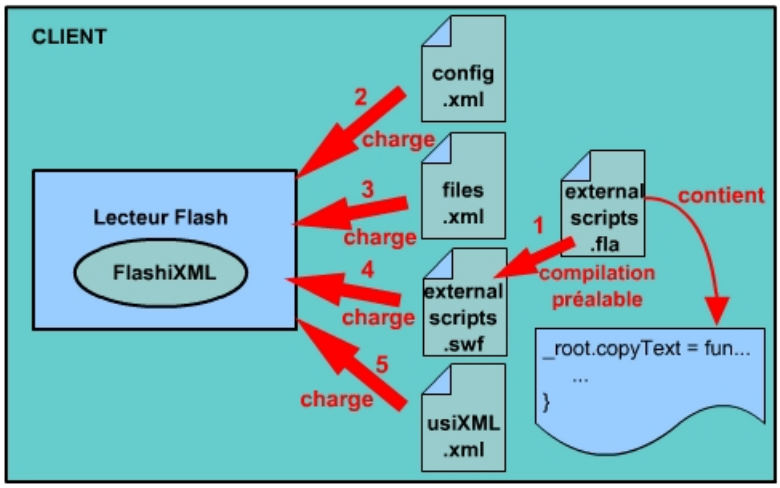

FIG. 8.36 – Procédé de chargement de scripts externes sans serveur.

Compilation dynamique des scripts avec MING

La librairie MING de PHP (voir section 6.2, page 39) permet la compilation dynamique d'animations Flash. En outre, lorsque le paramètre système *standalone* du fichier de configuration (voir section 8.2) est mis à *false*, FlashiXML peut faire appel à PHP et à MING pour charger dynamiquement les fonctions externes à partir d'un simple fichier texte. De ce fait, il n'est plus nécessaire de pré-compiler les fonctions ActionScript au format swf, comme c'était le cas dans la rubrique précédente.

Pour plus de modularité, il est possible d'entreposer les scripts dans une base de données plutôt que dans un fichier texte. Grâce à cela, il sera envisageable de les éditer facilement à distance. La figure 8.37 illustre le procédé de chargement de fonctions externes compilées dynamiquement.

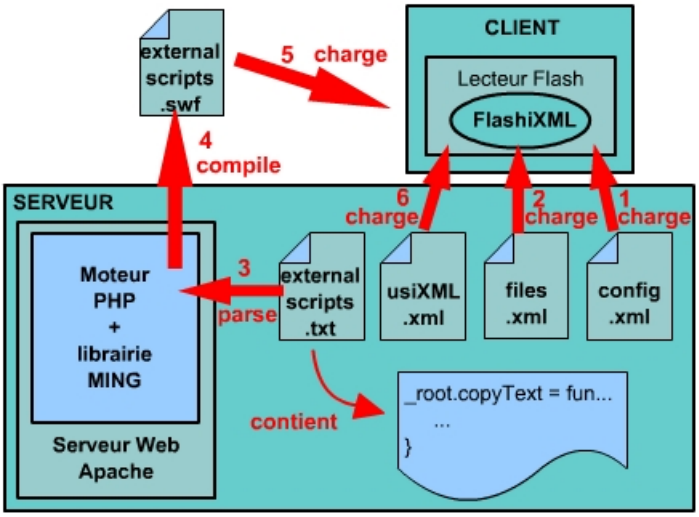

FIG. 8.37 – Procédé de chargement de scripts externes compilés dynamiquement avec la librairie MING de PHP.

L'annexe D contient le code PHP-MING permettant la compilation dynamique de code ActionScript au format *SWF* à partir d'un fichier texte.

8.5.5 Conclusion

La balise *behavior* de usiXML permet de définir le comportement que doit adopter un élément graphique lors de la survenance d'un événement. Il est possible de définir un comportement différent pour chaque type d'événement (clic de souris, survol de souris,...). Un comportement correspond à une série d'actions qui peuvent être de deux types :

– des transitions graphiques qui permettent de changer l'état de visibilité d'un élément graphique ;
– des appels de fonctions.

Lorsqu'une description usiXML fait appel à des fonctions, celles-ci doivent avoir été préalablement définies dans un fichier spécial dont l'emplacement est communiqué à FlashiXML par l'intermédiaire du fichier *files* (voir 8.2.1).

Lorsque FlashiXML fonctionne en *mode serveur*, le fichier contenant les fonctions externes peut être un simple fichier *texte* qui sera compilé dynamiquement par la librairie MING de PHP. Lorsque FlashiXML est lancé en mode *standalone* le fichier de fonctions externes doit avoir été préalablement compilé au format *SWF*.

Comme nous pouvons le constater, la balise *behavior* permet de définir des interfaces comprennant de la *navigation*, mais également de la *programmation*.

8.6 Etude des configurations possibles

8.6.1 Introduction

La section 8.5.4 a permis :
- d'illustrer l'utilisation de FlashiXML avec ou sans serveur (mode standalone) ;
- de décrire la manière dont FlashiXML parvient à compiler des scripts à la volée à l'aide des librairies MING lorsque le *mode serveur* est activé.

Dans la présente section, nous analyserons les possibilités d'emplacement des ressources. En effet, nous avons vu, à la section 8.5.4 que les scripts à compiler peuvent être entreposés soit dans de simples fichiers textes, soit dans une base de données. Par contre, rien n'a été précisé en ce qui concerne les autres ressources. La section courante vise dès lors à analyser les différentes répartitions possibles, entre CLIENT et SERVEUR, des entités nécessaires à la génération de l'interface.

8.6.2 CLIENT uniquement

La première configuration possible correspond à l'hypothèse où le paramètre *standalone* s'est vu attribuer la valeur true. FlashiXML devra alors fonctionner indépendamment de tout serveur. De ce fait :
- FlashiXML sera incapable de compiler dynamiquement les éventuelles fonctions externes utilisées par la description usiXML. Celles-ci devront dès lors avoir été pré-compilées au format *SWF* ;
- le *fichier files* ne pourra pas être généré dynamiquement. La liste des descriptions usiXML proposées à l'utilisateur au lancement de l'application sera donc statique.

Cette solution présente toutefois l'avantage de permettre l'exécution d'interfaces faisant appel à des fonctions externes mais qui doivent être capables de s'exécuter :
- sans serveur local ;
- sans connection internet ;
- sur un support en lecture-seule tel un CD-Rom.

Cette configuration, qui correspond en fait au comportement d'un compilateur, est illustrée à la figure 8.38.

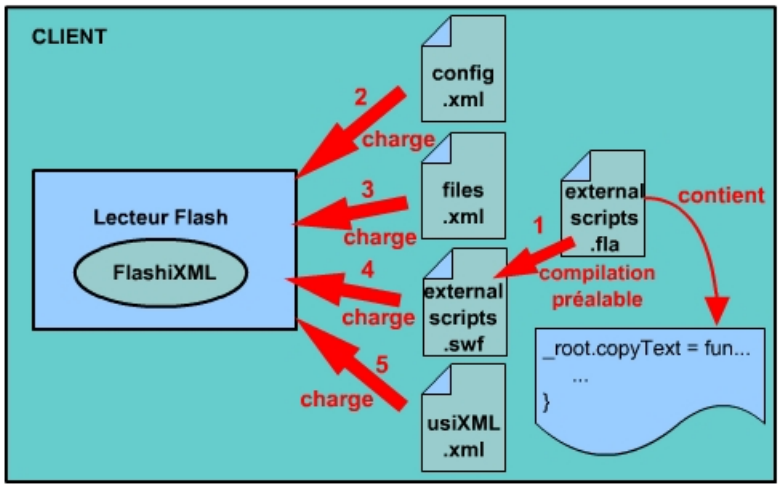

FIG. 8.38 – Illustration de la configuration *CLIENT uniquement* de FlashiXML dont le comportement est semblable à celui d'un compilateur.

8.6.3 CLIENT - SERVEUR local

La configuration *CLIENT - SERVEUR local* illustrée à la figure 8.39 est une configuration intermédiaire entre les configurations *CLIENT seul* et *CLIENT-SERVEUR distant*. En imaginant que PHP et la librairie MING soient disponibles sur le serveur local, cette configuration permet :

- de compiler dynamiquement les fonctions externes ;
- de générer dynamiquement le *fichier files* contenant la liste des descriptions à proposer à l'utilisateur lors du lancement de l'application ;
- d'entreposer les descriptions des interfaces dans une base de données locale, ce qui peut augmenter la facilité de mise à jour de ces dernières.

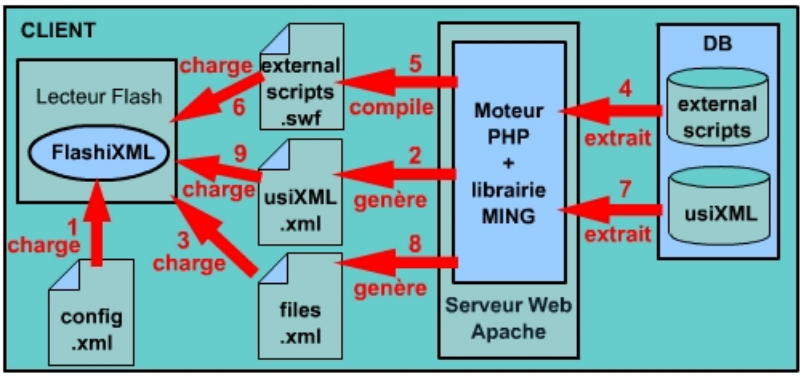

FIG. 8.39 – Illustration de la configuration *CLIENT - SERVEUR local* de FlashiXML.

8.6.4 CLIENT - SERVEUR distant

La configuration *CLIENT - SERVEUR distant* présente l'énorme avantage de permettre d'entreposer le *fichier files*, les descriptions d'interfaces ainsi que les méthodes externes qu'elles utilisent, sur un serveur distant. Lorsqu'un CLIENT éxécute FlashiXML, ce dernier rapatrie les différentes ressources depuis le serveur distant. Dès lors, toute modification d'interface sur le serveur sera perçue par tous les CLIENTS dès le lancement suivant de FlashiXML. Cette configuration permet de faciliter grandement la mise à jour des applications.

Les figures 8.40 et 8.41 illustrent deux variantes de la configuration *CLIENT - SERVEUR distant*. Dans la première, le fichier de configuration de FlashiXML se trouve sur le serveur distant (commun à tous les utilisateurs). Dans la seconde, le fichier de configuration est entreposé sur la machine CLIENT (personalisable).

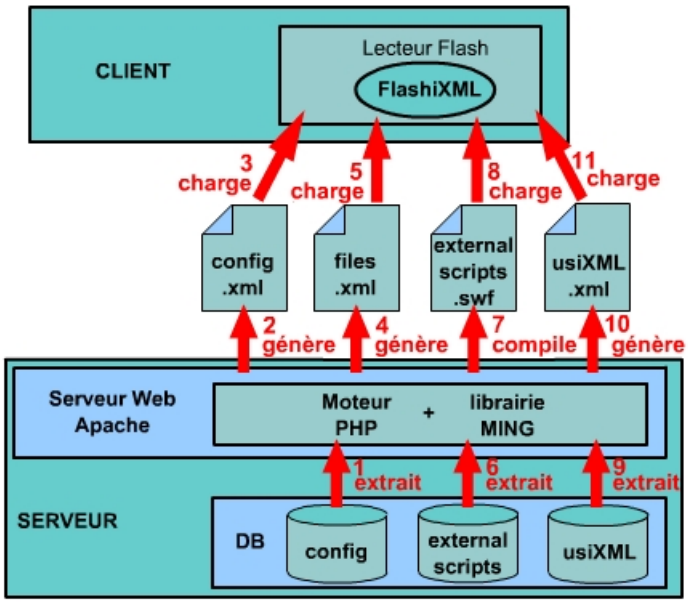

FIG. 8.40 – Illustration de la configuration *CLIENT - SERVEUR distant* de FlashiXML avec fichier de configuration commun.

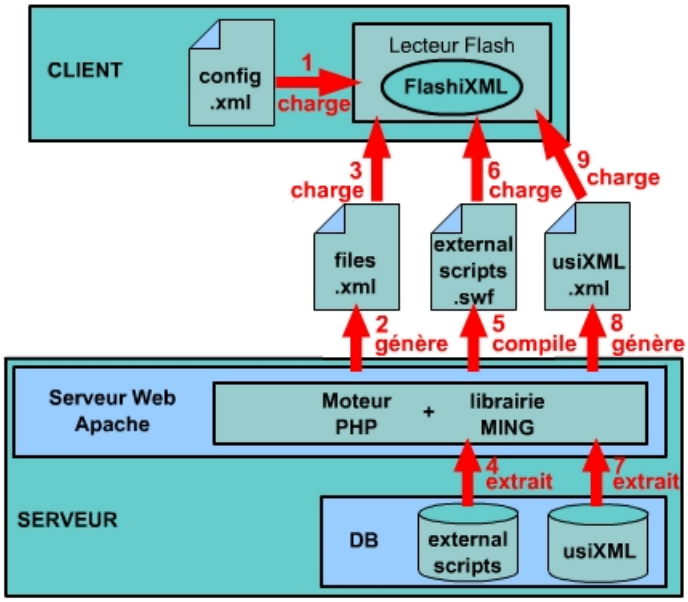

FIG. 8.41 – Illustration de la configuration *CLIENT - SERVEUR distant* de FlashiXML avec fichier de configuration personnalisé.

8.6.5 SERVEUR distant uniquement

Le fonctionnement de la configuration *SERVEUR distant uniquement*, illustré à la figure 8.42 est très proche du fonctionnement d'un site internet contenant une animation Flash. La différence avec la configuration *CLIENT - SERVEUR distant* réside dans le fait que l'application FlashiXML en elle-même se trouve également sur le serveur. La machine CLIENT ne nécessite donc aucune pré-installation de FlashiXML. La seule chose dont elle doit disposer est un navigateur internet muni du plug-in Flash.

En vue d'exécuter l'application, l'utilisateur ouvre son navigateur afin d'envoyer une requête HTTP au serveur distant. Le serveur renvoie alors une page internet contenant l'application FlashiXML au format *SWF*. Une fois la page téléchargée, l'application se lance automatiquement et rapatrie toutes les resources nécessaires depuis le serveur.

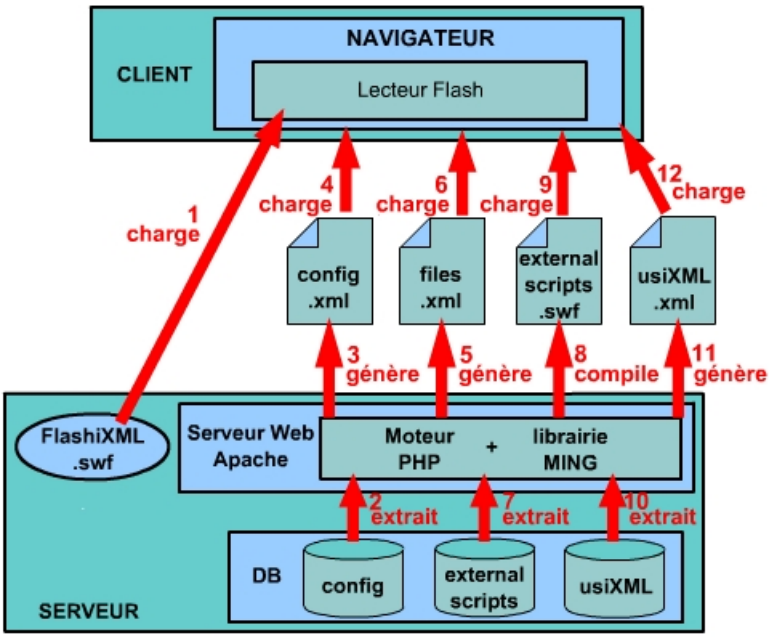

FIG. 8.42 – Illustration de la configuration *SERVEUR distant uniquement* de FlashiXML.

8.6.6 Conclusion

La variété de configurations possibles de FlashiXML lui confère une grande souplesse quand au type d'applications qui peuvent être générées. Ainsi :

– la configuration *CLIENT uniquement* est comparable au résultat d'un compilateur et permet donc des applications fonctionnant sans serveur (par exemple, une présentation sur CD-Rom) ;
– la configuration *CLIENT - SERVEUR distant* correspond aux méthodes orientées serveur *Flex* et *Lazlo* (section 2) ;
– la configuration *SERVEUR distant uniquement* imite, quant à elle, le fonctionnement d'un site internet comportant une animation Flash ;
– la configuration *CLIENT - SERVEUR local* permet de profiter de la compilation dynamique des fonctions externes utilisées par la description usiXML, et ce, sans connexion à internet ni à un serveur distant.

8.7 Gestion des interfaces multilingues

8.7.1 Introduction

Le *contextModel* et le *resourceModel* de usiXML introduits à la section 4.4.3 visent à définir des interfaces multilingues en permettant, pour tout élément graphique contenant du texte, de définir ce dernier dans chaque langue souhaitée. Parmi ces éléments, se trouvent : le *box*, le *button*, le *checkBox*, le *comboBox*, le *radioButton*, le *textComponent* ainsi que la *window*.

La figure 8.43 illustre le code usiXML d'une application contenant une fenêtre, un box et un bouton et qui définit un contenu en français et en anglais pour le bouton.

```
<window id="w0" defaultTitle="w0">
    <box id="B0" defaultTitle="b0">
        <button id="bt1" defaultContent="bt1" content="/resourceModel/cioRef/bt0"/>
    </box>
</window>
<contextModel>
    <context id="FR1" name="FR1">
        <ns1:userStereotype id="user1" language="francais"/>
    </context>
    <context id="EN1" name="EN1">
        <ns1:userStereotype id="user2" language="english"/>
    </context>
</contextModel>
<resourceModel>
    <cioRef cioId="bt1" xmlns="">
        <resource content="Envoyer" contextId="FR1"/>
        <resource content="Send" contextId="EN1"/>
     </cioRef>
</resourceModel>
```

FIG. 8.43 – Code usiXML illustrant la gestion des contenus multilingues.

8.7.2 Gestion des contenus en fonction de la langue

Dans l'hypothèse où le paramètre *languageSelection* du fichier de configuration de FlashiXML est mis à la valeur *true*, il convient de proposer le choix de langue à l'utilisateur et de générer l'interface en fonction de ce choix. Pour ce faire, il est nécessaire de pouvoir :

- identifier les langues disponibles ;
- identifier, pour chaque élément graphique, le contenu à afficher en fonction de la langue choisie.

Ceci se fait par l'intermédiaire du *LanguageManager*, du *ContextManager* et du *ResourceManager* selon le diagramme de la figure 8.44.

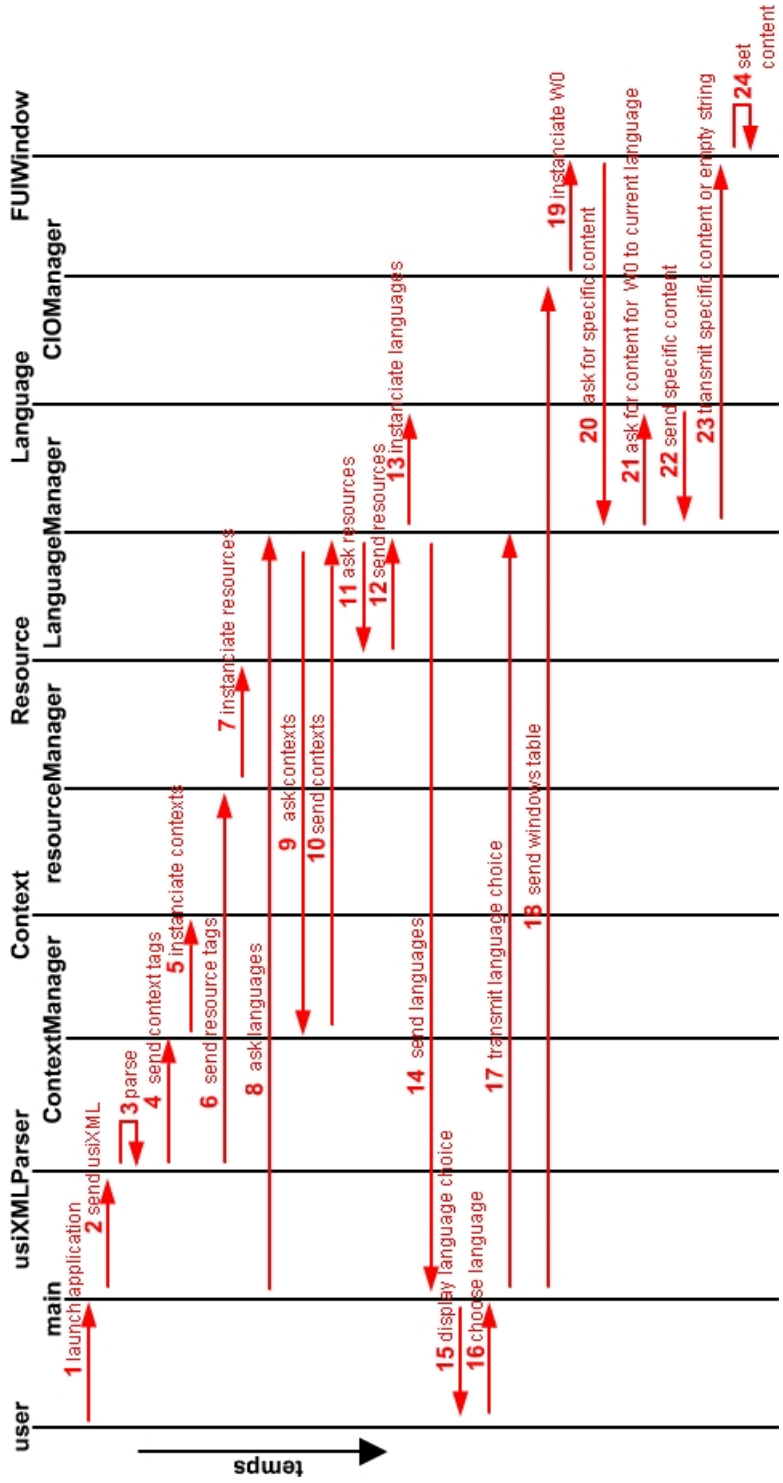

FIG. 8.44 – Diagramme de séquence de la gestion du contenu des éléments graphiques en fonction du choix de la langue de l'utilisateur.

8.8 Gestion des Fenêtres et du Menu

8.8.1 Gestion des icônes

Lorsque la valeur *true* est attribuée au paramètre de gestion de navigation *displayWindowBar* du fichier de configuration, FlashiXML affiche une barre d'icônes dans le bas de l'écran. Une icône est associée à chaque fenêtre de l'application, visible ou non. Une pression sur l'icône aura un effet différent en fonction de l'état de la fenêtre :

- lorsque la fenêtre est invisible, l'effet sera de la faire ré-apparaître à proximité de l'icône en taille réduite. Ensuite, un redimensionnement et un déplacement progressifs sont appliqués jusqu'à ce que la taille et la position initiales soient à nouveau atteintes. La figure 8.45 illustre cet effet ;
- lorsque la fenêtre est invisible, elle sera simplement ramenée au premier plan.

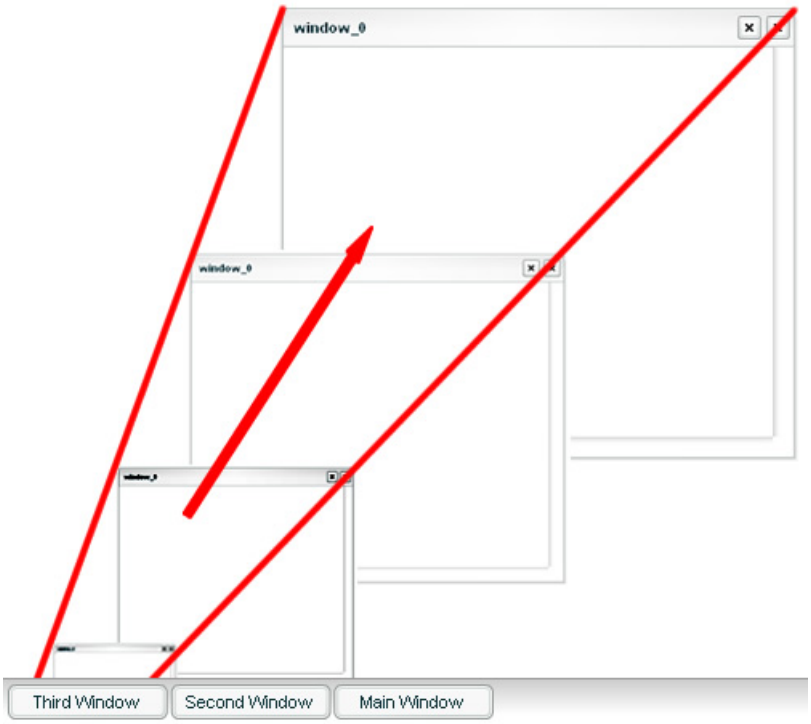

FIG. 8.45 – Illustration du WindowManager et de la *restauration* d'une fenêtre.

8.8.2 Menu utilisateur

Le paramètre système *displayMenu* du fichier de configuration permet au concepteur de proposer un menu déroulant à l'utilisateur. La section 8.2 étudie en détail les options de paramétrage de ce menu. La figure 8.46 illustre le résultat obtenu lorsque tous les paramètres sont activés.

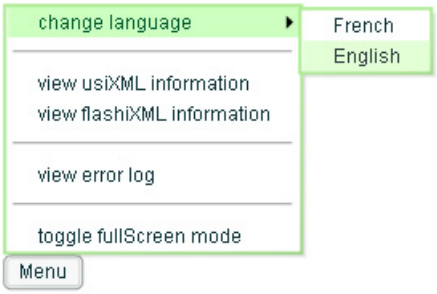

FIG. 8.46 – Illustration du menu de navigation de FlashiXML.

8.9 Gestion et affichage des erreurs (Log)

8.9.1 Introduction

Bien que certains logiciels de type *WYSIWYG*[19] permettant de générer facilement des interfaces usiXML soient en cours de développement, il est encore souvent nécessaire de créer ou d'adapter les descriptions usiXML manuellement. De ce fait, il est fréquent qu'une erreur se glisse dans la description. Par exemple, en faisant un copier-coller, il est possible que deux éléments graphiques aient le même *id*, identiant qui devrait en principe être *unique*.

FlashiXML peut, suite à une erreur, réagir de façon imprévisible, ce qui rend ces erreurs difficiles à déceler. A défaut de corriger les erreurs le gestionnaire des erreurs, permet de prévenir le concepteur de certaines erreurs fréquentes que FlashiXML aurait détectées au cours du procédé de génération.

C'est lors des phases de développement et de test des descriptions usiXML que le gestionnaire des erreurs est le plus utile. Son accès peut être activé ou désactivé en jouant sur les paramètres systèmes du fichier de configuration.

[19] What You See Is What You Get

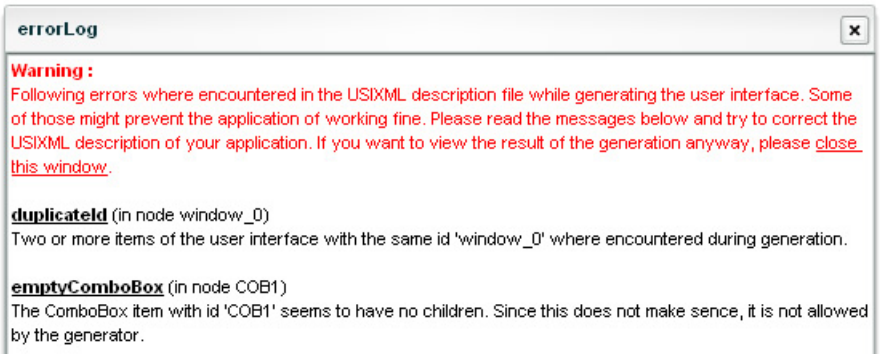

FIG. 8.47 – Illustration du gestionnaire des erreurs de FlashiXML.

8.9.2 Erreurs gérées au stade actuel

Le tableau 8.14 reprend la listes des erreurs gérées actuellement par FlashiXML ainsi que leur description.

propriété	description
duplicatedId	indique que deux ou plusieurs éléments de la description usiXML possèdent le même identifiant unique
emptyComboBox	indique que la description usiXML contient un élément *comboBox* ne comportant aucun fils

TAB. 8.14 – Erreurs actuellement détectées par FlashiXML et affichées par le gestionnaire des erreurs.

8.10 Récapitulation du procédé de génération

8.10.1 Illustration par diagramme de séquence de la dynamique de génération

Afin de récapituler l'ensemble des fonctionnalités étudiées dans ce chapitre de manière concise, les figures 8.50 et 8.50 reprennent le diagramme de séquence de l'interface exemple dont la description usiXML est reprise à la figure 8.48.

Cette application exemple contient une fenêtre, un box, un bouton et une case texte. En outre, deux actions sont associées à l'événement *relâcher* du bouton :

- un appel à la transition Tr0 dont l'effet est de rendre le bouton invisible ;
- un appel à la fonction externe *emptyText* avec un paramètre de valeur t0, qui correspond à l'identifiant de la case texte. La figure 8.49 reprend le code ActionScript de la fonction externe *emptyText*.

Enfin, la description usiXML de l'interface définit un contenu en Français ainsi qu'un contenu en Anglais pour la case texte. De ce fait, un choix de langue sera proposé à l'utilisateur.

Il convient de signaler que le processus de gestion des langues de FlashiXML est volontairement simplifié dans le diagramme de séquence. Le lecteur intéressé pourra se référer à la section 8.7 pour plus de détails à ce sujet.

```
<window id="w0" defaultTitle="w0">
    <box id="b0" defaultTitle="b0">
        <textComponent id="t0" defaultContent="text" content="/resourceModel/cioRef/bt0">
        <button id="btn0" defaultContent="apply">
            <behavior id="behav0">
                <event id="evt0" eventType="release" eventContext="btn0">
                <action id="act0">
                    <transition transitionIdRef="Tr0">
                    <methodCall methodName="emptyText">
                        <methodCallParam name="p0" value="t0"/>
                    </methodCall>
                </action>
            </behavior>
        </button>
    </box>
</window>
<graphicalTransition id="Tr0" transitionType="close">
    <source sourceId="btn0">
    <target targetId="btn0">
</graphicalTransition>
<contextModel>
    <context id="FR0" name="FR0">
        <ns1:userStereotype id="user0" language="francais"/>
    </context>
    <context id="EN0" name="EN0">
        <ns1:userStereotype id="user1" language="english"/>
    </context>
</contextModel>
<resourceModel>
    <cioRef cioId="t0" xmlns="">
        <resource content="Bonjour monde" contextId="FR0"/>
        <resource content="Hello world" contextId="EN0"/>
     </cioRef>
</resourceModel>
```

FIG. 8.48 – Code usiXML d'une interface multilingue contenant une fenêtre, un *box*, un *textComponent* et un *buttonComponent*.

```
_root.emtyText(id)  = function{
    temp = _root.getItemGUIById(id);
    temp.setText("");
}
```

FIG. 8.49 – Code ActionScript de la fonction externe emptyText(String) utilisée par la description usiXML de la figure 8.48.

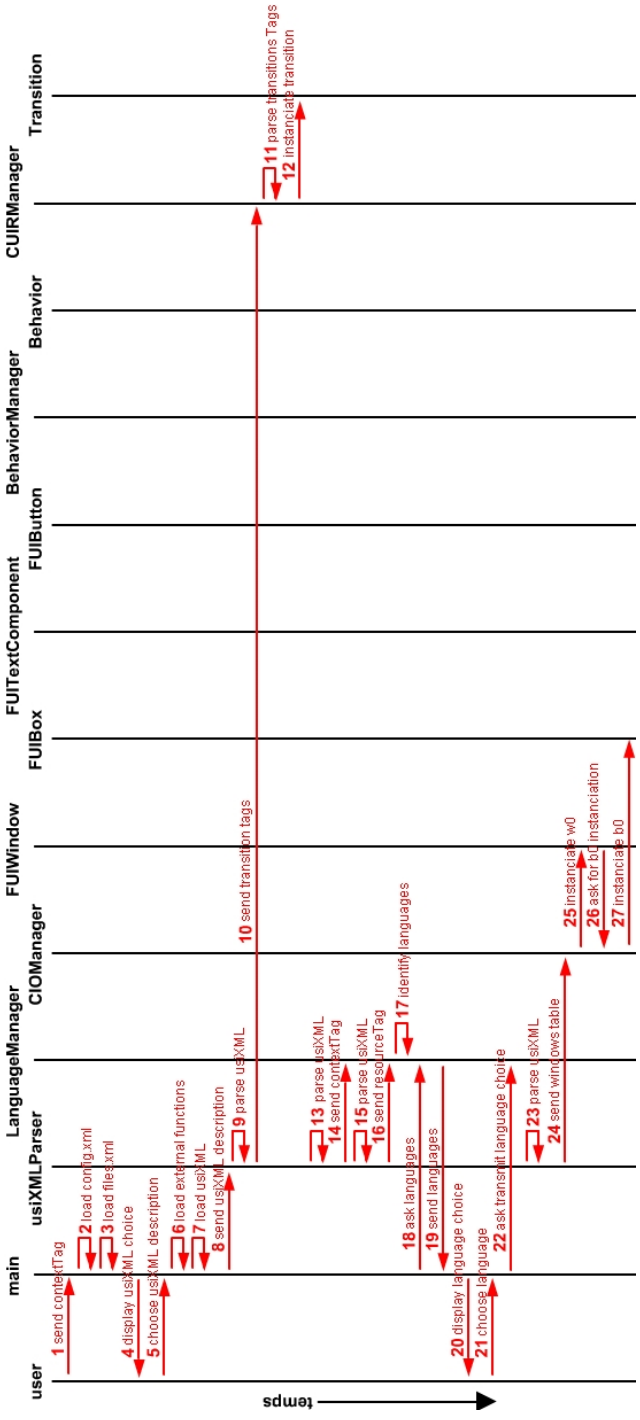

FIG. 8.50 – Diagramme de séquence de la génération de l'interface décrite à la figure 8.48 (1/2).

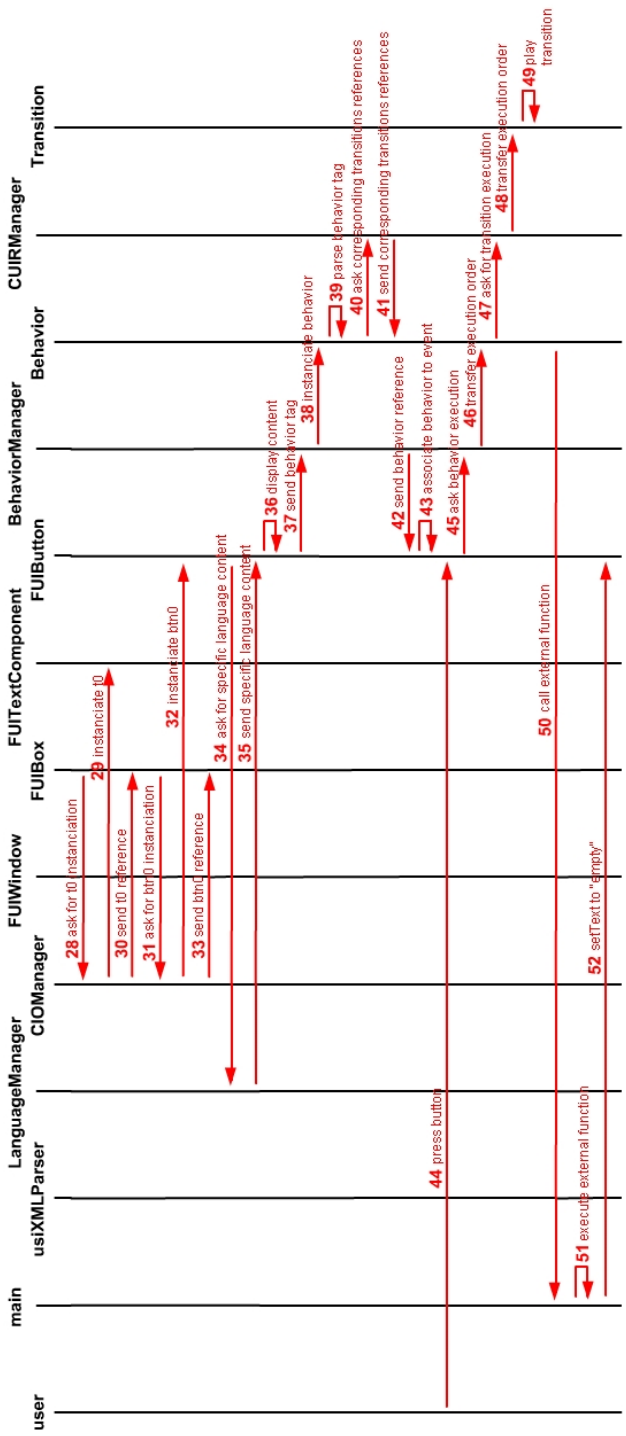

FIG. 8.51 – Diagramme de séquence de la génération de l'interface décrite à la figure 8.48 (2/2).

8.10.2 Diagramme de classe de FlashiXML

Les figures de cette section présentent le diagramme de classe de FlashiXML. Ainsi, la figure 8.52 reprend le diagramme principal. Ensuite, les figures 8.53 et 8.54 précisent respectivement la hiérarchie des classes *Item* et *CUIR*.

La figure 8.53 fait le lien entre les éléments de usiXML, ceux de Flash et ceux de FlashiXML. Dès lors, afin de mettre en évidence cette comparaison, nous avons ajouté les composants Flash utilisés par FlashiXML ainsi que les propriétés privées des classes de FlashiXML.

Dans ce cadre, il est important de rappeler que le préfixe *FUI* des classes FlashiXML est l'abréviation de *Flash User Interface* et non celle de *Final User Interface* au sens de usiXML qui correspond ici aux composants Flash utilisés.

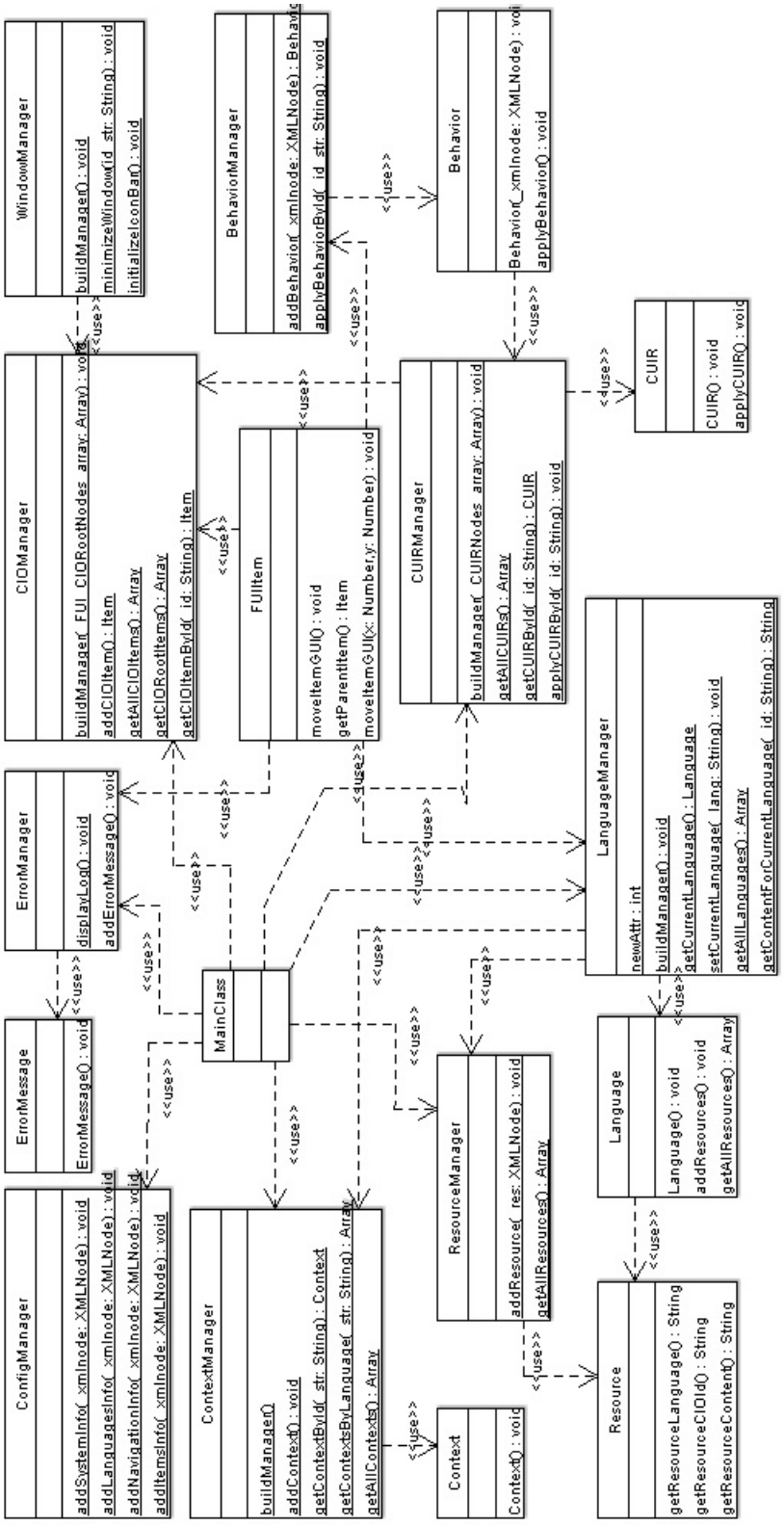

FIG. 8.52 – Diagramme de classe de FlashiXML (1/3).

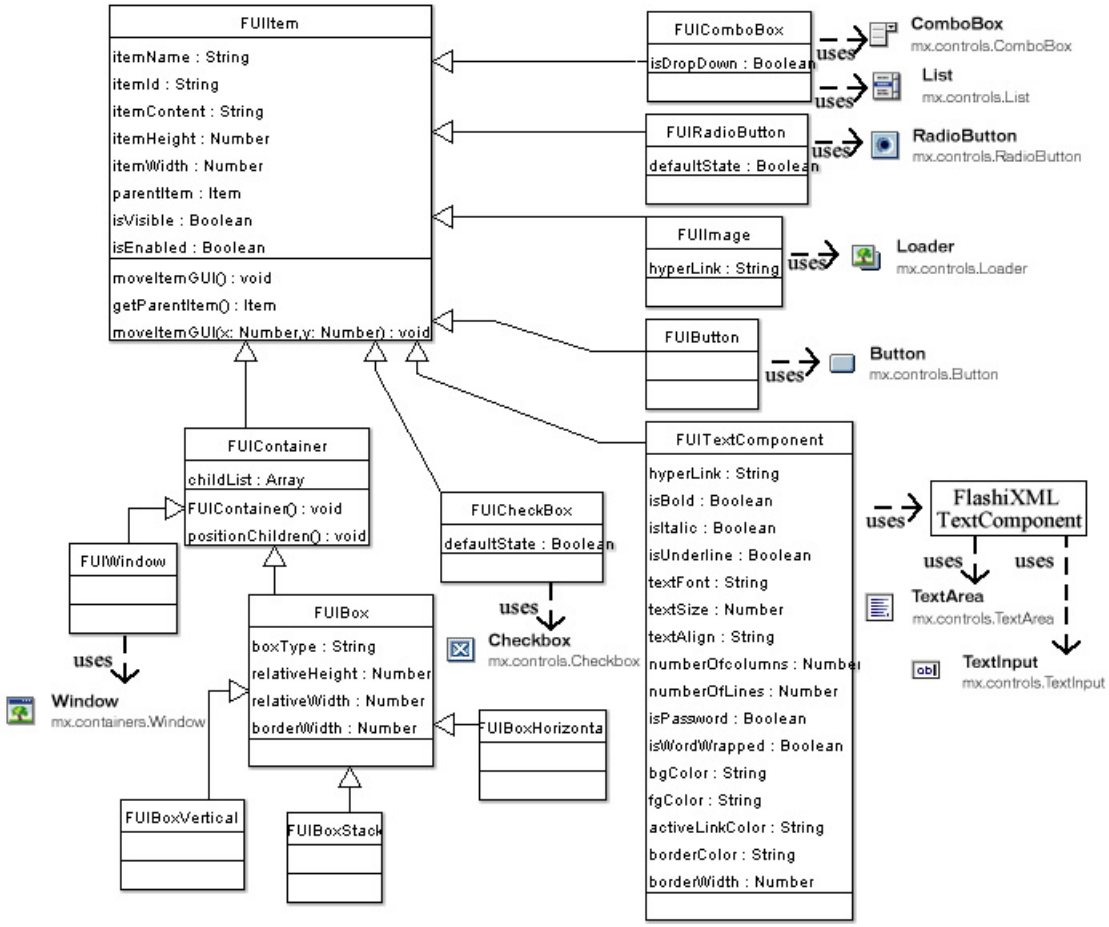

FIG. 8.53 – Diagramme de classe de FlashiXML (2/3). Hiérarchie de la FUIItem.

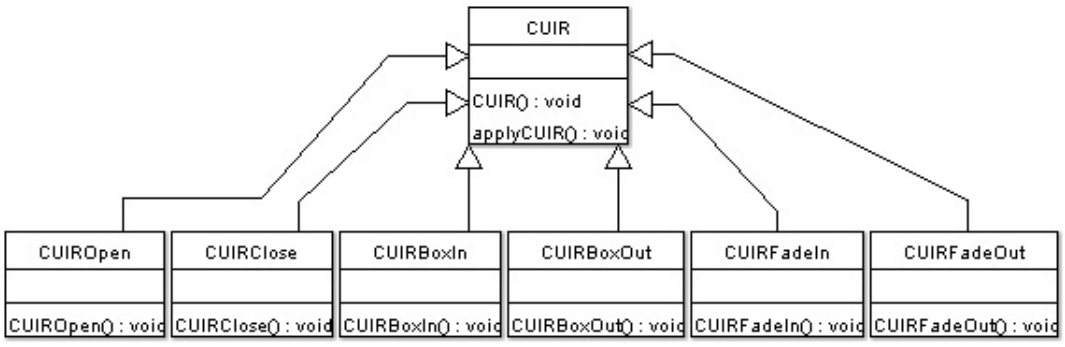

FIG. 8.54 – Diagramme de classe de FlashiXML (3/3). Hiérarchie de la classe CUIR.

8.11 Conclusion

Ce chapitre a décrit le fonctionnement du générateur d'interface vectorielle, baptisé FlashiXML que nous avons implémenté sur base des choix de langages source (usiXML) et cible (Flash) effectués dans les chapitres 4 et 5, des règles de correspondances entres ces langages établies dans le chapitre 7 et de l'architecture hybride en pour laquelle nous avons opté dans le chapitre 3.

Les principales caractéristiques de ce générateur sont :

- la possibilité de fonctionner, soit, selon le fonctionnement d'un interpréteur orienté serveur (avec compilation dynamique des fonctions externes à la description d'interface) ou selon le fonctionnement d'un compilateur générant une interface indépendante ;
- la grande portabilité liée d'une part au côté vectoriel de Flash et à la bonne distribution du plug-in nécessaire et, d'autre part, à la possibilité de définir en usiXML des interfaces indépendamment de leur contexte d'utilisation ;
- la grande variété de configurations et de répartitions de ressources sur différentes machines ;
- la possibilité d'associer des comportements (transitions graphiques et appels aux fonctions externes) aux éléments graphiques de l'interface ;
- la possibilité de générer des interfaces multilingues.

Tout au long du développement du générateur, deux problèmes majeurs ont été rencontrés :

- l'indépendance par rapport à la plate-forme, la résolution et au langage entrainent une complexité accrue de la génération d'interface. En effet, un certain nombre de choix et de calculs sont déplacés du designer de l'interface vers le programmeur du générateur. C'est le cas par exemple du positionnement et du dimensionnement des éléments graphiques étudiés à la section 8.4.4 ;
- bien que Flash possède, à priori, toutes les qualités requises pour la création du générateur, il semble que ce langage ne soit pas encore très stable. En effet, il arrive que Flash s'emmèle les pinceaux et mélange les propriétés privées des objets. Ceci entraine bien entendu la perte de tout l'avantage de la programmation orientée objet (puisque les notions d'encapsulation et de polymorphisme sont à remettre en doute). En outre, ce type d'erreurs est très difficile à repérer, d'autant plus que l'outil de *débugage* de Flash laisse à désirer. Ces problèmes sont probablement liés au jeune âge du langage ActionScript 2.0 de Flash qui date d'octobre 2003, mais n'en sont pas moins conséquents.

Quatrième partie

Exemples concrets de génération d'applications

Chapitre 9

Premier exemple : le slideshow

L'objectif de cette première application exemple est de tester la possibilité d'associer des transitions graphiques (voir 8.5.3) à la survenance d'un événement. En l'occurence, l'effet désiré est celui d'un bouton permettant de naviguer d'une photo à la suivante.

Comme c'est le cas pour toute description d'interface usiXML, la base du slideshow est constituée d'une fenêtre et de son *box* principal (ici de type vertical). La figure 9.1 illustre comment ce dernier est partagé en deux *box* de type *Stack*, ce qui implique que les éléments qu'ils comportent se superposeront.

Alors que le premier sous-box acceuillera les images, le second contiendra les boutons. En effet, bien qu'un seul bouton ne soit visible, la présence d'un bouton par image est nécessaire. Ceci est dû au fait que seul les transitions sont utilisées dans cet exemple et qu'il ne peut donc y avoir de script permettant au bouton d'afficher une image en fonction de celle qui est couramment affichée.

Le principe du fonctionnement du slideshow est dès lors très simple :
- au lancement de l'application, seuls la première image et le premier bouton ont leur propriété de visibilité à *true* :
- lorsque l'utilisateur enfonce le premier bouton (alors le seul visible) :
  - le premier bouton est rendu invisible et le deuxième est rendu visible ;
  - la première image est rendue invisible et la seconde est rendue visible.

Bien entendu, ces changements de visibilité sont effectués :

FIG. 9.1 – Illustration du résultat du slideshow et de sa décomposition en *box*.

- de manière abrupte (transitions open et close) en ce qui concerne les boutons ;
- de manière fluide (transition fadeIn, fadeOut, boxIn, boxOut) en ce qui concerne les images.

Le code source usiXML complet du slideshow se trouve en annexe A.

Chapitre 10

Deuxième exemple : la calculatrice

L'implémentation d'une calculatrice constitue un exemple classique en programmation. Dans le cadre de ce mémoire, il vise à démontrer la possibilité de générer des interfaces utilisateurs à partir d'une description usiXML faisant appel à des fonctions externes.

Rappelons que ces fonctions :
- doivent être écrites en ActionScript 2.0 ;
- peuvent être précompilées au format compressé de Flash (*SWF*) ou compilée dynamiquement par FlashiXML à l'aide de la librairie MING de PHP. Dans le deuxième scénario les fonctions peuvent être entreposées dans un fichier texte ou extraites d'une base de données.

La figure 10.1 illustre le résultat de l'application calculatrice qui, comme toute bonne calculatrice, permet d'additionner, soustraire, multiplier et diviser des nombres. Ces calculs impliquent l'exécution d'opérations telles que :
- enregistrer le premier terme de l'opération ;
- entreposer le type de l'opération ;
- récupérer le deuxième terme ;
- effectuer l'opération ;
- afficher le résultat.

Tous ces calculs se font par l'intermédiaire d'appels à des fonctions ActionScript chargées à partir d'un simple fichier texte et compilées dynamiquement. Nous ne détaillerons pas le fonctionnement de ces fonctions de calcul, mais le lecteur intéressé est invité à se référer au code source en annexe B.

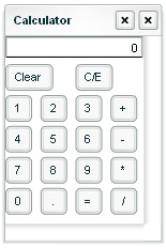

FIG. 10.1 – Illustration du résultat de la calculatrice.

Chapitre 11

Troisième exemple : le magasin virtuel

Lorsqu'il s'agit de mettre en évidence l'ergonomie d'une application, c'est souvent le *magasin virtuel* qui sert d'exemple. Les figures 11.1 et 11.2 illustrent les deux phases du magasin virtuel créé avec FlashiXML.

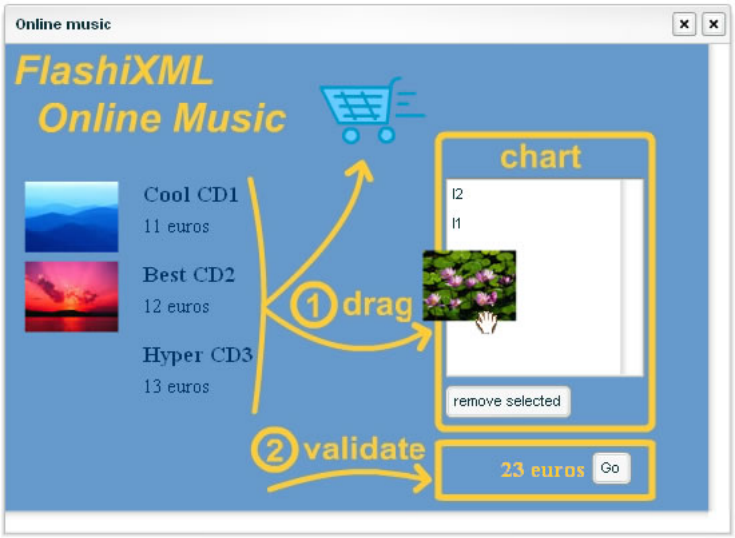

FIG. 11.1 – Illustration de la première étape du *magasin virtuel* de FlashiXML.

Dans la première phase, l'utilisateur est invité à glisser les articles de son choix dans le *panier* ou dans la *liste des produits déjà sélectionnés*, ce qui aura pour effet de les rajouter à la liste et de modifier le prix total des achats. A la figure 11.1 l'utilisateur est en train d'ajouter l'article *Hyper*

FIG. 11.2 – Illustration de la deuxième étape du *magasin virtuel* de FlashiXML.

CD3 dans la liste des éléments déjà sélectionnés.

Si toutefois l'utilisateur relâche l'article en dehors des deux zones indiquées, ce dernier rejoindra automatiquement son emplacement initial. En cas d'erreur, l'utilisateur peut sélectionner un élément de la liste et enfoncer le bouton *remove selected* afin d'écarter l'élément sélectionné.

Une fois satisfait de son choix, l'utilisateur est invité à confirmer le total calculé dynamiquement par l'application (grâce aux fonctions externes compilées dynamiquement), ce qui aura pour conséquence de basculer vers le deuxième volet de l'interface à l'aide d'une transition graphique.

Dans cette deuxième phase, l'utilisateur peut encoder les données demandées et envoyer le formulaire. Bien que cette dernière étape n'ait pas d'impact dans le cadre de cette application exemple, FlashiXML est capable de récolter les données du formulaire et de les envoyer par e-mail ou de les entreposer dans une base de données.

L'exemple du *magasin virtuel* de FlashiXML (dont le code est disponible sur le CD-Rom accompagnant le présent mémoire) permet ainsi de démontrer la possibilité de développer de véritables applications alliant fonctions externes et transitions graphiques et, profitant des avantages de Flash en matière d'animation vectorielle.

Cinquième partie

Conclusion générale

Chapitre 12

Rappel des objectifs et description de l'état actuel du système

Le générateur, baptisé FlashiXML, développé dans le cadre du présent mémoire a pour objectif de pallier les difficultés de mises à jour des interfaces utilisateur liées à l'évolution rapide des technologies actuelles.

Pour répondre à cet objectif, deux points principaux ont été mis en exergue. Le premier concerne la nécessité d'une interface générée à partir d'une description indépendante de son contexte d'exécution[1].

Dès lors, le choix du langage source s'est porté sur usiXML qui présente, outre une indépendance par rapport au contexte d'exécution, des avantages en matière de :
- couverture des éléments et des modèles ;
- qualité de définition de la sémantique ;
- support par les outils.

Le deuxième point est lié à la nécessité que l'interface générée soit vectorielle, et donc facilement redimensionnable, ce qui augmente sa portabilité. Nous avons dès lors choisi comme langage cible Macromedia Flash qui présente l'avantage de pouvoir être exécuté sur n'importe quelle plate-forme à condition que celle-ci dispose d'un plug-in.

Après avoir déterminé les langages source et cible, nous avons procédé à l'étude de leurs règles de correspondances. Cette étude a permis de mettre en évidence l'absence de certains composants Flash indispensables et, dès

[1] plate-forme, langage de génération, langue de l'utilisateur,...

lors, la nécessité de les implémenter.

En ce qui concerne l'architecture du générateur, nous avons opté en faveur d'une solution hybride permettant de reproduire, selon les besoins, le fonctionnement d'un interpréteur orienté serveur ou d'un compilateur. Grâce à ce choix notre générateur bénéficie des avantages de ces deux méthodes.

En outre, FlashiXML permet également :
- de générer des interfaces multilingues proposant un choix de langue à l'utilisateur lors du lancement de l'application ;
- de répartir les différentes ressources (fichier de configuration, description de l'interface,...) selon plusieurs configurations (client uniquement, client et serveur distant...) ;
- d'associer le déclenchement de comportements à la survenance d'événements des éléments graphiques de l'interface. Le résultat de ces comportements peut être une combinaison de transitions graphiques et/ou d'exécutions de fonctions externes à la description (pré-compilées ou compilées dynamiquement).

Chapitre 13

Analyse critique du système

Les principaux avantages que comporte le générateur d'interface FlashiXML développé dans le cadre du présent mémoire, sont, par rapport aux autres solutions existantes en matière de génération d'inferface :

- la possibilité d'être exécuté soit selon le fonctionnement d'une application compilée et indépendante, soit selon le fonctionnement d'un compilateur orienté serveur dynamique ;
- la portabilité. Le générateur peut être exécuté sur tout système possédant le plug-in Flash. Ce dernier est actuellement disponible d'une part, sur la majorité des systèmes d'exploitations d'ordinateurs, mais également sur la majorité des PDA ainsi que sur certains GSM et certains contrôleurs d'installations (contrôleurs d'alarme, de domotique, d'installations industrielles,...). En outre, le côté vectoriel de Flash lui permet de redimensionner facilement l'application en fonction de la résolution d'écran. Il est également à l'origine du faible poids de l'application (environ 100KB) ;
- les différentes configurations possibles. En effet, celles-ci permettent de répartir les différentes resources nécessaires (générateur, description de l'interface, fichier de configuration,...) de manière très flexible sur plusieurs machines (client, serveur distant, ...), ce qui permet d'automatiser la distributions de nouvelles versions de l'application ;
- le fait que la description d'interface soit indépendante du contexte et qu'une même description puisse donc être réutilisée facilement dans différents contextes (plate-forme, langue de l'utilisateur,...) sans devoir être adaptée. Ceci aillant pour conséquence de faciliter grandement la mise à jour.

A côté de ces nombreux avantages, FlashiXML présente toutefois quelques inconvénients tels le fait que :

- à ce stade, seuls les éléments de graphiques de base de usiXML ont été implémentés ;

– le générateur est relativement capricieux par rapport à la syntaxe du code usiXML ;
– les éléments de l'interface ne sont personalisables graphiquement ;
– lorsque FlashiXML fonctionne en mode *compilation dynamique*, la compilation se fait à l'aide de la librairie MING de PHP qui est encore en cours de développement et pourrait dès lors évoluer rapidement ;
– le langage Flash, dans lequel l'interface est générée, semble parfois réagir de manière imprévisible. A titre d'exemple, les règles d'encapsulation et de polymorphisme ne sont pas toujours respectées. En effet, lorsque plusieurs objets possèdent un tableau de références vers d'autres objets, il semble que Flash accumule l'ensemble des références vers tous les objets dans tous les tableaux. Suite à ces erreurs, certaines parties du code ont du être adaptées.

Chapitre 14

Directions pour l'évolution du développement

Le présent mémoire a permis d'explorer une nouvelle voie en matière de génération d'interfaces vectorielles qui s'est avérée intéressante. Toutefois, à ce stade, seules les bases ont pu être jetées et de nombreuses améliorations sont dès lors envisageables. Nous entrevoyons en particulier, et par ordre d'importance, les possibilités d'améliorations suivantes :

1. ré-implémenter les composants Flash fournis par Macromedia afin de les remplacer par des composants taillés sur mesure, ce qui présente plusieurs avantages :
   - Augmenter la souplesse d'utilisation des scripts.
   - la possibilité de traîter directement avec les composants et ainsi supprimer la couche intermédiaire *FUI* qui gère la correspondance entre les éléments usiXML et les composants Flash. Cette suppression aurait sans doute des effets positifs sur les performances de la génération qui, bien qu'elles paraissent satisfaisantes sur ordinateur, pourraient ne pas l'être sur des PDA ou des GSM.
   - la possibilité de compléter les composants qui ne possèdent actuellement pas toutes les propriétés nécessaires. A titre d'exemple, le composant Flash utilisé pour représenter une fenêtre n'est pas redimensionnable.
   - Les composants Flash ne sont personnalisables graphiquement que de manière statique. De ce fait, il est nécessaire de recompiler FlashiXML après chaque changement dans l'apparence des composants. En outre, une version différente de FlashiXML est alors nécessaire pour chaque interface ayant une apparence personnalisée. Il va de soi que tout l'intérêt de FlashiXML est alors perdu. Si les composants sont ré-implémentés, il est possible de prévoir la possibilité de changer l'apparence dynamiquement.
2. étudier la possibilité de charger dynamiquement du contenu à partir

d'une base de donnée *lors de la survenance d'un événement.*

3. étudier la possibilité de ré-implémenter FlashiXML sous forme de composant, ce qui permettrait :
   - d'intégrer un composant Flash dans une application Flash qui pourrait ensuite elle-même être chargée dans un application C++, .Net,...
   - d'intégrer plusieurs *composants FlashiXML* dans une application Flash. Par exemple, un site internet pourrait être décrit par un composant FlashiXML pour le menu et un par rubrique, chacun chargeant et générant le morceau d'interface qui le concerne. Ceci permettrait d'augmenter la modularité et la réutilisabilité des descriptions d'interfaces.

Sixième partie

Bibliographie

Références bibliographiques

[1] Scalable vector graphics (svg) : Xml graphics for the web, 2004. http ://www.w3c.org/Graphics/SVG/.

[2] A.Arsanjani, D.Chamberlain, and et al. (wsxl)web service experience language version, 2002, 2002. disponible à http ://www-106.ibm.com/develperworks/library/ws-wsxl2.

[3] P.Forbrig A.Müller and C.H.Cap. Model-based user interface design using markup concepts., 2001. In Ch.Johnson (Eds),editor, In Proc. Of 8th International Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'2001 (Glasgow,13-15 juin 2001),pages 16-27, Berlin,2001. Springer-Verlag.

[4] Waleed Anbar. Exploring version 2 of the macromedia flash mx 2004 component architecture. 2004. Disponible en ligne : http ://www.macromdia.com/devnet/mx/flash/articles/-component_architecture_print.html.

[5] Carlo Blatz et le Powerflasher. *Macromedia Flash MX ActionScript, Référence*. MICRO APPLICATION, 2002.

[6] G.Zimmermann, G.Vanderheiden, and A.Gilman. Universal remote console - prototyping for the alternate interface access standard., 2002.

[7] Andreas Hein. Building the footernav component. 2004. Disponible en ligne : http ://www.macromdia.com/devnet/mx/flash/-articles/footer_component.html.

[8] Macromedia Inc. Devnet resource kit volume 6, 2004. Disponible en ligne : http ://www.macromedia.com/software/drk/productinfo/-product_overview/volume6/.

[9] Macromedia Inc. Macromedia flex : the presentation tier solution for delivering enterprise rich internet applications, white paper, 2004.

[10] M.Argollo Jr. and C.Olguin. Graphical user interface portability, 1997. Cross Talk : The Journal of Defense Software Engineering,10(2) :14-17,1997.

[11] Chafic Kazoun. Skinning the flash mx 2004 components. 2004. Disponible en ligne : http ://www.macromdia.com/devnet/mx/flash/-articles/skinning_2004_print.html.

[12] Jody Keating and Fig Leaf Software. *Inside Flash.* New Riders Publishing, 2002.

[13] K.Luyten, C.Vandervelpen, and K.Coninx. Adaptable user interfaces in component based development for embedded systems., 2002. In Proceedings of the 9th Int. Workshop on Design, Specification,and Verification of Interactive Systems DSV-IS'2002, (Rostock, June 12-14,2002). Springer Verlag, 2002.

[14] Philipe Rebouillat & Vincent Leroudier. *Flash MX 2004 : faites vos jeux. Programmation et scénarios.* DUNOD, Planète numérique, 2003.

[15] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, and D. Trevisan. Usixml : A user interface description language for context-sensitive user interfaces, 2004. Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML : Advances on User Interface Description Languages" (Gallipoli, May 25, 2004), K. Luyten, M. Abrams, Q. Limbourg, J. Vanderdonckt (Eds.), Gallipoli, 2004, pp. 55-62. Accessible at http ://www.edm.luc.ac.be/uixml2004/-index.php ?selected=program.

[16] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lopez. a language supporting multi-path development of user interfaces, 2004. Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004 (Hamburg, July 11-13, 2004).

[17] M.Abrams, C.Phanouriou, A.L.Batongbacal, S.Williams, and title= J.Shuster, 1999.

[18] Roberto Tamassia Michael T. Goodrich. *Data structures and algorithms in Java, second edition.* John Wiley & Sons, Inc., 1997.

[19] Guylaine Monnier. *Flash MX : Animation, interactivité, ActionScript.* DUNOD, Planète numérique, 2002.

[20] Colin Moock. *ActionScript pour Flash MX : La référence.* O'Reilly, 2003.

[21] Paternò.F and Santoro.C. One model, many interfaces., 2002. In Ch Kolski and J.Vanderdonckt (Eds.),editors,Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes,15-17 May 2002), pages 143-154,Dordrecht,2002. Kluwer Academics Publishers.

[22] R.Merrick. Device independent user interfaces in xml,2001, 2001. disponible à http ://www.belchi.be/event.htm.

[23] Steve Roberts. Flash and svg, 2003. http ://broadway.cs.nott.ac.uk/-projects/SVG/flash2svg/.

[24] Steve Roberts. How flash animation is managed by an swf to svg translation packet, 2003. http ://broadway.cs.nott.ac.uk/projects/SVG/-flash2svg/swfsvganim.html.

[25] Steve Roberts. Semantics of macromedia's flash (swf) format and its relationschip to svg, 2003. http ://www.eprg.org/-projects/SVG/-flash2svg/swfformat.html.

[26] N. Souchon and J. Vanderdonckt. A review of xml-compliant user interface description languages, 2003. Proc. of 10th Int. Conf. on Design, Specification, and Verification of Interactive Systems DSV-IS'2003 (Madeira, 4-6 June 2003), Jorge, J., Nunes, N.J., Falcão e Cunha, J. (Eds.), Lecture Notes in Computer Science, Vol. 2844, Springer-Verlag, Berlin, 2003, pp. 377-391.

[27] G.A. Leierer & R. Stoll. *PHP4 & MySQL*. MICRO APPLICATION, Grand Livre, 2001.

[28] T.Ball, Ch.Colby, P.Danielsen, L.J. Jagadeesan, R.Jagadeesan, K.Läufer, P.Matag, and K.Rehor. Sisl : Several interfaces, single logic., 2000. Technical report, Loyola University, Chicago, January 6th,2000.

[29] Departement of Information Systems Université Catholique de Louvain. usixml website, 2004. Disponible à http ;//www.usixml.org.

[30] Matt Voerman. *Flash MX 2004 certified developper guide*. Macromedia Press, 2004.

[31] Xulplanet. Xul tutorial,2003., 2003. disponible à l'adresse http ://www.xulplanet.com/tutorials/xultu/.

Septième partie

Annexes

Annexe A

Code source de l'application exemple *Slideshow*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<uiModel xmlns="http://www.usixml.org">
<version modifDate="2004-08-09T13:39:56.198+02:00" xmlns="">1</version>
<authorName xmlns="">Vanden Berghe Youri</authorName>
<comment xmlns="">Manually written</comment>

<cuiModel id="calculatrice" name="calculatrice">

<window height="400" width="400" id="window_0" name="window_0">

<box borderWidth="0" isBalanced="false" relativeHeight="100"
relativeWidth="100" type="vertical" isVisible="true" id="box" name="box">

<box borderWidth="0" isBalanced="true" relativeHeight="70"
relativeWidth="100" type="stack" isVisible="true" id="box_I" name="box_I">
<imageComponent id="I0" defaultContent="data/images/photo1.jpg"
width="300" height="225" isVisible="true"/>
<imageComponent id="I1" defaultContent="data/images/photo2.jpg"
width="300" height="225" isVisible="false"/>
<imageComponent id="I2" defaultContent="data/images/photo3.jpg"
width="300" height="225" isVisible="false"/>
</box>


<box borderWidth="0" isBalanced="true" relativeHeight="30"
relativeWidth="100" type="stack" isVisible="true" id="box_B" name="box_B">
<button width="100" height="25" isVisible="true" id="B0"
defaultContent="next" name="B0">
```

```
<behavior id="behav0">
<event id="evt0" eventType="depress" eventContext="B0"/>
<action id="act0">
<transition transitionIdRef="Tr01"/>
<transition transitionIdRef="Tr02"/>
<transition transitionIdRef="Tr03"/>
<transition transitionIdRef="Tr04"/>
</action>
</behavior>
</button>

<button width="100" height="25" isVisible="false" id="B1"
defaultContent="next" name="B1">
<behavior id="behav1">
<event id="evt1" eventType="depress" eventContext="B1"/>
<action id="act1">
<transition transitionIdRef="Tr11"/>
<transition transitionIdRef="Tr12"/>
<transition transitionIdRef="Tr13"/>
<transition transitionIdRef="Tr14"/>
</action>
</behavior>
</button>

<button width="100" height="25" isVisible="false" id="B2"
defaultContent="next" name="B2">
<behavior id="behav2">
<event id="evt2" eventType="depress" eventContext="B2"/>
<action id="act2">
<transition transitionIdRef="Tr21"/>
<transition transitionIdRef="Tr22"/>
<transition transitionIdRef="Tr23"/>
<transition transitionIdRef="Tr24"/>
</action>
</behavior>
</button>
</box>

</box>
</window>

<graphicalTransition id="Tr01" transitionType="boxOut">
<source id="B0"/>
<target id="I0"/>
```

```
</graphicalTransition>
<graphicalTransition id="Tr02" transitionType="boxIn">
<source id="B0"/>
<target id="I1"/>
</graphicalTransition>
<graphicalTransition id="Tr03" transitionType="close">
<source id="B0"/>
<target id="B0"/>
</graphicalTransition>
<graphicalTransition id="Tr04" transitionType="open">
<source id="B0"/>
<target id="B1"/>
</graphicalTransition>

<graphicalTransition id="Tr11" transitionType="boxOut">
<source id="B1"/>
<target id="I1"/>
</graphicalTransition>
<graphicalTransition id="Tr12" transitionType="boxIn">
<source id="B1"/>
<target id="I2"/>
</graphicalTransition>
<graphicalTransition id="Tr13" transitionType="close">
<source id="B1"/>
<target id="B1"/>
</graphicalTransition>
<graphicalTransition id="Tr14" transitionType="open">
<source id="B1"/>
<target id="B2"/>
</graphicalTransition>

<graphicalTransition id="Tr21" transitionType="boxOut">
<source id="B2"/>
<target id="I2"/>
</graphicalTransition>
<graphicalTransition id="Tr22" transitionType="boxIn">
<source id="B2"/>
<target id="I0"/>
</graphicalTransition>
<graphicalTransition id="Tr23" transitionType="close">
<source id="B2"/>
<target id="B2"/>
</graphicalTransition>
<graphicalTransition id="Tr24" transitionType="open">
```

```
<source id="B2"/>
<target id="B0"/>
</graphicalTransition>
    </cuiModel>
</uiModel>
```

Annexe B

Code source de l'application exemple *Calculatrice*

B.1 Fonctions ActionScript

```
// called when equal sign is pressed
// finds back registered values and applies operation
_root.doOperation = function(){
    r1=Number(_root.reg1);
    r2=Number(_root.reg2);
    if(_root.operation == "add") {
        _root.result=r1+r2;
        _root.resetAfterOp();
    }
    if(_root.operation == "substract") {
        _root.result=r1-r2;
        _root.resetAfterOp();
    }
    if(_root.operation == "multiply") {
        _root.result=r1*r2;
        _root.resetAfterOp();
    }
    if(_root.operation == "divide") {
        _root.result=r1/r2;
        _root.resetAfterOp();
    }
};

// called when a number button is pressed
// saves values to registers
_root.addNumber = function(n){
```

```
    if(_root.currentRegister==undefined){
        _root.currentRegister="reg1";
    }
    if (_root.operation=="none" && _root.currentRegister=="reg2"){
        _root.reg1="";
        _root.setCurrentRegister();
    }
    _root[_root.currentRegister]+=n;
    _root.setDisplay(_root.currentRegister);
};

// clears the textComponent
_root.clearEntry = function(){
    _root[_root.currentRegister]="";
    _root.setDisplay(_root.currentRegister);
};

// resets all variables to initial value
_root.clearAll = function(){
    _root.reg1="";
    _root.reg2="";
    _root.setCurrentRegister();
    _root.setOperation("none");
    _root.setDisplay(_root.currentRegister);
};

// called when an operation button is pressed
// saves the operation to apply
_root.setOperation = function(operation){
    _root.operation=operation;
    _root.setCurrentRegister();
};

// sets textComponent content to the value of the given register
_root.setDisplay = function(register){
    temp = _root.getItemById("label_0");
    temp.setContent(_root[register]);
};

// sets the current register according to which is empty
_root.setCurrentRegister = function(){
    if (_root.reg1==""){
        _root.currentRegister="reg1";
    }
```

```
    else {
        _root.currentRegister="reg2";
    }
};

// saves operation result in register reg1 and displays it
// in the textComponent
_root.resetAfterOp = function(){
    _root.reg1=String(_root.result);
    _root.reg2="";
    _root.setDisplay("reg1");
    _root.setOperation("none");
};
```

B.2 Code usiXML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<uiModel xmlns="http://www.usixml.org">
<version modifDate="2004-08-09T13:39:56.198+02:00" xmlns="">1</version>
<authorName xmlns="">Vanden Berghe Youri</authorName>
<comment xmlns="">Manually written</comment>

<cuiModel id="calculatrice" name="calculatrice">

<window height="225" width="152" id="window_0" name="Calculator">

<box borderWidth="0" relativeWidth="100" isBalanced="false"
type="vertical" id="main">
<box borderWidth="0" isBalanced="false" relativeHeight="18"
relativeWidth="30" type="horizontal" id="calcDisplay">
<textComponent defaultContent="0" textHorizontalAlign="right"
numberOfColumns="43" isEditable="false" numberOfLines="1" name="TC1"
id="TC1"/>
</box>

<box borderWidth="0" isBalanced="true" relativeHeight="20"
relativeWidth="102" type="horizontal" id="boxTC" name="boxTC">
<button isEnabled="true" isVisible="true" defaultContent="Clear"
id="clear">
<behavior id="behavClear">
<event id="evtClear" eventType="release" eventContext="clear"/>
<action id="actClear">
<methodCall methodName="clearAll"/>
```

```
</action>
</behavior>
</button>
<button isEnabled="true" isVisible="true" defaultContent="C/E" id="CE">
<behavior id="behavCE">
<event id="evtCE" eventType="release" eventContext="CE"/>
<action id="actCE">
<methodCall methodName="clearEntry"/>
</action>
</behavior>
</button>
</box>


<box borderWidth="0" isBalanced="true" relativeHeight="20"
relativeWidth="102" type="horizontal" id="box_1" name="box_1">
<button isEnabled="true" isVisible="true" defaultContent="1" id="b1">
<behavior id="behav1">
<event id="evt1" eventType="release" eventContext="b1"/>
<action id="act1">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="1"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="2" id="b2">
<behavior id="behav2">
<event id="evt2" eventType="release" eventContext="b2"/>
<action id="act2">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="2"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="3" id="b3">
<behavior id="behav3">
<event id="evt3" eventType="release" eventContext="b3"/>
<action id="act3">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="3"/>
```

```
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="+" id="Add">
<behavior id="behavAdd">
<event id="evtAdd" eventType="release" eventContext="Add"/>
<action id="actAdd">
<methodCall methodName="setOperation">
<methodCallParam name="param1" value="add"/>
</methodCall>
</action>
</behavior>
</button>
</box>


<box borderWidth="0" isBalanced="true" relativeHeight="20"
relativeWidth="102" type="horizontal" id="box_2" name="box_2">
<button isEnabled="true" isVisible="true" defaultContent="4" id="b4">
<behavior id="behav4">
<event id="evt4" eventType="release" eventContext="b4"/>
<action id="act4">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="4"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="5" id="b5">
<behavior id="behav5">
<event id="evt5" eventType="release" eventContext="b5"/>
<action id="act5">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="5"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="6" id="b6">
<behavior id="behav6">
```

```
<event id="evt6" eventType="release" eventContext="b6"/>
<action id="act6">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="6"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="-" id="Substract">
<behavior id="behavSubstract">
<event id="evtAdd" eventType="release" eventContext="Substract"/>
<action id="actSubstract">
<methodCall methodName="setOperation">
<methodCallParam name="param1" value="substract"/>
</methodCall>
</action>
</behavior>
</button>
</box>


<box borderWidth="0" isBalanced="true" relativeHeight="20"
relativeWidth="102" type="horizontal" id="box_3" name="box_3">
<button isEnabled="true" isVisible="true" defaultContent="7" id="b7">
<behavior id="behav7">
<event id="evt7" eventType="release" eventContext="b7"/>
<action id="act7">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="7"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="8" id="b8">
<behavior id="behav8">
<event id="evt8" eventType="release" eventContext="b8"/>
<action id="act8">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="8"/>
</methodCall>
</action>
</behavior>
```

```
</button>

<button isEnabled="true" isVisible="true" defaultContent="9" id="b9">
<behavior id="behav9">
<event id="evt9" eventType="release" eventContext="b9"/>
<action id="act9">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="9"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="*" id="Multiply">
<behavior id="behavMultiply">
<event id="evtAdd" eventType="release" eventContext="Multiply"/>
<action id="actMultiply">
<methodCall methodName="setOperation">
<methodCallParam name="param1" value="multiply"/>
</methodCall>
</action>
</behavior>
</button>


</box>
<box borderWidth="0" isBalanced="true" relativeHeight="20"
relativeWidth="102" type="horizontal" id="box_4" name="box_4">
<button isEnabled="true" isVisible="true" defaultContent="0" id="b0">
<behavior id="behav0">
<event id="evt0" eventType="release" eventContext="b0"/>
<action id="act0">
<methodCall methodName="addNumber">
<methodCallParam name="param1" value="0"/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="." id="Dot">
<behavior id="behavDot">
<event id="evtDot" eventType="release" eventContext="Dot"/>
<action id="actDot">
<methodCall methodName="addNumber">
```

```
<methodCallParam name="param1" value="."/>
</methodCall>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="=" id="Equals">
<behavior id="behavEquals">
<event id="evtEquals" eventType="release" eventContext="Equals"/>
<action id="actEquals">
<methodCall methodName="doOperation"/>
</action>
</behavior>
</button>

<button isEnabled="true" isVisible="true" defaultContent="/" id="Divide">
<behavior id="behavDivide">
<event id="evtAdd" eventType="release" eventContext="Divide"/>
<action id="actDivide">
<methodCall methodName="setOperation">
<methodCallParam name="param1" value="divide"/>
</methodCall>
</action>
</behavior>
</button>
</box>

</box>
</window>
</cuiModel>
</uiModel>
```

Annexe C

Code source de l'application exemple *Magasin virtuel*

Pour une question de concision, le code de l'application *Magasin virtuel* n'a pas été inséré dans le présent mémoire. Il est toutefois disponible en version électronique sur le CD-Rom qui accompagnant le document.

Annexe D

Code source PHP-MING permettant la compilation dynamique de fonctions ActionScript

```
// set typical movie variables
ming_setScale(20.00000000);
ming_useswfversion(5);

// create swf movie
$movie=new SWFMovie();
$movie->setDimension(100,100);

// read the functions from the textFile
// $file variable recieved by GET method
$file = $_GET['file'];
$fd = fopen($file, "r");
$functions = fread($fd, filesize($file));
fclose($fd);

// parse functions and add them to the SWF movie
$functions_array = split("\r\n\r\n", $functions);
foreach($functions_array as $function){
$strAction = str_replace("\r\n", "", $function);
$movie->add(new SWFAction($strAction));
}

// save movie as functions.swf on disk
$movie->save("functions.swf");
```

```
// read functions.swf to be able to send it as a header in this page
header("Content-type: application/x-shockwave-flash");
$img = readfile("functions.swf");
echo $img;
```