Mémoire :

Implémentation d'un animateur de transitions entre interface graphique initiale et interface adaptée

Promoteur :
Jean Vanderdonckt

Mémoire présenté par
Charles-Eric Dessart
en vue de l'obtention du titre de
Master 60 crédits en sciences de gestion

Année académique 2010-2011

Université catholique de Louvain

Remerciements

Je tiens à remercier dans un premier temps mon promoteur, M. Jean Vanderdonckt, pour m'avoir accompagné et guidé tout au long de l'élaboration de ce mémoire. Je lui témoigne toute ma reconnaissance pour cette expérience enrichissante et pleine d'intérêts qu'il m'a permis de traverser durant cette année.

Je remercie aussi ma famille et mes amis, et plus particulièrement mes parents, pour le soutien dont ils m'ont fait part tout au long de mes études.

Finalement, je remercie Jean-Jacques Goldman, Vaya con dios, Rossini, et tous les autres pour avoir égayé mes longues journées de travail.

Table des matières

Table des figures

Introduction

De nos jours, il n'est pas rare d'utiliser une même application sur différentes plates-formes comme par exemple *iTunes*©. Cette application est utilisable sur un ordinateur mais aussi sur un *smartphone*. L'explosion du nombre de plates-formes augmente ce phénomène. L'utilisateur final se retrouve de plus en plus face à des interfaces qu'il doit réapprendre à utiliser.

Cette exportation d'applications sur différentes plates-formes pose une double problématique. Premièrement, le coût de développement des applications augmente de manière significative à cause des contraintes imposées par chaque plate-forme. La plupart des applications disponibles sur plusieurs plates-formes disposent d'une équipe de développement par plate-forme [3]. Et deuxièmement, l'utilisateur finale éprouve des difficultés à se réapproprier une interface disponible sur une nouvelle plate-forme. Cette perturbation de l'utilisateur est appelée la déstabilisation cognitive.

Ce mémoire s'insère dans ce contexte particulier. Il aborde les deux problématiques en ayant pour but la conception et le développement d'un prototype de transitions animées entre deux interfaces. La déstabilisation cognitive peut être réduite si l'utilisateur final constate les changements qui ont eu lieu entre l'interface qu'il avait l'habitude d'utiliser et la nouvelle interface. La fonctionnalité principale du prototype est de scénariser une suite de transitions animées permettant de montrer à l'utilisateur tous ces changements. De plus, le langage utilisé pour spécifier l'interface originelle est l'*UsiXML*. Ce langage de spécification d'interfaces peut être utilisé pour représenter une interface en fonction de son contexte d'utilisation [5]. Ce mémoire apporte sa pierre à l'édifice en utilisant ce langage qui pourrait réduire énormément les coûts de développement s'il était plus largement utilisé.

L'approche suivie dans la réalisation d'un tel outil logiciel est celle préconisée en ingénierie des exigences logicielles. Cette approche permet d'aborder des problématiques complexes afin d'en tirer l'utile et l'essentiel pour la conception de logiciels critiques. La conception de logiciels adaptés aux besoins nécessite au préalable une compréhension précise du domaine ainsi que de bonnes hypothèses sur l'environnement du logiciel. Sans quoi il y a un risque de ne pas atteindre les exigences souhaitées lors de l'élaboration du logiciel en question.

Lors de ce travail, la collecte d'informations s'est limitée à plusieurs articles ainsi qu'à plusieurs vidéos montrant des animations. Cette étape a permis de dégager une

vue d'ensemble du domaine afin de préciser la portée de l'étude.

Les objectifs principaux de ce mémoire sont :

- Premièrement, l'étude et l'analyse du besoin d'harmoniser la spécification d'interfaces et du problème corollaire que cela induit à savoir la déstabilisation cognitive.

- Deuxièmement, la conception d'un système de scénarisation permettant de montrer à l'utilisateur final les changements entre deux interfaces.

- Troisièmement, à partir de ces analyses, le développement d'un prototype proposant toutes les caractéristiques indispensables.

Ce mémoire est organisé comme suit :

- Le *chapitre 1* se penche sur l'explosion du nombre d'applications utilisables sur plusieurs plates-formes ainsi que les problèmes que cela engendre.

- Le *chapitre 2* présente le langage de transition développé, les opérations d'adaptation existantes, les différents types d'animation ainsi que les différentes façons de contrôler une animation.

- Le *chapitre 3* détaille les fonctionnalités, l'architecture et l'implémentation du prototype développé.

- Le *chapitre 4* présente le fonctionnement du prototype sur deux cas d'étude.

Chapitre 1

Vers une multiplicité des interfaces graphiques selon la plate-forme

Ce chapitre aborde deux problématiques pour préciser le contexte dans lequel s'insère ce mémoire. La section 1.1 décrit le besoin d'harmoniser la spécification d'interfaces pour faciliter leur exportation sur de multiples plates-formes et leur adaptation en fonction de leur contexte d'utilisation. En 1.2, la section aborde les conséquences corollaires de l'adaptation d'interfaces en fonction de la plate-forme, notamment la déstabilisation cognitive. Enfin, la section 1.3 présente brièvement le langage *UsiXML*. Ce langage de spécification d'interfaces multi plates-formes est utilisé par l'application développée dans le cadre de ce mémoire.

1.1 Harmoniser la spécification d'interfaces

Les systèmes informatiques font partie intégrante des organisations modernes. Les utilisateurs de ces systèmes informatiques doivent, la plupart du temps, faire face à une variété de plates-formes, mobiles et fixes, pour lesquelles ils s'attendent à trouver les mêmes fonctionnalités et données. Les plates-formes informatiques sont définies comme toute combinaison de matériel informatique et de logiciel sur lesquelles une interface peut être affichée. Ces plates-formes informatiques diffèrent par :

- leur type, par exemple un ordinateur de bureau, un ordinateur portable, un téléphone portable, un *smartphone*, etc.
- la taille de leur écran.
- le nombre de couleurs supportées.
- la librairie graphique utilisée pour afficher une interface.
- leur accessoire de pointage tel que la souris, un stylet ou aucun accessoire.
- leur accessoire d'encodage tel qu'un clavier d'ordinateur, un clavier numérique, etc.

Les nouveaux appareils apparaissent de plus en plus vite sur le marché d'où la volonté d'harmoniser la spécification d'interfaces. Développer un langage de spécification

d'interfaces destinées à être affichées sur de multiples plates-formes, pour permettre une exportation aisée de celles-ci, est une tâche difficile et prenant beaucoup de temps. Les développeurs et les chercheurs sont d'accord sur le fait que les méthodes actuelles, les outils et les langages ne sont pas assez développés pour permettre d'harmoniser la spécification d'interfaces efficacement. Le développement d'interfaces utilisateurs multi plates-formes souffre des limitations suivantes [3] :

- *Un manque de connaissances et d'expérience* : développer une interface utilisateur qui est adaptée à une plate-forme donnée demande ; une connaissance des langages de programmation supportés sur la plate-forme ciblée, une connaissance des fonctionnalités de l'appareil, etc. Ces informations ne sont pas toujours disponibles ou assez précises.

- *Un manque de méthodologie* : Si des méthodes pour développer des interfaces classiques existent, des méthodes pour développer plusieurs versions d'une interface utilisateur pour de multiples plates-formes à la fois sont quasiment inexistantes.

- *Un manque d'outils* : Peu d'outils sont spécifiquement dédiés à la spécification et à la programmation d'interfaces multi plates-formes si l'on considère les créateurs d'interfaces, les outils de prototypage, les outils de génération basés sur les modèles, ou les environnements de développement d'interfaces en général.

Ces trois déficiences majeures entraînent les problèmes suivants [3] :

- *Des coûts de développement et de maintenance élevés* : l'interface utilisateur a toujours représenté une part importante dans le développement d'un logiciel, même pour les applications traditionnelles développées pour une seule plate-forme. La diversité des plates-formes, en plus de la complexité toujours croissante des systèmes informatiques, ont augmenté les coûts de développement.

- *Un manque de cohérence* : la cohérence est un principe de base pour la création d'interfaces utilisateur. Avec la multiplication des plates-formes, les utilisateurs s'attendent à retrouver les mêmes fonctionnalités d'une plate-forme à une autre. Cependant, le développement est souvent géré par plusieurs équipes (chaque équipe développant une interface pour une plate-forme donnée) et à des moments différents, ceci ayant un impact significatif sur la cohérence.

- *Un manque d'adaptation* : une interface, en plus de devoir être utilisable sur toutes les plates-formes, doit être adaptée à chacune d'elles. L'adaptation de chaque interface est souvent négligée à cause du coût de développement élevé. Le résultat est généralement obtenu via des méthodes automatiques de *reauthoring* qui ne sont pas assez efficaces.

- *Un manque de réutilisabilité* : il existe un manque de techniques permettant de reproduire un composant graphique, une structure logique ou un *design* sur

plusieurs plates-formes.

– *Un manque de techniques explicitant la généricité* : les développeurs manquent de techniques leur permettant de spécifier une interface utilisateur de manière abstraite suivant la plate-forme et le contexte.

Cette volonté d'harmoniser la spécification d'interfaces met en lumière un autre problème : l'adaptation des interfaces. Bien qu'une interface devrait être représentable sur n'importe quelle plate-forme pour les raisons expliquées ci-dessus, elle ne sera généralement pas adaptée à toutes les plates-formes. Par exemple, il peut être nécessaire de mieux organiser les composants graphiques dans l'espace pour palier à une plate-forme avec un écran plus petit. Le chapitre suivant aborde la problématique de l'adaptation des interfaces graphiques.

1.2 Adaptation des interfaces

L'adaptation d'une interface utilisateur consiste en la modification partielle ou totale d'une interface de manière à satisfaire les exigences d'un utilisateur ou de la plate-forme où est affichée l'interface. Une adaptation est catégorisée en fonction de l'entité qui dirige le processus d'adaptation :

– L'adaptabilité est une adaptation qui permet à un utilisateur de modifier une interface.
– *L'adaptivité* est une adaptation qui est dirigée par le système lui même.

Il existe aussi des initiatives d'adaptation menées à la fois par l'utilisateur et le système.

L'adaptivité, bien qu'elle soit assez lourde à développer, engendre de nombreux bénéfices et est largement utilisée dans de nombreux domaines tels que l'automobile, le commerce électronique, l'algorithmie et les systèmes d'information. L'adaptivité incombe de nombreuses conséquences, notamment ; la *perturbation de l'utilisateur* causée par un comportement hasardeux de l'interface et *la perturbation cognitive* qui se manifestent lorsque l'utilisateur, confronté à la nouvelle interface, doit réapprendre à utiliser l'interface en imaginant des correspondances entre l'interface avant et après son adaptation. Entre l'interface avant l'adaptation et l'interface après l'adaptation, il n'y a aucun indice expliquant à l'utilisateur les changements qui ont eu lieu. Ce manque d'information sur l'adaptation renforce la perturbation cognitive. La psychologie cognitive réfère ce phénomène comme étant une *déstabilisation cognitive*. Ce phénomène signifie que l'utilisateur est mentalement déstabilisé quand il est confronté à des événements inattendus, sans précédents ou à du contenu non prédit. L'utilisateur reste à ce stade de la déstabilisation cognitive jusqu'à ce qu'une *re-stabilisation* restaure une relation entre l'ancien et le nouveau contenu. L'utilisateur ne souffre pas de ces conséquences avec l'adaptabilité étant donné que l'utilisateur garde le contrôle (il sait ce qu'il modifie), en opposition avec l'adaptivité où c'est le système qui prend le contrôle. La réduction de la déstabilisation cognitive est une problématique que l'outil d'animation

développé dans le cadre de ce mémoire tente de résoudre en montrant à l'utilisateur ce qui a changé voir expliquer pourquoi [6].

1.3 UsiXML

Dans le cadre de ce mémoire, le langage utilisé pour spécifier une interface devant être adaptée est l'*UsiXML*. Ce langage est indépendant de la plate-forme où est affichée l'interface et du contexte d'utilisation. L'explosion du nombre de plates-formes implique le développement de ce genre de langage pour éviter l'utilisation de langages plus contraignants selon les plates-formes/contextes. La déstabilisation cognitive étant une problématique corollaire de l'explosion du nombre de plates-formes, l'utilisation de l'*UsiXML* apparaît judicieuse pour garder une cohérence et pour expérimenter ce langage qui est encore au stade du développement. Cette section aborde brièvement la structure et les particularités de ce langage.

Le langage de spécification d'interface *UsiXML* permet à des développeurs de décrire plusieurs aspects d'une interface utilisateur. Selon ses exigences, un développeur peut adopter plusieurs points de vue d'une même interface. Dans les premières étapes de développement, il peut choisir de spécifier seulement les fonctionnalités de haut niveau ou le domaine des objets. Par la suite, il a toujours l'opportunité de détailler l'interface à l'aide de contrôles graphiques. Les différentes vues d'une interface, appelées *modèles* en *UsiXML*, sont catégorisées par plusieurs couches d'abstraction, toutes définies par le *Unified Reference Framework* [1].

Une spécification *UsiXML* est une combinaison de plusieurs modèles. Aucun de ces modèles n'est obligatoire et toute combinaison de modèles est autorisée. *UsiXML* propose huit types de modèle, comme illustré par la Figure 1.1 : un modèle des tâches, un modèle du domaine, le modèle *AUI*, le modèle *CUI*, un modèle de *mapping*, un modèle contextuel, un modèle des ressources et un modèle des transformations.

Les modèles des tâches et du domaine font référence au niveau *Tasks&Concepts* du *Unified Reference Framework*. Le modèle des tâches est une description des tâches effectuées par un utilisateur en interaction avec le système, alors que le modèle du domaine est une description des objets et des classes vues et manipulées par l'utilisateur.

Le modèle *AUI* (Abstract User Interface) fait référence au niveau d'abstraction suivant dans le *United Reference Framework*. Il est utilisé pour spécifier quels groupes de tâches et de concepts du domaine seront présentés ensemble (par exemple dans la même fenêtre).

Le modèle *CUI* (Concrete User Interface) est une spécification détaillée de l'apparence et du comportement des éléments d'une interface graphique.

Le modèle de *mapping* permet d'établir une relation entre les modèles ou des éléments de modèles (par exemple, entre une tâche du modèle des tâches et un *widget* du *CUI* qui permet l'exécution de cette tâche).

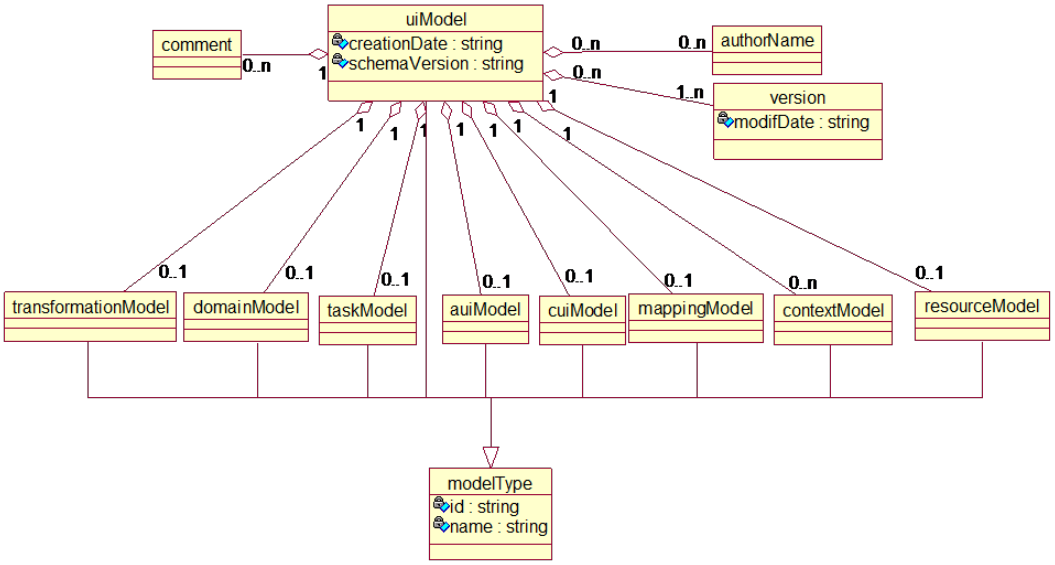

Figure 1.1 *Les différents types de modèles en UsiXML*

Le modèle contextuel est composé de trois sous-modèles ; le modèle utilisateur, le modèle de l'environnement et le modèle de plate-forme :

- Le modèle utilisateur décompose la population des utilisateurs en stéréotypes, décrits par des attributs tels que l'expérience avec le système ou la tâche, la motivation, etc.

- Le modèle de l'environnement décrit toutes les propriétés intéressantes de l'environnement où a lieu l'interaction avec le système ou la tâche. Les propriétés peuvent être physiques (par exemple, un environnement bruillant) ou psychologique (par exemple le niveau de stress).

- le modèle de plate-forme capture les attributs en relation avec la combinaison entre la machine et le logiciel où l'interface utilisateur est sensée être déployée.

Le modèle des ressources contient les éléments (titre, astuce, etc.) spécifiques à un contexte donné (par exemple la langue de l'utilisateur). Les ressources sont liées aux objets du *CUI* ou du *AUI*.

Pour terminer, le modèle des transformations permet la spécification de règles de transformation sous la forme de graphes.

Chapitre 2

Conception d'un outil d'animation

Ce chapitre présente les caractéristiques attendues de l'outil d'animation. La section 2.1 explique la caractéristique principale du langage utilisé par le prototype à savoir la dégradation d'une interface. La section 2.2 présente toutes les opérations d'adaptation que le langage doit implémenter. La section 2.3 propose des exemples de paires opérations d'adaptation/transitions animées. Et enfin la section 2.4 présente toutes les actions de contrôle de l'animation utilisées par le prototype.

2.1 Adaptation guidée par un langage de transitions animées

Le développement d'un langage permettant de spécifier des transitions animées pour montrer le processus d'adaptation d'une interface découle du concept de *Graceful Degradation*. Ce concept, dans le cadre des interfaces utilisateur, est une méthode pour adapter facilement une interface à une plate-forme selon une approche semi-automatique, de transformation et basée sur des modèles. Chaque adaptation se fait à l'aide de règles de transformation, produisant alors une ou plusieurs interfaces adaptées à des plates-formes plus contraignantes. Le but de ce mémoire est de proposer une animation de toutes les transitions effectuées lors d'une adaptation pour réduire la déstabilisation cognitive.

Les règles de transformation prennent comme entrée une interface destinée à une plate-forme donnée et produisent de plus petites interfaces en sortie, d'où l'utilisation du terme *dégradation* pour définir le processus d'adaptation. L'appellation *graceful degradation* est apparue pour la première fois dans le milieu des systèmes de sécurité critique tel que l'aérospatial. Il dénote la possibilité d'un système à continuer de fournir un service en fonction du degré de disponibilité du matériel informatique. L'idée est que chaque système est susceptible de mal fonctionner, soit à cause d'un composant défectueux ou absent, ou à cause d'un environnement trop rigoureux pour le système. Sur base de ces mauvaises conditions, le système est supposé ne pas crasher complètement, mais de continuer à proposer ses fonctions de base avec une performance moindre qu'à l'habitude. Dans le cadre de ce mémoire, le terme est légèrement

différent puisque ce ne sont pas des adaptations de système dont il est question mais de l'adaptation d'interfaces. L'adaptation n'est pas la conséquence d'une défaillance technique mais des possibilités réduites de la plate-forme cible [3].

Cette méthode est basée sur un ensemble de règles de transformation appelée *graceful degradation rules*. Cette méthode respecte un compromis entre, d'un coté, la facilité d'utilisation et, d'autre part, la consistance entre les différentes versions de l'interface adaptée. La Figure 2.1 présente un schéma du concept de *graceful degradation*.

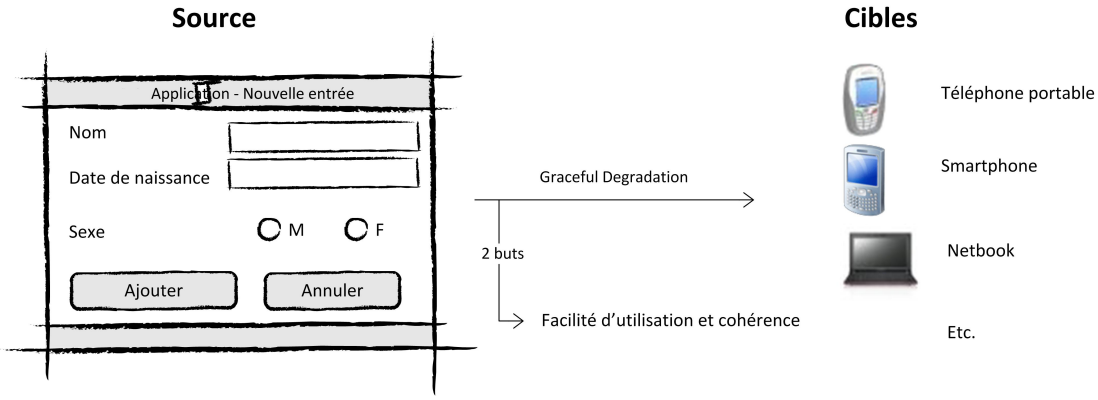

Figure 2.1 *Graceful Degradation*

2.2 Opérations d'adaptation

Une opération d'adaptation est définie comme étant n'importe quelle transformation exécutée sur un élément d'une interface. Une adaptation est une suite d'opérations d'adaptation qui permettent à l'utilisateur de comprendre tout le processus d'adaptation en partant de l'interface originelle jusqu'à l'interface finale. Chaque opération d'adaptation produit une interface intermédiaire qui consiste en une étape dans le processus d'adaptation. La Figure 2.2 synthétise le processus d'adaptation. En général, l'utilisateur ne voit pas ces interfaces intermédiaires, il ne voit que l'interface originelle et l'interface finale. C'est à cause de ce fait que l'utilisateur souffre de perturbations cognitives. La séquence entière des opérations d'adaptation exécutées pour adapter une interface est appelée un *scénario d'adaptation*. Les opérations d'adaptation sont réparties en cinq catégories [6] :

- *Opérations de redimension* : permettent de modifier la taille d'un composant dans le but d'optimiser l'esthétique de l'interface. Par exemple, il peut être utile de réduire la longueur et la hauteur de plusieurs champs pour permettre un réalignement de ceux-ci.

- *Opérations de relocation* : permettent de changer la place d'un composant sur l'interface. Par exemple, il peut être utile de rassembler plusieurs boutons à un

même endroit pour gagner de la place.

– *Opérations de transformation de composant(s)* : permettent de replacer un ou plusieurs composants par un ou plusieurs composants graphiques. Les composants substituant doivent assurer la même tâche que les composants d'origine. Il est cependant envisageable de dégrader les fonctionnalités des nouveaux composants. Par exemple, un ensemble de liste déroulantes pourrait être remplacé par une seule liste déroulante multi-sélections.

– *Opérations de transformation d'image* : permettent de modifier la taille, la surface et la qualité d'une image de manière à satisfaire les contraintes imposées par la nouvelle plate-forme.

– *Opérations de découpage* : permettent de diviser un ou plusieurs groupes de composants en plusieurs groupes de composants qui seraient affichés séparément. Par exemple, il peut être utile de séparer en plusieurs onglets un gros conteneur de composants pour économiser de la place à l'écran.

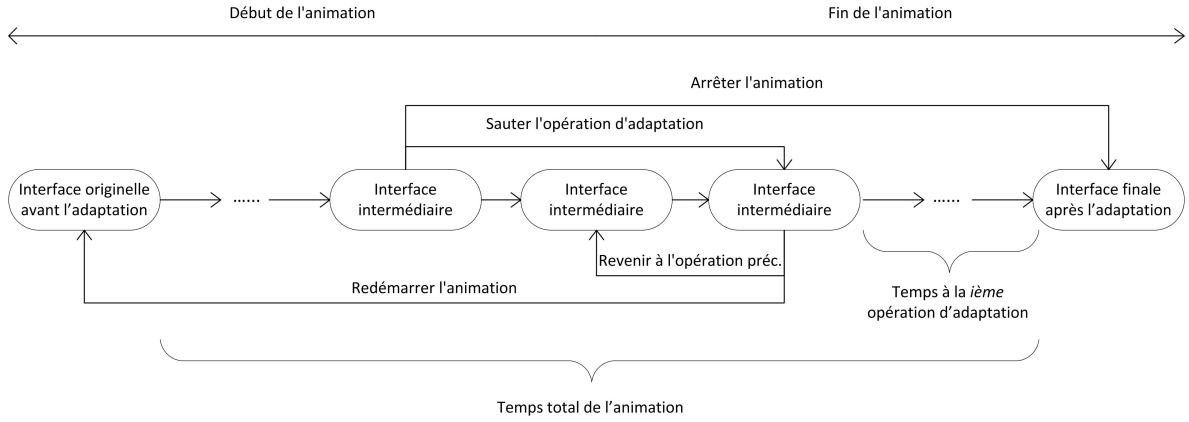

Figure 2.2 *Ligne du temps du processus d'animation*

Une opération d'adaptation doit pouvoir être exécutée sur un ou plusieurs composants graphiques. Pour cela, il est nécessaire de réfléchir à un mécanisme de sélection. Un *Selector* consiste en une définition d'un ensemble de composants graphiques qui seront affectés par une opération d'adaptation. Quatre types de *selector* sont considérés [6] :

– *universalSelector* : applique l'opération d'adaptation à tous les composants graphiques de l'interface.

– *elementTypeSelector* : applique l'opération d'adaptation à tous les composants graphiques d'un certain type (par exemple : tous les champs de texte).

- *classSelector* : applique l'opération d'adaptation à tous les éléments sélectionnés par un algorithme spécifique (par exemple : toutes les listes déroulantes de moins de 10 éléments).

- *idSelector* : applique l'opération d'adaptation au composant graphique correspondant à l'identifiant spécifié.

2.3 Types d'animation

Ce chapitre présente les différentes catégories d'animation existantes ainsi que quelques exemples d'animation pour tel ou tel type d'opération d'adaptation. En l'état actuel des choses, il y a, d'un coté, de nombreux manuels expliquant les différents types d'animation pour un usage particulier et d'un autre coté, des principes abstraits proposés par la psychologie cognitive pour réduire les perturbations cognitives. Cette situation ne permet pas de proposer un modèle expliquant objectivement quel type d'animation utiliser pour telle opération d'adaptation.

Les différentes catégories d'animation sont classées en fonction de leurs propriétés visuelles [6] :

- La famille *F1* rassemble les animations qui recouvrent simplement l'ancien composant par un nouveau composant graphique.

- La famille *F2* rassemble les animations qui divisent l'ancien composant en plusieurs parties. Ces parties bougent pour laisser place au nouveau composant. Par exemple, une porte automatique à deux battants est une *animation* de type *F2* car la porte est divisée en deux parties qui, en se déplanant chacune vers la gauche et vers la droite, font apparaître ce qu'il y a derrière la porte.

- La famille *F3* rassemble les animations qui déplacent l'ancien composant pour laisser la place au nouveau composant. Par exemple, le visionnage de dias est une *animation* de type *F3* car chaque nouvelle dia apparaît de la droite et déplace la dia précédente vers la gauche.

- La famille *F4* rassemble les animations qui divisent l'ancien composant en plusieurs parties. Ces parties disparaissent pour laisser entrevoir le nouveau composant. Par exemple, l'ouverture d'un volet vénitien est une animation de type *F4*. Les lattes du volet représentent toutes les parties de l'ancien composant et lorsque le volet est ouvert, la vision des lattes disparaît laissant apparaître le paysage dehors.

- La famille *F5* rassemble les animations qui n'induisent pas de mouvement de l'ancien ou du nouveau composant. L'ancien composant devient invisible et laisse place au nouveau composant qui apparaît.

Icône	Nom	Fam.	Contexte général d'utilisation
	*Horizontal scroll from right*	*F1*	Afficher le prochain élément d'une série d'éléments graphiques.
	*Horizontal scroll from left*	*F1*	Afficher l'élément précédent d'une série d'éléments graphiques.
	*Vertical scroll from bottom*	*F1*	Présenter un raisonnement itératif, un long texte ou un mouvement.
	*Vertical scroll from top*	*F1*	Revenir en arrière sur un raisonnement itératif, un long texte ou un mouvement.
	*Diagonal replacement from top/bottom left corner*	*F1*	Retourner à une page, un écran ou un élément graphique précédent.

Figure 2.3 *Définitions d'animations majeures - Partie 1 [6]*

Les Figures 2.3 et 2.4 présentent quelques animations qui sont considérées comme les plus connues. Aucune étude n'ayant été menée pour déterminer quelle animation utiliser pour telle opération d'adaptation (et vice-versa), il est nécessaire de faire preuve de bon sens et de se baser sur l'expérience des développeurs d'interfaces graphiques pour utiliser la meilleure animation possible pour une opération d'adaptation. La Figure 2.5 motive ainsi la sélection de quelques paires d'opération d'adaptation/transition animée.

2.4 Contrôle de l'animation

Un facteur clé dans l'utilisation d'une animation pour diminuer la déstabilisation cognitive réside dans la possibilité qu'a l'utilisateur à contrôler le déroulement de l'animation. L'utilisateur doit pouvoir contrôler le scénario (voir Figure 2.2) complètement soit en accélérant l'animation soit en revenant en arrière. Ces actions doivent être proposées à l'utilisateur sous forme de raccourcis [6].

Icône	Nom	Fam.	Contexte général d'utilisation
	*Diagonal replacement from top/bottom right corner*	*F1*	Aller à une page, un écran ou un élément graphique suivant.
	*Bam door close*	*F2*	Fermer une fenêtre intermédiaire (par exemple un splash screen ou une fenêtre *About*), terminer une animation, signifier la fin d'un jeu.
	*Bam door open*	*F2*	Ouvrir une fenêtre intermédiaire (par exemple un splash screen ou une fenêtre *About*), démarrer une animation, signifier le début d'un jeu.
	*Venetian blinds*	*F4*	Présenter un sujet complètement différent, donner un sentiment de coordination, assurer une transition importante.
	*Iris open*	*F4*	Montrer des détails plus spécifiques d'un sujet.
	*Iris close*	*F4*	Montrer des informations plus générales d'un sujet.

Figure 2.4 *Définitions d'animations majeures - Partie 2 [6]*

Opération d'adaptation	**Transition animée : Justification**
Augmenter la largeur d'un élément graphique	*Horizontal scroll from left* : Cette opération minimise le changement visuel puisque seule la partie droite de l'élément graphique va changer. Pour un champs de texte par exemple, cette animation est particulièrement appropriée car elle donne l'impression que le champs de texte s'étend réellement.
Augmenter la hauteur d'un élément graphique	*Vertical scroll from bottom* : Cette opération minimise le changement visuel puisque seul la partie du haut de l'élément graphique va changer.
Afficher un nouvel élément graphique	*Iris open* : Cette opération induit un affichage progressif du nouvel élément graphique. Cela donne l'illusion que l'élément graphique vient du vide.
Cacher un élément graphique	*Iris close* : Cette opération induit une disparition progressive de l'élément graphique. Cela donne l'illusion que l'élément graphique disparaît dans le vide.
Substituer un élément graphique par un autre	*Bam door open* : Cette opération affecte l'aspect visuel de l'ancien élément graphique et du nouveau dans leur entièreté.
Distribuer une série d'éléments graphiques dans une série de conteneurs	*Bam door open ou Iris open* : Ces opérations peuvent s'appliquer à des régions entières d'une interface pour éviter d'exécuter une multitude de petites animations pour chaque élément graphique.
Déplacer un élément graphique	N'importe quelle animation de *F5* : Intuitivement, faire bouger l'élément graphique jusqu'à sa nouvelle position semble être la meilleure solution. En pratique, cela est moins intéressant car un long déplacement impliquerait une trop longue transition. C'est pourquoi il est suggéré d'utiliser une animation de la famille *F5* qui fait disparaître l'élément graphique de sa position d'origine et le fait réapparaître à sa nouvelle position.

Figure 2.5 *Table de mapping entre une opération d'adaptation et une transition animée [6]*

Skip termine la transition courante et saute à la prochaine transition animée prévue par le scénario. Cette action est motivée par le fait que l'utilisateur doit pouvoir stopper une transition lorsque celle-ci est comprise par l'utilisateur.

Break termine la transition courante. L'utilisateur doit pouvoir stopper l'animation à tout moment, cette action est une des plus importantes.

Return termine la transition courante et relance l'animation à la transition précédente. Cette action est motivée par le fait que l'utilisateur doit pouvoir revenir à une transition antérieure quand il n'a pas compris celle-ci.

Restart relance l'animation depuis le début. Cette action est motivée par le fait que l'utilisateur doit pouvoir rejouer entièrement le scénario de transitions dans le cas où il n'aurait pas compris en quoi l'interface a changé depuis le début de l'animation.

User-Break stoppe momentanément la transition courante tant que l'utilisateur presse la barre d'espace. Cette action est motivée par le fait que l'utilisateur doit pouvoir faire une pause pour comprendre l'opération d'adaptation en cours.

Acceleration augmente la vitesse de l'animation en cours. Cette action est motivée par le fait que l'utilisateur doit pouvoir accélérer l'animation quand il n'a pas de soucis de compréhension dans le cas où les opérations d'adaptation lui paraissent logiques. Il existe aussi le principe inverse pour l'action *Deceleration*.

Chapitre 3

Description du prototype développé

Dans ce chapitre, plusieurs aspects du prototype développé sont décrits. Dans la section 3.1, les fonctionnalités du prototype sont présentées. En 3.2, la syntaxe du langage de transition est spécifiée. En 3.3, les architectures logiques et physiques du prototype sont présentées. La section 3.4 discute des choix d'implémentation du prototype. En 3.5, les limitations du prototype sont présentées. Et enfin, le point 3.6 décrit la création d'une animation de transition depuis la création de l'interface originelle.

3.1 Fonctionnalités

Dans le tableau suivant sont listés les différentes fonctionnalités prévues lors de la spécification du prototype. Toutes les fonctionnalités n'ont pas été implémentées dans le prototype mais les principales pour la création d'une animation le sont.

Fonctionnalités implémentées	**Fonctionnalités à implémenter**
Langage *Simple Interface Transition* (SIT) : - Grammaire hors-contexte gérant les *integers*, *boolean* et *string*. - Parseur et interpréteur SIT. - Système de *plugins* pour ajouter de nouveaux *classSelector* ou règles de substitution.	**Langage *Simple Interface Transition* (SIT) :** - Gestion de variables, création de structures d'itérations et autres concepts pour spécifier le comportement d'un *classSelector* directement dans le code SIT.

Opérations d'adaptation : - *set* : Modifie la propriété d'un ou plusieurs composants. - *substitute* : Substitue un ou plusieurs composants selon une règle de substitution. - *contact* : Diminue la largeur et/ou la hauteur d'un ou plusieurs composants. - *extend* : Augmente la largeur et/ou la hauteur d'un ou plusieurs composants. - *remove* : Supprime un ou plusieurs composants de l'interface. - *changebox* : Déplace un ou plusieurs composants dans un autre conteneur. - *changerow* : Modifie la hauteur de chaque ligne d'un conteneur. - *changecolumn* : Modifie la largeur de chaque colonne d'un conteneur.	**Opérations d'adaptation :** - *distribute* : Déplace plusieurs composants dans plusieurs conteneurs. - *play* : Affiche un message pour l'utilisateur.
Animations : - *wipe* : fait disparaître un ancien composant et apparaître un nouveau composant. - *horizontal scroll left/right*	**Animations** - Proposer plus d'animations notamment des familles *F1* à *F4*. - Choix de l'animation pour chaque opération d'adaptation.
Actions de contrôle : - *skip*, *break*, *return*, *restart* Voir section 2.4 pour plus d'informations.	**Actions de contrôle :** - *user-break*, *acceleration*, *deceleraction* Voir section 2.4 pour plus d'informations.
Méthodes de sélection des composants : - *universalSelector*, *elementTypeSelector*, *classSelector* et *idSelector*. Voir section 2.2 pour plus d'informations.	
En général : - Utiliser *UsiXML* comme langage de spécification d'interface. - Utiliser *C#* pour développer le prototype. - Gestion des erreurs de *parsing* et d'interprétation.	**En général :** - Gestion des événements dans *UsiXML*. - Adaptation de tous les éléments graphiques d'*UsiXML*.

3.2 Création de la grammaire du langage

Aux vues des fonctionnalités implémentées et futures citées au point 3.1, le langage *SIT* est spécifié par une grammaire hors-contexte. Les grammaires non contextuelles sont suffisamment puissantes pour décrire la syntaxe de la plupart des langages de programmation, avec au besoin quelques extensions ; la syntaxe de la majorité des langages de programmation est en fait spécifiée à l'aide de grammaires non contextuelles. La Figure 3.1 présente la grammaire du langage *SIT* sous forme *BNF*. La syntaxe a été simplifiée pour des raisons de lisibilité (pas de guillemets ni de majuscules). Certains terminaux représentent un ensemble de valeurs :

- *@control* : contient tous les types de contrôle de l'interface.
- *#identifier* : contient tous les identifiants des éléments de l'interface.
- *%selector* : contient tous les noms de plugin disponibles pour sélectionner un ou plusieurs éléments.
- *property* : contient toutes les propriétés des contrôles de l'interface.
- *constant* : représente un nombre entier.
- *string* : représente une chaîne de caractères.
- *lambda* : représente un symbole vide.

Le parseur *SIT* utilise la méthode dite *LL(1)* pour vérifier qu'un code source SIT est cohérent par rapport à la grammaire. L'analyse LL est une analyse descendante pour un sous-ensemble de grammaire non contextuelle. Les parseurs *LL(1)* sont populaires car ils valident l'utilisation d'un symbole en fonction du symbole suivant ce qui les rend très efficaces.

Dans la syntaxe de la Figure 3.1, on retrouve bien toutes les opérations d'adaptation, différents *selector* et types de valeur cités au point 3.1 :

- L'opération d'adaptation *substitute* est représentée par le symbole non terminal ⟨*stmtsubstitute*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à substituer. L'élément substituant est sélectionné par un *elementTypeSelector*. L'opération d'adaptation requiert aussi de définir l'identifiant du conteneur où sera placé l'élément substitutant. Ensuite, entre parenthèse, il est nécessaire de spécifier une chaine de caractères qui sera utilisée comme identifiant pour le nouvel élément substituant. Pour terminer, il faut définir la place du nouveau composant dans le conteneur. Chaque conteneur est quadrillé en lignes et en colonnes, il possible d'ajouter une ligne et/ou une colonne grâce aux terminaux *rowinsert* et *colinsert*, et ce, pour éviter que l'élément substituant se superpose à un autre élément graphique.

- L'opération d'adaptation *contract* est représentée par le symbole non terminal ⟨*stmtcontract*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à contracter. Il est aussi nécessaire de fournir la valeur de rétrécissement en pixels de la hauteur et de la largeur du ou des éléments à contracter.

⟨*program*⟩ : := ⟨*statement*⟩ ⟨*program*⟩
| eof

⟨*statement*⟩ : := ⟨*stmtsubstitute*⟩
| ⟨*stmtcontract*⟩
| ⟨*stmtextend*⟩
| ⟨*stmtremove*⟩
| ⟨*stmtchangebox*⟩
| ⟨*stmtchangecolumns*⟩
| ⟨*stmtchangerows*⟩
| ⟨*stmtset*⟩

⟨*stmtsubstitute*⟩ : := substitute ⟨*selector*⟩ by @control in #identifier (string) where ⟨*rowtype*⟩ constant , ⟨*coltype*⟩ constant ;

⟨*stmtcontract*⟩ : := contract ⟨*selector*⟩ of constant constant ;

⟨*stmtextend*⟩ : := extend ⟨*selector*⟩ of constant constant ;

⟨*stmtremove*⟩ : := remove ⟨*selector*⟩ ;

⟨*stmtchangebox*⟩ : := changebox ⟨*selector*⟩ to #identifier where ⟨*rowtype*⟩ constant , ⟨*coltype*⟩ constant ;

⟨*stmtchangecolumns*⟩ : := changecolumns #identifier to ⟨*defval*⟩ ;

⟨*stmtchangerows*⟩ : := changerows #identifier to ⟨*defval*⟩ ;

⟨*stmtset*⟩ : := set ⟨*selector*⟩-¿property to ⟨*value*⟩ ;

⟨*selector*⟩ : := #identifier
| all
| @control
| %selector ⟨*selector_param*⟩

⟨*selector_param*⟩ : := string ⟨*selector_param_ll1*⟩
| constant ⟨*selector_param_ll1*⟩
| ⟨*boolean*⟩ ⟨*selector_param_ll1*⟩

⟨*selector_param_ll1*⟩ : := lambda
| ⟨*selector_param*⟩

⟨*defval*⟩ : := constant ⟨*defval_ll1*⟩

⟨*defval_ll1*⟩ : := lambda | ⟨*defval*⟩

⟨*rowtype*⟩ : := rowinsert | row

⟨*coltype*⟩ : := colinsert | col

⟨*value*⟩ : := ⟨*boolean*⟩| constant | string

⟨*boolean*⟩ : := true | false

Figure 3.1 *Syntaxe du langage SIT sous forme BNF*

- L'opération d'adaptation *extend* est représentée par le symbole non terminal ⟨*stmtextend*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à étendre. Il est aussi nécessaire de fournir la valeur d'agrandissement en pixels de la hauteur et de la largeur du ou des éléments à étendre.

- L'opération d'adaptation *remove* est représentée par le symbole non terminal ⟨*stmtremove*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à supprimer.

- L'opération d'adaptation *changebox* est représentée par le symbole non terminal ⟨*stmtchangebox*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à déplacer. L'opération d'adaptation requiert de définir l'identifiant du conteneur où sera placé l'élément substitutant ainsi que sa place au sein du conteneur. Comme pour ⟨*stmtsubstitute*⟩, il est possible d'insérer une ligne ou une colonne avec les terminaux ⟨*rowinsert*⟩ et ⟨*colinsert*⟩.

- L'opération d'adaptation *changerows* est représentée par le symbole non terminal ⟨*stmtchangerows*⟩. Le conteneur à modifier est sélectionné par un *idSelector*. Les différentes valeurs de hauteur des lignes du conteneur sont définies par une suite de valeurs dont la somme vaut 100 et dont le nombre est égale au nombre de lignes du conteneur.

- L'opération d'adaptation *changecolumns* est représentée par le symbole non terminal ⟨*stmtchangecolumns*⟩. Le conteneur à modifier est sélectionné par un *idSelector*. Les différentes valeurs de largeur des colonnes du conteneur sont définies par une suite de valeurs dont la somme vaut 100 et dont le nombre est égale au nombre de colonnes du conteneur.

- L'opération d'adaptation *set* est représentée par le symbole non terminal ⟨*stmtset*⟩. Tous les types de *selector* peuvent être utilisés pour définir le ou les éléments à modifier. Si plusieurs éléments sont sélectionnés, ils doivent être de même type. Concernant le nom de la propriété à utiliser, il correspond à un nom de propriété spécifié dans le langage *XAML*, la section 3.4 explique en détail ce choix.

3.3 Architectures

Dans cette section, la structure du prototype est décrite en termes de composants logiciels et de relations entre ces composants. Dans le point 3.3.1, l'architecture logique de l'application et, en 3.3.2, l'architecture physique sont présentées.

3.3.1 Architecture logique

Le prototype utilise une architecture *Interpreter* [2], son architecture logique est donc composée de terminaux, non-terminaux, arbre syntaxique, etc. Cette architecture est illustrée par la Figure 3.2.

Dans les points suivants, les différents composants logiciels de l'architecture logique sont listés et détaillés.

Les symboles

Le composant *symbole* est un élément lexical utilisé pour spécifier les règles de production qui constituent une grammaire formelle. Il existe deux types de symbole : les terminaux et les non-terminaux représentés respectivement par les composants *Terminal* et *NonTerminal*. Un terminal est une chaîne de caractères qui peut être utilisée comme entrée ou résultat d'une règle de production et ne peut pas être divisé en unités plus petites. La Figure 3.2 présente une partie des terminaux utilisés par l'intépreteur tels que les différents *selectors*, les caractères spéciaux, etc. Un non-terminal est une combinaison de plusieurs terminaux.

L'analyseur lexical

L'analyse lexicale se trouve tout au début de la chaîne d'interprétation et est représentée par le composant *AnalyseurLexical*. L'analyseur lexical est défini par un ensemble de règles, généralement appelées expressions régulières. Celles-ci définissent les séquences de caractères possibles qui sont utilisées pour former des symboles. L'analyseur lexical échange des données avec le composant *LecteurSIT* qui contient le code source SIT à interpréter.

Le parseur LL1

Le parseur LL1 est représenté par le composant *ParseurLL1*. Le parseur charge la grammaire représentée par le composant *Grammaire* transmise par le composant *LecteurBNF*. Le parseur vérifie que la suite de symboles envoyée par l'analyseur lexical est cohérente avec la grammaire. Le résultat du parseur est l'arbre syntaxique représenté par le composant *ArbreSyntaxique* qui est utilisé ensuite par l'interpréteur pour exécuter le code SIT.

L'interpréteur

L'interpréteur est représenté par le composant *Interpréteur* et est composé de l'analyseur lexical et du parseur LL1. Il utilise notamment le créateur d'animation représenté par le composant *CréateurAnimation* ainsi que les instances des composants *Selector[Plugin]* et *Substitute[Plugin]*. Ces deux composants permettent de charger à la volée des plugins pour définir des *classSelector* et des règles de substitution. L'interpréteur utilise le composant *LecteurUsiXML* pour charger l'interface graphique à adapter.

3.3.2 Architecture physique

La transition de l'architecture logique vers l'architecture physique n'implique aucun changement au niveau des composants, seules les relations inter-composants se voient modifiées. Entres autres, les liens USE sont transformés en liens CALL. Cette architecture est illustrée par la Figure 3.3.

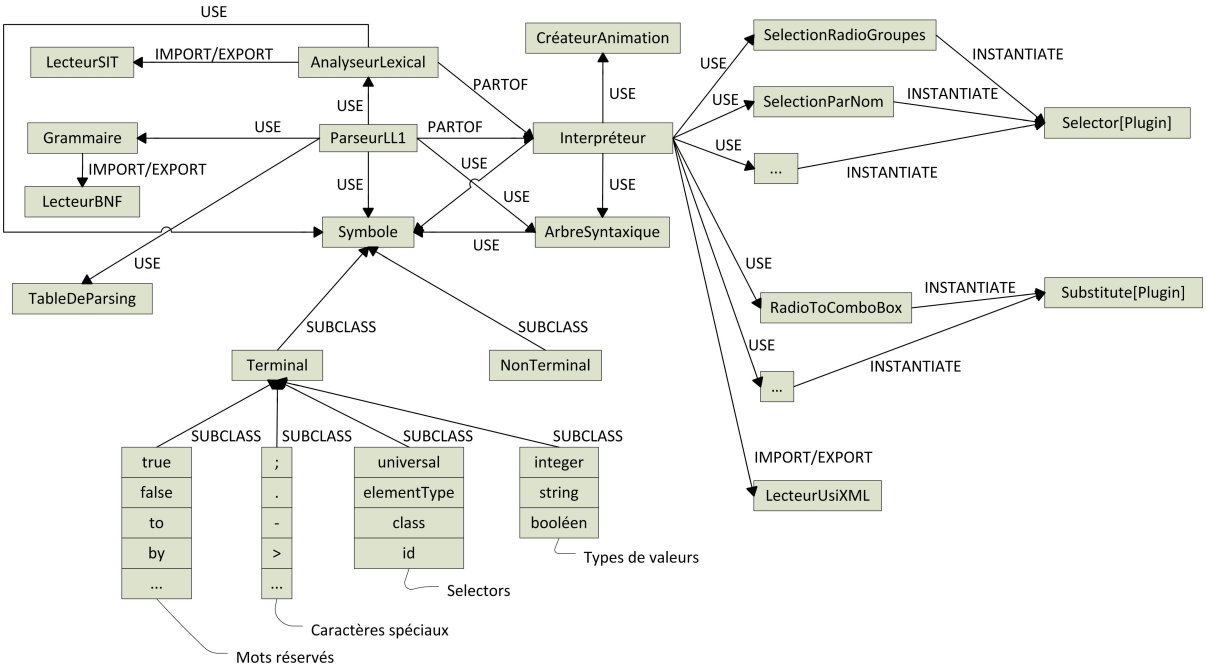

Figure 3.2 *Architecture logique*

3.4 Approbation des choix d'implémentation

Dans le cadre de ce mémoire, l'outil d'animation à développer doit utiliser deux technologies à savoir le *C#* pour le développement même du prototype et l'*UsiXML* comme langage de spécification d'interface. Bien que ces deux technologies aient été suggérées, ce chapitre revient sur les raisons qui ont poussé à ces choix.

La technologie *C#* a été choisie pour deux raisons. Tout d'abord, pour développer un interpréteur avec une grammaire hors-contexte il faut que la grammaire du langage utilisé pour développer l'interpréteur soit elle aussi hors-contexte. Le *C#* répond à ce premier critère. Deuxièmement, le langage *C#* permet d'utiliser la librairie *.NET*. Cette librairie propose un concept de *Storyboard* très intéressant pour le prototype. Une *Storyboard* est une ligne du temps contenant des informations pour animer des éléments d'une interface. Les fonctionnalités d'une *Storyboard* permettent entre autre de combiner des effets sur des éléments graphiques et de contrôler une animation.

La technologie *UsiXML* a été choisie pour son indépendance vis à vis des langages de programmation qui l'interprètent et son rôle important au sein de la recherche du département *ISYS*. *UsiXML* est conforme aux normes *XML* et capture l'essence même de ce qu'une interface est ou devrait être indépendamment des caractéristiques physiques de la plate-forme.

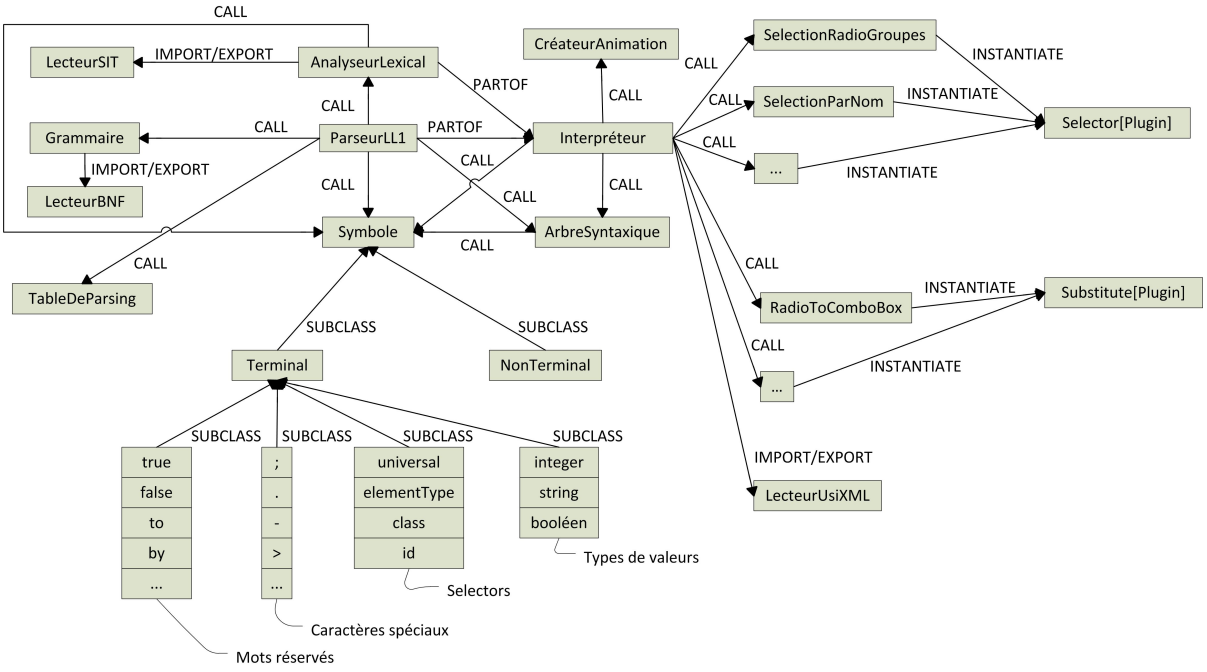

Figure 3.3 *Architecture physique*

Cependant, il n'existe pas d'interpréteur *UsiXML* en *C#*. Le langage de spécification d'interface utilisé en *C#* est le *XAML*. Tout comme l'*UsiXML*, il est aussi conforme *XML*. De ce fait, pour le prototype, tous les fichiers *UsiXML* sont transformés en fichier *XAML* grâce à une feuille de transformation *XSLT* [4]. C'est pour cette raison que tous les éléments graphiques implémentés en *UsiXML* ne sont pas tous utilisables car la feuille de transformation ne transforme qu'un ensemble d'éléments graphiques. Seuls les éléments *box*, *inputText*, *outputText*, *comboBox*, *listBox*, *button* et *radioButton* sont utilisables. L'utilisation du *XAML* pose aussi une deuxième contrainte. Les *classSelectors* et les propriétés de l'opération d'adaptation *SET* se rapportent à la sémantique de *XAML*. Par exemple, la propriété *text* d'un champs de texte en *UsiXML* porte le nom *Content* en *XAML*. C'est ce dernier nom de propriété qui est utilisé pour définir la propriété à changer pour l'opération *SET*. La Figure 3.4 montre un schéma de l'outil d'animation en accord avec les technologies choisies.

3.5 Limitations du prototype

Le prototype développé possède certaines limitations concernant l'utilisation des opérations d'adaptation. Ces limitations se justifient dans le sens où leur résolution n'a pas été jugée prioritaire. Elles pourraient faire l'objet d'une future amélioration du prototype ou prise en compte si le prototype venait à être réimplémenté.

Les commandes *contract*, *extend*, *changerows* et *changecolumns* modifient la taille d'un ou plusieurs composants graphiques. Tout composant modifié par ces opérations

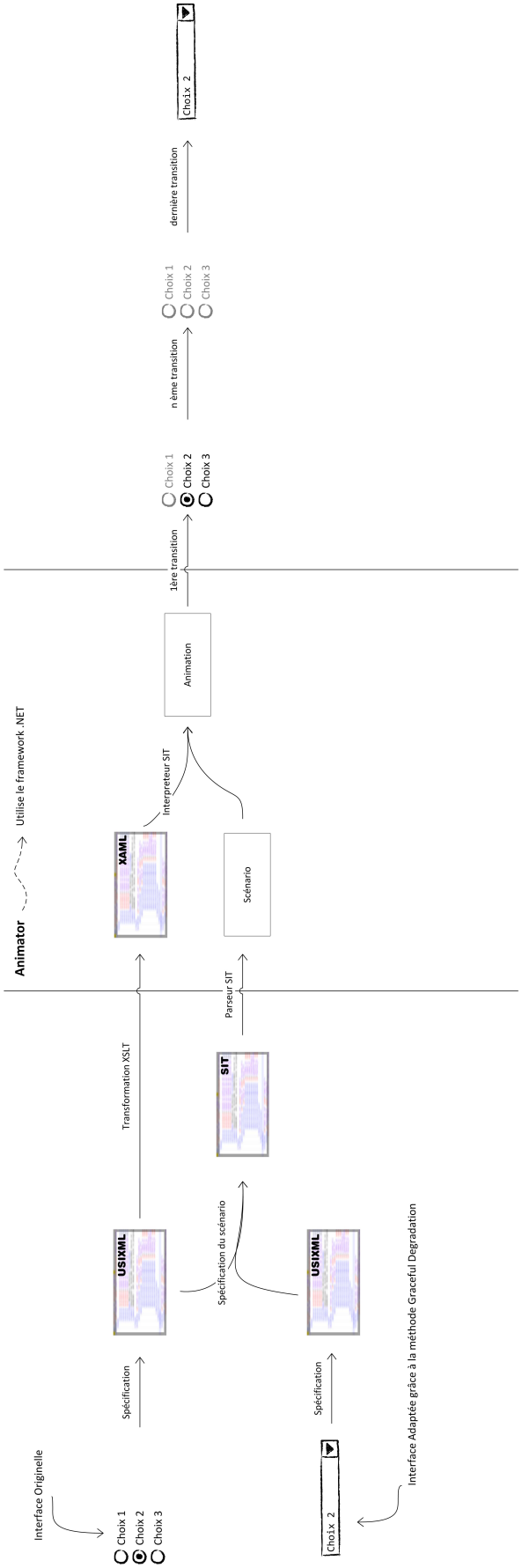

Figure 3.4 *Processus d'adaptation rendu par un scénario de transitions*

perd sa propriété de redimensionnement en fonction de la fenêtre. En d'autres mots, là où un composant était redimensionné en fonction de la taille de la fenêtre, après l'opération, sa taille ne change plus.

La commande *set* ne peut modifier que des propriétés dont le type de valeur est une chaîne de caractères, un nombre entier ou un booléen.

Les *selectors* ne sélectionnent que des composants graphiques autres que *Grid* et dont le parent et grand-parent sont des *Grid*. Exception faite pour la fenêtre en elle-même qui peut être sélectionnée mais seulement pour modifier ses propriétés de largeur et de longueur.

3.6 Scénario

Le scénario en Figure 3.6 montre la dynamique de création d'une animation. La spécification de l'interface et du scénario de transition se font à l'aide d'outils extérieurs notamment *GracefulDegradation* pour adapter l'interface originelle. Ensuite la spécification du scénario est envoyée au parseur SIT et la spécification de l'interface est envoyée à l'interpréteur qui va pouvoir créer l'animation grâce à l'arbre syntaxique fourni par le parseur SIT. Pour terminer, l'utilisateur lance l'animation.

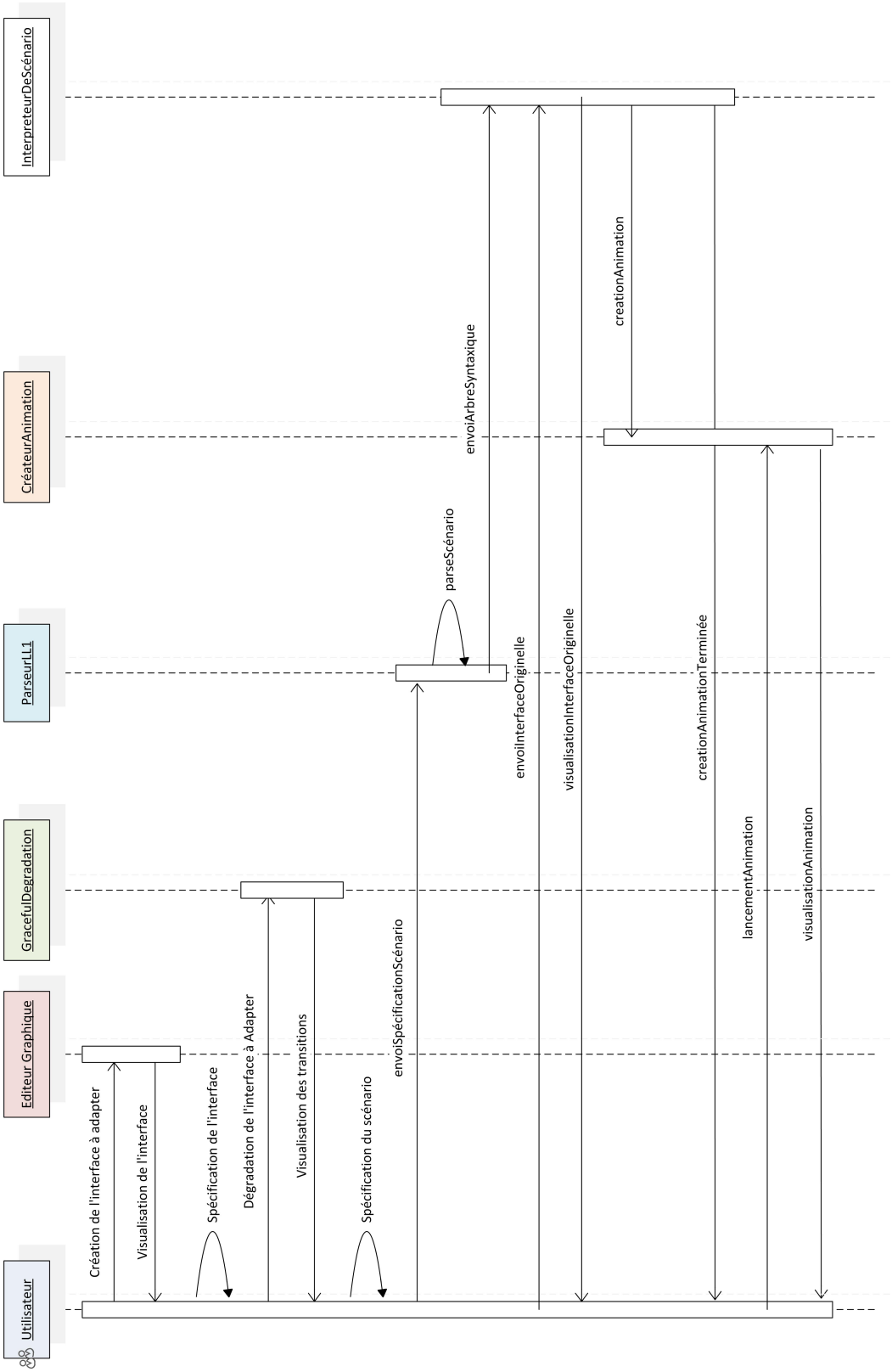

Figure 3.5 *Scénario décrivant la création d'une animation*

Chapitre 4

Application sur deux cas d'étude

Dans ce chapitre, la procédure de chargement et de lancement d'un animation est expliquée à la section 4.1. En 4.2, plusieurs exemples utilisant des opérations d'adaptation élémentaires sont présentées. En 4.3, un premier cas d'étude est présenté, il consiste en la dégradation maximale de la taille d'une interface. En 4.4, un deuxième cas d'étude est présenté, il consiste en la substitution de boutons *radios* de plusieurs manières.

4.1 Chargement et lancement d'une animation

Le chargement d'une animation requiert de spécifier un fichier contenant le code *UsiXML* et d'un fichier contenant le code SIT. Pour ce faire, il suffit de cliquer sur les bouton *Fichier UsiXML* et *Fichier Sit*, et de sélectionner les fichiers à charger grâce à une boite de dialogue. La Figure 4.2 montre que ces deux fichiers ont bien été chargés grâce au label *OK* apparaissant en dessous des boutons. Lorsque les deux fichiers sont chargés, l'animation peut être lancée en cliquant sur le bouton *Lancer l'animation* qui s'est dégrisé. Une fois l'animation lancée, les boutons en dessous de l'interface *Précédent*, *Pause* et *Suivant* permettent respectivement de reculer, de mettre en pause et d'avancer dans l'animation.

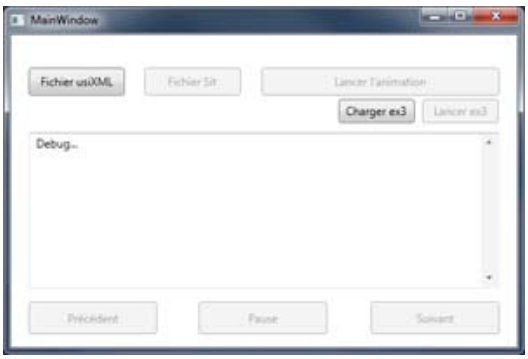

Figure 4.1 *Interface au démarrage*

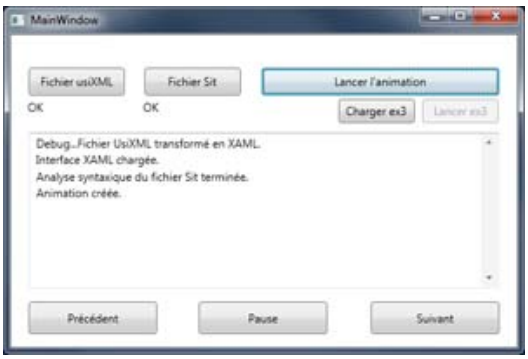

Figure 4.2 *Animation lancée*

4.2 Opérations d'adaptation élémentaires

Cette section présente des exemples élémentaires d'opération d'adaptation. La sous-section 4.2.1 explique comment contracter et étendre un bouton. En 4.2.2, l'exemple montre comment supprimer un bouton. En 4.2.3, il est expliqué comment déplacer un bouton. La sous-section 4.2.4 explique comment modifier la propriété d'un bouton. En 4.2.5, l'exemple propose de modifier la hauteur et la largeur du quadrillage d'un conteneur. Et enfin en 4.2.6, deux exemples présentent les *classSelectors* et les *elementTypeSelectors*. Les exemples sur la substitution sont présentés aux sections 4.3 et 4.4.

4.2.1 Contracter et étendre un bouton

L'interface représentée par la Figure 4.4 a été modifiée par l'opération d'adaptation définie à la ligne 1. Le bouton *Cancel* qui possède l'identifiant *button_0* a été contracté de 20 pixels de chaque coté du bouton et n'a pas été contracté en hauteur.

```
1. CONTRACT #button_0 OF 20 0;
2. EXTEND #button_0 OF 130 0;
```

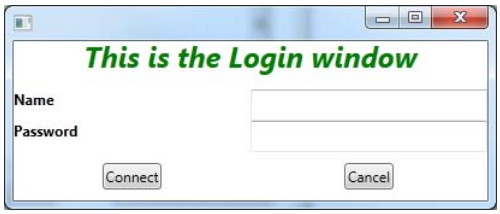

Figure 4.3 *Interface sans contraction*

Figure 4.4 *Contraction d'un bouton*

L'interface représentée par la Figure 4.6 a été modifiée par l'opération d'adaptation définie à la ligne 2. Le bouton *Cancel* qui possède l'identifiant *button_0* a été étendu de 130 pixels de chaque coté du bouton et n'a pas été étendu en hauteur.

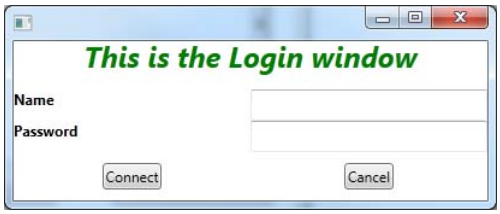

Figure 4.5 *Interface sans extension*

Figure 4.6 *Extension d'un bouton*

4.2.2 Supprimer un bouton

L'interface représentée par la Figure 4.8 a été modifiée par l'opération d'adaptation définie ci-dessous. Le bouton *Cancel* qui possède l'identifiant *button_0* a été supprimé de l'interface.

```
1. REMOVE #button_0;
```

Figure 4.7 *Interface sans suppression*

Figure 4.8 *Suppression d'un bouton*

4.2.3 Déplacer un bouton

L'interface représentée par la Figure 4.10 a été modifiée par l'opération d'adaptation définie ci-dessous. Le bouton *Cancel* qui possède l'identifiant *button_0* a été déplacé dans le conteneur dont l'identifiant est *box5*. Le bouton est situé à la 2ème ligne (indexée par le numéro 1) et dans la 3ème colonne (indexée par le numéro 2) du conteneur. Le mot clé *colinsert* signifie que le bouton doit être inséré dans une nouvelle colonne.

```
1. CHANGEBOX #button_0 TO #box5 WHERE ROW 1, COLINSERT 2;
```

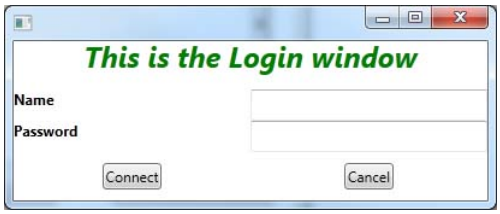

Figure 4.9 *Interface sans déplacement*

Figure 4.10 *Déplacement d'un bouton*

4.2.4 Modifier le label d'un bouton

L'interface représentée par la Figure 4.12 a été modifiée par l'opération d'adaptation définie ci-dessous. Le label du bouton *Cancel* qui possède l'identifiant *button_0* a été modifié pour devenir *test*. La propriété *Content* est une propriété de la sémantique de *XAML* représentant la valeur du label d'un bouton.

```
1. SET #button_0->Content TO "test";
```

Figure 4.11 *Interface sans modification*

Figure 4.12 *Modification du label d'un bouton*

4.2.5 Modifier la hauteur et la largeur du quadrillage d'un conteneur

L'interface représentée par la Figure 4.14 a été modifiée par l'opération d'adaptation définie à la ligne 1. Les hauteurs des lignes du conteneur ayant pour identifiant *box_0* ont été modifiées par les valeurs 20, 20 et 60. Ces valeurs représentent le pourcentage de place attribué à chaque ligne.

```
1. CHANGEROWS #box_0 TO 20 20 60;
2. CHANGECOLUMNS #box3 TO 25 75;
```

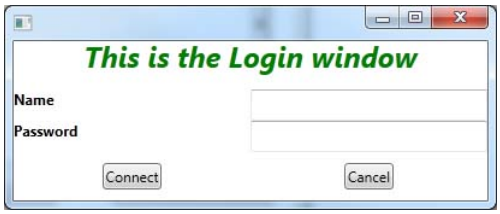

Figure 4.13 *Interface sans modification*

Figure 4.14 *Chgt. de la hauteur d'un conteneur*

L'interface représentée par la Figure 4.16 a été modifiée par l'opération d'adaptation définie à la ligne 2. Les largeurs des colonnes du conteneur ayant pour identifiant *box3* ont été modifiées par les valeurs 25 et 75. Ces valeurs représentent le pourcentage de place attribué à chaque colonne.

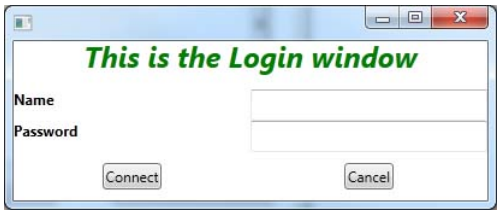

Figure 4.15 *Interface sans modification*

Figure 4.16 *Chgt. de la largeur d'un conteneur*

4.2.6 Sélectionner plusieurs éléments grâce aux *classSelectors* et aux *elementTypeSelectors*

Le prototype développé propose deux *plugins* de type *classSelector* pour sélectionner plusieurs composants graphiques : *SelectName* qui permet de sélectionner les composants graphiques dont l'identifiant contient la chaîne de caractères passée en paramètre et *SelectGroupType* qui permet de sélectionner les boutons *radios* d'un groupe correspondant à la chaîne de caractères passée en paramètre. *SelectName* est le *plugin* utilisé à la ligne 1. Tous les composants sélectionnés, à savoir les deux boutons *Connect* et *Cancel*, voient leur label modifié par la valeur *test*. Le *plugin SelectGroupType* est présenté à la section 4.4. Le résultat est illustré par la Figure 4.18.

Il est aussi possible de sélectionner plusieurs composants graphiques avec un *classSelector*. La ligne 2 utilise un *classSelector* qui sélectionne tous les composants graphiques de type *Button* et modifie la valeur du label par *test*. Le résultat est illustré par la Figure 4.18.

```
1. SET %SelectName "button"->Content TO "test";
2. SET @Button->Content TO "test";
```

Figure 4.17 *Interface sans modification*

Figure 4.18 *Interface modifiée*

4.3 Dégradation maximale de la taille d'une interface

L'interface représentée par la Figure 4.19 est dans ce cas sujette à une adaptation visant à réduire au maximum la taille qu'elle occupe. Pour ce faire, le code SIT ci-dessous est appliqué à l'interface. Le résultat est l'interface représentée par la Figure 4.27.

```
1. SUBSTITUTE #listbox_component_19 BY @ComboBox IN #box3 ("newComboBox") WHERE ROW 0, COL 0;
2. CONTRACT #newComboBox OF 90 50;
3. CHANGEBOX #button_1 TO #box3 WHERE ROW 0, COLINSERT 1;
4. SET #button_1->Content TO "GO!";
5. CHANGEBOX #button_0 TO #box3 WHERE ROW 0, COLINSERT 2;
6. SET #button_0->Content TO "[X]";
7. SET #label_0->FontSize TO 12;
8. CONTRACT #window_0 OF 40 120;
```

Tout d'abord, la *ListBox* contenant les noms est substituée par une *ComboBox*. Cette adaptation est réalisée par l'opération *substitute* à la ligne 1 et est représentée par la Figure 4.20. La *ListBox* ayant pour identifiant *listbox_component_19* est substituée

par la *ComboBox* ayant pour identifiant *newComboBox* et est insérée dans le conteneur *box3*. Ce conteneur ne contient qu'un élément d'où l'insertion à la position 0, 0.

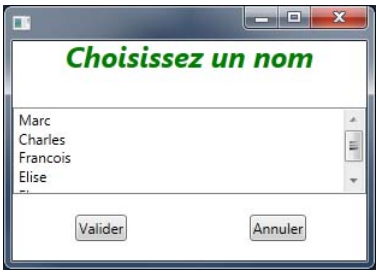

Figure 4.19 *Interface 1 avant l'adaptation*

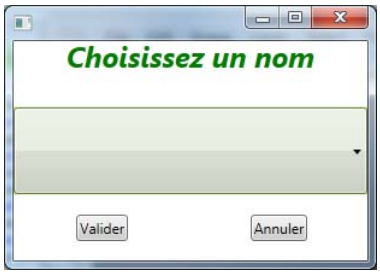

Figure 4.20 *Interface 1 après op. ligne 1*

Ensuite la *ComboBox* est contractée de 90 pixels en longueur et de 50 pixels en hauteur pour prendre moins de place via l'opération *contract* à la ligne 2. La Figure 4.21 présente cette transformation. Puis à la ligne 3, c'est le bouton *Valider* qui est déplacé grâce à l'opération *changebox* dans le conteneur *box3* auquel on ajoute une deuxième colonne (indexée par le numéro 1) avec le mot clé *colinsert*. Le déplacement est représenté par la Figure 4.22.

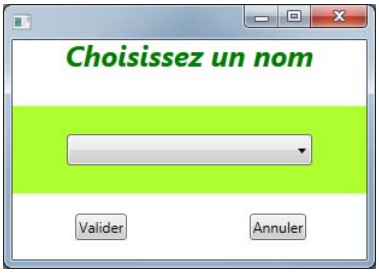

Figure 4.21 *Interface 1 après op. ligne 2*

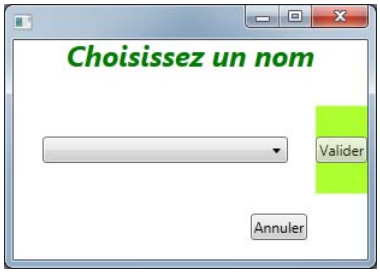

Figure 4.22 *Interface 1 après op. ligne 3*

Ensuite le bouton *Valider* est renommé par *Go !* grâce à l'opération *set* de la ligne 4 qui modifie la propriété *Content* du bouton. Le renommage est représenté par la Figure 4.23.

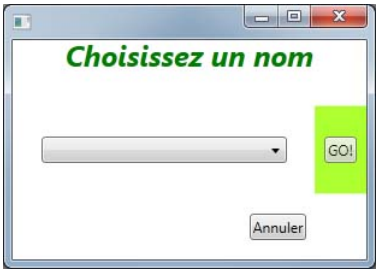

Figure 4.23 *Interface 1 après op. ligne 4*

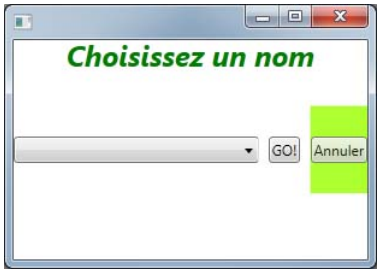

Figure 4.24 *Interface 1 après op. ligne 5*

Ensuite le bouton *Annuler* est soumis aux mêmes adaptations que le bouton *Valider*. Ensuite, la taille de police du label *Choisissez un nom* est diminuée grâce à l'opération *set* de la ligne 7 qui modifie la propriété *FontSize* du label. Le changement de taille de caractère est représenté par la Figure 4.26.

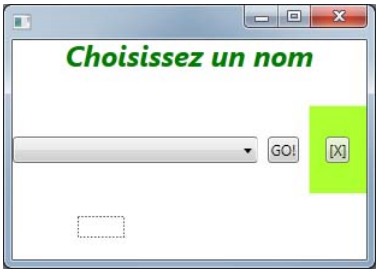

Figure 4.25 *Interface 1 après op. ligne 6*

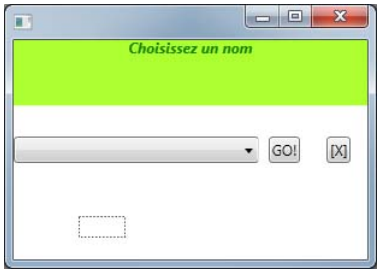

Figure 4.26 *Interface 1 après op. ligne 7*

Et pour terminer, la fenêtre est réduite de 40 pixels en longueur et 120 pixels en hauteur à la ligne 8 par l'opération *contract*. Le résultat final est représenté par la Figure 4.27.

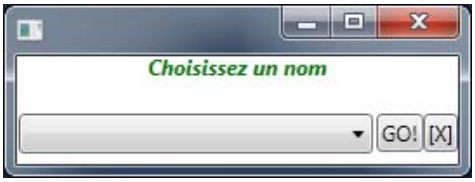

Figure 4.27 *Interface 1 après l'adaptation*

4.4 Substitutions de boutons *radios*

L'interface représentée par la Figure 4.28 est dans ce cas sujette à une adaptation visant à substituer, de plusieurs manières, les boutons *radios*. Pour ce faire, le code SIT ci-dessous est appliqué à l'interface. Le résultat est l'interface représentée par la Figure 4.32.

```
1. SUBSTITUTE %SelectRadioGroup "groupeType" BY @ComboBox IN #box110 ("newComboBoxType") WHERE ROW 0, COL 0;
2. SUBSTITUTE %SelectRadioGroup "groupeEtudiant" BY @CheckBox IN #box120 ("newCheckBoxEtudiant") WHERE ROW 0, COL 0;
3. CONTRACT #window_0 OF 50 50;
4. CHANGEROWS #box1 TO 25 25 25 25;
```

Les boutons *radios* définissant le genre du contact sont substitués par une *ComboBox* grâce à l'opération *substitute* à la ligne 1. Les boutons *radios* sont sélectionnés grâce à un *classSelector* appelé *SelectRadioGroup*. Ce *classSelector* sélectionne les boutons *radios* dont le groupe est égal à la chaîne de caractères passée en paramètre, dans notre cas : *groupeType*. La nouvelle *ComboBox* est placée dans le conteneur *box110* à la position 0, 0 c'est à dire à la place du bouton *radios* avec le label *Homme*. Cette substitution est représentée par la Figure 4.29.

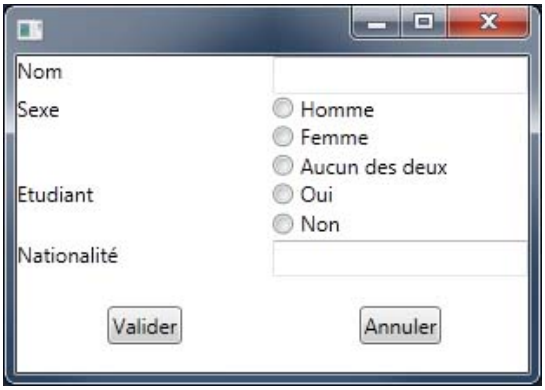

Figure 4.28 *Interface 2 avant l'adaptation*

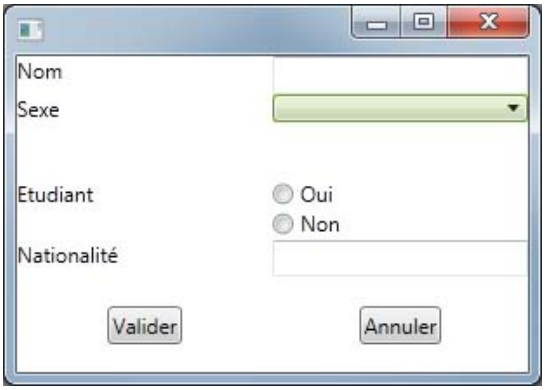

Figure 4.29 *Interface 2 après op. ligne 1*

Les boutons *radios* définissant si le contact est un étudiant sont substitués par une *CheckBox* grâce à l'opération *substitute* à la ligne 2. Les boutons *radios* sont aussi sélectionnés grâce au *classSelector SelectRadioGroup*. La nouvelle *ComboBox* est placée dans le conteneur *box120* à la position 0, 0 c'est à dire à la place du bouton *radios* avec le label *Oui*. La règle de substitution *RadiosButton* vers *CheckBox* ne coche la nouvelle *CheckBox* que si le *RadiosButton* portant le label *Oui* est coché. Cette substitution est représentée par la Figure 4.30. Ensuite la fenêtre est réduite de 50 pixels en longueur et en largeur grâce à l'opération *contract* à la ligne 3. Cette opération est représentée par la Figure 4.31.

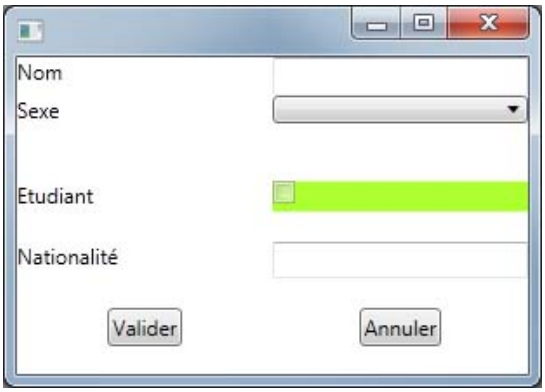

Figure 4.30 *Interface 2 après op. ligne 4*

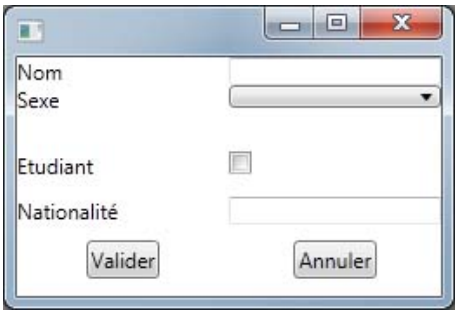

Figure 4.31 *Interface 2 après op. ligne 5*

Pour terminer, la hauteur du quadrillage principale de la fenêtre est redéfini à la ligne 4 de manière à diminuer la place prise par les boutons *radios* avant l'adaptation et d'augmenter la place attribuée aux champs Nom et Nationalité. Cette opération est réalisée grâce à *changerows*. Etant donné qu'il y a 4 champs donc 4 lignes sur l'interface, on attribue comme hauteur à chaque ligne un pourcentage de 25% ce qui donne la même hauteur à chaque ligne.

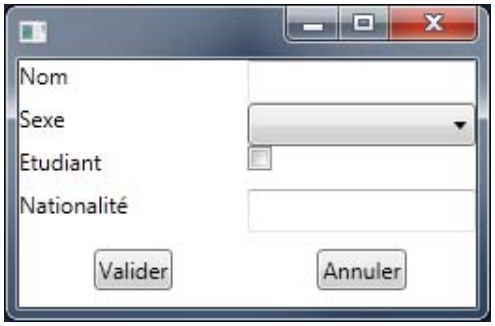

Figure 4.32 *Interface 2 après l'adaptation*

Conclusion

Dans un premier temps, les contributions du mémoire sont résumées. Ensuite, les limites du prototype sont présentées et discutées. Finalement, une note personnelle sur le mémoire est abordée.

Contributions

Comme expliqué dans l'introduction, la scénarisation de transitions animées entre deux interfaces s'insère dans un dessein plus vaste, à savoir la diminution des coûts de développement et de la déstabilisation cognitive.

Dans ce mémoire, une première approche concernant le développement d'un langage de transitions est proposée. Trois éléments essentiels sont ressortis du prototype implémenté :

- **La conversion d'*UsiXML* en *XAML*** : A défaut de disposer d'un langage capable d'interpréter du code *UsiXML*, il a fallu mettre au point une feuille de transformation *XSLT* pour convertir l'*UsiXML* en *XAML*.

- **L'interprétation de SIT** : Le langage de transition *Simple Interface Transition* a du être spécifié par une grammaire hors-contexte pour permettre au langage de proposer toutes les fonctionnalités requises.

- **Création d'une *Storyboard*** : Une *Storyboard* est une séquence d'animations qui est utilisée pour montrer à l'utilisateur final les transitions entre une interface originelle et une interface adaptée.

Le prototype a été développé en *C#*. Ce choix vient de la possibilité de manipuler facilement des composants graphiques grâce à la libraire *.NET*.

Réflexion sur le prototype

Le prototype développé montre l'intérêt et la viabilité de la scénarisation de transitions animées. En effet, l'application est en mesure de manipuler n'importe quelle interface pour peu qu'elle utilise les éléments graphiques gérés par le prototype. Les fonctionnalités implémentées permettent de modifier la place, certaines propriétés et les dimensions des éléments graphiques. Il est aussi possible de sélectionner un ou

plusieurs éléments graphiques selon 4 stratégies et de les substituer selon des règles spécifiées grâce à un système de *plugins*.

Néanmoins, le prototype possède ses limites ; soit en raison des fonctionnalités non implémentées, soit en raison des choix d'implémentation.

Concernant les fonctionnalités non implémentées, citons trois points importants :

- Seules les transitions de type *wipe* et *horizontal scroll left/right* ont été implémentées. Il serait intéressant de proposer plus de types de transition.

- Les opérations d'adaptation *distribute* et *play* sont deux opérations non implémentées mais qui ont, au même titre que les autres, une grande utilité.

- La grammaire de SIT ne gère pas les structures d'itérations, les variables et d'autres concepts qu'il serait intéressant d'implémenter pour permettre de créer des règles de substitution et des *classSelectors* dans le code SIT.

Concernant les limitations dues aux choix d'implémentation, l'utilisation d'une feuille de transformation *XSLT* pour convertir le code *UsiXML* en *XAML* s'est avérée être un frein dans le développement du prototype. Tout d'abord parce que la syntaxe *XSLT* est répétitive et peu maintenable lorsqu'il y a beaucoup d'éléments à convertir. C'est pourquoi seul un ensemble d'éléments graphiques sont convertis d'*UsiXML* en *XAML*. Ensuite, le chargement dynamique d'interface en *XAML* ne permet pas de gérer les événements contenus dans le fichier *XAML*. La solution à ces problèmes serait de créer un interpréteur *UsiXML* en *C#* qui servirait d'interface entre la sémantique du *UsiXML* et du *XAML*.

Réflexion sur le mémoire

Le travail que j'ai effectué m'a passionné. Au commencement, le sujet de ce mémoire m'est apparu comme un challenge dont les tenants et aboutissants étaient assez vagues. J'ai beaucoup appris sur les différentes manières d'adapter une interface.

Bien que le prototype développé reste en soi un prototype "jetable", j'espère que cette première approche permettra de mieux appréhender le problème de la déstabilisation cognitive à l'avenir.

Bibliographie

[1] UsiXML Consortium. Usixml v1.8.0 - documentation, 2007.

[2] Ralph Johnson John Vlissides Eric Gamma, Richard Helm. *Design Patterns : Elements of Reusable Object-Oriented Software*. 2009.

[3] Murielle Florins. *Graceful Degradation : a Method for Designing Multiplatform Graphical User Interfaces*. PhD thesis, Universite catholique de Louvain, 2006.

[4] Jean Vanderdonckt Francisco J. Martinez Ruiz, Jaime Munoz Arteaga. Transformation of xaml schema for ria using xslt & usixml.

[5] Benjamin Michotte Laurent Bouillon Daniela Trevisan Murielle Florins Jean Vanderdonckt, Quentin Limbourg. Usixml : a user interface description language for specifying multimodal user interfaces.

[6] Jean Vanderdonckt. Showing user interface adaptivity by animated transitions. 2010.