CHAPTER 1

Modélisation des interactions abstraites

Sommaire

1.1 Introduction

Les approches de migration d'UI génériques et réutilisables que nous avons étudiées dans le chapitre précédent se basent sur des modèles abstraits qui décrivent indépendamment des plateformes et des applications les différents aspects[1] des UI.

La migration d'UI vers les tables interactives implique un changement de modalité d'interactions car elles offrent des nouveaux types d'interactions. L'objectif de mise en place d'un mécanisme d'équivalences dynamiques entre instruments d'interactions des plateformes source et cible et qui permet la prise en en compte des spécificités[2] de la plateforme cible. Nous nous basons sur l'idée de décrire les correspondances entre les instruments d'interactions de la source et la cible à l'aide d'un modèle d'interactions abstraites.

Ce chapitre propose à la section 1.2 un modèle d'interactions abstraites qui décrit les activités atomiques possibles sur les composants graphiques dans l'objectif de décrire des équivalences. Ce modèle identifie l'ensemble des primitives d'interactions des composants graphiques, la section 1.3 présente les primitives d'interactions par rapport aux composants graphiques instanciés dans une UI et ceux d'une bibliothèque graphique. La section 1.3.2 propose un ensemble d'opérateurs d'équivalence entre les composants graphiques en se basant sur les primitives d'interactions.

[1] Interactions [GH95, KSM99], Structure, Positionnement [PSS09, VLM+04] et le Style

[2] Ces spécificités concernent les instruments d'interactions, les types d'UI et les guidelines associées.

1.2 Primitives d'interactions: un modèle des interactions abstraites

Les modèles des systèmes interactifs [NC94, PHR02, Weg97] distinguent deux types d'interactions entre les utilisateurs et une UI [W3C03]: les interactions en entrée et les interactions en sortie. Les interactions en entrée permettent de fournir des données ou d'invoquer des fonctionnalités du NF grâce à une UI et à des dispositifs d'entrée (souris, clavier, microphone, camera de reconnaissance gestuelle, écran tactile, accéléromètre, etc.). Les interactions en sortie permettent d'effectuer des rendus des données du NF à travers des dispositifs de sortie (tel un écran, des hauts parleurs, etc.). Les interactions (en entrée ou en sortie) se déclinent en plusieurs modalités d'interactions en fonction des langages et des dispositifs d'entrée ou des dispositifs de sortie utilisés. Chaque modalité d'interaction est utilisée comme un canal de communication entre un utilisateur et une application pour transmettre ou acquérir des informations [Nig94].

Une **primitive d'interaction** est une décomposition atomique d'une interactions[3] sur les composants graphiques d'une UI.

En considérant que les composants graphiques sont caractérisés par les données qu'ils contiennent, leurs propriétés graphiques et leurs comportements. Et qu'Ils appartiennent à des bibliothèques graphiques qui permettent de décrire des UI graphiques 2D, des images de synthèse en 3D, jeu vidéo, des graphiques de données [BCW+06, Lon10, SWND03], etc. Les interactions en entrée en modifient l'état d'un composant graphique soit à travers ses propriétés (données contenues, taille, position et représentation) ou par son comportement (tels les appels des fonctionnalités du NF et les interactions sur d'autres composants graphiques d'une UI, etc.). Les interactions en sortie quant à elles permettent d'afficher des données provenant du NF ou de changer les propriétés graphiques des composants graphiques de l'UI.

Les primitives d'interactions sont des interactions abstraites indépendantes des dispositifs physiques qui sont dérivés en interactions concrètes. Par exemple l'édition dans un champ de texte nécessite une sélection de ce champ de texte (**WidgetSelection** ou **Navigation**) avec une souris ou un clavier puis l'édition(**Edition**) de son contenu.

Cette section présente une caractérisation des primitives d"interactions en fonction des deux types d'interactions (entrée et sortie) ainsi qu'une méta modélisation des primitives d'interactions pour le processus de migration.

1.2.1 Les primitives d'interactions en entrée

Elles décrivent de manière abstraite toutes les actions des utilisateurs ou des autres composants graphiques qui permettent de modifier l'état d'un composant graphique. Nous avons identifiés comme interactions en entrée sur les contenus l'édition (**Data Edition**), la sélection (**Data Selection**) et le déplacement (**Data Move In** et **Data Move Out**) de contenus. Elles sont indépendantes des dispositifs d'interaction en entrée et sont effectuées avec des dispositifs de manipulation directe (souris, écran tactile, reconnaissance gestuelle, etc.) ou en combinant plusieurs mode d'interaction (clavier et souris).

Primitive d'interaction 1 : Data Edition
Elle consiste à modifier les données d'un composants graphiques qui ont des contenus.

[3] en entrée par l'utilisateur et en sortie par le NF

La modification du contenu d'un composant graphique peut se faire avec un clavier, une reconnaissance vocale, gestuelle ou de forme en fonction des plateforme. Dans le cadre des tables interactive, l'édition se fait avec un clavier virtuelle ou à main levée sur une écran tactile.

Primitive d'interaction 2 : Data Selection
Elle permet d'accéder aux contenus d'un composant graphique.

Les composants graphiques peuvent contenir plusieurs données, dans le cas des listes par exemple elle permet la sélection d'un contenu précis. Elle se fait à l'aide d'un clavier, d'une souris, d'un écran tactile, d'un dispositif de reconnaissance gestuelle, etc.

Primitive d'interaction 3 : Data Move In
Elle permet de recevoir le contenu d'un autre composant graphique de type compatible.

Les composants graphiques peuvent s'échanger des contenus, le drag et le drop permettent de déplacer un contenu à l'aide d'une souris ou d'un geste tactile. Cette interaction n'est pas que liées à une souris, elle peut être migré sur une table interactive en utilisant l'écran tactile ou émulée avec un objet tangible.

Primitive d'interaction 4 : Data Move Out
Elle permet d'exporter le contenu vers un autre composant graphique.

Cette primitive d'interaction correspond à la première action d'un drag-drop. Elle est indépendant des dispositifs physiques et peut être émulée sur une table interactive.

Les trois primitives d"interactions en entrée sur le(s) contenu(s) des composants graphiques décrit de manière exhaustive l'ensemble des actions possible qu'un utilisateur peut faire sur un élément graphique contenant des données utilisable par le noyau fonctionnel. En effet, si nous considérons le(s) contenu(s) d'un composant graphique comme une donnée, ces primitives d"interactions permettent la création, la modification, l'accès et la suppression des données (CRUD [D. 06]) d'un composant graphique.

Par ailleurs, les actions utilisateurs sur une UI[4] ne concernent pas que le contenu des composants graphiques, leurs propriétés graphiques sont modifiées aussi par soit un déplacement (**Widget Move**), soit une rotation sans changement de position (**Widget Rotation**), soit un redimensionnement (**Widget Resize**). Un composant graphique peut aussi être sélectionné de manière directe (**Widget Selection**) ou en naviguant à travers un container (**Navigation**). Les primitives d"interactions sur ces propriétés graphiques sont indépendantes des dispositifs d'interaction en entrée. En effet elles peuvent être effectuées avec des dispositifs de manipulation directe (souris, écran tactile, reconnaissance gestuelle, etc.) ou en combinant plusieurs mode d'interaction (clavier et souris).

Primitive d'interaction 5 : Widget Move
Elle permet d'exprimer le changement de la position d'un composant graphique.

[4]Ces actions concerne les UI en 2D telles les GUI classiques

La position des composants graphiques peuvent être changée par un utilisateur à l'aide d'un dispositifs de manipulations directes ou des raccourcis d'un clavier. Cette primitive d'interaction est définie pour les composants graphiques déplaçable à l'aide.

Primitive d'interaction 6 : Widget Rotation
Elle permet d'exprimer le changement d'orientation d'un composant graphique.

L'orientation des composants graphiques est une caractéristique importante dans le cadre des UI collaboratives et co localisée. Nous considérons le changement d'orientation comme une primitive d'interaction car elle peut être définie pour des écran utilisables par plusieurs personnes tels qu'une table interactive, une tablette tactile, un ordinateur portable avec écran pliable, etc. La rotation de composant graphique n'est pas définie pour les composants graphiques de la plateforme desktop. Dans le cadre de la migration vers des plateformes multi utilisateurs cette primitive d'interaction est indispensable.

Primitive d'interaction 7 : Widget Resize
Elle permet de modifier les dimensions d'un composant graphique.

Les dimensions d'une composant graphique peuvent être redéfinie par un utilisateur d'une UI par une interaction. Cette action peut être effectuée avec un clavier ou une souris sur un desktop et à l'aide d'un geste tactile sur une table interactive.

Primitive d'interaction 8 : Widget Selection
Elle permet d'exprimer la sélection immédiate de composants graphiques avec un dispositif de manipulation directe.

Primitive d'interaction 9 : Navigation
Elle permet d'exprimer la sélection d'un composant graphique de manière séquentielle

Les primitives d"interactions **Widget Selection** et **Navigation** décrivent les différents types de sélections d'un composant graphique par un utilisateur.

Les primitives d"interactions ci-dessus[5] caractérisent les actions utilisateurs sur les propriétés graphiques(position, orientation, taille et représentation) d'un composant graphique.

Primitive d'interaction 10 : Activation
Elle permet de décrire les interactions des composants graphiques avec le NF d'une applications

L'activation représente le lien entre la structure d'une UI les fonctionnalités. Concrètement, dans le cadre d'une architecture MVC, cette primitive permet de représenter une appel de méthode du contrôleur par un élément de la vue.

[5]Widget Move, Widget Rotation, Widget Resize, Widget Selection et Navigation

Résumé

Les primitives d"interactions en entrée caractérisent les actions possibles des utilisateurs sur le(s) contenu(s), la taille, la position, l'orientation, et la représentation des composants graphiques d'une UI.

En considérant l'artéfact d'UI de l'application CBA **??** par exemple, la liste d'images permet aux dessinateurs de BD de sélectionner des images prédéfinies et de l'ajouter à sa BD. Les éléments de cette liste peuvent être ajoutés au canevas de dessin par une interactions de glisser-déposer (drag-drop). Le tableau 1.1 présente les primitives d"interactions de la liste d'images et de la liste déroulante (ComboBox) de la figure 1.1

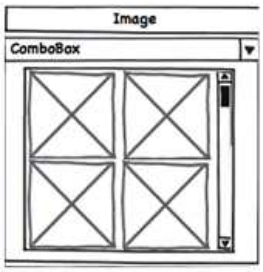

Figure 1.1: Artéfact d'UI

Composants graphiques	**Primitives d'interaction en entrée**
Liste d'images	Widget Selection, Navigation, Widget Display, Data Selection, Data Move Out, Activation
Liste déroulante	Widget Selection, Navigation, Data Selection, Data Display, Activation

Table 1.1: Exemple de primitives d'interactions en entrée

1.2.2 Les primitives d'interactions en sortie

Elles décrivent de manière abstraite les feed-back, les réponses ou les réactions du NF sur les différents éléments d'une UI. Elles concernent le(s) contenu ou la représentation d'un composant graphique. Nous avons identifiés deux primitives d'interactions en sortie: la primitive d'interactions **Data Display** et la primitive d'interactions **Widget Display**

Primitive d'interaction 11 : Data Display
Elle permet d'exprimer la modification du contenu d'un élément graphique par le NF

Certains composants graphiques sont utilisés pour afficher des données provenant du NF, cette primitive exprime par exemple la possibilité pour le NF de présenter des données provenant du modèle dans cas d'une architecture MVC par exemple.

Primitive d'interaction 12 : Widget Display
Elle exprime les actions du NF sur l'aspect visuel d'un composant graphique

Cette primitive d'interactions représente la mise à jour des propriétés graphiques (visibilité, position, taille) d'un composant graphique de la Vue par le Contrôleur ou Modèle dans le cadre d'une architecture MVC par exemple.

Résumé

Les primitives d'interactions en sortie caractérisent les réactions et les feedback du NF sur les composants graphiques d'une UI.

Considérons par exemple la boîte de dialogue de la figure 1.2 de l'application CBA, elle est affichée après une demande de fermeture d'une document. La boîte de dialogue aura la primitive d'interactions **Widget Display** tels que décrit dans le tableau 1.2. Le message est affiché dans un label qui obtient le nom du fichier à partir du modèle, ce label aura donc la primitive d'interactions **Data Display**.

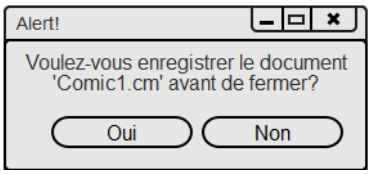

Figure 1.2: Boîte de dialogue

Composants graphiques	**Primitives d'interactions en sortie**
Label Message	Data Display
Boîte de dialogue	Widget Display

Table 1.2: Exemple de primitives d'interactions en entrée

1.3 Primitives d'interactions et Composants graphiques

Les composants graphiques appartiennent à des bibliothèques graphiques (ou boîtes à outils) et sont utilisés pour décrire des UI. Les composants graphiques qui constituent une UI sont des instances des composants graphiques d'une bibliothèque graphique. Les composants graphiques instanciés n'implémentent pas systématiquement toutes les interactions possibles de son type. En considérant l'illustration de la figure 1.3 du lien entre une instance et son type d'une ListBox. De manière intrinsèque, le composant ListBox définie les primitives d'interactions **Data Move Out** et **Data Move In** car il supporte l'interaction glisser-déposer (drag-drop). Cependant de manière effective, il est possible de l'instancier sans implémenter ces primitives d'interactions dans le cas d'une liste qui ne permet pas d'effectuer un glisser-déposer de son contenu.

Nous constatons que pour des composants graphique qu'il existe des **primitives d'interactions intrinsèques** et des **primitives d'interactions effectives**.

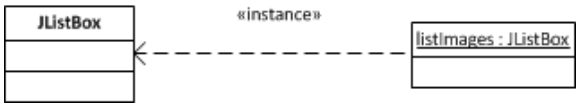

Figure 1.3: Lien type-instance des composants graphiques

Dans le cadre d'un processus de migration, l'on remarque aussi que pour les primitives d'interactions des éléments de l'UI à migrer sont des **primitives d'interactions effectives** car elles sont issues des instances des composants graphiques. Les composants graphiques de l'UI cible seront instanciés à partir de la bibliothèque graphique de la plateforme d'arrivée et ces instances devront conservés au moins les même primitives d'interactions que l'UI de départ.

Les primitives d'interactions présentées à ci-dessus(cf. section 1.2) sont utilisées pour décrire l'ensemble des interactions possibles sur les composants graphiques. Elles font parties des modèles abstraits d'UI avec les modèles de structures, de layout et de styles. Les primitives d'interactions ne décrivent pas les aspects structurels d'une UI tels que les types des données, la cardinalités des données ou le regroupement des composants graphiques. Dans notre processus de migration d'UI vers les tables interactives, nous nous basons sur deux modèles abstraits (les primitives d'interactions et un modèle de structure) pour décrire les mécanismes automatiques de migration. Nous résumons les différents éléments intervenants dans notre solution par la figure 1.4, en effet l'UI finale est décrite par des instances des éléments d'une bibliothèque graphique. Et les composants graphiques d'une bibliothèque graphiques sont décrite par un modèle de composant graphique générique, chaque composant graphique abstrait définit des primitives d'interactions intrinsèques. Par ailleurs les éléments d'une UI finale sont modélisés à l'aide d'un modèle de structure[6] dont chaque instances implémente des primitives d'interactions effectives.

Cette section présente d'abord un modèle de composant graphique ainsi que les règles qui permettent d'identifier les **primitives d'interactions intrinsèques** à chaque composant graphique. Ensuite cette section présente un modèle de structure d'une instance d'UI ainsi que les règles d'identifications des **primitives d'interactions effectives**.

1.3.1 Un modèle abstrait de composants graphiques

Nous proposons dans cette section une formalisation des différentes caractéristiques d'un composant graphique dans un méta modèle(cf figure 1.5), ceci dans le but de faciliter la définition des règles d'identification des primitives d'interactions des composants graphiques d'une UI à migrer.

"Widget is a user interface object which defines specific interaction behaviour and a model of information presented to the user" [Cre01]

Nous complétons cette définition en prenant en compte les caractéristiques graphiques telles que la taille, la position, l'orientation et la structure du Widget. Nous ne considérons pas les caractéristiques liées aux styles de présentation ou au layout car notre processus de migration permet aux développeurs de personnaliser l'UI pendant la migration.

Widget Le méta modèle de la figure 4-2 décrit un composant graphique par la classe *Widget* avec son nom (*name*) et le nombre d'éléments qu'il peut contenir (*cardinaltity*). Le champ *name* permet de l'identifier de manière unique dans une bibliothèque graphique. La cardinalité permet quand à

[6]Le modèle de structure décrit les types de données, la cardinalité des données et le regroupement des éléments graphiques d'une UI

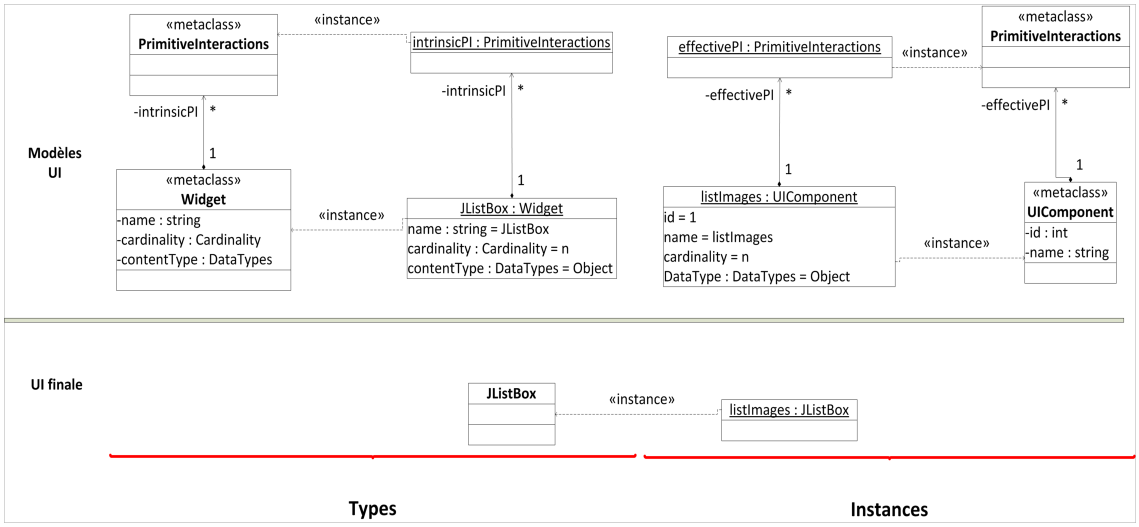

Figure 1.4: Modèles abstraits d'UI et UI finale

elle de préciser le nombre de données ou de *Widget* maximal qu'il peut contenir. Un composant graphique nécessite dans certains cas d'autres composants graphiques pour ses contenus, l'association *containedWidget* représente les widgets utilisés comme item. Par exemple en XAML [Mic12] les ListBox nécessitent des ListBoxItem. La classe Widget correspond à tous les composants graphiques d'une bibliothèque graphique.

Behaviour Le méta modèle de la figure 1.5 considère que les comportements des composants graphiques consistent à modifier ou à sélectionner son contenu, sa taille, sa position ou sa structure. Il considère aussi qu'un comportement fait aussi appel à des fonctionnalités. La classe *Behaviour* est un attribut de *Widget*, qui est caractérisée par les types de comportements (*type*) et les propriétés qu'elle impacte (*targetProperty*). Nous identifions trois types de comportements (*BehaviorType)* parmi les caractéristiques d'un *Widget* :

1. la modification (*Change*) de valeurs du contenu (*Content*), des propriétés graphiques (*Size, Position, Orientation*)

2. la sélection (*Select*) d'un contenu ou du Widget lui-même (*WidgetStructure*)

3. et l'appel d'une fonctionnalité (*Call*).

L'attribut *targetProperty* de la classe *Behaviour* permet de préciser le type de propriété modifiée par un comportement, dans le cas d'un comportement de type *Call* alors ma propriété cible est nulle. Si un comportement est de type *Select* alors la propriété cible est de type *WidgetStructure* ou *Content*. En effet la sélection s'effectue uniquement sur un composant graphique ou sur son contenu. Si un comportement est de type *Change* alors la propriété cible est de type *Size*, *Position*, *Orientation* ou *Content*. En effet, le contenu, la taille, la position ou l'orientation d'un composant graphique sont modifiés.

Un comportement peut combiner plusieurs types par exemple une liste déroulante permet de sélectionner un item de la liste et aussi de déclencher une fonctionnalité. Elle aura un attribut *Behaviour*

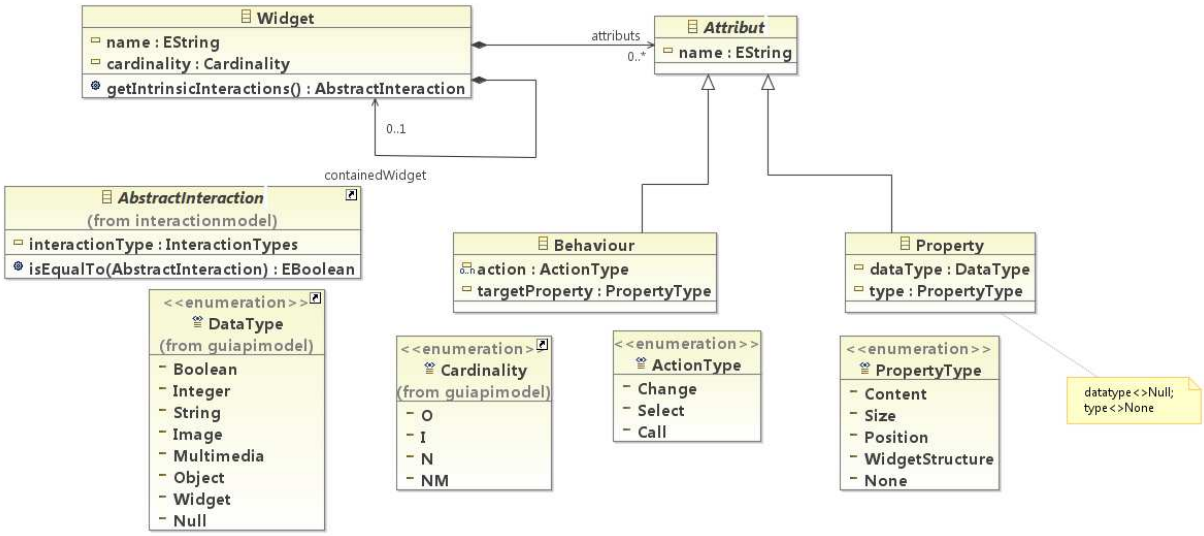

Figure 1.5: Méta modèle de composant graphique

avec les types *Change* et *Call* sur la propriété de type *Content*. Un *Widget* peut avoir plusieurs comportements pour exprimer toutes ses interactions intrinsèques, en effet les comportements décrivent l'ensemble des interactions possibles d'un composant graphiques, ces comportements sont spécifiques à une bibliothèque graphique.

La classe *Behaviour* est implémentée de différentes manières par les bibliothèques graphiques. Les comportements de type *Change* et *Select* sont identifiés en fonction des événements, des méthodes ou des propriétés des composants graphiques. Par exemple en Java Swing, la propriété *isEditable* permet de dire qu'un composant graphique définit de manière intrinsèque un comportement pour changer son contenu. La méthode *addActionListener* permet de dire qu'un composant graphique définit de manière intrinsèque un comportement pour faire appel à des fonctionnalités et la méthode *getSelectedItem* permet de dire qu'un composant graphique défini de manière intrinsèque un comportement pour sélectionner son contenu. Les comportements de la bibliothèque graphique XAML sont identifiés à partir des événements et des propriétés ; par exemple l'événement *SelectedText* permet d'exprimer le comportement de sélection d'un contenu, et la propriété *AllowDrop* permet d'exprimer que le contenu peut être changé.

Pour chaque bibliothèque graphique, une table décrivant les correspondances entre les types de comportement et les caractéristiques des composants graphiques qui permet de les identifier de manière intrinsèque est fournie, des exemples de cette table sont décrits de manière exhaustive pour les bibliothèques graphiques XAML et XAML Surface à l'annexe (réf).

Property La classe *Property* (cf. figure 1.5) permet de décrire les attributs statiques des *Widget* qui sont impactés par les comportements. Les types des propriétés sont identifiés de manière spécifique à chaque bibliothèque graphique en décrivant une correspondance entre les propriétés des composants graphiques et les types de propriétés génériques du méta modèle de composant graphique. Pour chaque bibliothèque graphique, il faut fournir les différents noms des propriétés des composants graphiques correspondant aux différents types (*Content, Size, Position, Orientation, WidgetStructure*).

Les contenus d'un composant graphique sont de type booléen(*Boolean*), entier (*Integer*), chaîne de caractère(*String*), image (*Image*), son ou vidéo (*MediaElement*), des classes indépendantes de la bibliothèque graphique (*Object*), des composants graphiques (*Widget*). Tout autre type n'est pas pris en compte par ce modèle de composant graphique. L'identification des types est faite à l'aide d'une table de correspondances entre les types spécifiques aux bibliothèques graphique et les types décrit dans notre modèle. L'annexe (réf) présente des exemples de cette table de manière exhaustive pour les bibliothèques graphiques XAML et XAML Surface.

Le nom d'une propriété (tel que décrit dans la bibliothèque graphique) est conservé pendant la migration (*name)*, pour permettre à l'UI migré de préserver les ressources (étiquettes des widgets, etc.) de l'UI de départ, ceci permet de retrouver dans la structure analysable (*UIStructure*) la valeur de la propriété à partir de son nom.

Identification des primitives d'interactions intrinsèques

La méthode $getIntrinsicInteractions : \{Attributs\} \longrightarrow \{AbstractInteraction\}$ du méta modèle (cf. figure 1.5) permet d'identifier les primitives d'interactions d'intrinsèque d'un Widget. Cette méthode se base sur des règles pour identifier chaque primitive d'interaction.

Règle 1 : Widget Selection et Navigation
Widget Selection et **Navigation** sont définies de manière intrinsèque pour tous les widgets.

Règle 2 : Widget Resize
Widget Resize est définie de manière intrinsèque pour tous les widgets qui ont un comportement de changement des propriétés de type *Size*.

$$\exists prop \in Widget.attributs, prop.type = Size$$
$$\bigwedge$$
$$\exists beh \in Widget.attributsbeh.type = Change$$
$$\bigwedge$$
$$beh.targetProperty = Size$$

Règle 3 : Widget Move
Widget Move est définie de manière intrinsèque pour tous les widgets qui ont un comportement de changement des propriétés de type *Move.*

$$\exists prop \in Widget.attributs, prop.type = Position$$
$$\bigwedge$$
$$\exists beh \in Widget.attributs/beh.type = Change \bigwedge beh.targetProperty = Position$$

Règle 4 : Widget Rotation
Widget Rotation est définie de manière intrinsèque pour tous les widgets qui ont un comportement de changement des propriétés de type *Orientation*.

$$\exists prop \in Widget.attributs, prop.type = Orientation$$

$$\bigwedge$$

$$\exists beh \in Widget.attributs / beh.type = Change \bigwedge evt.targetProperty = Orientation$$

Règle 5 : Widget Display
Widget Display est définie de manière intrinsèque pour tous les widgets.

Règle 6 : Data Edition
Data Edition est définie de manière intrinsèque pour tous les widgets qui ont un comportement de changement des propriétés de type *Content* et dont le type de données contenu n'est pas un *Widget*.

$$\exists prop \in Widget.attributs / prop.type = Content$$

$$\bigwedge$$

$$prop.dataType \notin \{Widget,\ Null\}$$

$$\bigwedge$$

$$\exists beh \in Widget.attributs / beh.type = Change \bigwedge beh.targetProperty = Content$$

Règle 7 : Data Selection
Data Selection est définie de manière intrinsèque pour tous les widgets qui ont un comportement de sélection des propriétés de type *Content*

$$\exists prop \in Widget.attributs, prop.type = Content$$

$$\vee$$

$$\exists beh \in Widget.attributs / beh.type = Select \wedge beh.targetProperty = Content$$

Règle 8 : Data Move In
Data Move In est définie de manière intrinsèque pour tous les widgets qui ont un comportement de changement des propriétés de type *Content*

$$\exists prop \in Widget.attributs / prop.type = Content$$

$$\bigwedge$$

$$\exists beh \in Widget.attributs / beh.type = Change \bigwedge beh.targetProperty = Content$$

Règle 9 : Data Move Out
Data Move Out est définie de manière intrinsèque pour tous les widgets qui ont un comportement de sélection et de changement des propriétés de type *Content.*

$$\left\{\begin{array}{c}\exists\ prop1 \in Widget.attributs/\ prop1.type = Content \\ \wedge \\ \exists beh1 \in Widget.attibut\ /beh1.type = Select \\ \wedge\ beh1.targetProperty = Content\end{array}\right\}$$

$$\wedge$$

$$\left\{\begin{array}{c}\exists\ prop2 \in Widget.attributs,\ prop2.type = Content \\ \wedge \\ \exists\ beh2 \in Widget.attributs/beh2.type = Change \\ \wedge beh2.targetProperty = Content\end{array}\right\}$$

$$\wedge$$

$$prop1.valueDataType = prop2.valueDataType$$

Règle 10 : Data Display
Data Display est définie de manière intrinsèque pour tous les widgets qui ont des propriétés de type *Content.*

$$\exists\ prop \in Widget.attributs/\ prop.type = Content$$

Règle 11 : Activation
Activation est définie de manière intrinsèque pour tous les widgets qui ont des comportements de type *Call*

$$\exists\ evt \in Widget.attributs/beh.type = Call$$

1.3.2 Un modèle de structure d'UI

La structure des instances d'UI à migrer est décrite sous différents formats (XML, Archives Java, etc.) cependant tous ces formats peuvent être représentés à l'aide d'une structure arborescente. En effet, une fenêtre d'une UI peut être considérer comme la racine de l'arbre la représentant et les éléments de la fenêtre seront des nœuds fils. Si un ensemble de composants graphiques est regroupé dans un container alors ils seront représentés par nœud parent correspondant au container et des nœud fils correspondant aux composants de l'ensemble.

Les structures des UI sont modélisées au niveau CUI du CRF [CCB+02], les différents méta modèle du niveau CUI [W3C10, VLM+04, PSS09] caractérisent entre autre les liens de contenance entre les éléments d'une UI, le cardinalités et les types de données que contiennent les composants graphiques. Les modèles de CUI caractérisent aussi le style ou layout dans le cadre des UI graphiques, par exemple le langage UsiXML a plusieurs balises de layout pour le modèle CUI telque *box, groupBox, flowBox,* etc. [SC12].

Dans notre contexte, nous considérons qu'un modèle de CUI capable de décrire les liens de contenance, la cardinalité et les types de données d'une UI peut être utilisés. Le but d'un modèle de structure pour le processus de migration est de conserver les **données** et les **liens entre les composants graphiques** de l'UI de départ.

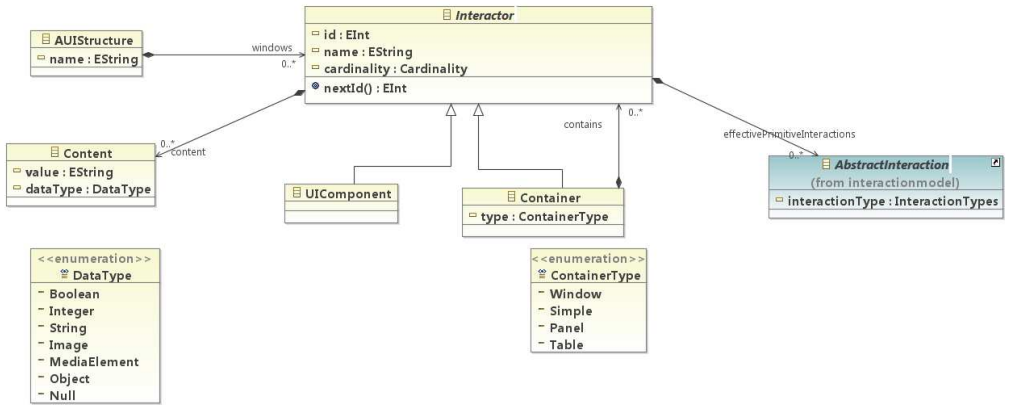

Figure 1.6: Méta modèle de structure

Contenu	Type de container
$\{UIComponent\}$	**Homogène**: données du même type **Hétérogène**: données de type différents **Racine**: si pas de parent
$\{UIComponent\} \cup \{Container\}$	**Hétérogène:** si a un parent **Racine**: si pas de parent
$\{Container\}$	**Récursif:** si contient des containers **Racine**: si pas de parent

Table 1.3: Types de container

Le modèle de structure que nous utilisons pour la migration comporte des instances de composants graphiques d'une UI que nous représentons dans un modèle abstrait sous forme d'*interacteurs*. Ces *interacteurs* qui sont représentés par la classe abstraite *Interactor* (cf. figure 1.6) sont caractérisées par leur identifiant unique pour une instance d'UI et leur nom qui correspond au nom du composant graphique de l'UI à migrer. Ils préservent les valeurs des données structurelles de l'UI à migrer par la classe *Content* , par exemple les étiquettes des labels, des boutons, des images, etc. On distingue deux types d'instances de widgets :

- *UIComponent* représentant l'ensemble des composants graphiques ne pouvant pas contenir de composant graphique. Ils sont identifiés à partir des widgets dont les contenus ne sont pas d'autres widgets,

- et *Container* représentant l'ensemble des composants graphiques pouvant contenir d'autres instances de widgets. Ils sont identifiés à partir des widgets pouvant contenir d'autres widgets.

Types de Container

Les Container peuvent être catégorisés en fonction des interacteurs qu'ils contiennent. Nous caractérisons dans le tableau 1.3 quatre types de container. Les types de containers sont identifiés à partir de la structure d'une UI de manière récursive en identifiant d'abord les types des interacteurs fils.

Container Racine C'est la d'une structure d'UI. Il peut contenir des containers et des interacteurs simple.

Container Récursif C'est un container ne contenant que des *Container*. Ce type permet d'identifier un regroupement de plusieurs containers sur le quel on pourra appliquer des transformations. Un container de type *Récursif* ne peut donc pas contenir des interacteurs de types *UIComponent*. En considérant la figure 1.7, le container vert de est une illustration d'un *Container* de type *Récursif*.

Container Homogène C'est un container contenant uniquement des *UIComponent* dont les données du même type et de même cardinalité. Un container de ce type permet de définir un groupe de composants graphique de même type (tel qu'une liste, un tableau, un menu, etc.). Le container bleu de la figure 1.7 est une illustration d'un *Container* de type *Homogène*.

Container Hétérogène C'est un container pouvant contenir à la fois des*UIComponent* et des *Container*. Ce type permet d'identifier les instances de widgets pouvant contenir des données non structurées et qui sont interprétés de diverses manières pendant la migration. En considérant la figure 1.7 Le container rouge est une illustration d'un *Container* de type *Hétérogène*.

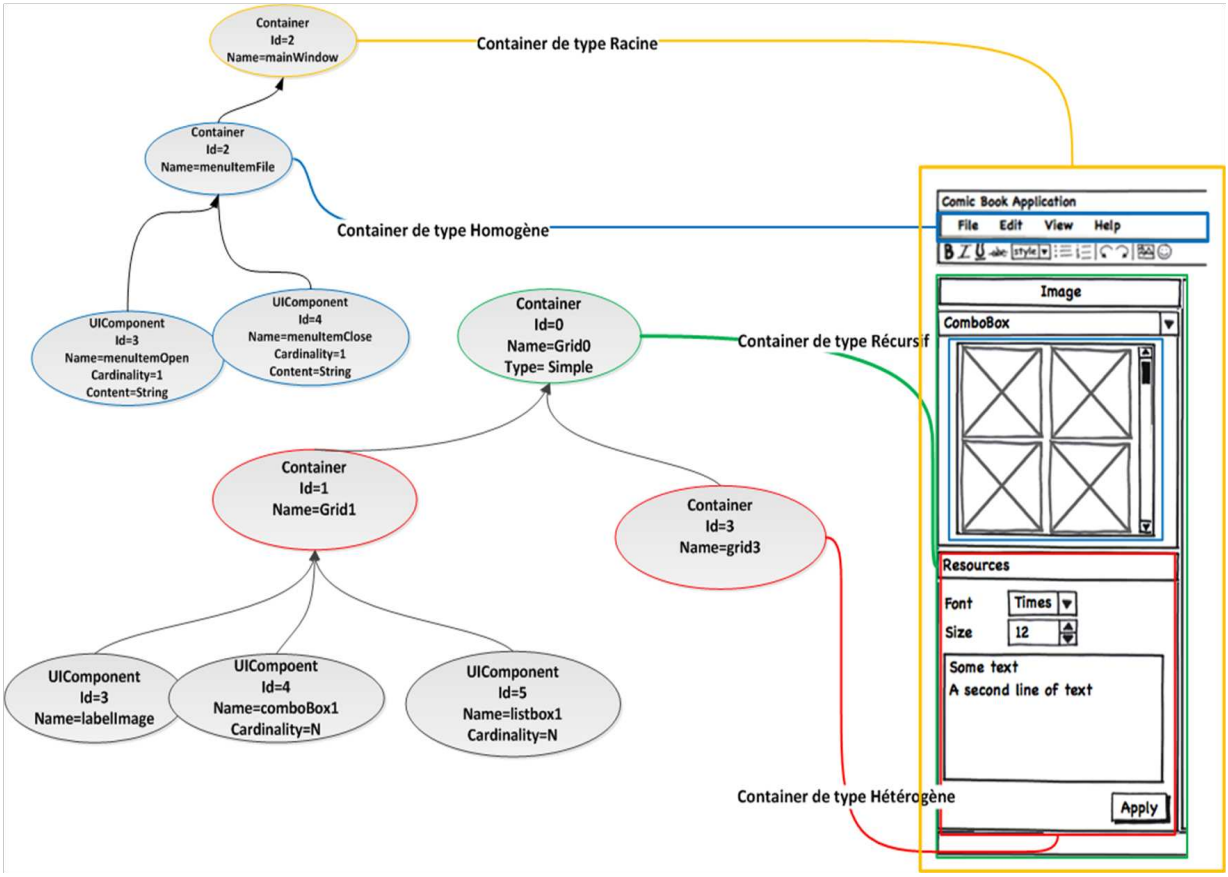

Figure 1.7: Illustration des types de container

Données des interacteurs

Le modèle de structure présenté à la figure 1.6 permet de préserver les données d'une UI à migrer. Le processus de migration dans notre cas permet un changement de la modalité d'interactions mais les données contenues dans une UI ou échangées entre l'UI et NF doit être préserver car le NF est réutilisés.

Le modèle de structure 1.6 propose un ensemble de type de données classiques d'UI. *DataType* est considéré comme un modèle de type de données abstrait qui sera instancié en fonction de chaque langage de programmation. Par exemple le type *MediaElement* représente un objet de type vidéo et le type *Object* désigne des types complexes.

Primitives d'interactions effectives

Les primitives d'interactions réellement utilisées par chaque interacteur sont identifiées à partir de la structure de l'UI de départ. Cependant pour préserver l'assemblage des composants graphiques de l'UI à migrer, on extrait au moyen d'interacteurs cette structure arborescente à l'aide du méta modèle *AUIStructure*. Il décrit la structure de l'UI comme un ensemble d'arbre dont les racines sont des fenêtres, les nœuds des interacteurs et les arcs la relation de contenance entre les interacteurs. Chaque interacteur a une méthode qui permet d'identifier les primitives d'interactions effectives en fonction du nœud correspondant dans la structure analysable de départ et du modèle de *Widget*.

On considère que l'UI de départ est une structure analysable *UIStructure* qui est constitué d'un ensemble d'arbres (*Tree*) qui décrit les différentes fenêtres de l'UI à migrer.

$$UIStructure = \bigcup_{i=0}^{N} Tree_i$$

$$Tree_i = \langle root,\ Node,\ Contains \rangle$$

La méthode $getEffectiveInteractions : WidgetNode \rightarrow \{AbstractInteraction\}$ du méta modèle (cf. figure 1.5) permet d'identifier les primitives d'interactions utilisées par un interacteur. Cette méthode se base sur des règles effectives suivantes.

Règle 12 : Widget Selection et Navigation
Widget Selection et **Navigation** sont définies de manière effective pour les interacteurs accessible dont les propriétés de type *WidgetStructure*(cf. figure 1.5)sont de type booléen et avec une valeur différente de false.

$$\exists prop \in Widget.attributs, prop.type = WidgetStructure$$

$$\wedge$$

$$\exists att \in Node / prop.dataType = Boolean$$

$$\wedge$$

$$att.name = prop.name \wedge node.value \neq False$$

Règle 13 : Widget Resize
Widget Resize est définie de manière effective pour les interacteurs redimensionnables dont les propriétés de type *Size* sont de type booléen et avec une valeur différente de false.

$$\exists prop \in Widget.attributs, prop.type = Size$$
$$\wedge$$
$$\exists att \in Node / prop.dataType = Boolean$$
$$\wedge$$
$$att.name = prop.name \wedge node.value \neq False$$

Règle 14 : Widget Mode
Widget Move est définie de manière effective pour les interacteurs déplaçables dont les propriétés de type *Move* sont de type booléen et avec une valeur différente de false.

$$\exists prop \in Widget.attributs, prop.type = Move$$
$$\wedge$$
$$\exists att \in Node / prop.dataType = Boolean$$
$$\wedge$$
$$att.name = prop.name \wedge node.value \neq False$$

Règle 15 : Widget Rotation
Widget Rotation est définie de manière effective pour les interacteurs orientables dont les propriétés de type *Orientation* sont de type booléen et avec une valeur différente de false.

$$\exists prop \in Widget.attributs, prop.type = Orientation$$
$$\wedge$$
$$\exists att \in Node / prop.dataType = Boolean$$
$$\wedge$$
$$att.name = prop.name \wedge node.value \neq False$$

Règle 16 : Widget Display
Widget Display est définie de manière effective pour tous les interacteurs.

Règle 17 : Data Edition
Data Editionest définie de manière effective pour tous les interacteurs accessibles dont les widgets pour lesquels la primitive est définie de manière intrinsèque.

Règle 18 : Data Selection
Data Selection est définie de manière effective pour tous les interacteurs accessibles dont les widgets pour lesquels la primitive est définie de manière intrinsèque.

Règle 19 : Data Move In
Data Move In est définie de manière effective pour tous les interacteurs accessibles dont les widgets pour lesquels la primitive est définie de manière intrinsèque

Règle 20 : Data Move Out
Data Move Out est définie de manière effective pour tous les interacteurs qui définissent de manière effective **Data Selection** et **Data Edition**.

Règle 21 : Data Display
Data Display est définie de manière effective pour tous les interacteurs qui ont un contenu ou dont la propriété de contenu est modifiée dans une méthode.

Règle 22 : Activation
Activation est définie de manière effective pour tous les interacteurs qui font appel à une méthode du contrôleur (dans une architecture MVC)

1.3.3 Synthèse des modèles abstraits

Cette section a présenté un modèle de composants graphiques et un modèle de structure pour une instance d'UI. Le modèle de composant graphique décrit de manière abstraite tous les aspects des widgets d'une bibliothèque graphique. En effet, le méta modèle de la figure 1.5 décrit le contenu, la cardinalité, les propriétés graphiques (position et taille), les comportements et les interactions d'un composant graphique indépendamment de sa représentation par une boîte à outils donnée.

Par ailleurs le modèle de structure (cf. figure 1.6) représente les liens de contenances et les données d'une instance d'UI à migrer. Ce modèle est certes composé d'instances de composants graphiques mais il ne décrit pas tous les aspects d'une UI.

Dans le cadre de la migration, les applications à migrer sont abstraits dans le modèle de structure *AUIStructure* en exprimant aussi les primitives d'interactions effectives. L'application source ainsi abstrait est transformée pour être conforme aux recommandations de la plateforme cible. La section suivante présente le mécanisme de migration basée sur les primitives d'interactions et les modèles décrits dans cette section.

1.4 Opérateurs d'équivalences des composants graphiques basés sur des primitives d'interactions

Les *Widget* ou *Interactor* sont caractérisés respectivement par leurs primitives interactions intrinsèques et effectives qui sont des sous-ensembles des primitives d'interactions. Nous définissons trois opérateurs de comparaisons de ces sous-ensembles de primitives d'interaction: l'égalité($=$), l'inclusion($\subseteq$) et la contenance($\supseteq$).

Égalité des deux sous ensemble de primitives d'interactions
$=: \{PrimitiveInteractions\} \times \{PrimitiveInteractions\} \rightarrow Boolean.$
Si toutes les primitives d'interactions de deux sous-ensemble sont identiques.

Inclusion des primitives d'interactions
$\subseteq: \{PrimitiveInteractions\} \times \{PrimitiveInteractions\} \rightarrow Boolean$
Si toutes les primitives d'interaction du sous ensemble de gauche se retrouvent dans le sous-ensemble

de droite. Ceci est possible si une instance de *Widget* n'utilise pas toutes ses primitives d'interaction intrinsèques.

Contenance des primitives d'interactions

$\supseteq$: $\{PrimitiveInteractions\} \times \{PrimitiveInteractions\} \rightarrow Boolean$

Ceci est possible si la bibliothèque graphique ne contient pas de widgets avec le sous ensemble de primitives d'interaction intrinsèques équivalent à l'interacteur. Dans ces cas on considère les widgets qui n'implémentent pas les primitives d'interactions concernant les propriétés graphiques tout en conservant les autres primitives d'interactions. Les primitives d'interaction **Widget Move**, **Widget Rotation** et **Widget Resize** sont les seules qui peuvent être omis par l'opérateur de comparaison$\supseteq$.

1.4.1 Équivalences des interacteurs

La sélection est un processus qui permet de retrouver l'ensemble des widgets équivalents à un interacteur dans la bibliothèque graphique de la plateforme d'arrivée. Ce processus se base sur des opérateurs de comparaison qui permettent d'établir les équivalences entre les interacteurs et les widgets. Cette section présente les différents opérateurs de comparaison et l'algorithme de sélection des widgets équivalents.

Les opérateurs de comparaison

La comparaison entre les interacteurs et les widgets se basent sur les caractéristiques qui leurs sont communes telles que les primitives d'interactions (effectives et intrinsèques), le type et la cardinalité de données.

La comparaison des cardinalité est effectuée pour vérifier si les widgets équivalents supportent le même nombre de données que l'interacteur à migrer. Elle est possible grâce à la fonction *equalCard* : $Interactor \times Widget \rightarrow Boolean$. $\forall i \in \{Interactor\} \wedge \forall w \in \{Widget\}$,

$$equalCard\,(i,w) = \begin{cases} True,\ i.content.Cardinality = w.Cardinality \\ False,\ else \end{cases}$$

La prise en compte de la caractéristique types de données pour comparer les interacteurs et les widgets à pout but de vérifier si l'adaptation des widgets équivalents nécessite l'ajout d'un adaptateur de types de données. En effet si la cardinalité et les primitives d'interaction d'un *Widget* correspondent à ceux d'un interacteur alors ce Widget sera proposé au concepteur comme équivalent même si son utilisation nécessite un travail supplémentaire (écriture du code pour adaptateur de type).

La fonction $equalDataType : Interactor \times Widget \rightarrow Boolean$ permet vérifier l'égalité des types de données des interacteurs et des widgets. $\forall i \in \{Interactor\}, \forall w \in \{Widget\}$,

$$equalDataType(i,w) = \begin{cases} True, \left(\begin{array}{c} \exists att \in w.attibuts, \\ i.Content.dataType = att.dataType \\ \vee \\ i.Content.dataType = Null \\ \wedge \\ \nexists att \in w.attibuts / att.type = Content \end{array} \right) \\ \\ False, else \end{cases}$$

Les opérateurs de comparaison combinent les trois caractéristiques en considérant d'abord la cardinalité des données, car il n'y a pas d'équivalence entre un interacteur et un Widget s'ils n'ont pas la même cardinalité. Ensuite, les opérateurs considèrent les types des données des interacteurs et enfin les sous-ensembles de primitives d'interaction. La combinaison des ces opérateurs des caractéristiques nous permet de relever quatre cas d'équivalences signifiant pour le processus de migration : le cas d'une équivalence stricte suivant les trois caractéristiques ci-dessus, le cas d'une équivalence large , le cas d'une équivalence simple et le cas d'une équivalence faible (6.2.1.4).

Équivalence stricte

Il ya une équivalence stricte entre un interacteur et un Widget s'ils ont des cardinalités égales, les même types de données et des primitives d'interaction égales. La
fonction $strongEquivalent : Interactor \times Widget \rightarrow Boolean$ permet de vérifier cette équivalence, $\forall\, i \in \{Interactor\}, \forall w \in \{Widget\}$,

$$strongEquivalent(i,w) = \begin{cases} True, & \begin{pmatrix} equalCard(i,w) \\ \wedge \\ equalDataType(i,w) \\ \wedge \\ i = w \end{pmatrix} \\ False, else \end{cases}$$

Equivalence large

Il y a une équivalence large entre un interacteur et un Widget s'ils ont des cardinalités égales, les mêmes types de données et si les primitives d'interaction de l'interacteur sont incluses dans celles du Widget. La fonction $largeEquivalent : Interactor\ X\ Widget \rightarrow Boolean$ permet de vérifier cette équivalence.$\forall\, i \in \{Interactor\}, \forall w \in \{Widget\}$,

$$largeEquivalent(i,w) = \begin{cases} True, & \begin{pmatrix} equalCard(i,w) \\ \wedge \\ equalDataType(i,w) \\ \wedge \\ i \subseteq w \end{pmatrix} \\ False, else \end{cases}$$

Équivalence simple

Il y a une équivalence simple entre un interacteur et un Widget s'ils ont des cardinalités égales, les types de données différentes et si les primitives d'interaction de l'interacteur sont égales ou incluses dans celles du Widget. La fonction $simpleEquivalent : Interactor\ X\ Widget \rightarrow Boolean$ permet de

vérifier cette équivalence. $\forall\, i \in \{Interactor\}, \forall w \in \{Widget\}$,

$$simpleEquivalent(i,w) = \begin{cases} True, & \left(\begin{array}{c} equalCard\,(i,w) \\ \wedge \\ !equalDataType\,(i,w) \\ \wedge \\ (i \subseteq w \vee i = w) \end{array} \right) \\ False, else \end{cases}$$

Équivalence faible

La fonction $lowEquivalent : Interactor\ X\ Widget \rightarrow Boolean$ permet de vérifier cette équivalence. $\forall\, i \in \{Interactor\}, \forall w \in \{Widget\}$,

$$lowEquivalent(i,w) = \begin{cases} True, & \left(\begin{array}{c} equalCard(i,w) \\ \wedge \\ !equalDataType(i,w) \\ \wedge \\ (i \supseteq w) \end{array} \right) \\ False \end{cases}$$

1.4.2 Correspondances entre interacteurs et composants graphiques

1.5 Synthèse

[TODO] Les primitives d'interaction présentées dans ce chapitre permettent la représentation des interactions intrinsèques aux composants graphiques et effectives aux interacteurs. Cette représentation offre des éléments de comparaison entre interacteurs de l'UI à migrer et les composants graphiques de la plateforme d'arrivée. Les algorithmes d'équivalence utilisés par le processus de migration s'appuient sur les primitives d'équivalence pour la sélection des widgets correspondants aux interacteurs à migrer. Les interacteurs permettent de décrire les instances d'UI et leurs interactions effectives indépendamment des bibliothèques graphiques tout en conservant leurs ressources et les liens entre eux. La structure analysable *UIStructure* fournie en entrée est représentée de manière abstraite par un ensemble d'arbre d'interacteurs.

Cette structure abstraite est utilisée par le processus de migration pour établir les équivalences avec les widgets de la plateforme cible et pour identifier et transformer les groupes d'interacteurs en fonction des guidelines. Par ailleurs cette modélisation des interactions et de la structure des éléments d'une UI ne prend pas en compte toutes les préoccupations liées à la conception des UI adaptables aux contextes d'usage telles que définies par CRF. En effet, les modélisations pour l'adaptation des styles de présentation de l'UI [réf style], du layout [réf layout] des composants graphiques ou des comportements de l'UI de façon globale [réf tâche] ne sont pas prises en compte dans la modélisation proposée dans ce chapitre. Le processus de migration que nous proposons permet aussi d'assister les développeurs pendant le choix des styles et présentation ou du layout suivant les guidelines de la plateforme d'arrivée.

Bibliography

[BCW+06] Doug A Bowman, Jian Chen, Chadwick A Wingrave, John Lucas, Andrew Ray, Nicholas F Polys, Qing Li, Yonca Haciahmetoglu, Seonho Kim, Robert Boehringer, and Tao Ni. New Directions in 3D User Interfaces. *The International Journal of Virtual Reality*, (0106):1–30, 2006.

[CCB+02] Gaëlle Calvary, Joëlle Coutaz, Laurent Bouillon, Murielle Florins, Quentin Limbourg, L. Marucci, Fabio Paternò, Carmen Santoro, N. Souchon, David Thevenin, and Jean Vanderdonckt. CAMELEON Project. Technical report, 2002.

[Cre01] Murray Crease. *A Toolkit Of Resource-Sensitive, Multimodal Widgets Murray Crease*. PhD thesis, 2001.

[D. 06] D. Heinemeier Hansson. World of Resource, 2006.

[GH95] Hans-w Gellersen and Hans-W. Gellersen. Modality Abstraction : Capturing Logical Interaction Design as Abstraction from "User Interfaces for All". In *1st ERCIM Workshop on "User Inter- faces for All". ERCIM*, 1995.

[KSM99] Lanyan Kong, Eleni Stroulia, and Bruce Matichuk. Legacy Interface Migration : A Task âˆ’ Centered Approach. 1999.

[Lon10] Nguyen Hoang Long. *Web Visualization of Trajectory Data using Web Open Source Visualization Libraries Web Visualization of Trajectory Data using Web Open Source Visualization Library*. PhD thesis, INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION ENSCHEDE, THE NETHERLANDS, 2010.

[Mic12] Microsoft WPF. WPF, 2012.

[NC94] Laurence Nigay and Joëlle Coutaz. *Conception et modélisation logicielles des systèmes interactifs: application aux interfaces multimodales = Software design and implementation of interactive systems: a case study of multimodal interfaces*. PhD thesis, 1994.

[Nig94] Laurence Nigay. *Conception et modélisation logicielles des systèmes interactifs : application aux interfaces multimodales*. PhD thesis, UNIVERSITÆ JOSEPH FOURIER - GRENOBLE 1, 1994.

[PHR02] Jenny Preece, Sharp Helen, and Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction*. Chichester edition, 2002.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. *ACM Transactions on Computer-Human Interaction*, 16(4):1–30, November 2009.

[SC12] Carlos Eduardo Silva and José Creissac Campos. Can GUI Implementation Markup Languages Be Used for Modelling ? In *Human-Centered Software Engineering*, pages 112–129, 2012.

[SWND03] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide : the official guide to learning openGL, version 1.4*. 4th ed edition, 2003.

[VLM+04] Jean Vanderdonckt, Quentin Limbourg, Benjamin Michotte, Laurent Bouillon, Daniela Trevisan, and Murielle Florins. USIXML : a User Interface Description Language for Specifying Multimodal User Interfaces The Reference Framework used for Multi-Directional UI Development. *Language*, pages 19–20, 2004.

[W3C03] W3C. W3C Multimodal Interaction Framework, 2003.

[W3C10] W3C Incubator Group. Model-Based UI XG Final Report, 2010.

[Weg97] Peter Wegner. Why Interaction Is More Powerful Than Algorithms, 1997.