Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données

Arnaud Blouin

université angers

Année : 2009
N° d'ordre : 1025

Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données

Thèse de doctorat

Spécialité : Informatique

École Doctorale STIM

Présentée et soutenue publiquement

Le 30 novembre 2009

À Angers

Par **Arnaud BLOUIN**

Devant le jury ci-dessous :

Rapporteurs :	Gaëlle Calvary,	Professeur, Grenoble INP
	Jean-Marc Jézéquel,	Professeur, IRISA
Examinateurs :	Jin-Kao Hao,	Professeur, Université d'Angers
	Emmanuel Pietriga,	Chargé de recherche, INRIA Saclay
Directeur de thèse :	Stéphane Loiseau,	Professeur, Université d'Angers
Co-encadrant :	Olivier Beaudoux,	Enseignant-chercheur, Groupe ESEO

Laboratoire d'Étude et de Recherche en Informatique d'Angers
2, Bd Lavoisier, 49045 Angers Cedex 01

Remerciements

Je tiens, tout d'abord, à remercier mon co-encadrant, Olivier Beaudoux, pour m'avoir proposé ce sujet de thèse et encadré pendant ces trois années. Les travaux réalisés n'auraient pas été les mêmes sans sa rigueur scientifique et sa présence. Je remercie également mon directeur de thèse, Stéphane Loiseau, pour ses conseils et ses points de vue.

Je remercie tout particulièrement Gaëlle Calvary et Jean-Marc Jézéquel pour m'avoir fait l'honneur d'évaluer ma thèse et pour leurs remarques pertinentes.

Merci à Jin-Kao Hao et à Emmanuel Pietriga d'avoir accepté de faire partie du jury de cette thèse.

Merci au Groupe ESEO pour m'avoir accueilli et permis de réaliser des enseignements.

Ces trois années n'auraient pas été les mêmes sans les autres doctorants, à savoir Sébastien, Matthias, Vincent et Matthieu, les membres du GRI et ceux de l'équipe TRAME. En un mot : merci !

Un grand merci à mes amis pour les vacances, les week-ends et les soirées passés ensemble.

Pour terminer, un merci tout particulier à Mathilde pour sa présence et sa patience lorsque je n'ai ni l'une ni l'autre de ces qualités.

Table des matières

Partie II Manipulation de données avec Malan

Introduction

Les données auxquelles un utilisateur peut accéder requièrent l'utilisation de systèmes interactifs pour être manipulées. Nous appelons *système interactif* (SI) un système informatique qui nécessite des interactions de la part de l'utilisateur pour réaliser des actions. Le développement de SI ne se limite plus uniquement aux SI de bureau s'exécutant sur une unique plate-forme d'exécution. L'avènement du web a engendré l'apparition d'un nouveau type de SI : les applications Internet riches (RIA - *Rich Internet Application* [MacDonald, 2007]). Alors que les pages web restaient jusqu'alors peu interactives, les RIA visent à fournir les mêmes capacités en terme d'interaction que les SI de bureau afin de manipuler et visualiser les données du web. Le web est maintenant un nouvel espace d'exécution accessible depuis une pléthore d'appareils de différentes tailles et pourvus de divers périphériques d'entrée (HID - *Human Interface Device*). Les écrans tactiles des téléphones portables et les gyroscopes de l'IPhone et de la Wii font partie de cette nouvelle génération de HID. La conception des SI a dû ainsi s'adapter pour prendre en compte ces nouveaux HID. Pour cela, la recherche en ingénierie des interactions homme-machine a fourni bon nombre de modèles conceptuels et de techniques de description d'interactions visant à faciliter la conception de SI requérant des HID modernes ou non fondés sur des interfaces classiques (WIMP - W*indow, Icon, Menu, Pointing device*) ; on parle alors d'interfaces post-WIMP.

Motivations et objectifs

L'ingénierie du logiciel s'intéresse à trois aspects majeurs du développement de SI : *la liaison entre les données sources et leurs présentations cibles* ; *la conception de la facette interactive des SI* ; *l'exécution d'un SI sur différentes plates-formes d'exécution* aux caractéristiques différentes, sans revoir entièrement son développement. Les différentes évolutions du web, des données et des appareils sur lesquels s'exécutent les SI, ainsi que les nouvelles techniques d'interaction modernes, amènent à revoir la manière de traiter ces trois aspects.

Le premier aspect à aborder concerne l'établissement de *liaisons* entre des données. Cet aspect n'est pas nouveau : le domaine des bases de données l'utilise pour créer un lien entre différentes bases, afin de fusionner différentes sources de données ou de créer des présentations à partir d'un ensemble de données [Ullman, 1988; Ullman, 1989]. C'est sur cette dernière utilisation que se fonde la séparation entre les données sources d'un SI et leurs présentations cibles. Ainsi, les plates-formes RIA actuelles permettent la définition de liens entre les composants d'une interface homme-machine (IHM) et des données sources ; ce principe est appelé « *data binding* ». Les modèles de « *data binding* » sont cependant conçus pour la liaison de l'interface vers les données en ne séparant pas la définition des « *bindings* » de celle de l'interface. Les langages de transformation permettent, au contraire, de construire des présentations cibles à partir de

données sources. Ils n'autorisent cependant pas d'établir un lien *durable* entre les données et les présentations et sont, de ce fait, non adaptés aux SI.

Notre premier objectif est donc de proposer un langage de correspondance, indépendant de toute plate-forme de données et d'IHM, dédié au lien entre les données et leurs présentations, et assurant la validité des données manipulées.

Le deuxième aspect à traiter concerne la facette *interactive* des SI. De nombreux travaux de recherche ont apporté des techniques et des modèles pour faciliter ou améliorer la description et la gestion de l'interaction. Par exemple, la modélisation d'une interaction telle que le « glisser-déposer » peut s'effectuer par le biais d'une machine à états, remplaçant ainsi l'utilisation des « callbacks » sujette à erreurs [Appert et Beaudouin-Lafon, 2008]. De même, l'interaction instrumentale est un modèle conceptuel dans lequel l'instrument est une métaphore du monde réel où un individu dispose d'instruments pour manipuler des objets [Beaudouin-Lafon, 2000]. Cependant, les plates-formes de développement modernes ne prennent pas en considération ces travaux et restent centrées sur le concept de « *widget* », ignorant ainsi les notions fondamentales d'interaction, d'action et d'instrument.

Notre deuxième objectif est de proposer un modèle d'interaction fondé sur les principes essentiels de travaux de recherche en interaction, en apportant des améliorations visant à faciliter la conception de SI WIMP ou post-WIMP.

Un troisième aspect à considérer est l'indépendance de la plate-forme d'exécution. Exécuter un même SI sur différentes plates-formes d'exécution disposant de caractéristiques variées, comme la taille de l'écran ou les HID disponibles, nécessite que différentes versions de ce SI, chacune adaptée à une plate-forme d'exécution donnée, aient été développées. Pour éviter d'avoir à développer chacune de ces versions, il est possible de réaliser une modélisation s'échelonnant sur différents niveaux d'abstraction. Par exemple, une approche fondée sur l'IDM (Ingénierie Des Modèles [Bézivin, 2004]) consiste à débuter la conception par la spécification des tâches que l'utilisateur peut effectuer et des données manipulées par ces tâches. A partir de ces deux modèles, une interface abstraite (AUI - *Abstract User Interface*), indépendante de toute plate-forme et ne contenant pas d'information graphique est créée. Cette interface abstraite est elle-même utilisée pour la conception d'une interface concrète (CUI - *Concrete User Interface*), dépendante d'une plate-forme donnée, avec laquelle le code source de l'IHM est généré [Calvary *et al.*, 2003]. Ce processus permet ainsi de factoriser la conception d'un SI. Le choix des plates-formes sur lesquelles il doit s'exécuter s'effectue uniquement lors de la réalisation de l'interface concrète, ce que ne permettent pas les plates-formes de développement de SI actuelles. Cet aspect se place dans le cadre de l'adaptation au contexte d'usage d'un SI Le contexte d'usage se réfère au triplet : plate-forme d'exécution, utilisateur et environnement physique.

Notre troisième et dernier objectif est d'intégrer les deux objectifs précédents dans ce processus IDM. Les notions d'utilisateur et d'environnement ne sont toutefois pas étudiées. De même, la plasticité des interfaces, c.-à-d. la capacité d'une interface à s'adapter au contexte d'usage dans le respect de son utilisabilité [Thevenin et Coutaz, 1999], n'est pas abordée.

Contributions

Les travaux réalisés dans cette thèse proposent, au travers de deux contributions, une prise en compte originale des trois aspects majeurs du développement des SI présentés précédemment.

La première contribution consiste en un langage de correspondance, appelé Malan (*a MApping LANguage*), dédié à l'établissement de liens entre les données sources d'un SI et leurs présentations cibles. Malan est construit selon deux principes fondamentaux. D'une part, ce langage travaille au niveau des *schémas*, c.-à-d. que les liens, appelés *correspondances de schémas*, sont établis entre le schéma des données et celui de la présentation, tous deux décrits par des diagrammes de classes UML. Ce principe assure la validité des données manipulées. D'autre part, ce langage est dédié au lien « données-présentation », ce qui induit : une expressivité élevée, pour notamment calculer des paramètres graphiques et manipuler des données, conjugée à une syntaxe la plus simple possible ; une indépendance envers les plates-formes de données ; la possibilité d'interpréter directement une correspondance de schémas par un interpréteur dédié qui crée et gère un ensemble de liaisons actives entre les instances sources et cibles lors de l'exécution du SI ; la génération de transformations à partir d'une correspondance de schémas.

La seconde contribution consiste en un modèle conceptuel, appelé Malai, dédié à la conception de la facette interactive des SI en relation avec Malan. Ce modèle réunit les principes du modèle d'action de Norman [Norman et Draper, 1986], de la manipulation directe [Shneiderman, 1982], du concept d'interacteur [Myers, 1990], de l'interaction instrumentale [Beaudouin-Lafon, 2000] et du modèle DPI [Beaudoux et Beaudouin-Lafon, 2001]. Malai vise à unifier ces modèles de manière à en retenir les avantages tout en apportant les améliorations suivantes :

1. considérer les *actions*, qu'un utilisateur effectue sur un SI, comme des objets à part entière possédant leur propre cycle de vie ;
2. modéliser les *interactions* qu'un utilisateur réalise à l'aide de HID sous la forme de machines à états et non sous la forme d'écouteurs ;
3. utiliser la notion d'*instrument*, métaphore de l'outil du monde réel avec lequel un individu interagit pour produire des actions, comme médiateur entre l'utilisateur et les données du SI ;
4. définir le *feed-back intérimaire* de l'instrument permettant à l'utilisateur d'être tenu au courant de l'état du système et des différentes actions qu'il peut accomplir.

Malai divise un SI en plusieurs éléments fondamentaux : Les *données* à manipuler, l'*interface*, les *présentations* des données (abstraites et concrètes) et les *instruments* dont dispose l'utilisateur (utilisant les *évènements* produits par les HID, les *interactions* que l'utilisateur peut réaliser pour produire des actions et les *actions* que l'utilisateur peut effectuer).

Ces deux contributions suivent la démarche IDM en différents niveaux d'abstraction : modèle de tâche, interface abstraite, interface concrète et interface finale. Dans le cadre de nos travaux, une interface abstraite se compose des présentations abstraites et des actions. L'interface concrète complète l'interface abstraite en y apportant les présentations concrètes, les correspondances de schémas Malan entre les présentations abstraites et ces dernières, les interactions et les instruments. Nous n'abordons cependant pas dans cette thèse la génération du code source d'une interface finale. De même, le modèle de tâche n'est pris en compte dans nos travaux.

Organisation du manuscrit

Ce manuscrit est structuré en quatre parties. La première présente les principes généraux du développement de SI et se divise en deux chapitres. Le chapitre 1 présente les principales

plates-formes de données ainsi que différentes plates-formes d'exécution d'applications de la littérature et montre en quoi ces dernières intègrent mal : la liaison entre les données sources et leurs présentations ; la conception de la facette interactive des SI ; l'exécution d'un SI sur différentes plates-formes d'exécution aux caractéristiques différentes. Le chapitre 2 est consacré à la présentation de l'IDM et, en particulier, à celle des environnements de développement fondés sur les modèles. De tels environnements permettent de considérer le développement des SI de manière plus abstraite que les plates-formes du premier chapitre. C'est dans cette approche IDM que se basent nos travaux.

La deuxième partie se focalise sur le lien entre les données et leurs présentations cibles. Le chapitre 3 est un état de l'art des différentes approches de la manipulation de données dans le contexte des SI. A partir de l'évaluation de la capacité des approches présentées à répondre au problème de la liaison entre des données sources et leurs présentations cibles, un constat est dressé. Ce constat motive les travaux introduits dans les chapitres 4 et 5. Le chapitre 4 présente le *modèle de données* du langage Malan. Le chapitre 5 décrit de manière théorique (présentation du langage abstrait) et pratique (présentation du langage concret) le *modèle de correspondance* du langage Malan, lequel se repose sur le modèle de données.

La troisième partie porte sur la facette interactive des SI. Le chapitre 6 est un état de l'art des principaux modèles d'interaction et d'action : modèles conceptuels, modèles de description d'interaction et modèles de description d'actions. Ces différents modèles sont à la base du modèle conceptuel Malai dont les parties statique et dynamique sont respectivement présentées dans les chapitres 7 et 8.

La quatrième partie regroupe deux études de cas illustrant comment Malan et Malai sont utilisés pour la conception de deux SI aux caractéristiques différentes. Le chapitre 9 présente la conception d'un éditeur de dessins vectoriels. Cet exemple met en exergue les capacités de Malai à modéliser la description et l'utilisation d'interactions complexes au sein d'un SI. Le chapitre 10 est dédié à la conception d'un agenda standard permettant de visualiser et d'éditer des évènements d'une semaine donnée. Du fait de sa complexité, le lien entre les données et la présentation cible, établi *via* le langage Malan, est le point central de cette étude de cas.

Première partie

Environnements de développement et plates-formes d'exécution de SI

Introduction

Cette partie est consacrée à une présentation détaillée des plates-formes d'exécution d'applications et des environnements de développement qui y sont liés. Le premier chapitre porte sur les plates-formes d'exécution d'applications : les différentes plates-formes de données utilisées à l'heure actuelle sont présentées ; les principales caractéristiques des plates-formes de développement d'IHM (Interface Homme-Machine) sont discutées. Ce chapitre met en exergue la présence des trois aspects de développement des SI dans les plates-formes de développement actuelles présentés en introduction (*cf.* page 1), à savoir : la liaison entre les données sources et leurs présentations cibles *;* la conception de la facette interactive des SI *;* l'exécution d'un SI sur différentes plates-formes d'exécution. Le second chapitre présente, quant à lui, un cadre de travail, en l'occurrence l'IDM, proposant une solution à l'aspect relatif aux plates-formes d'exécution sur lequel se fondent nos travaux.

Le second chapitre introduit les principes des environnements de développement fondés sur les modèles, dont l'objectif central est de considérer la conception de SI à un niveau plus abstrait que les plates-formes d'exécution d'applications. Ce point de vue s'avère intéressant pour adapter un SI à différentes plates-formes d'exécution

Chapitre 1

Plates-formes d'exécution de systèmes interactifs

Sommaire

1.1 Introduction

La manipulation de données par l'utilisateur est une des finalités majeures des SI. Ces données sont représentées physiquement sous différentes formes mais convergent globalement vers le format XML et les bases de données relationnelles notamment depuis l'apparition du web 2.0 [Abiteboul *et al.*, 2000]. Parmi les composants formant le web 2.0, les applications Internet (RIA - *Rich Internet Application*), dont le but est de fournir le même niveau d'interactivité que les applications standards, ouvrent une nouvelle voie concernant la conception de SI.

Ce premier chapitre se divise ainsi en deux parties. La section 1.2 est une présentation des plates-formes de données majeures. La section 1.3 porte sur les principales plates-formes de définition d'IHM.

1.2 Plates-formes de données

Cette section présente trois plates-formes de données largement utilisées : les modèles de données semi-structurées, relationnel et XML. Un même exemple, inspiré de Abiteboul *et al.* (2000), est utilisé pour les introduire. Il consiste en des données décrivant des personnes possédant un nom, un prénom, un numéro de téléphone, une adresse postale, une adresse électronique ainsi que des collègues de travail. La dernière section précise pourquoi les documents textuels non XML ne sont pas pris en compte dans cette thèse.

1.2.1 Modèle de données semi-structurées

Les données semi-structurées, également désignées comme « auto-descriptives » ou « sans schéma », correspondent à des données ne disposant pas de définition séparée de leur structure ou de leur type [Abiteboul *et al.*, 2000]. Un des avantages de ce modèle est la flexibilité dans la définition des données : une valeur n'est pas nécessairement toujours de même type et des données peuvent manquer dans les structures. Un fichier bibtex peut, par exemple, être considéré comme semi-structuré puisqu'il peut être privé de certains champs. De plus, certains problèmes comme l'analyse de génomes [Hernandez et Kambhampati, 2004], nécessitent que le format utilisé puisse être adapté rapidement à tout changement, comme c'est le cas avec les données semi-structurées [Abiteboul, 1997].

Les données semi-structurées sont généralement décrites en utilisant une syntaxe simple, comme la syntaxe sous forme de paires « étiquette-valeur ». Dans l'exemple ci-dessous, décrivant l'identité d'une personne, le symbole « : » sépare l'étiquette de sa valeur :

```
1 { personne : {nom : {nomFamille : "Blouin", prénom : "Arnaud"},
2                téléphone : 0241866745,
3                email : "arnaud.blouin@eseo.fr"},
4   personne : {nom : "Olivier", email : "olivier.beaudoux@eseo.fr"} }
```

Ce type de représentation de données se formalise également sous la forme d'un arbre où les valeurs correspondent aux feuilles et les étiquettes aux arcs de l'arbre. La figure 1.1 décrit l'exemple précédent de manière arborescente.

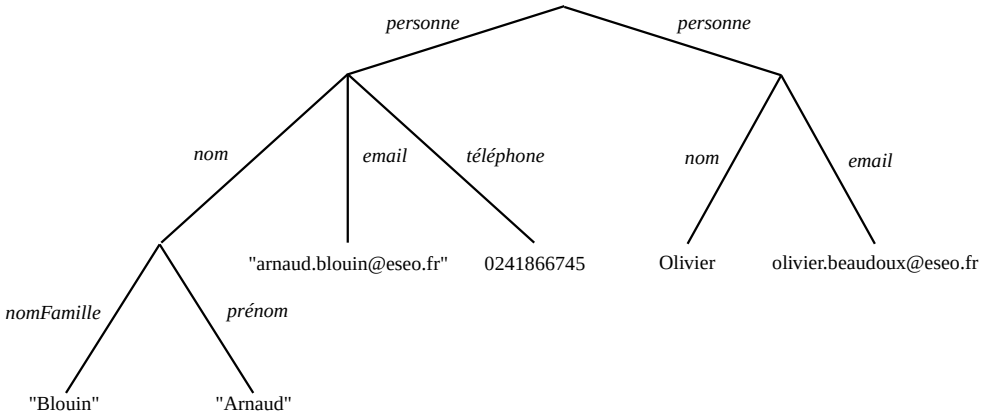


Fig. 1.1: Représentation de données semi-structurées sous la forme d'un arbre

L'absence de schéma, si elle favorise la flexibilité, ne permet pas : d'interroger ou de restructurer des données semi-structurées étant donné que leur structure n'est connue que partiellement ; de valider la cohérence des données par rapport à leur définition. Les modèles relationnel et XML, présentés dans les sections suivantes évitent de telles limitations.

1.2.2 Modèle relationnel

A l'origine, le modèle relationnel a été développé par Codd (1970) dans le but d'organiser des données bancaires ; il a ensuite permis le développement de nombreux systèmes de gestion de bases de données (SGBD). Une base de données relationnelle est décrite par un ensemble de relations formant un schéma relationnel, contrairement au modèle de données semi-structurées qui ne définit pas sa structure. Une autre différence réside dans le fait que le modèle relationnel ne s'occupe pas de la manière dont les données sont stockées, mais définit leur structure au niveau logique. La relation suivante correspond au schéma relationnel décrivant des entités de type `personne` :

```
1 personne(nomFamille, prénom, téléphone, email, identifiant)
```

Les données décrites dans la section 1.2.1 peuvent être représentées sous la forme d'une table (figure 1.2) ou de manière arborescente (semblable à la figure 1.1).

personne :

nomFamille	prénom	email	téléphone	identifiant
Blouin	Arnaud	arnaud.blouin@eseo.fr	0241866745	e1
	Olivier	olivier.beaudoux@eseo.fr		e2

Fig. 1.2: Représentation d'une base de données relationnelle sous la forme d'une table

Un point important du modèle relationnel est la gestion des entités. Par exemple, la définition de parenté entre des personnes s'exprime en utilisant les identifiants d'entités permettant ainsi à

une personne de faire référence à d'autres. Les données se représentent désormais sous la forme d'un graphe comme l'illustre la figure 1.3 dans laquelle `e1` et `e2` servent de références pour accéder aux objets auxquels elles correspondent.

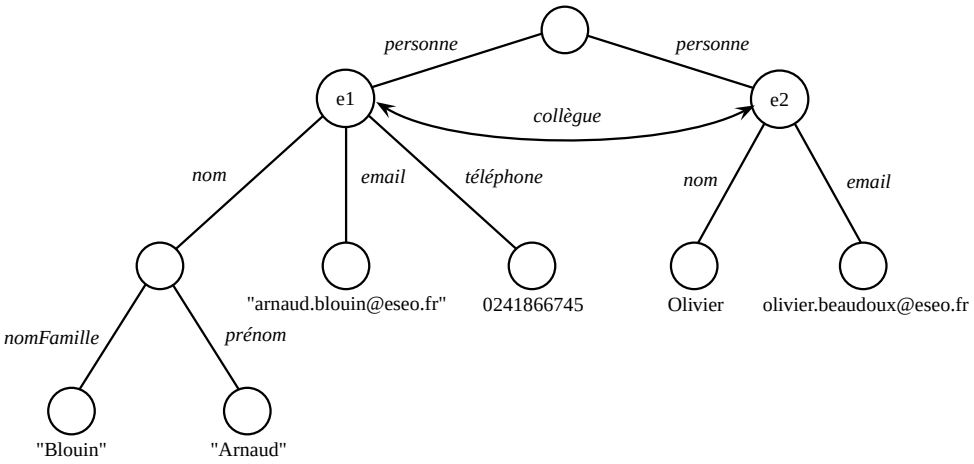


Fig. 1.3: Représentation de données relationnelles sous la forme d'un graphe

1.2.3 Modèle de documents XML

XML (*eXtensible Markup Language*) est un standard du W3C permettant la définition de langages à balises [W3C, 2006a]. Les origines de ce métalangage sont doubles. Premièrement, XML est un sous-ensemble de SGML, un langage de présentation de données et de documents structurés [ISO8879, 1986]. Ce dernier a pour principal objectif de séparer la structure d'un document de sa représentation, mais a l'inconvénient d'être complexe et rigide. Deuxièmement, HTML est un langage permettant la définition de pages web. Il est apparu en 1991 et est devenu une recommandation du W3C en 1997. Bien que populaire du fait de sa simplicité, il s'agit d'un langage limité sémantiquement, séparant mal la structure de la présentation. L'idée du langage XML est de garder la puissance de SGML et la simplicité d'HTML.

L'avantage principal d'XML est d'être un langage de référence pour la structuration de données dans des documents textuels. Ce principe favorise l'interopérabilité entre des données et, par conséquent, entre les systèmes les manipulant, ainsi que l'emploi de mêmes outils pour le traitement de documents XML (p. ex. la transformation de documents ou la validation). De plus, XML rend les données pérennes dans le temps.

XML a hérité de la structuration par balisage de SGML : une balise est un mot suivi d'un ensemble, éventuellement vide, d'attributs, encadrés par `'<'` et `'>'`. Les données d'un document XML sont soit représentées sous la forme d'attributs, soit encadrées de manière récursive par une balise ouvrante `<nom>` et une balise fermante `</nom>`, comme l'illustre la figure 1.4.

Tout document XML doit respecter un ensemble de règles pour être déclaré *« bien formé »* : il doit exister une balise englobant toutes les autres ; les balises ne se chevauchent pas ; une balise ouverte est toujours fermée plus loin. Ces règles facilitent la compréhension d'un document XML aussi bien pour un individu que pour un analyseur syntaxique.

```
<personnes>
   <personne id="p1">
      <nom>
        <prénom>Arnaud</prénom>
        <nomFamille>Blouin</nomFamille>
      </nom
      <téléphone>0241866745/<téléphone>
      <email>arnaud.blouin@eseo.fr</email>
      <collègues ref="id2" />
   </personne>
   <personne id="p2">
      <nom><prénom>Olivier</prénom></nom>
      <collègues ref="id1" />
   </personne>
</personnes>
```

Fig. 1.4: Exemple d'un document instance XML

Un document *valide* est un document bien formé conforme à une grammaire définie par ailleurs. Il existe plusieurs langages de description de grammaires XML dont les plus connus sont DTD (*Document Type Definition*) [W3C, 1998], XML Schema [W3C, 2007a] et Relax NG [OASIS, 2001], tous les trois comparés dans [Lee et Chu, 2000].

1.2.4 Document texte

Les données textuelles non XML existent dans différents métiers, tels que les langages de programmation. Cependant, nous n'abordons pas dans cette thèse ce type de données sachant qu'il s'agit d'un domaine très particulier, traité par les grammaires en ce qui concerne les langages de programmation. Il est toutefois possible de modéliser le langage abstrait d'un langage de programmation pour être utilisé dans notre approche.

1.3 Plates-formes d'IHM

Les plates-formes de développement d'IHM consistent en des langages et des librairies dédiés au codage de SI permettant, entre autres, la manipulation de données. Initialement spécialisées pour la définition d'applications de bureau standards, des plates-formes dédiées au développement d'applications Internet riches (RIA - *Rich Internet Application*) sont récemment apparues. L'ensemble de ces plates-formes offrent des fonctionnalités intéressantes pour notre problématique dont, notamment, le lien entre les données et les présentations (appelé « *data binding* ») et l'utilisation d'un langage pour décrire les IHM.

Après avoir fourni une vue d'ensemble des plates-formes considérées dans la section 1.3.1, la section 1.3.2 et les suivantes comparent ces plates-formes d'IHM en fonction de quatre critères majeurs :

1. le langage utilisé pour définir l'interface ;
2. les spécificités de la liaison entre les données et la présentation ;
3. le modèle de définition d'actions, dont le but est de modifier les données sources ou des paramètres de l'interface ;

4. le modèle de description d'interactions que l'utilisateur effectue à l'aide de périphériques d'entrée pour réaliser des actions.

Ces quatre critères couvrent la définition des SI dédiés à la manipulation de données en évaluant comment les interactions sont définies, comment les actions produites de manière conséquente aux interactions sont modélisées, comment l'IHM sur laquelle l'interaction est effectuée est décrite, et comment les données sont liées à l'IHM.

1.3.1 Vue d'ensemble des plates-formes

Nous avons sélectionné pour cette présentation les deux plates-formes d'IHM les plus utilisées actuellement, à savoir Swing [Elliott *et al.*, 2002] et SWT - JFace [Scarpino *et al.*, 2004]. Swing est une boîte à outils fondée sur le modèle conceptuel MVC (*Modèle-Vue-Contrôleur*) qui fournit un ensemble de « *widgets*[1] » pour la définition d'applications Java. SWT est une bibliothèque graphique Java développée par Eclipse. JFace est une extension de SWT fondée sur le modèle conceptuel MVC.

En ce qui concerne la définition de pages web interactives, les limites du langage HTML ont conduit, au final, au développement de plates-formes d'applications Internet (RIA - *Rich Internet Application*). Celles-ci visent à donner aux sites web les mêmes capacités, en terme d'interaction et d'interface, que les applications de bureau classiques. Actuellement, les trois plates-formes RIA majeures sont Silverlight [MacDonald, 2007], JavaFX [Morris, 2009] et Flex [Kazoun et Lott, 2007].

1.3.2 Evaluation de la définition de l'interface

Alors que la plupart des plates-formes d'applications standards définissent des interfaces de manière implémentatoire en utilisant des librairies de développement spécifiques (p. ex. AWT, Swing, SWT et JFace pour le langage Java), les plates-formes d'applications Internet utilisent un langage de haut niveau pour décrire statiquement les interfaces (UIDL - *User Interface description Language*). L'atout majeur des UIDL, dans leur ensemble, est de rendre la définition des interfaces indépendantes de la plate-forme d'exécution. Les UIDL de Flex et de Silverlight ont cependant été conçus dans le cadre d'une utilisation pour une plate-forme d'IHM spécifique ce qui en réduit l'indépendance. Ce principe, à la base des environnements de développement fondés sur les modèles (*cf.* chapitre 2), n'est pas nouveau : déjà en 1991, Nakatsuyama *et al.* (1991) proposaient une plate-forme de développement séparant le code du SI de la description de l'IHM, principe à la base des travaux en architecture logicielle des SI. La figure 1.8 présente l'interface Flex correspondant au code de la figure 1.5. Cette interface affiche, dans une table, les données utilisées dans la section 1.2, décrivant des personnes.

[1] Le terme « *widget* » désigne un composant graphique comme un bouton par exemple.

```
[Bindable] public class Personne {
 public var prenom      : String;
 public var nomFamille  : String;
 public var email       : String;
}
```

(a) Programme

```
<mx:Script>
  import models.users.User;
  [Bindable] public var personnes : Array;
</mx:Script>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Canvas>
  <mx:DataGrid height="100%" width="100%"
       dataProvider="{personnes}">
   <mx:columns>
    <mx:DataGridColumn dataField="prenom" headerText="Prénom"/>
    <mx:DataGridColumn dataField="nomFamille" headerText="Nom"/>
    <mx:DataGridColumn dataField="email" headerText="Courriel"/>
   </mx:columns>
  </mx:DataGrid>
 </mx:Canvas>
</mx:Application>
```

(b) Description de l'interface

Fig. 1.5: Exemple d'une interface Flex

```
public class Personne {
  public var prenom      : String;
  public var nomFamille  : String;
  public var email       : String;
}
```

(a) Programme

```
var personnes : Personne[];
Stage {
 title:  "Personnes"
 scene: Scene {
  width  : 400
  height : 300
  content: SwingList {
  items: bind for (p in personnes)
   SwingListItem{text:"{p.prenom} {p.nomFamille} {p.email}"}
}}}
```

(b) Description de l'interface

Fig. 1.6: Exemple d'une interface JavaFX

```
public abstract class Personne:ObservableObject{
 private string prenom;
 private string nom;
 public string Prenom{
  get{ return prenom; }
  set{ prenom = value;
       ObjectChanged("Prenom"); }
 }
 public string Nom{
  get{ return nom; }
  set{ nom = value;
       ObjectChanged("Nom"); }
}}
```

(a) Programme

```
<Window x:Class="VuePersonnes"
 Title="Personnes" Height="300" Width="300">
 <Grid>
  <ListView  Name="listePersonnes"><ListView.View>
   <GridView><GridView.Columns>
    <GridViewColumn Header="Prénom"
     DisplayMemberBinding="{Binding Path=Prenom}"/>
    <GridViewColumn Header="Nom"
     DisplayMemberBinding="{Binding Path=Nom}"/>
   </GridView.Columns></GridView>
  </ListView.View></ListView>
 </Grid>
</Window>
```

(b) Description de l'interface

```
public partial class VuePersonnes: Window {
 private ObservableCollection<Personne> personnes;
 public VuePersonnes.() {
  InitializeComponent();
   personnes = new ObservableCollection<Personne>();
    listePersonnes.ItemsSource = personnes;
}}
```

(c) Définition du binding

Fig. 1.7: Exemple d'une interface Silverlight

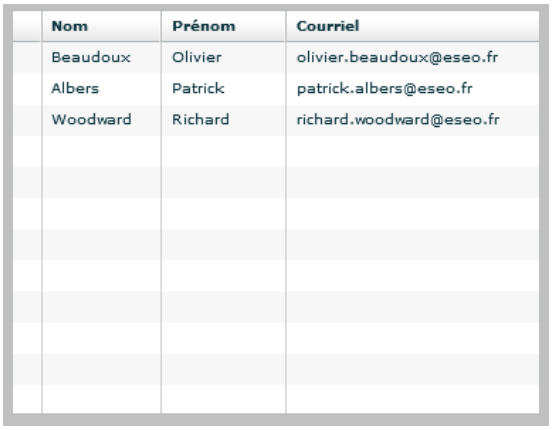


Fig. 1.8: Interface Flex

La figure 1.5a présente le programme Flex du SI. Il contient la classe `Personne` elle-même composée de trois attributs : `prénom`, `nomFamille` et `email`. La figure 1.5b décrit l'interface Flex du SI. Les lignes 1 à 4 indiquent les données utilisées dans l'interface, à savoir un tableau d'instances de `Personne`. Les lignes 5 à 16 définissent les éléments de l'interface et les données qu'ils affichent. En l'occurrence, il s'agit d'une liste dont les données sources correspondent au tableau `personnes`. Chacune des trois colonnes de la liste affiche un des trois attributs d'une personne.

JavaFX repose sur l'extension du langage JavaScript, comme l'illustre la figure 1.6 décrivant la même interface que précédemment. La figure 1.6a présente le programme JavaFX du SI, très proche de celui du Flex (*cf.* figure 1.5a). La figure 1.6b décrit l'interface JavaFX dans un formalisme non XML. La variable `personnes`, ligne 6, correspond à un tableau des instances de `Personne` à afficher. Les lignes 7 à 16 définissent une interface composée d'une liste. Cette dernière affiche les attributs des instances du tableau `personnes`. Cette liste et ces instances sont liées grâce à l'expression « *bind for* », ligne 13.

Tout comme Flex, Silverlight propose un UIDL, appelé XAML, fondé sur le langage XML. Le code XAML de la figure 1.7b décrit une interface équivalente à celle de la figure 1.5b. De même, le modèle de données de la figure 1.7a est équivalent à celui de l'interface Flex (*cf.* figure 1.5a). La description de l'interface et les données sources sont liées par le binding défini dans la figure 1.7c. Ce binding spécifie que les objets de la liste sont une liste de personnes. Pour une présentation plus exhaustive des différents UIDL existants fondés sur le langage XML, le lecteur intéressé peut se référer à [Souchon et Vanderdonckt, 2003].

1.3.3 Evaluation de la liaison dynamique entre les données et l'interface

Le mécanisme de la liaison dynamique entre les données et l'interface est présent dans les plates-formes RIA sous le nom de « *data binding* ». Il se place dans le cadre de MVC, dans lequel une vue écoute son modèle (relation *observable-observateur*). Ses buts sont d'initialiser les composants de l'interface qui affiche une donnée et de répercuter automatiquement toute modification subie par la donnée sur ces composants.

Le modèle de « data binding » que propose Flex possède, dans une majeure partie, les mêmes capacités que celui de WPF. Le code Flex suivant, tiré de la figure 1.5a, lie l'attribut `prenom` à une colonne de la table. L'attribut `dataProvider` définit la source du « binding », en l'occurrence un ensemble de personnes, alors que l'attribut `dataField` spécifie que la première colonne de la table correspond au nom de chacune des personnes.

```
<mx:DataGrid height="100%" width="100%" dataProvider="{personnes}">
  <mx:columns>
    <mx:DataGridColumn dataField="prenom" headerText="Prénom"/>
...
  </mx:columns>
</mx:DataGrid>
```

JavaFX propose un modèle de « data binding » plus puissant que ceux de Silverlight et de Flex : il est intégré dans le langage et permet du « binding » entre deux ensembles. Par exemple, le code JavaFX suivant, extrait de la figure 1.6, définit, à l'aide d'un « *bind for* », que chaque personne de l'ensemble `personnes` correspond à un élément de la liste :

```
items: bind for (p in personnes)
            SwingListItem {value:p text:"{p.prenom} {p.nomFamille} {p.email}"}}
```

Le « data binding » de Silverlight permet de créer un lien dynamique entre un ou plusieurs objets sources et un objet cible de manière unidirectionnelle (de la source vers la cible) ou bidirectionnelle (un lien est également établi de la cible vers la source pour modifier cette dernière). Si la bidirectionnalité n'est pas possible de manière automatique, un filtre peut être placé entre la source et la cible pour réaliser des traitements (p. ex. une conversion ou un calcul). La source d'un « binding » Silverlight est soit un paramètre d'un composant graphique, soit une propriété d'une classe ou d'un objet (un champ d'une base de données, un nœud ou un attribut d'un document XML, *etc.*). Le code Silverlight suivant, extrait de la figure 1.7, définit un « binding » entre la première colonne d'une liste et le prénom d'une personne de manière bidirectionnelle.

```
<TextBox Text="{Binding Path=Prenom, Mode=TwoWay}" Grid.Column="0"/>
```

JFace utilise la notion de « data binding » pour lier différents objets entre eux. Les « bindings » doivent être programmés au lieu d'être définis dans la description de l'interface. Le modèle de « data binding » de JFace ne permet pas d'opérations complexes de mise en correspondance des objets. Par exemple, il n'est pas possible de définir des « bindings » utilisant plusieurs sources. L'exemple qui suit montre comment lier un élément du document XML de la figure 1.4 à un champ « *texte* ».

```
Element peElt = (Element)doc.getElementsByTagName("personne").item(0);
Element prElt = (Element)peElt.getElementsByTagName("prénom").item(0);
Text text     = new Text(composite, SWT.BORDER);
context.bindValue(SWTObservables.observeText(text, SWT.Modify),
    DOMObservables.observeAttrValue(realm, preElt, "value"), null, null);
```

L'aspect concernant la liaison entre les données sources et les présentations cibles doit être traité indépendamment des plates-formes d'exécution. Cependant, chacun des modèles de « data binding » présentés dans cette section est associé à une plate-forme. Une de nos contributions vise à proposer un langage de correspondance indépendant de toute plate-forme d'IHM et dédié au lien entre les données sources et leurs présentations cibles.

1.3.4 Evaluation de la définition d'actions et d'interactions

Que cela concerne les plates-formes d'applications standards ou RIA, aucun modèle d'action, dans lequel une action est considérée comme un objet à part entière, n'est proposé. Swing et JFace proposent la définition d'actions s'associant à différents composants graphiques. Cependant, cette action correspond seulement à une méthode Java sur laquelle l'annotation `@action` est ajoutée. Cette modélisation limite l'utilisation d'actions en terme d'objet : il est par exemple impossible d'enregistrer l'action dans un historique pour être annulée puis rejouée.

En dépit de boîtes à outils issues de travaux de recherche, telles que SwingStates [Appert et Beaudouin-Lafon, 2008] et ICon [Dragicevic, 2004], permettant de définir des interactions, les plates-formes d'IHM sont toujours fondées sur la notion de « listener ». Les modèles sous-jacents à ces boîtes à outils sont présentés dans la section 6.3, page 100.

1.4 Conclusion

Nous avons présenté dans ce chapitre les plates-formes de données majeures, à savoir les données semi-structurées, relationnelles et XML. Nous avons ensuite introduit les principales plates-formes d'IHM dédiées au développement d'applications bureau (Java et SWT) ou pour le web (WPF, JavaFX et Flex).

Un premier constat est que les modèles XML et relationnel sont préférés à celui des données semi-structurées. L'avènement d'XML et de ses langages associés ainsi que l'utilisation de bases de données relationnelles en sont les principales causes. De plus, XML possède une souplesse d'expression presque équivalente à celle des données structurées (le schéma d'un document XML n'est pas obligatoirement défini et les données peuvent ne pas être complètes), et fournit une base commune pour structurer des données (la syntaxe XML) facilitant l'interopérabilité entre les systèmes et la création d'outils manipulant des documents XML.

Plates-formes	*Data binding*	Action	Interaction	Interface	Modèle conceptuel	Support
Java	non	pas de vrai modèle d'action	écouteur	Swing, AWT	MVC	bureau, applet web
JFace, SWT	basique, pour widget	idem	écouteur	JFace, SWT	MVC	bureau, plugin Eclipse
WPF, Silverlight	relation n-1, dépendant de l'interface	idem	écouteur	XAML (UIDL)	MVC	web, bureau
JavaFX	idem	identique à Java	écouteur	JavaFX Script (UIDL)	MVC	web, bureau, mobile
Flex	relation n-1	non	écouteur	MXML (UIDL)	MVC	web

Tab. 1.1: Caractéristiques des plates-formes IHM présentées

Concernant les plates-formes d'IHM, dont les caractéristiques générales sont résumées dans le tableau 1.1, celles dédiées aux RIA ont toujours pour limites l'absence de modèles d'action et d'interaction. Elles ont cependant apporté des améliorations comme l'utilisation du « data binding » pour lier dynamiquement des données sources à une interface, et d'un UIDL pour décrire de manière abstraite une interface. Ceci est une première étape vers la généralisation de langages abstraits pour définir les autres composants d'un SI comme les interactions, les actions, les données ou le lien entre les données et l'interface. Le chapitre suivant présente les modèles abstraits d'applications dont UML, utilisé pour représenter des données, et l'ingénierie des modèles dont le principe est de considérer le développement d'un SI comme une chaîne de modèles plus ou moins abstraits.

Chapitre 2

Environnements de développement fondés sur les modèles

Sommaire

2.1 Introduction

Développer des SI en partant d'un niveau plus abstrait que celui proposé par les boîtes à outils de développement n'est pas nouveau ; en 1985, le système COUSIN proposait déjà d'abstraire la définition d'une interface pour en générer ensuite le code [Hayes *et al.*, 1985]. De cette idée est apparue une pléthore de systèmes dédiés à la spécification plus ou moins abstraite de SI et à la génération plus ou moins automatique de leur code. Ces systèmes, appelés MB-UIDE (*Model-Based User Interface Development Environnement*), utilisaient déjà le terme de « *modèle* » et les notions « d'indépendance de la plate-forme » et de « transformation », montrant ainsi l'intérêt que porte l'IHM à l'IDM. Le langage de modélisation UML [OMG, 2007; Rumbaugh *et al.*, 2004b], standard de l'OMG[1], est tout d'abord utilisé pour la génération de code dans un langage orienté objet comme Java ou C++. Il est ensuite utilisé par de nouveaux MB-UIDE étendant UML à la modélisation d'IHM. Grâce à l'apparition du standard MDA en 2000 [OMG, 2001], variante de l'IDM [Bézivin, 2005], l'IHM bénéficie d'un standard sur lesquels les MB-UIDE peuvent se reposer pour répondre aux problèmes liés à l'évolution des dispositifs électroniques (PDA, téléphone portable, *etc.*) et des nouvelles méthodes pour interagir avec ces derniers (réalité mixte, interaction multimodale, *etc.*).

Les deux principaux avantages à modéliser de manière abstraite un SI concernent la réutilisation et l'enrichissement incrémental d'un modèle : un développeur peut ainsi partir d'une représentation abstraite d'un SI pour ensuite passer à des niveaux moins abstraits contenant de plus en plus d'informations sur la plate-forme cible.

Ce chapitre introduit tout d'abord l'ingénierie des modèles. Différents MB-UIDE sont ensuite présentés en fonction de leurs principales caractéristiques.

2.2 L'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM, ou MDE en anglais pour *« Model Driven Engineering »*) est un nouveau paradigme concernant l'ingénierie des logiciels, des systèmes et des données. Son idée centrale, résumée par le terme « tout est modèle », inspiré de l'expression « tout est objet » utilisée lors du développement de la programmation objet, est de considérer le cycle de développement d'un logiciel comme une chaîne de transformations de modèles. Les sections suivantes présentent les principes généraux de l'IDM, et un standard, appelé MDA, respectant ces principes.

2.2.1 Principes généraux

Fig. 2.1: Relations de base en IDM

[1] *Object Management Group*, http ://www.omg.org/

Les éléments de base de l'IDM sont le système, le modèle et le métamodèle (*cf.* figure 2.1). Le modèle est l'élément central de l'IDM, autour duquel se raccordent un ensemble de concepts formant l'ingénierie des modèles. Cependant, il n'existe pas de définition universelle de la notion de modèle bien qu'une majorité de définitions utilise les mêmes termes. Nous définissons un modèle de la manière suivante :

Définition : Modèle. *Un modèle est une représentation (ou abstraction) d'un système, décrit dans une intention particulière [Minsky, 1968].*

Pour qu'un modèle puisse être manipulé par une machine, il doit être modélisé dans un langage clairement défini ; il s'agit du métamodèle :

Définition : Métamodèle. *Un métamodèle est un modèle définissant le langage d'expression d'un modèle [OMG, 2006]. Un modèle et son métamodèle sont liés par la relation « conforme à ».*

Un métamodèle représente les concepts et relations utilisés par des modèles. Par exemple, la relation entre un modèle et son métamodèle peut être comparée à la relation entre un programme et le langage de programmation avec lequel il a été implémenté. C'est sur ces principes de base que s'appuie l'OMG pour définir son standard MDA.

2.2.2 MDA

MDA (*Model Driven Architecture*) est un standard industriel de l'OMG fondé sur l'IDM et sur un ensemble de standards de l'OMG dont UML [OMG, 2001]. MDA a apporté à l'IDM un ensemble de concepts clés. Après avoir défini un métamodèle comme étant un langage de description de modèles, un certain nombre de métamodèles ont été développés pour différents domaines. Pour éviter de voir se développer une pléthore de métamodèles non-interopérables, le méta-métamodèle MOF (*Meta-Object Facility*) fut développé sous la forme d'un formalisme générique pour la définition de métamodèles [OMG, 2006] :

Définition : Méta-métamodèle. *Un méta-métamodèle est un modèle décrivant un langage de métamodélisation. Un méta-métamodèle doit être réflexif pour limiter le nombre de niveaux d'abstraction.*

Ainsi, la figure 2.1 devient la figure 2.2 dans le cadre de MDA. Cette organisation à quatre niveaux de l'OMG est généralement appelée « organisation 3+1 » du faite de la différenciation entre le monde réel et le monde des modèles. En bas de l'organisation, le niveau M_0 correspond au système représenté par un modèle au niveau M_1. Un modèle est conforme à son métamodèle, défini au niveau M_2, lui même conforme à son méta-métamodèle au niveau M_3. Dans le contexte de MDA, le méta-métamodèle est le langage MOF, un métamodèle est le langage UML, et un modèle un modèle UML.

Un autre des principes clés de MDA est la séparation entre un modèle abstrait décrivant l'analyse et la conception d'un système (PIM, *Platform Independant Model*), et un modèle concernant le portage vers une plate-forme spécifique (PSM, *Platform Specific Model*) :

Définition : PIM. *Un PIM (Platform Independant Model) est un modèle métier neutre, indépendant de toute plate-forme d'exécution.*

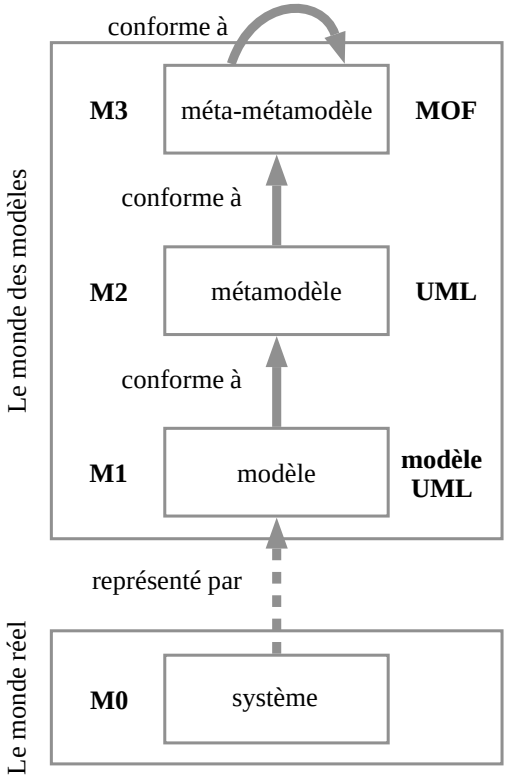


Fig. 2.2: Organisation 3+1 de l'OMG, inspiré de [Bézivin, 2005]

Un exemple de PIM est un modèle décrivant la structure de données sans spécifier la plate-forme de représentation.

Définition : PSM. *Un PSM (Platform Specific Model) est un modèle lié à une plate-forme. Il doit généralement être généré à partir d'un PIM pour obtenir un gain de productivité et pour générer plusieurs PSM à partir d'un même PIM.*

Le passage d'un PIM à un PSM fait intervenir un processus de transformation de modèles (voir la section 2.2.4) prenant en entrée un PIM et un modèle de description de plates-formes (PDM, *Platform Description Model*), comme l'illustre la figure 2.3.

Ce processus met en avant l'importance de la manipulation de modèles (transformation et correspondance de modèles) comme le détaille la section 2.2.4.

2.2.3 Espace technique et langage dédié

Le principe de hiérarchisation illustré par la figure 2.2 n'est pas nouveau. Dans le domaine XML, par exemple, un document XML est un modèle, un schéma XML un métamodèle et XML le méta-métamodèle. Chaque décomposition M1/M2/M3 pour un domaine donné forme un *espace technique* [Kurtev *et al.*, 2002] :

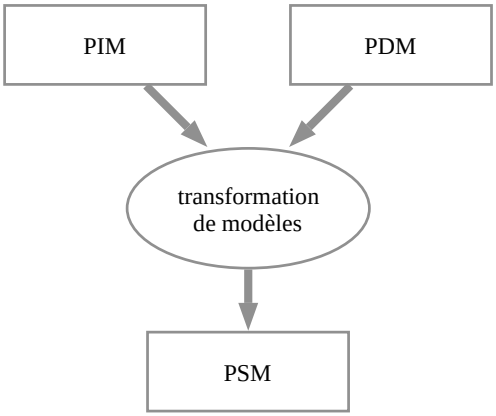


Fig. 2.3: Processus en Y du passage d'un PIM vers un PSM

Définition : Espace technique. *Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de métamodèles dont le sommet est occupé par une famille de (méta-) métamodèles similaires [Favre* et al.*, 2006].*

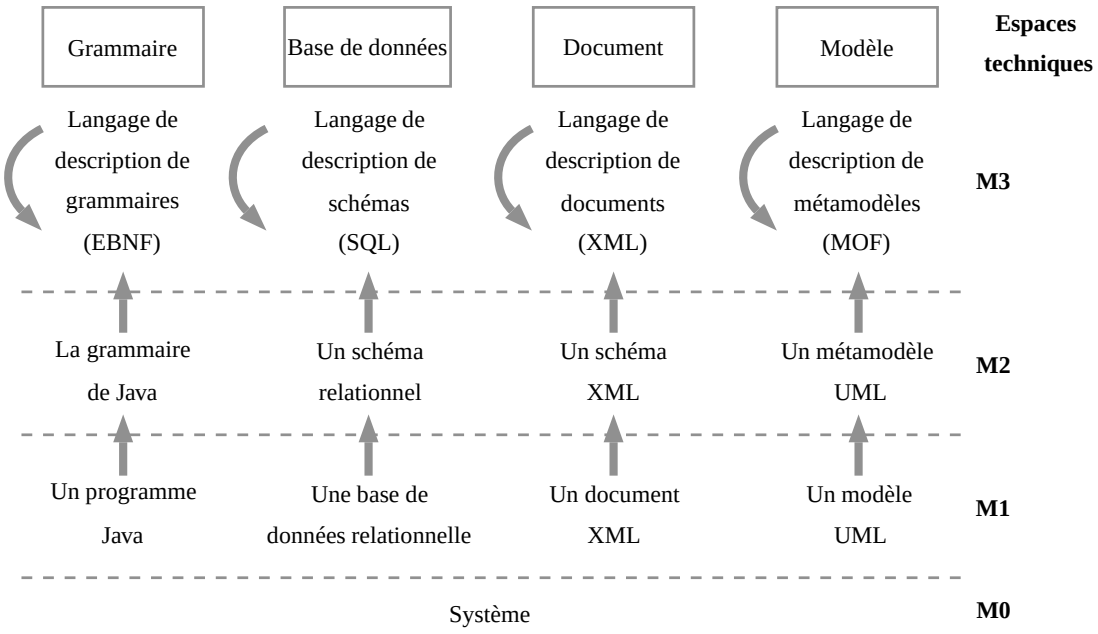


Fig. 2.4: Exemples d'espaces techniques

La figure 2.4 présente différents espaces techniques correspondant aux trois plates-formes de données présentées dans la section 1.2, page 9 (les espaces techniques des grammaires, des bases de données et d'XML) et à l'espace technique MDA.

La notion d'espace technique permet de choisir et d'utiliser un ensemble d'outils et de techniques adaptés à des besoins bien définis. Par exemple, pour modéliser des documents XML, il

est préférable d'utiliser l'espace technique XML plutôt que celui de MDA car ce dernier n'est pas dédié à ce type de problème. Le passage d'un espace technique à un autre peut s'effectuer par des transformations de modèles dont le principe est détaillé dans la section suivante. La notion de langage dédié est liée à celle de l'espace technique dans le sens où ce dernier contient des outils généralement dédiés à un domaine précis. Le langage ATL (section 3.5) est, par exemple, un langage dédié de l'espace technique MDA consacré à la transformation de modèles.

Définition : Langage dédié. *Un langage dédié (DSL - Domain Specific Language) est un langage de programmation ou de spécification exécutable qui offre, à l'aide de notations et d'abstractions appropriées, une puissance d'expression concentrée sur, et généralement limitée à, un domaine de problèmes particulier [van Deursen* et al.*, 2000].*

2.2.4 Transformation de modèles

La transformation de modèles est un outil primordial pour l'IDM dont le principe de base, illustré par la figure 2.5, est le passage d'un modèle source à un modèle cible *via* une transformation. Le modèle cible est conforme à un métamodèle cible et est obtenu par l'application de la transformation « source vers cible » appliquée sur le modèle source.

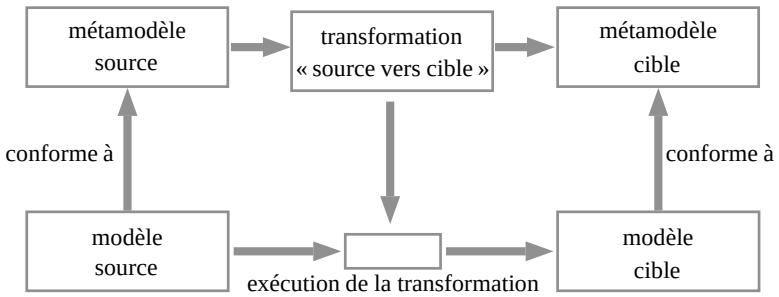


Fig. 2.5: Principe de la transformation de modèles

Il existe différentes approches de transformation de modèles parmis lesquelles : les langages de programmation ; les langages de transformation de modèles ; les outils de métamodélisation.

Les langages de programmation, tels que Java, peuvent être utilisés dans le cadre de la transformation de modèles par le biais de librairies de développement. L'inconvénient de cette approche est d'utiliser un langage non pensé pour la transformation de modèles. Son avantage est de réutiliser un langage connu, évitant ainsi la conception et l'apprentissage d'un nouveau langage.

Les langages de transformation de modèles, tels que QVT, ont pour but l'écriture de modèles de transformation. QVT (*Query View Transformation*) est une recommandation de l'OMG définissant un langage capable d'exprimer des requêtes, des vues et des transformations sur des modèles [OMG, 2005a]. QVT est un langage à la fois déclaratif et impératif composé de trois sous-langages appelés *Relations*, *Core*, *Operationnal Mappings*.

Les outils de métamodélisation, tels que Kermeta [Muller *et al.*, 2005], permettent de définir des métamodèles exécutables, c.-à-d. qu'ils peuvent spécifier à la fois la structure et le comportement des modèles. Selon Fleurey (2006), « *le langage Kermeta a été défini dans le but de prendre*

en compte les besoins liés à la validation et la vérification de méta-modèles et de transformations de modèles ».

Czarnecki et Helsen (2006) recensent et classifient de manière exhaustive les différentes approches de transformation de modèles en se fondant sur leurs fonctionnalités et leur type.

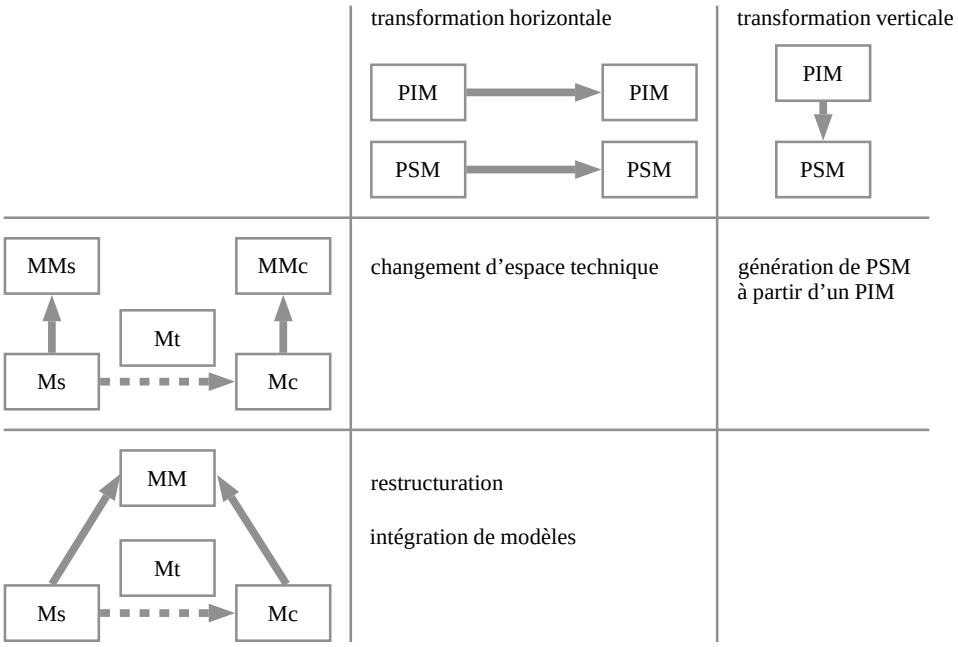


Fig. 2.6: Types de transformations et leurs principales utilisations, inspiré de [Combemale, 2008]

La transformation de modèles répond à différents problèmes résumés dans la figure 2.6. Dans le contexte de MDA, une transformation de modèles est une opération indispensable pour la réalisation et la gestion de PIM et de PSM. Elle peut, par conséquent, être réalisée de manière *horizontale* ou *verticale* :

- Le passage d'un PIM à un autre PIM, ou d'un PSM à un autre PSM est considéré comme une transformation horizontale. Si les métamodèles source *MMs* et cible *MMc* sont différents, ce processus permet le changement d'espace technique (p. ex. d'XML vers une base de données relationnelle). Dans le cas contraire, ce processus correspond à une restructuration de modèles ou à l'intégration d'un modèle dans un autre.
- Le passage d'un PIM à un PSM ou inversement correspond à une transformation verticale. La génération de code à partir d'un diagramme de classes UML et la rétro-ingénierie en sont deux exemples.

Le passage d'un PIM à un PSM peut se traduire par une transformation de type « modèle-vers-code ». Ce type de transformation peut être considéré comme un cas spécial de la transformation « modèle-vers-modèle » dans lequel le métamodèle d'un langage cible est fourni. Cependant, cela implique un changement de l'espace technique du code vers celui de l'IDM. Par conséquent, le code est souvent directement généré sous forme de texte à destination d'un compilateur dédié. Czarnecki et Helsen (2003) présentent en détails les différentes techniques de transformation « modèle-vers-code » et « modèle-vers-modèle ».

La section suivante présente l'utilisation de l'IDM et de la transformation de modèles dans le cadre de l'IHM.

2.3 Environnements fondés sur les modèles

Cette section présente différents MB-UIDE regroupés de manière chronologique : ceux développés avant et après la première version du standard MDA. Ce choix se justifie par le fait que, même si les concepts de l'IDM existaient avant le MDA, ce dernier les a formalisés pour fournir une base commune sur laquelle un grand nombre de systèmes et d'outils « post-MDA » se s'appuient.

2.3.1 MB-UIDE pré-MDA

Bien que développés avant l'apparition du standard MDA, nous considérons les MB-UIDE présentés dans cette section comme respectant les principes de l'IDM : chacun d'entre eux peut être considéré comme un espace technique regroupant différentes technologies dédiées à la modélisation et à la création de SI, au même titre que MDA est un espace technique fondé sur les langages MOF et UML.

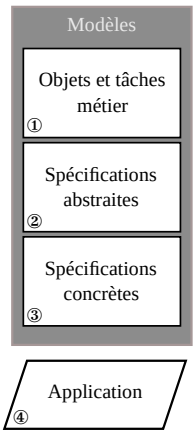


Fig. 2.7: Développement d'un SI en différentes étapes, adaptée de [Szekely, 1996]

L'évolution des MB-UIDE, dont les principaux sont amplement détaillés par Szekely (1996) et Silva (2000), s'est effectuée en plusieurs étapes successives. Les premiers environnements, tels que COUSIN [Hayes *et al.*, 1985], UIDE [Foley *et al.*, 1988], Mickey [Olsen, 1989], Jade [Zanden et Myers, 1990] et Humanoid [Szekely *et al.*, 1992], utilisaient un langage de description d'IHM pour en générer le code. Afin d'améliorer la qualité des IHM générées, de nouveaux MB-UIDE utilisant différents modèles complémentaires aux modèles de description d'IHM sont apparus. Parmi ces modèles, on retrouve tout d'abord le modèle de tâches, décrivant les différentes tâches qu'un utilisateur peut réaliser sur les SI. Les environnements comme DIANE+ [Tarby, 1993], TADEUS [Elwert et Schlungbaum, 1995], EXPOSE [Gorny, 1995], AME [Märtin, 1996] et JANUS [Balzert *et al.*, 1996] sont les principaux exemples mettant en œuvre les modèles de tâches. Les modèles de données (souvent appelés modèles du domaine), de présentation et d'utilisateurs sont également

utilisés en plus du modèle de tâches, respectivement, pour représenter les données manipulées par les SI, définir la structure abstraite de l'IHM et spécifier les caractéristiques des utilisateurs. TRIDENT [Vanderdonckt et Bodart, 1993], FUSE [Lonczewski et Schreiber, 1996], ADEPT [Wilson *et al.*, 1993], MECANO [Puerta, 1996], MOBI-D [Puerta, 1997] et TEALLACH [Griffiths *et al.*, 1998] sont des exemples de tels MB-UIDE.

Un concept important qu'ont apporté certains de ces environnements est la notion d'interfaces abstraite et concrète. La figure 2.7 résume l'architecture de ces MB-UIDE dans laquelle le développement d'un SI se divise en quatre principales étapes : la définition des modèles de tâches et des données (étape ①) ; la spécification de l'interface abstraite (étape ②) ; celle de l'interface concrète (étape ③) ; la génération du code (étape ④). Les concepts définis par MDA (voir section 2.2.2), notamment l'indépendance de la plate-forme et la représentation d'un système sous la forme de modèles, sont déjà présents dans ces MB-UIDE.

Les MB-UIDE présentés précédemment se fondent cependant tous sur un formalisme différent pour représenter ces modèles. Les MB-UIDE fondés sur UML ont, quant à eux, l'avantage d'utiliser un langage de modélisation largement adopté et donnant la possibilité, au travers de ses différents diagrammes, de décrire une partie des SI. Parmi ces MB-UIDE, UMLi [Silva et Paton, 2000] et l'extension d'UML pour les interfaces WIMP[2] [Almendros-Jiménez et Iribarne, 2008] en sont les deux principaux exemples. Ils utilisent et étendent certains diagrammes UML, comme les diagrammes de classes, d'activité et de cas d'utilisation, pour décrire différentes parties d'un SI.

2.3.2 MB-UIDE post-MDA

Les MB-UIDE post-MDA ont l'avantage de pouvoir s'appuyer sur des concepts et des outils communs et pré-existant pour modéliser un SI. De plus, ces MB-UIDE utilisent avantageusement la transformation de modèles pour passer d'un niveau d'abstraction à un autre.

Un des environnements IDM les plus connus est certainement GMF (*Graphical Modeling Framework*) développé par Eclipse [Moore *et al.*, 2004]. GMF ne se place pas dans l'espace technique de MDA mais définit son propre espace technique, appelé EMF (*Eclipse Modeling Framework*) dans lequel le langage Ecore est le méta-métamodèle. GMF est dédié à la génération d'éditeurs de diagrammes à partir d'un ensemble de modèles décrivant les données sources et différentes parties de l'interface (palettes d'outils, présentations, *etc.*). Si GMF a pour principal avantage de générer intégralement le code des éditeurs de diagrammes, ces derniers ont des interfaces graphiques stéréotypées desquelles il est difficile de s'éloigner.

Une majorité des MB-UIDE post-MDA s'intéresse à de nouveaux problèmes issus de l'évolution des dispositifs électroniques (PDA, téléphone portable, *etc.*) et aux nouvelles façons d'interagir avec ces derniers (réalité mixte, interaction multimodale, *etc.*). S'ils gèrent tous les interactions multimodales, certains, comme OpenInterface [Serrano *et al.*, 2008] et ASUR [Gauffre *et al.*, 2008], sont orientés pour la conception d'interactions et d'interfaces post-WIMP ; d'autres, comme CAMELEON [Calvary *et al.*, 2003], UsiXML [Vanderdonckt, 2005] et TERESA [Paternò *et al.*, 2008], tendent à répondre au problème de l'adaptation des IHM au contexte.

[2] « *window, icon, menu, pointing device* »

2.4 Conclusion

Un point commun à tous les MB-UIDE les plus récents présentés dans ce chapitre est l'utilisation d'un processus de conception, représenté par la figure 2.8, divisé en quatre niveaux :

1. *les modèles de tâches et des données sources* définissent les tâches que les utilisateurs peuvent réaliser sur les données sources ;
2. *l'interface abstraite* fournit une description indépendante d'une plate-forme et de toute organisation logique des éléments de l'interface ;
3. l'*interface concrète,* contrairement à celle abstraite, décrit l'aspect graphique de l'interface en fonction d'une plate-forme donnée ainsi que les interacteurs utilisés ;
4. l'*interface finale* correspond au code source du SI.

Une telle décomposition a pour avantages d'enrichir progressivement la spécification d'un SI et de pouvoir réutiliser les modèles définis. Une interface abstraite sert, par exemple, à la création de plusieurs interfaces concrètes, dans le cadre d'un SI multi-plateformes. Cependant, une des limites majeures à la modélisation abstraite est le pouvoir d'expression des MB-UIDE : les SI générés se restreignent généralement à un domaine d'étude (p. ex. les éditeurs de diagrammes pour GMF) [Myers *et al.*, 2000]. Une solution à ce problème consiste à ajouter des modèles afin de décrire plus en détails les SI. Ce principe introduit un nouvel inconvénient, l'augmentation de la complexité du processus de spécification, qui réduit du même coup l'attractivité et la faisabilité de la méthode [Vanderdonckt, 2008]. D'autres limites sont également présentes, comme la maintenance des modèles qui peut s'avérer compliquée lorsqu'ils sont liés entre eux.

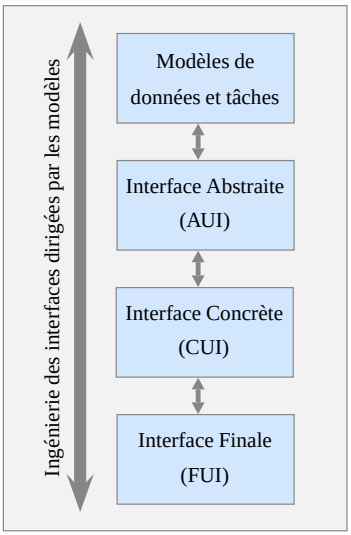


Fig. 2.8: Ingénierie des interfaces dirigée par les modèles

Le tandem IDM-IHM ne se limite pas à la conception de SI mais concerne également la composition et l'adaptation d'IHM. La composition d'IHM consiste à fusionner différentes IHM en une unique [Lepreux *et al.*, 2007; Joffroy, 2009; Déry-Pinna *et al.*, 2009]. L'adaptation d'IHM consiste à modifier l'IHM de manière dynamique, c.-à-d. lors de l'exécution du SI, ou statique

lors d'un changement de contexte d'usage (changement d'utilisateur, de plate-forme d'exécution, sonore, lumineux, *etc.*) [Sottet *et al.*, 2007a; Blumendorf *et al.*, 2008].

Deuxième partie

Manipulation de données avec Malan

Introduction

Cette partie est dédiée à la présentation du langage de correspondance Malan (*a MApping LANguage*) et de l'état de l'art qui s'y rapporte. Le premier chapitre est un état de l'art des différentes méthodes de manipulation de données pour les SI : les principales approches de manipulation de données sont présentées ; le lien entre des données sources d'un SI et ses présentations est discuté. Ce chapitre met en avant l'absence de langage de correspondance respectant les contraintes imposées par l'établissement et la gestion des liens « données-présentations ».

Le deuxième et le troisième chapitres détaillent le langage de correspondance Malan. Le deuxième chapitre décrit le modèle de données de ce langage. La représentation des données est présentée et le typage est ensuite décrit. Le troisième chapitre introduit le modèle de correspondance de Malan, fondé sur le modèle de données du second chapitre. Il se consacre à une présentation détaillée des langages abstrait et concret de Malan.

Chapitre 3

Manipulation de données pour les systèmes interactifs : un état de l'art

Sommaire

3.1 Introduction

Parmi les formats de données prédominants, XML et les relationnelles sont certainement ceux les plus utilisés du fait du consensus établi autour du premier l'établissant comme le langage de description de données, et de la nécessité de centraliser les données pour le second.

Au cœur des SI, la manipulation de données se divise en deux domaines d'application [Blouin *et al.*, 2008b]. Le premier est appelé la *translation de schémas* puisque son but est d'établir un pont entre deux ensembles de données homogènes exprimant un même concept. Il permet ainsi l'interopérabilité entre des bases de données et les SI qui les utilisent. Le second domaine est appelé la *transformation de schémas* étant donné que les schémas sources et le schéma cible n'expriment pas le même concept. Ce principe permet la création de présentations.

La manipulation de données des SI (MDSI) peut être divisée en trois groupes. Le premier rassemble les techniques et les langages manipulant directement les données, c.-à-d. les instances, tels qu'XSLT et XQuery. Leur principal inconvénient est qu'ils ne vérifient pas la structure des données, ce qui rend la manipulation sujette à erreur. Pour éviter cet inconvénient, les techniques du deuxième groupe travaillent au niveau des schémas, au lieu d'effectuer des transformations entre des données. Le troisième groupe comprend les approches dédiées à la transformation dans le cadre d'IDM (*cf.* chapitre 2, page 19). Même si l'IDM et ses langages associés (p. ex. ATL, QVT) visent à répondre à un problème plus général que celui de la MDSI, une transformation de modèles peut être développée dans ce contexte. Les approches fondées sur les schémas sont dépendantes d'une plate-forme de données spécifique et travaillent, par conséquent, dans un espace technique particulier, à savoir XML ou relationnel. En travaillant dans l'espace technique des modèles, les approches IDM sont, quant à elles, indépendantes de ces plates-formes. Cependant, ces approches doivent définir le passage entre leur espace technique et ceux du relationnel et de l'XML.

La section suivante présente une taxonomie des domaines d'applications et une taxonomie des différentes approches. Les sections 3.3, page 38, 3.4, page 41 et 3.5, page 44 détaillent les méthodes et les langages des trois groupes introduits ci-dessus, à savoir : les approches fondées sur les instances, celles fondées sur les schémas et celles dirigées par les modèles. La section 3.6, page 47 est consacrée au problème de la transformation de données en présentation.

3.2 Taxonomies

3.2.1 Taxonomie des domaines d'applications

Nous considérons que la manipulation de données par les systèmes interactifs (MDSI) se limite aux documents XML [W3C, 2006a] et aux bases de données. Par conséquent, les besoins des SI concernent l'*interopérabilité* entre les bases de données et les documents XML qu'ils manipulent ainsi que la *création de présentations*. Ces besoins relèvent des deux domaines de la manipulation de données présentés par la figure 3.1 : la translation et la transformation de schémas.

Une manipulation de données est dite *translation de schémas* lorsque la sémantique du ou des schémas sources est la même que celle du schéma cible. La translation de schémas permet l'interopérabilité entre des données exprimant un même concept mais de manière différente. Il est courant, en effet, que des données hétérogènes soient représentées dans différents formalismes (p.

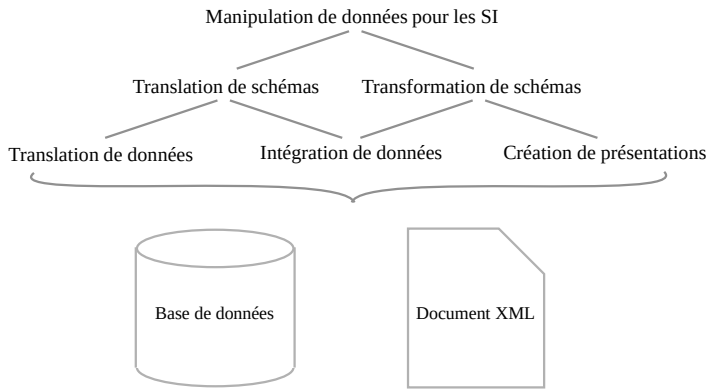
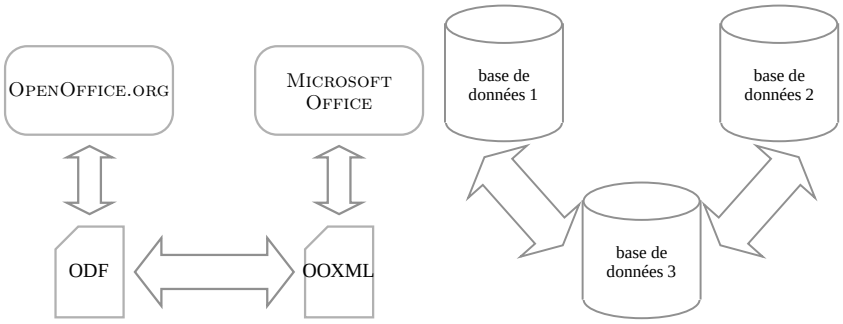


Fig. 3.1: Différents domaines d'application de la manipulation de données pour les SI

ex. schéma relationnel, DTD ou XML Schema) ou dans un formalisme identique mais de manière distincte. Notons que, bien qu'à l'origine la translation de schémas concernait essentiellement le domaine des bases de données, elle s'est étendue au domaine des documents et accentuée depuis l'avènement de l'XML comme l'illustre la figure 3.2a. La translation de schémas se divise en deux sous-domaines majeurs : la translation de données et l'intégration de données.

La *translation de données*, également appelée l'échange de données, est le problème visant à faire passer des données structurées d'un schéma source vers un schéma cible [Fagin *et al.*, 2003; Abiteboul *et al.*, 2002; Atzeni, 2006; Kolaitis, 2005]. La figure 3.2a présente un exemple de translation de données où le but est de créer un pont entre un document au format ODF (*Open Document Format*) et un autre au format OOXML (*Office Open XML*), ces documents définissant le même concept (la notion de document bureautique) différemment.



(a) La translation de données (b) L'intégration de données

Fig. 3.2: Principe de la translation et de l'intégration de données

L'intégration de données vise à combiner des données résidant dans différentes sources afin de fournir à l'utilisateur une vue unifiée et globale de ces données [Batini *et al.*, 1986; Lenzerini, 2002]. La figure 3.2b illustre le principe de l'intégration de données avec un exemple : les sources (les bases de données 1 et 2) contiennent les données d'origine tandis que la cible (la base de données 3) fournit une vue regroupant tout ou partie des données sources.

Une manipulation de données est dite *transformation de schémas* lorsque la sémantique du ou des schémas sources diffère de celle du schéma cible. La transformation de schémas peut être divisée en deux sous-domaines majeurs : l'intégration de données et la création de présentations.

L'intégration de données, présentée précédemment en tant que translation de données, peut également être considérée comme de la transformation de schémas dans certains cas. En effet, si les schémas sources et le schéma cible ont la même sémantique, alors il s'agit d'une translation de schémas. Par exemple, la fusion de deux schémas définissant chacun le concept d'emploi du temps, en un schéma global ayant le même but, est un problème de translation. Par contre, la fusion d'un schéma spécifiant le concept d'emploi du temps avec un autre schéma définissant la scolarité d'étudiant, en un schéma global dédié à l'emploi du temps d'étudiants, est considérée comme de la transformation de schémas. Dans certains cas, la classification d'une intégration de données peut être subtile et dépend essentiellement du contexte et de la sémantique des schémas concernés.

L'autre sous-domaine majeur de la transformation de schémas est la *création de présentations*. Abiteboul (1999) présente les vues comme des outils permettant à des utilisateurs de voir des données de différents points de vue. Cependant, dans le contexte des SI, de telles transformations doivent être incrémentales pour mettre à jour l'interface dont les données sources ont été modifiées sans avoir à reconstruire l'ensemble de cette interface. Cet aspect sera détaillé dans la section 3.6.

3.2.2 Taxonomie des approches

La figure 3.3 synthétise notre classification des différents grands groupes d'approches de la MDSI qui ressortent de la littérature.

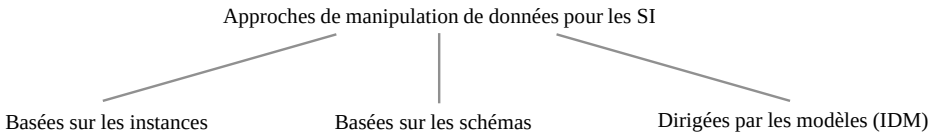


Fig. 3.3: Classification des approches de la manipulation des données pour les SI

Les approches *fondées sur les instances*, c'est-à-dire sur les données elles-mêmes et non sur leur schéma, s'appliquent principalement à la manipulation de documents structurés. Cette tendance s'est accentuée depuis le consensus établissant XML comme le langage standard pour le stockage et l'organisation des données dans les documents textuels.

Par ailleurs, les approches *fondées sur les schémas* considèrent le schéma comme l'élément central de la manipulation. Ces approches s'appliquent aux bases de données, dont le schéma est généralement défini sous forme relationnelle, mais également aux documents structurés (schéma XML).

Enfin, tout comme celles fondées sur les schémas, les approches de *l'ingénierie dirigée par les modèles* (IDM) ne sont pas centrées sur les données (ou modèles) mais sur leur schéma (ou méta-modèle). La majorité des approches fondées sur les schémas peuvent ainsi être considérées comme étant fondées sur l'IDM. Notre classification sépare cependant ces deux types d'approches pour la raison suivante : les premières se limitent principalement aux bases de données et aux documents XML, et aux problèmes associés (p. ex. la fusion des bases de données). Les secondes

visent à répondre des problèmes différents, tels que le passage d'un espace technique à un autre et les transformations « modèle-vers-modèle », « modèle-vers-texte » et « texte-vers-modèle » (voir section 2.5).

Dans les trois sections suivantes, nous détaillons les groupes exposés dans la section 3.3 notamment en présentant les différents langages et cadres de travail existants.

3.2.3 Exemple illustratif

Pour illustrer les approches majeures présentées dans les sections 3.3, 3.4 et 3.5, un même exemple de manipulation de données est utilisé. La source et la cible de cette manipulation sont respectivement les documents XML et SVG des figures 3.4a et 3.4b. Cet exemple vise à créer une présentation graphique en SVG [W3C, 2003] à partir de données XML représentant un blog. Dans un souci de simplicité, les commentaires pouvant être attachés aux billets d'un blog ont été omis.

```
<blog nom="mon Blog">
<billets>
  <billet num="1">
   <titre>Mon premier
    billet</titre>
   <contenu>Ceci est
    le contenu de mon
    premier billet.
   </contenu>
   <date>17-11-2008</date>
   <auteur>Arnaud B.</auteur>
  </billet>
 </billets>
</blog>
```

(a) Un fichier XML

```
<s:svg xmlns:s="http://www.w3.org/2000/svg">
 <s:g font-size="24" fill="#000000">
  <s:text x="-280" y="20" text-anchor="middle"
   transform="rotate(-90)">mon Blog</s:text>
  <s:g id="1">
   <s:rect y="30" x="40" fill="#e6e6e6"
    stroke="#000000" height="180" width="630"/>
   <s:text font-weight="bold" y="60" x="60">Mon
    premier billet</s:text>
   <s:text y="125" x="55">Ceci est le contenu
    de mon premier billet.</s:text>
   <s:text font-style="italic" text-anchor="end"
    y="195" x="620">Arnaud B. - 11-17-2008</s:text>
  </s:g>
 </s:g>
</s:svg>
```

(b) Exemple d'une présentation SVG

Fig. 3.4: Exemple du blog

3.3 Approches fondées sur les instances

Les approches fondées sur les instances travaillent directement sur des données. L'avantage de ce principe est de permettre le développement de manipulations sans avoir recours à la définition du schéma des données qui, dans certains cas, n'existe pas ou n'est pas disponible. Cela peut s'avérer être un inconvénient lors de manipulations de données dont le schéma est complexe, les données cibles pouvant alors rapidement devenir non conformes à leur schéma.

Etant donné que chaque approche recensée utilise un langage de programmation, nous les avons classées en fonction de leur type : les langages déclaratifs et ceux de requêtes. Il existe également des approches qui étendent des langages de programmation courants afin de faciliter la manipulation de données XML, comme XJ [Harren *et al.*, 2005] et XACT [Kirkegaard *et al.*, 2004] pour Java, XTATIC pour C# [Gapeyev *et al.*, 2006], HaXml pour Haskell [Wallace et Runciman, 1999] ou XCentric pour Prolog [Coelho et Florido, 2007]. De plus, des méthodes graphiques permettent la définition de manipulations, comme VXT pour des programmes Circus et XSLT

[Pietriga *et al.*, 2001], visXcerpt pour des programmes Xcerpt [Berger *et al.*, 2003], ou encore l'éditeur commercial *Stylus Studio*[1] pour des programmes XSLT et XQuery.

3.3.1 Les langages déclaratifs

La programmation déclarative s'applique de manière homogène à la manipulation de données. En effet, un programme déclaratif se compose d'un ensemble de règles décrivant quelles données doivent être manipulées, les opérations à réaliser et les données cibles, sans stipuler comment la manipulation doit être effectuée.

```
1  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
2   version="1.0" xmlns:s="http://www.w3.org/2000/svg">
3   <xsl:output method="xml" encoding="utf-8" indent="yes"/>
4   <xsl:template match="blog">
5    <s:svg>
6     <s:g font-size="24" fill="#000000">
7      <s:text x="{-count(billets/billet)*140}" y="20" text-anchor="middle"
8       transform="rotate(-90)"><xsl:value-of select="@nom" /></s:text>
9      <xsl:apply-templates select="billets"/>
10    </s:g>
11   </s:svg>
12  </xsl:template>
13  <xsl:template match="billet">
14   <s:g id="{@num}">
15    <xsl:variable name="y"><xsl:value-of select="(position()-2)*130"/></xsl:variable>
16    <s:rect y="{$y+30}" x="40" fill="#e6e6e6" stroke="#000000" height="180" width="630"/>
17    <s:text font-weight="bold" y="{$y+60}" x="60"><xsl:value-of select="titre"/></s:text>
18    <s:text y="{$y+125}" x="55"><xsl:value-of select="contenu"/></s:text>
19    <s:text font-style="italic" y="{$y+195}" x="620" text-anchor="end">
20     <xsl:value-of select="concat(concat(auteur, ' - '), date)"/></s:text>
21   </s:g>
22  </xsl:template> <xsl:template match='text()|@*' />
23 </xsl:stylesheet>
```

Fig. 3.5: Transformation du blog en XSLT

Le langage déclaratif le plus connu est certainement XSLT [W3C, 2007b; Kay, 2004b; Drix, 2002]. Ce langage transforme des documents XML en d'autres documents textuels, XML ou non. Le principe général de la programmation en XSLT consiste à définir des règles de transformation (*templates*) à appliquer au document source XML. La navigation dans le document source XML s'effectue de manière arborescente grâce au langage XPath [W3C, 2006b; Kay, 2004a]. L'exemple suivant présente une transformation XSLT de l'exemple du blog (*cf.* figure 3.4).

Le programme XSLT de la figure 3.5 se compose de deux règles : la première, ligne 4, définit les actions à réaliser, en l'occurrence la création du corps du document SVG cible, lorsque le nœud courant a pour nom « blog ». Cette règle en appelle une seconde, ligne 13, qui construit un fragment SVG pour chaque billet.

3.3.2 Les langages de requêtes

Les langages de requêtes du web ont pour but de sélectionner, *via* des requêtes, des données dans des bases de données et, plus récemment, dans des documents structurés.

[1] http ://www.stylusstudio.com/

XQuery [W3C, 2006c] est un langage de requêtes pour documents XML, dont il est souvent dit qu'il est pour XML ce qu'est SQL pour les bases de données. Si XQuery et XSLT présentent des similarités en termes de fonctionnalité (ils analysent tous les deux des sources XML, et XQuery peut être considéré comme un langage de transformation), leur principale différence est culturelle : le premier est orienté pour les bases de données alors que le second l'est pour les documents.

Ces deux langages font référence à XPath : XSLT l'utilise de manière indépendante afin de parcourir un document XML, alors qu'XQuery est une surcouche d'XPath.

```
1  declare namespace svg="http://www.w3.org/2000/svg";
2  let $doc := doc("blog.xml")
3  return
4   <s:svg xmlns:s="http://www.w3.org/2000/svg">
5    <s:g font-size="24" fill="#000000">
6     <s:text x="{-count($doc/blog/billets/billet)*140}" y="20"
7      text-anchor="middle" transform="rotate(-90)">
8      {data($doc/blog/@nom)}</s:text> {
9       for $i in 1 to count($doc/blog/billets/billet) return
10       <s:g id="{data($doc/blog/billets/billet[$i]/@num)}">
11        <s:rect y="{($i -1)*260+30}" x="40" fill="#e6e6e6" stroke="#000000"
12         height="180" width="630"/>
13        <s:text font-weight="bold" y="{($i -1)*260+60}" x="60">
14         {data($doc/blog/billets/billet[$i]/titre)}</s:text>
15        <s:text y="{($i -1)*260+125}" x="55">
16         {data($doc/blog/billets/billet[$i]/contenu)}</s:text>
17        <s:text font-style="italic" y="{($i -1)*260+195}" x="620" text-anchor="end">
18         {concat(concat(data($doc/blog/billets/billet[$i]/auteur),' - '),
19          data($doc/blog/billets/billet[$i]/date))}</s:text>
20       </s:g> }
21    </s:g>
22   </s:svg>
```

Fig. 3.6: Transformation du blog en XQuery

La figure 3.6 définit un programme XQuery reprenant le problème de la transformation du blog en un document SVG. Ce programme interroge le document XML source, spécifié à la ligne 2, et retourne les données cibles. Pour chaque billet du blog, l'itération à la ligne 9 lance une requête pour construire un bloc d'objets SVG.

Une des particularités d'XQuery est d'être considéré comme un langage de transformation puisque les données cibles peuvent être restructurées, contrairement à des langages de requêtes classiques tels que SQL. XQuery est donc un langage de manipulation de données réalisant aussi bien de la transformation que de la translation de schémas.

Xcerpt est un langage de requêtes pour le web [Berger *et al.*, 2003]. La structure de ses programmes se compose en un ensemble de requêtes de construction contenant trois blocs : la construction des données cibles (`CONSTRUCT`), l'origine de données sources (`FROM`) et le *pattern matching* à appliquer sur les données sources (`WHERE`). Tout comme XQuery, Xcerpt structure les données cibles mais n'autorise cependant pas certaines instructions itératives nécessaires lors de transformations de schémas complexes. Par exemple, la position des fragments textuels SVG correspondant aux billets du blog ne peut être calculée dans ce langage.

3.3.3 Discussion

Le principal avantage des approches fondées sur les instances, telles qu'XSLT et XQuery, est qu'elles facilitent le développement de transformations simples, comme la transformation d'un document XML en un document HTML. Cependant, ces approches ne peuvent pas assurer la conformité des données manipulées par rapport à leur schéma. Pour contourner ce problème, certaines approches permettent de valider les données cibles générées en utilisant un schéma [W3C, 2007b]. Le fait de travailler au niveau des instances tout en voulant valider les données utilisées est sujet à erreur puisque cela mélange les instances et leur schéma. Les approches fondées sur les schémas ont pour but d'éviter ces problèmes.

3.4 Approches fondées sur les schémas

Les approches fondées sur les schémas considèrent le schéma des données comme l'élément central d'une manipulation de données. Elles permettent ainsi de manipuler les données sources et cibles conformément à leur schéma respectif.

```
1 start   = element blog    { attribute nom   {text},
2                             element billets {element billet {Billet}*}}
3 Billet = attribute num    {xsd:integer},
4          element titre    {text},
5          element contenu  {text},
6          element date     {text},
7          element auteur   {text}
```

Fig. 3.7: Le schéma du blog de la figure 3.4a

La figure 3.7 présente le schéma du blog représenté en Relax NG Compact [OASIS, 2002]. La ligne 1 définit l'élément `blog` comme la racine du schéma, sachant qu'un blog est composé d'un nom (ligne 1) et d'un ensemble de billets (de la ligne 3 à 7). Le schéma SVG est disponible sur le site du W3C[2].

Les approches fondées sur les schémas utilisent soit directement des types correspondant à la structure des données dans les transformations, soit une correspondance de schémas à partir de laquelle une transformation peut être générée.

3.4.1 Les langages de transformation typés

Certains langages de transformation considèrent le schéma comme un ensemble de types directement utilisables dans une transformation ; la conformité des données manipulées est ainsi naturellement garantie. On peut noter que les langages recensés sont des langages fonctionnels. Ce paradigme de programmation place les fonctions en tant qu'objets de première classe dont les calculs transforment les paramètres d'entrée en données de sortie [Hudak, 1989]. De plus, le *pattern matching* permet la sélection des données d'entrées en fonction de critères. Ces avantages ont motivé le développement de nombreux langages fonctionnels manipulant des données structurées ou semi-structurées.

[2]http ://www.w3.org/TR/2002/WD-SVG11-20020108/SVG.xsd

```
1  module blogVersSVG {
2  const blog: [XMLTree] => [XMLTree]=
3   pam x: [XMLTree], y:[XMLTree]
4    var s: String.(*( x # [?blog]++ ?x => [|
5     blog # <label=%'blog', attr=[{nom=?nomBlog}], sub=?subs>
6      => y+= [<label='s:svg',
7                attr =[{xmlns:s='http://www.w3.org/2000/svg'}],
8                sub  =[<label='s:g',
9                         attr =[{font-size='24'}, {fill='#000000'}],
10                        sub  =[<label='s:text',
11                                attr=[{x='-280'}, {y='20'}, {text-anchor='middle'},
12                                      {transform='rotate(-90)'}],
13                                sub=[nomBlog]>,
14                                billets(subs)]>]>]|]))
15 const billets: [XMLTree] => [XMLTree]=
16  pam x: [XMLTree], y:[XMLTree]
17   var s: String.(*( x # [?billet]++ ?x => [|
18    billet # <label=%'billet', attr=[{num=?numBil}],
19               sub  =[<label=%'titre',    sub=?titreBil>,
20                      <label=%'contenu', sub=?contBil>,
21                      <label=%'auteur',  sub=?autBil>,
22                      <label=%'date',    sub=?dateBil>]>
23     => y+= [<label='s:g', attrs=[{id=numBil}],
24               sub  =[<label='s:rect',
25                        attr=[{y=(pos-1)*260+30},{x='40'},{fill='#e6e6e6'},
26                              {width='630'},{height='180'}, {stroke='#000000'}]>,
27                      <label='s:text', sub=[titreBil],
28                        attr =[{font-weight='bold'}, {y=(pos-1)*260+60}, {x='60'}]>
29                      <label='s:text', sub=[contBil],
30                        attr =[{y=(pos-1)*260+125},{x='55'}]>,
31                      <label='s:text', sub=[autBil+' - '+dateBil,
32                        attr =[{font-style='italic'},{y='{(pos-1)*260+195}'},
33                               {x='620'},{text-anchor='end'}]]>]>]|])) }
```

Fig. 3.8: Transformation du blog en Circus

XDuce[Hosoya et Pierce, 2003] est un langage fonctionnel au typage statique manipulant des données XML. Ce langage possède un *pattern matching* puissant utilisant les opérations sur les arbres et les expressions régulières. Développé en parallèle à XDuce, CDuce[3] est également un langage fonctionnel à typage statique pour XML [Benzaken *et al.*, 2003; Frisch, 2004], partageant de nombreux principes avec XDuce comme son typage fort et son *pattern matching*. CDuce peut être vu comme une extension de XDuce permettant en plus l'utilisation d'itérateurs, la surcharge de fonctions et se voulant plus général (moins orienté XML). Dans le même registre, le langage OCaml+XDuce, fusion des langages XDuce et OCaml, combine les meilleures propriétés des deux langages [Frisch, 2006].

Circus est un langage typé spécialisé dans la transformation de structures s'adaptant à la manipulation de documents XML [Vion-Dury, 1999; Vion-Dury *et al.*, 2002]. Il s'agit d'une extension du lambda-calcul typé non récursif rendant possible la définition de modules. La figure 3.8 présente un programme Circus de notre exemple dans lequel la définition des types a été volontairement omise. Ce programme Circus se compose d'une première méthode (ligne 2) créant le corps du document SVG. Cette méthode en appelle une seconde (ligne 15) qui construit la représentation SVG des billets du blog.

[3]Un prototype est disponible à l'adresse suivante : http ://www.cduce.org

Du fait de leur puissant *pattern matching* et de leur utilisation d'expressions régulières pour le traitement de données, ces langages ont tous la même capacité à réaliser des translations et des transformations de schémas.

3.4.2 Les approches fondées sur la correspondance de schémas

Les approches fondées sur la correspondance de schémas permettent d'établir des liens, appelés correspondances, entre des schémas sources et cibles. Ces approches diffèrent de celles fondées sur les instances : elles définissent des *relations* entre deux schémas et non un *programme* qui implémente ces relations. La spécification de correspondances est un paradigme utilisé originellement dans le domaine des bases de données et dont l'enjeu est double [Raffio *et al.*, 2008; Roth *et al.*, 2006] :

1. pouvoir générer, à partir d'une correspondance de schémas, des transformations dans différents langages manipulant des données instances de ces schémas ;
2. capturer la relation entre des schémas pour faciliter la gestion des changements affectant des schémas.

Cette section se focalise sur la capacité des approches présentées à gérer le premier point puisque notre problématique concerne la manipulation de données et non l'évolution de schémas.

Clio est un outil IBM permettant la définition graphique de correspondances entre un schéma source et un schéma cible [Miller *et al.*, 2000; Miller *et al.*, 2001; Yan *et al.*, 2001; Popa *et al.*, 2002; Haas *et al.*, 2005]. Ces correspondances sont enrichissables *via* un éditeur d'expressions (expressions arithmétiques par exemple). Clio génère des transformations XQuery, SQL ou XSLT à partir de correspondances de schémas.

La figure 3.9 présente l'interface de Clio lors de la définition de la correspondance entre le schéma de la figure 3.7, représenté dans la partie gauche, et celui de SVG dans la partie droite. Les correspondances sont établies entre les attributs des schémas. Dans notre exemple, la première correspondance concerne l'attribut `nom` de l'élément `blog` et un élément SVG `text` ; les autres concernent les éléments `titre`, `contenu`, `date`, `auteur` et `num` des billets, et des éléments SVG `text`. Chaque élément cible est défini soit par une correspondance, soit par une valeur fixe définie par le biais de l'éditeur d'expression (voir la fenêtre à gauche de la figure 3.9). La transformation d'un blog en un document SVG n'est pas réalisable avec Clio puisque certaines opérations impératives, comme l'itération, ne sont pas permises. Il est cependant possible de contourner ce problème en appelant directement les fonctions d'un langage de transformation *via* l'éditeur d'expressions, comme l'illustre la figure 3.9 avec l'appel de la fonction XSLT `count`. L'inconvénient de ce processus est la dépendance à un langage de transformation, ce qui contredit l'un des principes fondamentaux de la correspondance de schémas.

Influencé par Clio, Clip est un langage graphique réalisant le tracé de correspondances entre des éléments d'un schéma source et d'un autre cible [Raffio *et al.*, 2008]. La différence entre ces deux approches est que Clip définit graphiquement la sémantique des correspondances. Le principal inconvénient de Clip est le même que la plupart des langages graphiques, à savoir le manque de lisibilité et donc de compréhension lorsque le nombre et la complexité des correspondances augmentent.

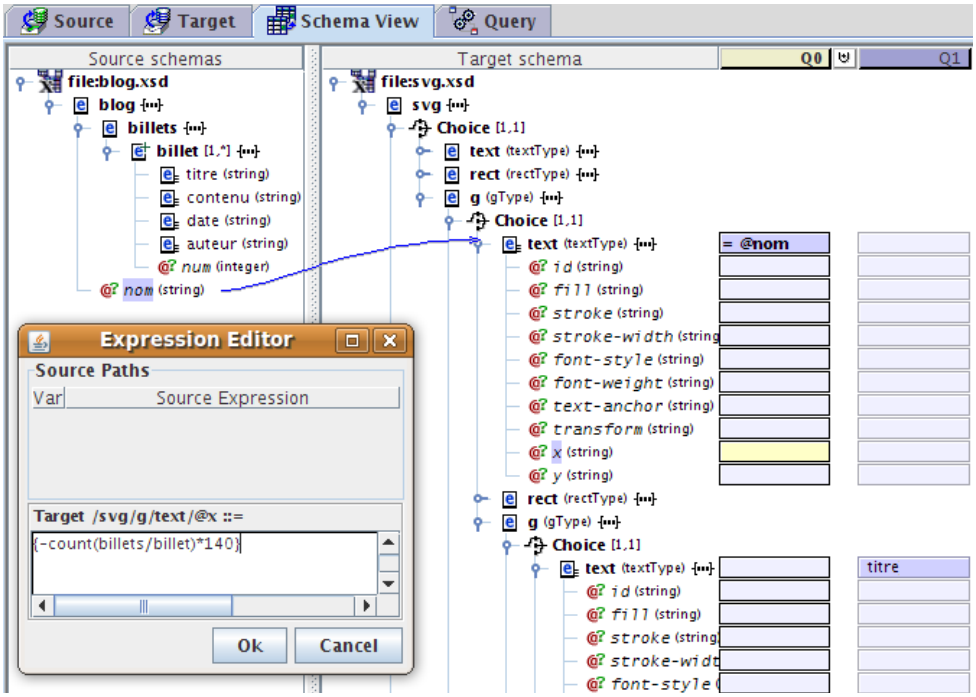


Fig. 3.9: Interface de Clio pour l'exemple du blog

3.4.3 Discussion

Les approches fondées sur les schémas répondent certainement le mieux au problème de la MDSI. En effet, ces approches garantissent la validité des données et demeurent indépendantes des processus de transformation ; les transformations peuvent ainsi être générées dans différents langages à partir d'une même correspondance de schémas. De plus, une correspondance de schémas peut être utilisée pour définir un lien durable entre des données sources et une présentation, comme le détaille la section 3.6. L'utilisation du principe des correspondances permet également l'application d'opérations importantes sur des sources de données et leur schéma, comme le *schema matching*[4] et le *schema merging*, visant notamment à simplifier la définition de correspondances.

3.5 Approches dirigées par les modèles

Bien que MDA, dont les principes sont détaillés dans la section 2.2.2, page 22, soit appliquée essentiellement à l'ingénierie du logiciel, cette approche peut être utilisée dans le cadre de la MDSI. La section suivante donne les différences fondamentales entre les approches fondées sur les schémas et celles fondées sur MDA ainsi que la transformation de modèles appliquée au problème de la MDSI ; la section 3.5.2 conclut cette section par une discussion sur ce problème.

[4]Le terme *matching* n'est pas à prendre au sens de *correspondance* ; le *schema matching* est une opération visant à trouver de manière (semi-)automatique des correspondances entre des schémas possédant des similitudes.

3.5.1 Différences entre les approches fondées sur les schémas et celles fondées sur MDA

Contrairement à MDA, les approches fondées sur les schémas, présentées dans la section 3.4, se restreignent à deux représentations de données : les documents XML et les bases de données. Les espaces techniques (*cf.* section 2.2.3, page 23) permettent aux approches MDA de passer de l'espace technique MDA, dédié à la description de systèmes sous la forme de modèles, aux espaces techniques de différentes représentations de données à l'aide de langages dédiés, tels que les langages Sintaks [Muller *et al.*, 2006] et TCS (*Textual Concrete Syntax*) [Jouault, 2006], comme l'illustre la figure 3.10.

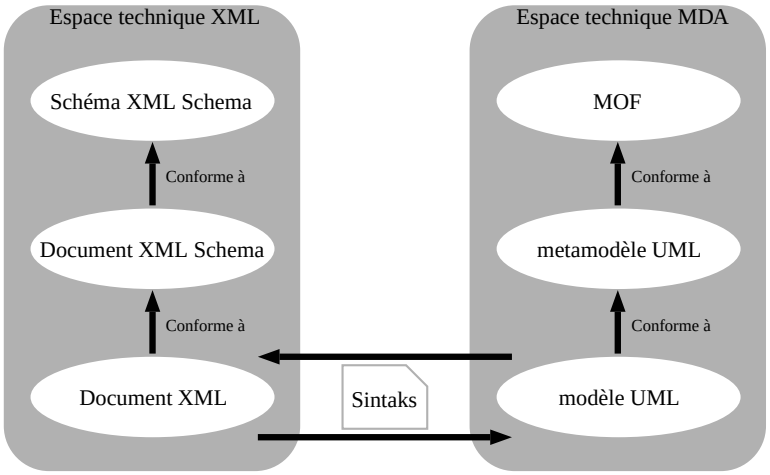


Fig. 3.10: Différences entre l'approche fondée sur le schéma et l'IDM

Les espaces techniques assurent ainsi une grande flexibilité par rapport à la représentation de données. Cependant, dans le contexte de la MDSI où les données sont soient des bases de données, soient des documents XML, cette flexibilité voit son utilité réduite et provoque alors une contrainte : pour des données représentées dans un formalisme donné, il est nécessaire de spécifier le passage entre ce formalisme et sa représentation dans l'espace technique MDA. Par exemple, étant donné un modèle, il est nécessaire de définir le passage de ce modèle vers sa représentation XML, relationnelle, *etc.*

La transformation de modèles, détaillée dans la section 2.2.4, est une opération essentielle dans le cadre de MDA. La figure 3.11 présente un programme ATL pour notre exemple du blog[5]. Ce programme est constitué d'une règle de transformation, `blog2svg` définissant les éléments sources et cibles concernés en utilisant les opérateurs `from` et `to`. Cette règle décrit de manière déclarative la structure générale du modèle cible (lignes 4 à 19), ainsi que la transformation des billets en éléments SVG (lignes 20 à 54).

[5]Les détails techniques de cette transformation de modèles (métamodèles, modèles, *etc.*) sont décrits dans l'annexe D, page 231.

```
1  module Blog2SVG;
2  create OUT : SVG from IN : Blog;
3  rule blog2svg {
4    from b : Blog!Blog to s : SVG!Svg (
5          children <- g1,
6          version <- '1.1',
7          namespace <- 'http://www.w3.org/2000/svg' ),
8      g1 : SVG!G (
9          fontSize <- 24,
10         fill <- '#000000',
11         groupContent <- titreBlog,
12         groupContent <- billetsBlog ),
13     titreBlog : SVG!Text (
14         position <- coordTitreBlog,
15         textAnchor <- 'middle',
16         attribute <- rotateTitreBlog,
17         content <- b.nom ),
18     coordTitreBlog : SVG!AbsoluteCoord ( x <- (0-140) * b.billets->size(), y <- 20 ),
19     rotateTitreBlog : SVG!Rotate(angle <- 0-90),
20     billetsBlog : distinct SVG!G foreach(b1 in b.billets) (
21         id <- b1.num,
22         groupContent <- rectBillet,
23         groupContent <- titreBillet,
24         groupContent <- contenuBillet,
25         groupContent <- dateBillet ),
26     rectBillet : distinct SVG!Rect foreach(b1 in b.billets) (
27         position <- rectPos,
28         fill <- '#e6e6e6',
29         stroke <- '#000000',
30         size <- rectDim ),
31     rectPos : distinct SVG!AbsoluteCoord foreach(b1 in b.billets)
32               ( x <- 40, y <- (b.billets->indexOf(b1)-1)*130+30 ),
33     rectDim : SVG!Dimension ( width <- 630, height <- 180 ),
34     titreBillet : distinct SVG!Text foreach(b1 in b.billets) (
35         position <- coordTitre,
36         attribute <- fontWeight,
37         content <- b1.titre ),
38     coordTitre : distinct SVG!AbsoluteCoord foreach(b1 in b.billets)
39               ( x <- 60, y <- (b.billets->indexOf(b1)-1)*130+60 ),
40     fontWeight : SVG!FontWeight ( bold <- true ),
41     contenuBillet : distinct SVG!Text foreach(b1 in b.billets) (
42         position <- coordContenu,
43         content <- b1.contenu ),
44     coordContenu : distinct SVG!AbsoluteCoord foreach(b1 in b.billets) (
45         x <- 55,
46         y <- (b.billets->indexOf(b1)-1)*130+125 ),
47     dateBillet : distinct SVG!Text foreach(b1 in b.billets) (
48         fontStyle <- 'italic',
49         textAnchor <- 'end',
50         position <- coordDate,
51         content <- b1.auteur + ' - ' + b1.date ),
52     coordDate : distinct SVG!AbsoluteCoord foreach(b1 in b.billets) (
53         x <- 620,
54         y <- (b.billets->indexOf(b1)-1)*130+195 )
55 }
```

Fig. 3.11: Transformation du blog en ATL

3.5.2 Discussion

En complément de la transformation de modèles, le tissage de modèles[6] (*model weaving*) permet la spécification de correspondances entre les éléments des différents métamodèles [Fabro et Valduriez, 2007] à la manière de la spécification d'une correspondance de schémas pour les approches fondées sur les schémas. L'ensemble de ces correspondances forme un modèle de tissage conforme au métamodèle de tissage. Les buts sont multiples et équivalents à ceux de la correspondance de schémas, à savoir : la génération de transformations de modèles, l'application d'opérations sur des sources de données (*merging* et *matching*), ou encore la capture et la maintenance de la relation sémantique entre des modèles.

Bien que le tissage de modèles rende possible des opérations essentielles sur les données, en particulier sur les bases de données, l'approche MDA aspire à répondre à un problème beaucoup plus large que celui de la manipulation de données pour les SI. Elle nécessite ainsi la définition de règles de passage entre l'espace technique IDM et ceux de l'XML et du relationnel. Nos travaux se placent dans une approche IDM en utilisant des règles de passage automatique qui apportent cependant des contraintes de modélisation.

3.6 Transformation des données en présentations

Dans le contexte des SI, la création de présentations a également pour but de maintenir un lien durable entre des données sources et leurs présentations cibles. Ce lien doit être durable dans le sens où toute modification des données sources doit provoquer une mise à jour de la présentation cible. Il peut être implémenté par une transformation *active* laquelle se fonde sur le modèle « d'observable-observateur ». Il peut également être réalisé par une transformation *incrémentale* qui utilise une transformation classique (p. ex. un programme XSLT) avec un processeur incrémental, lequel calcule la différence entre deux états successifs du même document source pour en déduire les fragments cibles à reconstruire. Cette propriété de durabilité est particulièrement utile lors de la manipulation d'ensembles de données volumineux. De plus, dans le contexte des SI, ce lien permet la manipulation (création, modification et suppression) de données sources par le biais de la présentation.

Des travaux ont été effectués pour développer des processeurs de transformation XSLT incrémentaux [Villard et Layaïda, 2002; Onizuka *et al.*, 2005], sans toutefois assurer la validité des données cibles générées. Concernant XQuery, la recommandation XQuery Update Facility (XQUF) [W3C, 2008] vise à étendre ce langage pour le support de mises à jour incrémentales, ainsi que pour la modification des données sources. Ainsi, XQUF fournit la possibilité de :

- supprimer un nœud source ;
- ajouter un nœud source ;
- modifier des propriétés d'un nœud source ;
- dupliquer un nœud source déjà existant.

La figure 3.12 présente un exemple de requête XQUF ajoutant un billet dans un blog. On y trouve la fonction `insert` définissant le nœud à insérer, suivie de la fonction `after` spécifiant la position où doit être inséré le nouveau nœud dans l'arbre source. De même que pour le langage XSLT, XQUF travaille au niveau des instances et se limite au traitement de documents XML.

[6] Le terme « tissage » est ici improprement utilisé dans le sens où il se réfère à des opérations sur des modèles (« matching », *etc.*) et non au tissage d'aspect [Kiczales *et al.*, 1997].

```
1 insert <billet num="{count(document(blog.xml)/blog/billets/billet)+1}">
2         <titre>Mon deuxième billet</titre>
3         <contenu>Ceci est mon 2ième billet.</contenu>
4         <date>16-02-2009</date>
5         <auteur>Arnaud B.</auteur>
6        </billet>
7 after document(blog.xml)/blog/billets/billet[last()]
```

Fig. 3.12: Requête XQuery Update Facility ajoutant un billet au blog

Entièrement dédié à la définition de liens « données-présentation », eXAcT (*XML Active Transformation*) [Beaudoux, 2005c] est un processus de transformation définissant comment un document DOM source est transformé en un autre document DOM cible correspondant à une présentation SVG, Draw2D ou X3D. Dans la continuité d'eXAcT, AcT est un environnement dédié à la spécification graphique et textuelle de correspondances [Beaudoux *et al.*, 2009]. Il reprend les principes du processus eXAcT en se fondant sur l'IDM : une correspondance AcT est établie entre un métamodèle source et un autre cible. Elle est ensuite compilée en une transformation active s'exécutant sur la plate-forme .NET. Ces travaux rejoignent ceux initialement proposés par Akehurst (2000) et Varró *et al.* (2002) pour des problèmes non dédiés au lien « données-présentation ». Ainsi, dans AcT, une transformation utilise un modèle de données source (base de données ou document XML), conforme au métamodèle source, pour créer ou mettre à jour un modèle de présentation cible (un graphe de scène WPF), conforme au métamodèle cible, comme l'illustre la figure 3.13.

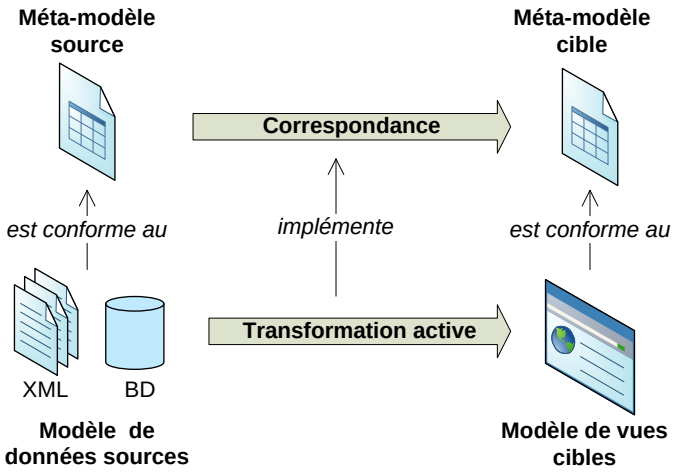


Fig. 3.13: Principe de la correspondance et de la transformation dans AcT, extrait de [Beaudoux *et al.*, 2009]

Toujours dans le domaine de l'IDM, la spécification du langage QVT préconise qu'un processeur de transformation QVT puisse mettre à jour incrémentalement un modèle cible.

La définition de liens « données-présentation » est également abordée dans les plates-formes RIA sous le terme de *data binding*, comme le détaille le chapitre 1.3, page 12.

3.7 Conclusion

Etant toutes « *batch* », les transformations fondées sur les instances, sur les schémas ou dirigées par les modèles, présentées dans la première partie de ce chapitre, ne sont pas adaptées à la transformation active de données en présentations. Le tableau 3.1 synthétise les principales méthodes en fonction des caractéristiques intervenant dans le lien données-présentations, à savoir : la méthode est-elle active, modifie-t-elle les données sources et les données cibles, travaille-t-elle au niveau des instances, des schémas ou des modèles ? L'absence d'information relative à certaines caractéristiques des méthodes est symbolisée par l'abréviation « *n/a* ». Certains travaux, comme IncXSLT et XTim ont permis d'utiliser une transformation « *batch* » avec un processeur incrémental afin d'éviter la reconstruction systématique de la cible à chaque changement de la source. Les transformations actives, telles que eXAcT et AcT, sont optimisées pour la mise à jour de la cible *via* un processus « observable-observateur » et sont ainsi préférables à ces méthodes incrémentales. Bien que les travaux d'Akehurst proposent un modèle d'implémentation active de correspondances, la modification du modèle source, essentielle dans le cadre des SI, n'est pas abordée. A l'inverse, la spécification du langage QVT précise que l'implémentation de ce langage doit permettre une mise à jour incrémentale du modèle cible sans toutefois fournir des détails précis sur ce processus.

Ce constat illustre le manque d'outils et de méthodes pour le problème de la transformation de données en présentations. Le chapitre suivant présente le langage de correspondance Malan dédié à la génération d'une transformation active AcT à partir d'une correspondance de schémas Malan.

	Actif	Modifie Source	Source	Cible	Approche
incXSLT	Incrémental	Non	XML	Texte, XML	Instance
XTim	Incrémental	Non	XML	Texte, XML	Instance
Akehurst	Oui	n/a	Modèle	Modèle	IDM
QVT	Incrémental	Oui	Modèle	Modèle	IDM
eXAcT	Incrémental	Oui	DOM	DOM	Instance
AcT	Oui	En perspective	Modèle	Modèle	IDM
XQuery UF	n/a	Oui	XML	Texte, XML	Instance

Tab. 3.1: Classification des caractéristiques des méthodes pour la définition du lien entre des données sources et une présentation d'un SI

Chapitre 4

Modèle de données du langage Malan

Sommaire

4.1 Introduction

Les travaux présentés dans ce chapitre portent sur le modèle de données du langage de correspondance Malan (*a MApping LANguage*). Le modèle de données d'un langage spécifie, d'une part, la manière dont les données sont représentées et, d'autre part, le typage du langage. Dans notre cahier des charges du langage Malan figurent deux critères relatifs au modèle de données. Tout d'abord, un tel langage doit être indépendant des différentes plates-formes de données que peut utiliser un SI. Pour cela, ce langage doit travailler au niveau des schémas des données et non au niveau des instances. Ensuite, le langage de correspondance doit disposer de types primitifs utilisables dans la définition de données. A partir de ces deux besoins, les choix relatifs au modèle de données du langage Malan sont présentés dans ce chapitre.

Ce chapitre s'organise de la manière suivante : la section 4.2 présente comment les données sont représentées ; la section 4.3 détaille le typage utilisé par Malan.

4.2 Représentation des données

Nous présentons dans cette section le principe de la correspondance au niveau des schémas des données, ainsi que les avantages qui ont motivé le choix du diagramme de classes UML pour représenter les données dans Malan.

4.2.1 Principe

Afin d'assurer la validité des données manipulées, le langage Malan travaille au niveau des schémas : une correspondance de schémas s'établit entre le schéma des données sources et celui des données cibles (*cf.* figure 4.1), tous deux décrits par des diagrammes de classes UML. Dans le cadre de la conception de SI, ce principe permet également d'être indépendant des plates-formes de données qu'ils peuvent utiliser.

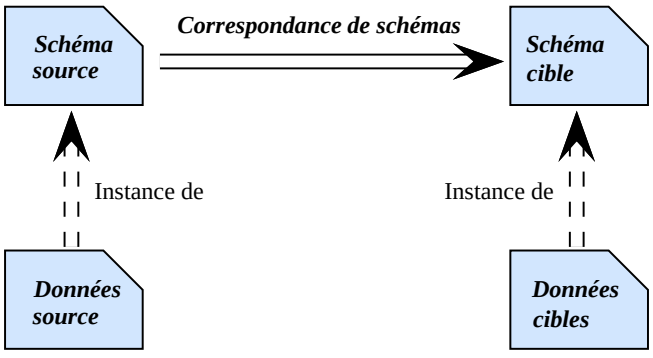


Fig. 4.1: Principe de la correspondance de schémas au sein de Malan

Nous considérons Malan dans une approche IDM définissant son propre espace technique dans lequel le diagramme de classes UML correspond au métamodèle. Pour éviter toute confusion avec le sens du terme « modèle » et pour reprendre le vocabulaire déjà existant du domaine

des données XML et relationnelles, nous utilisons, dans ce chapitre, les termes « données » et « schéma » pour désigner, respectivement, un modèle et un métamodèle.

4.2.2 Diagramme de classes UML

Malan utilise le diagramme de classes UML pour représenter les données. Ce choix se justifie pour les raisons suivantes :

1. UML, et en particulier les diagrammes de classes, est dédié à la modélisation de systèmes et est devenu l'outil principal de l'IDM pour définir les modèles et les métamodèles.
2. Dans notre contexte, les données sources correspondent à des documents XML ou des données relationnelles, tandis que les données cibles représentent des graphes de scènes. L'utilisation d'UML dans le cadre d'XML et des données relationnelles a été largement étudiée sans présenter de limite majeure [Soutou, 2007].
3. Les profils UML permettent d'étendre UML à la modélisation de domaines initialement non prévus.

Le diagramme de classes de la figure 4.2 décrit un blog qui définit un nom et se compose d'un ensemble de billets. A chaque billet correspond une date, un numéro, un titre, un contenu et un auteur, lequel possède un nom et un prénom. Cet exemple montre bien l'indépendance envers la plate-forme de données puisque les données instances de ce schéma peuvent être un document XML, une base de données relationnelle ou un document semi-structuré. La passerelle entre le diagramme de classes et les plates-formes de données est donc à la charge de l'implémentation de Malan. L'exemple du blog est utilisé dans la section 4.3 pour illustrer les types que fournit Malan.

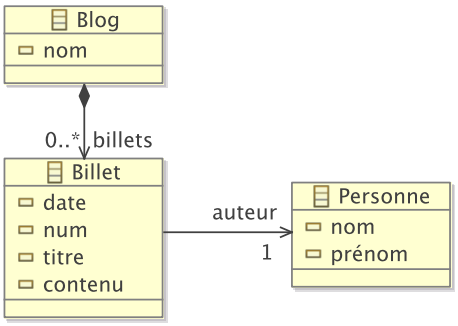


Fig. 4.2: Diagramme de classes d'un blog

Les définitions suivantes, largement inspirées de Berardi *et al.* (2005), sont utilisées dans le chapitre suivant pour la spécification du modèle de correspondance du langage Malan. Elles décrivent formellement un sous-ensemble des diagrammes de classes UML. Une classe correspond à un prédicat unaire $\mathcal{C}$ tel que $\mathcal{C}(x)$ stipule que x est une classe. Le prédicat $abs(x)$ stipule, quant à lui, qu'une classe x est abstraite, tel que :

$$\forall x. \quad abs(x) \implies \mathcal{C}(x) \tag{4.1}$$

Un attribut se formalise par le prédicat binaire $\mathcal{A}(x, \omega)$ où ω est le type de l'attribut ($\Gamma \vdash \omega$ stipule que ω est un type) et x la classe à laquelle il appartient, tel que :

$$\forall x, \omega. \quad \mathcal{A}(x, \omega) \implies \mathcal{C}(x) \wedge \Gamma \vdash \omega \tag{4.2}$$

La cardinalité $[i..j]$ d'un attribut de type ω d'une classe x définit que l'attribut est associé à chaque instance de x avec au minimum i et au maximum j instances du type ω, tel que :

$$\forall x. \quad \mathcal{C}(x) \implies i \leq \#\{\omega / \mathcal{A}(x, \omega)\} \leq j \tag{4.3}$$

Une opération est un prédicat n-aire $\mathcal{O}(x, \omega, \omega_1, \ldots, \omega_n)$ où x correspond à la classe de l'opération, $\omega_1, \ldots, \omega_n$ aux types de ses paramètres et ω au type de la valeur de retour, comme le spécifie la formule 4.4.

$$\mathcal{O}(x, \omega, \omega_1, \ldots, \omega_n) \implies \mathcal{C}(x) \wedge \Gamma \vdash \omega \bigwedge_{i=1}^{n} \Gamma \vdash \omega_i \tag{4.4}$$

Concernant les associations, les agrégations et les compositions, aucune distinction n'est faite entre ces trois types de relations et sont formalisées par le prédicat binaire $\mathcal{R}(x, x')$ où x et x' sont respectivement la classe source et la classe cible de la relation, comme le définit la formule suivante.

$$\forall x, x'. \quad \mathcal{R}(x, x') \implies \mathcal{C}(x) \wedge \mathcal{C}(x') \tag{4.5}$$

La formule 4.6 formalise la relation d'héritage entre une classe fille $C(x)$ et une classe mère $C(x')$ représentée par le symbole $\implies$. Cette relation est transitive : étant donnés x, x' et x'', si $\mathcal{C}(x'') \implies \mathcal{C}(x')$ et que $\mathcal{C}(x') \implies \mathcal{C}(x)$, alors $\mathcal{C}(x'') \implies \mathcal{C}(x)$.

$$\forall x, x'. \quad \mathcal{C}(x) \implies \mathcal{C}(x') \tag{4.6}$$

4.3 Typage

Avant de présenter les différents types que possède le langage Malan, les notations et les conventions utilisées pour définir ces types sont brièvement introduites.

4.3.1 Notations

Les lettres κ, ρ, α et δ sont réservées pour représenter les types *classe*, *relation*, *attribut* et *fonction/opération*. D'autres symboles sont également utilisés pour représenter différentes classes de types : le type υ regroupe tous les types primitifs (entier, flottant, booléen et chaîne de caractères) ; le type ϕ rassemble les types primitifs avec celui définissant une liste d'objets d'un même type ; τ réunit les types *classe* et *liste de classes* avec ϕ ; de même, ω rassemble le type τ avec les types *relation* et *liste de relations* ; enfin, σ regroupe le type τ avec le type *attribut* α.

$$\begin{aligned}
\text{Type } \textit{classe } \kappa \\
\text{Type } \textit{relation } \rho \\
\text{Type } \textit{attribut } \alpha \\
\text{Type } \textit{fonction/opération } \delta \\
\text{Type } \upsilon &::= \textit{int} \mid \textit{float} \mid \textit{bool} \mid \textit{string} \\
\text{Type } \phi &::= \upsilon \mid \textit{list} < \tau > \\
\text{Type } \tau &::= \phi \mid \kappa \mid \textit{list} < \kappa > \\
\text{Type } \omega &::= \tau \mid \rho \mid \rho < \tau > \\
\text{Type } \sigma &::= \tau \mid \alpha
\end{aligned}$$

Soient β et β' deux types, l'opérateur binaire $\preceq$ est utilisé pour définir que β est un sous-type de β' tel que $\beta \preceq \beta'$. Cet opérateur de sous-typage possède les propriétés suivantes :

– *réflexivité* : un type β est un sous-type de lui-même : $\forall \beta, \beta \preceq \beta$;

– *transitivité* : soient β, β' et β'' trois types tels que si $\beta \preceq \beta'$ et $\beta' \preceq \beta''$, alors $\beta \preceq \beta''$.

La relation de typage porte sur des triplets (Γ, a, β), où β est un type, a une expression et Γ un environnement de typage (également appelé contexte de typage). Elle est habituellement formalisée à l'aide d'assertions logiques, appelées *jugements*, notée $\Gamma \vdash a : \beta$ et se lisant « l'expression a a pour type β dans le contexte Γ ». Dans la suite du manuscrit, le symbole Γ représentera l'environnement de typage de Malan. Les règles d'inférences établissent la validité d'un jugement en fonction de celle d'autres jugements ; ainsi, la règle 4.7 définit que si les jugements $\Gamma \vdash x : \tau$ et $\Gamma \vdash y : \tau$ sont vrais alors $\Gamma \vdash x + y : \tau$ est vrai.

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \tau}{\Gamma \vdash x + y : \tau} \tag{4.7}$$

4.3.2 Définition des types

Le typage facilite la manipulation des éléments d'un diagramme de classes UML lors de la définition de correspondances. Le langage Malan possède ainsi six types différents :

Types primitifs. Les *entiers*, les *flottants*, les *chaînes de caractères* et les *booléens* sont des types sémantiquement équivalents à ceux du langage Java. Des variables de ces types peuvent être déclarées dans une fonction en utilisant, respectivement, les mots-clés `int`, `float`, `string` et `bool`. La sémantique des opérations autorisées sur les types primitifs est définie dans l'annexe A.2.

Classe. Une classe est un type utilisable dans le code d'une correspondance de schémas. Par exemple, la fonction suivante possède un paramètre de type `Billet` issu du diagramme source :

```
1 function int calculerPositionY(Billet b, int supp) {
2   return (position(b)-1)*130+30+supp;
3 }
```

La notion d'héritage est représentée en utilisant l'opération de sous-typage : soient A et B deux classes, tel que A hérite de B, alors $A \preceq B$.

Malan n'autorise pas l'instanciation d'une classe issue d'un diagramme source ou cible. Par exemple, il n'est pas possible de déclarer et d'instancier une variable de type `Billet` de la manière suivante : `Billet b = new Billet()`. Ce choix se justifie par le fait que Malan est un langage de correspondance permettant la navigation dans les schémas source et cible mis en relation. L'instanciation des objets cibles, conforment au schéma cible, est effectuée par l'implémentation des correspondances.

Liste. Une liste est un ensemble ordonné d'objets de même type. Elle est notamment utilisée pour représenter les relations, de cardinalité $0..n$ ou $1..n$, d'un diagramme de classes. Une liste peut être elle-même typée à la manière des « *generics* » en Java ; le type de la relation `billets` de la classe `Blog` (figure 5.1, page 64) est `list<Billet>,` où `<Billet>` stipule que la liste ne contient que des objets de type `Billet`. Une liste se déclare comme suit :

```
1 "list" ("<" TYPE ">")? ID
```

où `TYPE` correspond au type des éléments de la liste et `ID` à son nom. Il est possible de réaliser des opérations ensemblistes sur des variables de type `list`, dont les principales sont :

- la concaténation ; il est possible de concaténer deux listes (de même type si celui-ci est spécifié) :

$$\frac{\Gamma \vdash e_1 : list < \tau > \quad \Gamma \vdash e_2 : list < \tau >}{\Gamma \vdash e_1 + e_2 : list < \tau >} \tag{4.8}$$

- L'insertion d'un objet (de même type que celui de la liste si ce dernier est spécifié) au début ou à la fin d'une liste est également possible :

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : list < \tau >}{\Gamma \vdash e_1 + e_2 : list < \tau >} \tag{4.9}$$

$$\frac{\Gamma \vdash e_1 : list < \tau > \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : list < \tau >} \tag{4.10}$$

- La suppression d'un élément dans une liste ainsi que la suppression, dans une liste, des éléments d'une autre liste s'effectue de la manière suivante :

$$\frac{\Gamma \vdash e_1 : list < \tau > \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 - e_2 : list < \tau >} \tag{4.11}$$

$$\frac{\Gamma \vdash e_1 : list < \tau > \quad \Gamma \vdash e_2 : list < \tau >}{\Gamma \vdash e_1 - e_2 : list < \tau >} \tag{4.12}$$

Les listes dont le type de leurs objets n'est pas précisé se déclarent simplement `list`. De plus, la syntaxe concrète visant à obtenir la cardinalité d'une liste `l` s'obtient *via* l'utilisation de l'opérateur `|l|`, dont la sémantique est décrite par la formule 4.13. La syntaxe abstraite de cet opérateur, utilisée dans les formules de ce chapitre, est la suivante : $\#\{l\}$.

$$\frac{\Gamma \vdash exp : \omega \quad \omega \preceq list}{\Gamma \vdash |exp| : int} \tag{4.13}$$

L'accès aux éléments d'une liste s'effectue à l'aide d'un indice définissant la position de l'élément ciblé en respectant le format suivant :

```
1 LISTE_SELECTION = ID "[" EXP_ARITHMETIQUE "]"
```

où `ID` correspond au nom de la liste et `EXP_ARITHMETIQUE` à l'indice. Pour illustrer ce principe, prenons les exemples suivants :

- `billets[i]` retourne le ième billet de la liste, avec $i \in [1; |billets|]$;
- `billets[1]`, sélectionne le premier billet de la composition `billets` ;
- `billets[|billets|]`, sélectionne le dernier billet.

La formule 4.14 définit la sémantique de ce type d'expression.

$$\frac{\Gamma \vdash exp : \omega \quad \Gamma \vdash exp' : int \quad \omega \preceq list < \tau >}{\Gamma \vdash exp[exp'] : \tau} \tag{4.14}$$

Il est également possible de sélectionner des éléments d'une liste, dans un intervalle donné, en respectant la grammaire suivante et en se basant sur la notion de sélection dans une liste expliquée précédemment.

```
1 INTERVALLE = "," ID "=" EXP_ARITHMETIQUE ".." EXP_ARITHMETIQUE
```

Les deux `EXP_ARITHMETIQUE` de la grammaire ci-dessus définissent l'intervalle représenté par la variable `ID`. Expliquons ce principe par des exemples :

- `billets`[i], i=1..|`billets`|, sélectionne tous les billets ;
- `billets`[i], i=|`billets`|..1, sélectionne tous les billets mais dans le sens inverse ;
- `billets`[j], j=1..|`billets`|/2, sélectionne la première moitié des billets ;
- si l'ordre n'a pas d'importance, il est possible d'écrire directement `billets` qui est sémantiquement équivalent à `billets`[i], i=1..|`billets`|.

La définition de l'intervalle doit toujours se placer en fin d'instruction, comme l'illustre l'instruction suivante :

```
1 billets[i] -> gg[i], i=1..|billets|
```

Tout attribut d'une classe dont la cardinalité maximale est supérieure à 1 est considéré comme une liste et possède, par conséquent, les mêmes propriétés.

Il est également possible de sélectionner seulement certains éléments d'une liste en fonction de critères. Cette sélection par motif s'utilise de la même manière que la navigation dans une liste à la différence qu'une expression booléenne remplace l'expression arithmétique :

```
1 LISTE_MOTIF = ID "[" EXP_BOOLEENNE "]"
```

Le résultat d'une sélection par motif est une autre liste contenant les éléments correspondant au motif, comme le définit la formule 4.15.

$$\frac{\Gamma \vdash exp : list < \omega > \quad \Gamma \vdash exp' : bool}{\Gamma \vdash exp[exp'] : list < \omega >} \tag{4.15}$$

Prenons par exemple une classe `A'` héritant de la classe `A` et une liste `l` de type `list<A>`. L'expression `l[is A']` sélectionne les éléments de `l` dont le type est `A'`. De même, l'expression `billets[auteur.nom=="Arnaud"]` ne sélectionne que les billets dont l'auteur est Arnaud.

Concernant les relations dont la cardinalité maximale est supérieure à 1, nous les définissons comme étant un type ρ tel que $\rho < \tau > \preceq list < \tau >$, où $\rho < \tau >$ définit une relation liée à une

classe cible τ. Dire qu'une relation est un sous-type d'une liste permet de l'utiliser comme s'il s'agissait d'une liste.

Forçage de type

Malan autorise le forçage de type à condition qu'une relation de sous-typage existe entre les deux types concernés, comme le stipule la formule 4.16.

$$\frac{\Gamma \vdash type : \beta \quad \Gamma \vdash exp : \beta' \quad \beta' \preceq \beta}{\Gamma \vdash (type)exp : \beta} \tag{4.16}$$

D'un point de vue syntaxique, le forçage de type proposé par Malan est similaire à celui du langage Java : dans une correspondance, tout comme dans une fonction, le forçage d'un type vers un autre s'effectue en précédant l'expression à forcer par le type final :

```
1 FORCAGE_TYPE = (TYPE)EXPRESSION
2 TYPE         = ID ("." ID)*
```

Par exemple, L'instruction `alias titre (Text)g.elements[1]`, de la figure 5.2, force l'expression `g.elements[1]` à être du type `Text` au lieu de `Element`.

4.4 Conclusion

Ce premier chapitre présente le modèle de données du langage de correspondance Malan. Dans un premier temps, le choix de travailler au niveau des schémas et d'utiliser le diagramme de classes pour les représenter a été discuté. Dans un deuxième temps, le typage du langage Malan a été décrit de manière formelle.

Ce modèle de données constitue la base sur laquelle le modèle de correspondance du langage Malan se repose. Le typage statique proposé permet aux instructions des correspondances de classes d'utiliser et de naviguer dans les classes sources et cibles. Le travail au niveau des schémas assure, quant à lui, la validité des données manipulées et l'indépendance envers les plates-formes de données.

Le chapitre suivant complète la présentation du langage Malan en décrivant son modèle de correspondance.

Chapitre 5

Modèle de correspondance du langage Malan

Sommaire

5.1 Introduction

Ce chapitre présente le modèle de correspondance du langage Malan. Il spécifie comment les éléments des schémas source et cible peuvent être mis en relation. Ce modèle de correspondance se fonde sur le modèle de données défini dans le chapitre précédent. Les critères recensés qu'un modèle de correspondance doit respecter afin de pouvoir répondre au problème du lien « données-présentation » sont les suivants :

1. *Manipulation des données sources.* Lorsque des éléments sont mis en relation, il doit être possible de traiter les données sources concernées pour, par exemple, changer le format d'une valeur (p. ex., conversion en degré d'un angle initialement défini en radian). Ce critère n'est cependant pas spécifique à un langage de correspondance.

2. *Calcul des données cibles*, dont le « layout ». Travaillant au niveau de l'interface d'un SI, le modèle de correspondance doit être capable de calculer la disposition d'éléments de celle-ci.

3. *Lien actif entre les données sources et cibles.* Toute modification de la source doit provoquer automatiquement une mise à jour de la cible. Ce critère concerne essentiellement l'implémentation du langage même si ce dernier doit être conçu dans cette optique.

Ce chapitre s'organise de la manière suivante : la section 5.2 présente les caractéristiques générales du langage Malan ; la section 5.3 introduit l'exemple utilisé tout au long de ce chapitre ; les sections 5.4, 5.5, 5.6 et 5.7 décrivent le cœur du modèle de correspondance, respectivement, les correspondances de schémas, les correspondances de classes, les instructions et les fonctions ; la section 5.8 présente le métamodèle de Malan ; la section 5.9 décrit les contraintes de validité d'une correspondance de schémas ; la section 5.10 est une discussion relative à l'évaluation du langage ; la section 5.11 présente les différents cas d'utilisation du langage Malan.

5.2 Caractéristiques du langage

En Malan, la définition d'une correspondance de schémas s'effectue de manière déclarative ; ce choix se justifie par le fait que l'ordre des fonctions et des correspondances de classes à l'intérieur d'une correspondance de schémas n'a aucune incidence sur la sémantique de cette dernière. Il en est de même concernant la définition des instructions d'une correspondance de classes. Cependant, il est parfois difficile de définir une solution entièrement déclarative à un problème (notamment lors de calculs sur des données sources ou de « layout ») ; c'est pourquoi il est possible en Malan de définir des fonctions impératives, parallèlement aux correspondances de classes.

Une correspondance de classes est une relation de type $n - 1$, c.-à-d. qu'elle établit le lien entre n éléments sources et un élément cible. Nous considérons la présentation cible comme étant l'objet central de la correspondance de schémas : un élément cible de la présentation est une vue d'un ou plusieurs éléments sources. C'est pourquoi nous estimons que les relations $n - 1$ sont plus adaptées au problème du lien « données-présentation ».

5.3 Exemple

Afin d'illustrer les caractéristiques de Malan, l'exemple de la transformation d'un blog en un document XML, présenté dans la section 3.2.3, page 38 et illustré par la figure 5.1 et par le code de la figure 5.2, est utilisé dans cette section. La partie gauche de la figure 5.1 décrit le schéma source du blog, tandis que la partie droite correspond à une sous-partie du schéma cible SVG. La correspondance de schémas est représentée par les flèches partant des classes sources du blog vers des classes cibles SVG.

En plus des exemples et des détails formels, la grammaire du langage Malan est présentée par le biais de la forme Backus-Naur qui se compose des symboles suivants :

- * : 0 élément ou plus ;
- + : 1 élément ou plus ;
- ? : 0 ou 1 ;
- () : regroupement ;
- | : alternatives ;
- guillemets simples autour des littéraux.

La grammaire de Malan est décrite de manière complète dans l'annexe A.1.

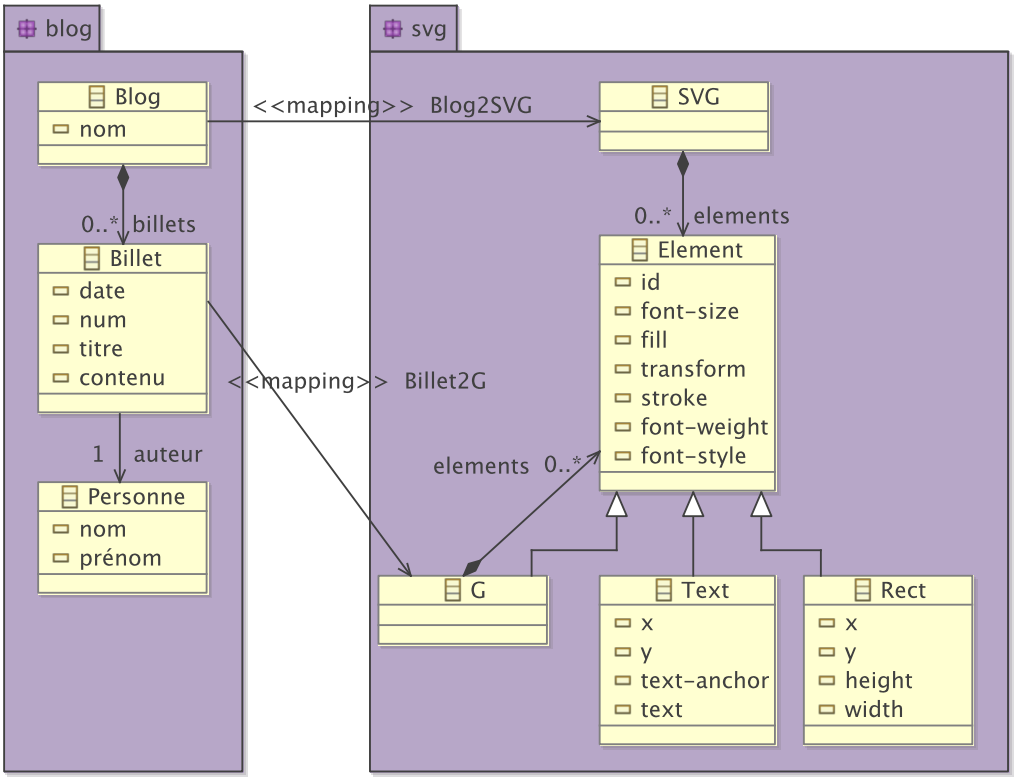


Fig. 5.1: Exemple de correspondance de schémas

```
"exemple/blog.uml/blog" -> "exemple/blog.uml/svg" {
 param couleurBillet

 blog2svg : Blog -> SVG {
   alias g            (G) elements[1]
   alias titre        (Text)g.elements[1]
   alias listeAuteurs (Text)g.elements[2]
   alias dateDernier  (Text)g.elements[3]
   alias gg           (G)g.elements[4]

   24              -> g.font-size
   "#000000"       -> g.fill
   -|billets|*140 -> titre.x
   20              -> titre.y
   "middle"        -> titre.text-anchor
   "rotate(-90)"  -> titre.transform
   nom             -> titre.text
   max(Blog.billets.date)                -> dateDernier.text
   billets.auteur.(nom+" "+prénom+", ") -> listeAuteurs
   |billets|       -> |gg|
   billets[i]      -> gg[i], i=[1..|billets|]
 }

 billet2g : Billet billet -> G {
   alias rect  (Rect)elements[1]
   alias text1 (Text)elements[2]
   alias text2 (Text)elements[3]
   alias text3 (Text)elements[4]

   couleurBillet -> rect.fill
   num        -> id
   calculerPositionY(billet, 0)  -> rect.y
   40         -> rect.x
   "#000000" -> rect.stroke
   180        -> rect.height
   630        -> rect.width
   "bold"     -> text1.font-weight
   calculerPositionY(billet, 30) -> text1.y
   60         -> text1.x
   titre      -> text1.#textContent
   calculerPositionY(billet, 95) -> text2.y
   55         -> text2.x
   contenu    -> text2.#textContent
   "italic"   -> text3.font-style
   "end"      -> text3.text-anchor
   calculerPositionY(billet, 165)-> text3.y
   620        -> text3.x
   auteur.prénom+' '+auteur.nom+' - '+date -> text[3].text
 }
 function int calculerPositionY(Billet b, int supp) {
   return (position(b)-1)*130+30+supp;
 }
}
```

Fig. 5.2: Code Malan de la correspondance de schémas de la figure 5.1

5.4 Correspondance de schémas

Cette section se consacre à la description de la correspondance de schémas dans le langage Malan. Le principe et la syntaxe de la correspondance de schémas sont tout d'abord présentés. La définition de paramètres est ensuite expliquée.

5.4.1 Principe et syntaxe

Définition : correspondance de schémas. *Soient $\mathcal{S}$ et $\mathcal{T}$ deux diagrammes de classes UML. Une correspondance de schémas est un prédicat $\mathcal{CS}(\mathcal{S}, \mathcal{T}, \Sigma)$ où $\mathcal{S}$ est appelé le schéma source et $\mathcal{T}$ le schéma cible. Σ est un ensemble des correspondances de classes décrites dans une logique $\mathcal{L}$ sur $\langle \mathcal{S}, \mathcal{T} \rangle$ et dont le but est de créer un lien entre les schémas source et cible.*

D'un point de vue syntaxique, une correspondance de schémas se définit par un en-tête et un corps. L'en-tête spécifie le chemin des diagrammes de classes UML source (le premier `CHEMIN_UML`) et cible (le second `CHEMIN_UML`), ainsi que le nom optionnel de la correspondance de schéma (`ID`). Le corps, encadré par les deux accolades, contient quant à lui la définition des correspondances de classes et des fonctions. Le tout doit respecter la grammaire suivante :

```
1 CORRESP_SCHEMAS = (ID ":")? CHEMIN_UML "->" CHEMIN_UML
2                   "{" (CORRESPONDANCE_CLASSE | FONCTION | PARAMETRE)* "}"
```

où `CHEMIN_UML` doit respecter la grammaire ci-dessous dans laquelle `CHEMIN` spécifie le chemin absolu ou relatif du fichier UML, `PACKAGE_UML` le nom du package UML du diagramme de classes concerné et `SEPARATEUR` le caractère de séparation de fichier. Plusieurs packages d'un même diagramme peuvent être spécifiés en les séparant par un point-virgule.

```
1 CHEMIN_UML = """ CHEMIN SEPARATEUR PACKAGE_UML (";" PACKAGE_UML)* """
```

Le code Malan suivant présente un exemple de définition de correspondance de schémas :

```
1 "exemple/blog.uml/blog" -> "exemple/blog.uml/svg" {
2   //...
3 }
```

5.4.2 Paramètre

Une correspondance de schémas Malan autorise la déclaration de paramètres utilisés comme des alias dans les correspondances de classes. Ce principe est notamment nécessaire dans le contexte d'un lien « données-présentation » où un utilisateur d'un SI peut faire varier certains paramètres de la présentation *via* l'interface.

```
1 "exemple/blog.uml/blog" -> "exemple/blog.uml/svg" {
2   param couleurBillet
3   billet2g : Billet billet -> G {
4     //...
5     couleurBillet -> rect.fill
6   }
7   //...
8 }
```

Le code ci-dessus, extrait de la figure 5.2, présente un exemple de déclaration et d'utilisation d'un paramètre. L'instruction `couleurBillet -> rect.fill`, ligne 5, utilise comme donnée source le paramètre `couleurBillet` déclaré à la ligne 2. La définition d'un paramètre doit respecter la grammaire suivante dans laquelle `ID` correspond au nom du paramètre.

```
1 PARAMETRE = "param" ID
```

Un paramètre se déclare dans une correspondance de schémas et en parallèle aux correspondances de classes. Il est considéré comme une donnée source supplémentaire venant compléter les données sources du SI.

5.5 Correspondance de classes

Cette section présente tout d'abord le principe et la syntaxe d'une correspondance de classes. Les différentes caractéristiques de la correspondance de classes sont ensuite décrites, à savoir l'utilisation d'alias, la navigation, la dépendance et le filtrage par motif.

5.5.1 Principe et syntaxe

Définition : correspondance de classes. *Soient $\mathcal{S}$ et $\mathcal{T}$ deux diagrammes de classes UML. Une correspondance de classes est un prédicat $\mathcal{CC}(s_1, \ldots, s_n, t, \Omega)$ tel que $(s_1, \ldots, s_n) \in \mathcal{S}$ et $C(s_i)$, avec i de 1 à n, $t \in \mathcal{T}$ et $C(t)$. Ω est un ensemble de prédicats, appelés « instructions », décrits dans une logique $\mathcal{L}$ sur $\langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{T} \rangle$, établissant des liens entre les classes sources $(s_1, \ldots, s_n)$ et la classe cible t.*

Une correspondance de classes possède la même structure qu'une correspondance de schémas : elle se compose d'un en-tête définissant des critères de sélection, le nom facultatif ainsi que les classes sources et l'unique classe cible, et d'un corps contenant un ensemble d'instructions. La grammaire d'une correspondance de classes est la suivante :

```
1  CORRESPONDANCE_CLASSE = (NOM_CORRESP ":")? CLASSE_SRC ALIAS_SRC?
2                          ("," CLASSE_SRC ALIAS_SRC?)* FILTRE? "->"
3                          CLASSE_CIBLE ALIAS_CIBLE? "{" INSTRUCTION* "}"
4  INSTUCTION      = CORRESPONDANCE_RELATIONS | CORRESPONDANCE_ATTRIBUTS
5                    INSTRUCTION_CONDITIONNELLE | REGROUPEMENT | ALIAS
6  CHEMIN_CLASSE   = ID ("." ID)*
7  ALIAS_CIBLE     = ID
8  ALIAS_SRC       = ID
9  NOM_CORRESP     = ID
10 CLASSE_SRC      = CHEMIN_CLASSE
11 CLASSE_CIBLE    = CHEMIN_CLASSE
```

où le premier `NOM_CORRESP` correspond au nom de la correspondance, `CLASSE_SRC` à ceux des classes sources. `CLASSE_CIBLE` spécifie la classe cible ainsi que son origine (*cf.* section 5.5.4). Il est possible de définir des alias, qui suivent le nom des classes (`ALIAS_SRC` et `ALIAS_CIBLE` de `CORRESPONDANCE_CLASSE`), pour utiliser les classes spécifiées sous un autre nom. Le filtre correspond à une instruction booléenne définissant les critères de sélection des classes sources (voir section 5.5.5). L'exemple suivant présente la déclaration d'une correspondance de classes

`Blog2SVG` ayant comme classe source `Blog` et comme cible `SVG`, dont les alias sont respectivement `bl` et `s`. L'expression `|bl.billets|>0` stipule que la correspondance de classes concerne uniquement les objets de type `Blog` dont le nombre de billets est supérieur à 0.

```
1 Blog2SVG : Blog bl [|bl.billets|>0] -> SVG s {
2   // Définition des instructions.
3 }
```

5.5.2 Alias

Un alias permet d'utiliser un attribut, une classe, une relation, un calcul ou une fonction sous un autre nom afin de simplifier l'écriture d'instructions. Il ne s'agit en rien d'une variable mais d'un raccourci d'écriture. La déclaration d'un alias peut en utiliser d'autres et s'effectue n'importe où dans une correspondance de classes, parallèlement à ses instructions, de la manière suivante :

```
1 ALIAS = "alias" ID EXPRESSION
```

où `ID` correspond au nom de l'alias, devant être unique dans la correspondance de classes, et `EXPRESSION` à sa valeur pouvant être :

- Une classe déclarée dans l'en-tête de la correspondance de classes, notamment pour éviter des ambiguïtés comme l'illustre l'exemple suivant dans lequel les deux classes sources portent le même nom :

```
1 package1.A, package2.A -> B {
2   alias A1 package1.A
3   alias A2 package2.A
4   //...
5 }
```

  Cette utilisation est équivalente à celle déclarant le nom de l'alias juste après la déclaration des classes :

```
1 package1.A A1, package2.A A2 -> B {
2   //...
3 }
```

- Un attribut ou une relation dont l'accès à partir de classes de la correspondance de classes courante est fastidieux, comme l'illustre l'exemple suivant :

```
1 Blog -> SVG {
2   alias g     (G) elements[1]
3   alias titre (Text)g.elements[1]
4
5   "middle" -> titre.text-anchor
6   //...
7 }
```

- Un calcul arithmétique ou le résultat d'une fonction. Dans ce cas, l'expression contient uniquement des éléments provenant des classes sources de la correspondance de classes, des fonctions et des constantes :

```
1 Blog -> SVG {
2   alias listeAuteurs billets.auteur.(nom+" "+prénom+", ")
3   alias date         max(Blog.billets.date)
4   //...
5 }
```

5.5.3 Navigation

Pour simplifier la définition des instructions des correspondances de classes, il est possible d'accéder, à partir d'une classe ou d'une relation, à leurs attributs et relations en utilisant l'opérateur « . ». Par exemple, l'expression `Blog.billets` accède à la relation `billets` de la classe `Blog`. Le code Malan suivant, extrait de la correspondance de classes `blog2SVG` définie dans la figure 5.2, est utilisé dans cette section afin d'illustrer le principe de la navigation.

```
1  blog2svg : Blog -> SVG {
2    //...
3    nom                                    -> titre.text
4    max(Blog.billets.date)                 -> dateDernier.text
5    billets.auteur.(nom+" "+prénom+", ") -> listeAuteurs
6  }
7  billet2g : Billet -> G {
8    //...
9    auteur.prénom+' '+auteur.nom+' - '+date -> text[3].text
10 }
```

Eléments de référence. Lorsqu'un identifiant v est utilisé en début d'expression sans aucun préfixe (par exemple `nom` et `titre` à la ligne 3 du code précédent), il est nécessaire d'en connaître l'origine (s'agît-il d'un alias, d'un attribut, *etc.* ?). Nous appelons *éléments de référence*, les objets servant à identifier l'origine de v. Dans une correspondance de classes, il existe par défaut deux ensembles d'éléments de référence : les éléments de référence sources $R_s = \{alias_s, classes_s\}$ et cibles $R_c = \{alias_c, classe_c\}$, où $alias_s$ et $alias_c$ sont les alias sur des éléments sources ou cibles, et $classes_s$ et $classe_c$ les classes sources et la classe cible de la correspondance de classes. Par exemple, l'instruction `auteur.prénom+' '+auteur.nom+' - '+date` contient deux identifiants sans préfixe dont il faut trouver l'origine : `auteur` et `date`. Pour cela, il faut tout d'abord vérifier s'ils correspondent au nom d'une des classes sources. Si la recherche ne donne rien, l'existence d'un attribut ou d'une relation portant leur nom va être ensuite vérifiée dans les classes sources. Si la recherche ne donne rien, ils seront recherchés parmi les alias. Dans le cas présent, les deux identifiants correspondent à deux attributs de la classe `Blog`.

Il en est de même pour les opérations d'une classe et les fonctions : la ligne 4 démarre par la fonction `max(...)` ; il s'agit soit d'une opération de la classe `Blog`, soit d'une fonction (les fonctions sont expliquées dans la section 5.7). En l'occurrence, étant donné que `Blog` ne possède pas d'opération `max`, il s'agit de la fonction prédéfinie `max`.

Lorsqu'une expression débute par un attribut, une relation ou une opération, il est conseillé d'ajouter le nom de la classe correspondante en préfixe afin de faciliter la compréhension du code. La ligne 4 possède, par exemple, l'expression `Blog.billets.date`, qui aurait également pu s'écrire `billets.date` mais qui a l'avantage d'être plus facilement compréhensible. La formule

5.1 spécifie la sémantique d'une expression de navigation lorsque son premier identifiant est une classe.

$$\frac{\Gamma \vdash exp : \kappa \quad \Gamma \vdash exp' : \omega}{\Gamma \vdash exp.exp' : \omega} \tag{5.1}$$

De plus, l'ajout de ce préfixe pour s'avérer obligatoire pour désambiguïser une expression, par exemple lorsque plusieurs classes possèdent des attributs ou/et des relations ayant le même nom.

Navigation dans les relations. La navigation dans une relation s'effectue de deux manières différentes. Premièrement, de manière identique aux listes, l'utilisation d'un indice définit la position de l'élément souhaité de la relation. Lorsque la cardinalité d'une relation est `0..1` ou `1..1`, la navigation dans celle-ci s'effectue comme s'il s'agissait d'une classe et non d'une liste. Par exemple, `billets[1].auteur.nom` accède à l'attribut `nom` de l'auteur du premier billet. `billets[1]` cible le premier élément de la relation `billets` (de cardinalité `0..n`), tandis que la relation `auteur`, de cardinalité `1..1`, est utilisée comme une classe. La formule 5.2 définit le typage d'une expression de navigation fondée sur une relation de cardinalité `1..1`, où $\#_{max}\{max\} = 1$ stipule que la cardinalité maximale de la relation *exp* est égale à 1.

$$\frac{\Gamma \vdash exp : \rho \quad \Gamma \vdash exp' : \omega \quad \#_{max}\{exp\} = 1}{\Gamma \vdash exp.exp' : \omega} \tag{5.2}$$

Il est également possible de ne pas utiliser d'indice lors de la navigation dans une relation de cardinalité `m..n` (avec $n > 1$ et $m \leq n$) ; dans ce cas, la valeur retournée n'est pas un élément unique mais est un ensemble d'éléments.

$$\frac{\Gamma \vdash exp : \rho \quad \Gamma \vdash exp' : \tau \quad \#_{max}\{exp\} > 1}{\Gamma \vdash exp.exp' : list < \tau >} \tag{5.3}$$

Par exemple, l'expression `billets.auteur.nom`, dont le typage est défini par la formule 5.3, retourne la liste du nom de l'auteur de chaque billet et est équivalente au pseudo-code suivant :

Algorithme 5.1 : Récupération du nom de l'auteur de chaque billet

```
1 début
2 |  list l = ∅
3 |  pour i de 1 à /billets/ faire
4 |  |  l = l + billets[i].auteur.nom
5 |  fin
6 |  retourner l
7 fin
```

Lorsque plusieurs relations de cardinalité `*..n` sont utilisées sans indice ou avec des intervalles dans une même expression, il en résulte une liste aplatie, dont le typage est défini par la formule 5.4.

$$\frac{\Gamma \vdash exp : \rho \quad \Gamma \vdash exp' : \rho' < \tau > \quad \#_{max}\{exp\} > 1 \quad \#_{max}\{exp'\} > 1}{\Gamma \vdash exp.exp' : list < \tau >} \tag{5.4}$$

Supposons par exemple que la relation `auteur` soit en fait une relation, appelée `auteurs`, de cardinalité `*..n` et non `1..1` ; dans ce cas, l'expression `billets.auteurs.nom` retourne la liste du nom des auteurs de chaque billet, en suivant l'algorithme suivant :

Algorithme 5.2 : Récupération du nom des auteurs de chaque billet

```
1 début
2 |  list l = ∅
3 |  pour i de 1 à /billets/ faire
4 |  |  pour j de 1 à /billets[i].auteurs/ faire
5 |  |  |  l = l + billets[i].auteurs[j].nom
6 |  |  fin
7 |  fin
8 |  retourner l
9 fin
```

Factorisation. L'instruction `billets.auteur.(nom+" "+prénom+", ")` (ligne 5) fournit la liste du nom adjoint au prénom des auteurs des billets. Le but de cette syntaxe de navigation est de factoriser le code d'une instruction lors de calculs arithmétiques. La grammaire suivante définit le format d'une expression de factorisation :

```
1 EXP_FACTOR = EXP_SELECTION "." "(" EXP_ARITHMETIQUE ")"
```

Une factorisation $exp.(exp')$ change les éléments de référence sources R_s en y ajoutant temporairement, jusqu'à la fin de l'expression de factorisation, la classe ou la relation issue de l'expression exp, que le notera $\mathcal{F}(exp)$, comme le résume la formule 5.5 :

$$exp.(exp') \,|\, \Gamma \vdash exp : (\kappa \vee \rho) \rightarrow R'_s = R_s \cup \mathcal{F}(exp) \tag{5.5}$$

Ainsi, les éléments de la classe ou de la relation $\mathcal{F}(exp)$ peuvent être directement utilisés pendant la factorisation : dans l'instruction `billets.auteur.(nom+" "+prénom+", ")`, la classe `Auteur` est, par exemple, ajoutée aux éléments de référence sources afin d'utiliser les attributs `nom` et `prénom` sans préfixe.

Le typage de ce genre d'expression est défini par les formules 5.6, 5.7 et 5.8 qui concernent, respectivement, les expressions de factorisation fondées sur une relation `1..1`, une relation `n..m` et une classe.

$$\frac{\Gamma \vdash exp : \rho \quad \Gamma \vdash exp' : \phi \quad \#_{max}\{exp\} = 1}{\Gamma \vdash exp.(exp') : \phi} \tag{5.6}$$

$$\frac{\Gamma \vdash exp : \rho \quad \Gamma \vdash exp' : \phi \quad \#_{max}\{exp\} > 1}{\Gamma \vdash exp.(exp') : list < \phi >} \tag{5.7}$$

$$\frac{\Gamma \vdash exp : \kappa \quad \Gamma \vdash exp' : \phi}{\Gamma \vdash exp.(exp') : \phi} \tag{5.8}$$

5.5.4 Dépendance d'une correspondance de classes

Il est nécessaire dans certains cas de spécifier une partie de l'origine d'une classe cible d'une correspondance de classes. La figure 5.3, par exemple, définit un fragment de diagramme représentant une partie d'une correspondance de schémas dans laquelle la classe `Face` est reliée à une autre classe par la relation `droite`. Puisque `Face` ne possède qu'une seule relation, la correspondance de classes fait par conséquent référence aux objets `Face` de cette relation `droite`.

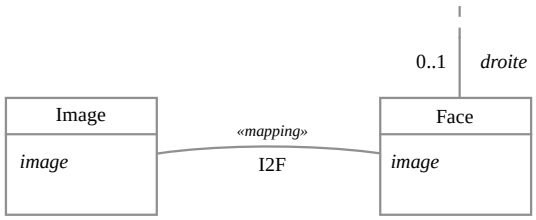


Fig. 5.3: Exemple de dépendance implicite

Lorsque plusieurs relations sont reliées à une classe, il est possible de définir plusieurs correspondances de classes utilisant cette classe (par exemple, une correspondance de classes pour chaque relation), comme l'illustre la figure 5.4. Dans un tel cas, il est alors nécessaire de spécifier explicitement la relation dont dépend chaque correspondance de classes.

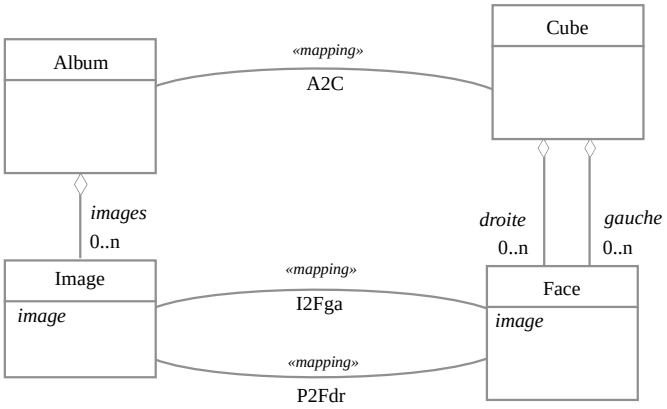


Fig. 5.4: Exemple de dépendance explicite

Le code suivant présente la correspondance de schémas de la figure 5.4, où `P2Fga` et `P2Fdr` définissent explicitement la relation dont elles dépendent (respectivement les agrégations `gauche` et `droite`) en ajoutant comme préfixe à la classe cible la relation concernée ainsi que la classe opposée, comme l'illustre les lignes 5 et 8 du code ci-dessous.

```
1 A2C : Album -> Cube {
2   images   -> droite
3   images   -> gauche
4 }
5 I2Fdr : Image -> Cube.droite.Face {
6   image -> image
```

```
7  }
8  I2Fga : Image -> Cube.gauche.Face {
9    negatif(image) -> image
10 }
```

La ligne 5 du code précédent définit que la cible de la correspondance de classes `I2Fdr` est la classe `Face` de la relation `droite`. Le préfixe `Cube.droite` ne doit pas être utilisé lors de l'utilisation de la classe `Face`. Par exemple, l'accès à l'attribut `image` de la classe `Face` ne change pas (il s'effectue toujours *via* `Face.image` ou `image`).

5.5.5 Filtrage par motif

Il est possible de spécifier un filtre lors de la définition d'une correspondance de classes de manière à restreindre les instances des classes sources concernées uniquement à celles respectant les critères du filtre. Ce principe est identique au filtrage par motif des listes (*cf.* section 4.3, page 55).

```
1 Billet [prénom=="Arnaud" && nom=="Blouin"] -> G {
2   alias nom Billet.auteur.nom
3   alias prénom Billet.auteur.prénom
4   //...
5 }
```

Le code Malan ci-dessus est un exemple de définition d'un filtre dont la condition à respecter, `prénom=="Arnaud" && nom=="Blouin"`, stipule que seules les instances de la classe `Billet` dont l'auteur est Arnaud Blouin sont concernées par la correspondance de classes. La grammaire suivante décrit le format d'un filtre qui se compose d'une paire de crochet encadrant une expression booléenne.

```
1 FILTRE = "[" EXPRESSION_BOOLEENNE "]"
```

5.6 Instruction

Cette section détaille les différents types d'instructions qu'une correspondance de classes peut contenir : les regroupements d'instructions, les correspondances de relations et d'attributs, et les instructions conditionnelles.

5.6.1 Regroupement d'instructions

Toujours dans l'optique de spécifier des correspondances de schémas de manière lisible et facilement compréhensible, le langage Malan rend possible la définition de regroupement d'instructions.

Un regroupement d'instructions réunit un ensemble d'instructions dans lequel l'élément cible par défaut n'est plus R_s, mais la classe cible spécifiée au début du bloc, dont le type est défini par la formule 5.9.

$$\frac{exp\{exp'\}}{\Gamma \vdash exp : \kappa} \tag{5.9}$$

Par exemple, le code Malan de la figure 5.2, qui n'utilise pas de regroupement, définit les éléments de `rect` de la manière suivante :

```
1 "#e6e6e6" -> rect.fill
2 "#000000" -> rect.stroke
3 180       -> rect.height
```

Les trois instructions ci-dessus utilisent le même préfixe, à savoir l'alias `rect` ; il est dans ce cas possible de les regrouper sous la forme d'un seul bloc ayant comme élément cible de référence `rect`, comme l'illustre le code suivant :

```
1 rect {
2   "#e6e6e6" -> fill
3   "#000000" -> stroke
4   180       -> height
5 }
```

Dans le regroupement `rect` précédent, les éléments de référence cibles R_s sont remplacés par l'élément cible du regroupement R'_s, comme le spécifie la formule 5.10 :

$$exp\{exp'\} \,|\, \Gamma \vdash exp : \kappa \implies R'_s = \{\mathcal{F}(exp)\} \tag{5.10}$$

Ainsi, seuls les éléments accessibles à partir de `rect` peuvent être utilisés ; les alias portant sur d'autres éléments cibles sont également inutilisables. De plus, tout alias défini dans un regroupement est utilisable uniquement dans les instructions du regroupement. Un regroupement peut en contenir d'autres comme le définit la grammaire ci-dessous, où `ID` est l'élément de référence du regroupement, qui ne peut être qu'un élément du schéma cible.

```
1 REGROUPEMENT = EXPRESSION_SELECTION (","  INTERVALLE)?
2                "{" INSTRUCTION* "}"
```

Le code suivant correspond à une partie de la correspondance de classes `billet2g` de la figure 5.2 à la différence que les instructions ont été factorisées en différents regroupements : `rect` de la ligne 6 à 13 et `text1` de la ligne 15 à 23.

```
1  billet2g : Billet -> G {
2    alias rect  (Rect)elements[1]
3    alias text1 (Text)elements[2]
4    // Premier bloc : l'élément cible par défaut est maintenant "rect"
5    // et non plus "G".
6    rect {
7      (position(billet)-1)*130+30  -> y
8      40                 -> x
9      "#e6e6e6"          -> fill
10     "#000000"          -> stroke
11     180                -> height
12     630                -> width
13   }
14   // Second bloc : "text1" est le nouvel élément cible par défaut.
15   text1 {
16     "bold"             -> font-weight
17     (position(billet)-1)*130+60  -> y
```

```
18      60          -> x
19      titre       -> #textContent
20      (position(billet)-1)*130+125 -> y
21      55          -> x
22      contenu     -> #textContent
23    }
24    //...
25  }
```

Le regroupement d'instructions s'applique également aux listes d'instances d'une classe. Par exemple, le regroupement suivant s'applique à chaque instance `Element` de la relation `elements`.

```
1  elements[i], i=1..|elements| {
2    i -> id
3  }
```

Malan n'autorise pas la définition d'un regroupement ayant comme élément de référence un élément source pour plusieurs raisons. Tout d'abord, un des principes fondamentaux de Malan est de se focaliser sur les éléments du schéma cible plutôt que sur ceux du schéma source : une correspondance de classes utilise un ou plusieurs éléments sources pour définir un seul élément cible. Les regroupements d'instructions vont dans ce sens en utilisant comme point central des instructions d'un regroupement l'élément cible de référence. Utiliser un élément source comme référence serait contraire à ce principe. De plus, il serait, par conséquent, confondant et éventuellement ambigu de pouvoir utiliser un élément source ou cible comme référence.

5.6.2 Correspondance de relations et d'attributs

Une classe cible possède des attributs et des relations qu'il est nécessaire de lier avec des classes sources. Les correspondances de relations et d'attributs répondent à ce besoin.

Correspondance de relations. Lorsque qu'une classe cible possède une relation, il est nécessaire d'en spécifier la cardinalité, les éléments qui la compose, ainsi que leur ordre. Nous appelons ce processus, *la correspondance de relations*, que nous formalisons à l'aide du prédicat $\mathcal{CR}(s_1, \ldots, s_n, c)$, défini ci-dessous, dans lequel c symbolise la classe cible de la relation à définir, et $s_1, \ldots, s_n$ les classes sources à utiliser pour définir c.

$$\mathcal{CR}(s_1, \ldots, s_n, c) \implies \bigwedge_{i=1}^{n} \mathcal{C}(s_i) \wedge \mathcal{C}(c) \wedge \{s_1, \ldots, s_n\} \in \mathcal{S} \wedge c \in \mathcal{T} \tag{5.11}$$

L'instruction à la ligne 5 du code ci-dessous, tiré de l'exemple de la figure 5.2, est une correspondance de relations liant la composition `billets` de la classe `Blog`, avec la composition `elements` de la classe `G` (voir la figure 5.1). De manière littérale, cette instruction se lit « pour tout billet de la relation `billets` à la position i, $i \in [1, |billets|]$, il existe un objet de type `Element` à cette même position i dans la relation `elements` ».

```
1  blog2svg : Blog -> SVG {
2    //...
3    alias gg (G)g.elements[4]
4    //...
5    billets[i]  -> gg[i], i=[1..|billets|]
```

```
6 }
```

Une telle instruction peut être établie uniquement s'il existe une correspondance de classes mettant en relation un `Billet` avec une sous-classe d'`Element` (la classe `Element` étant abstraite), telle que la correspondance de classes `Billet2G : Billet -> G`. Le principe de la navigation, expliqué dans la section 5.5.3, est utilisé pour ordonnancer les éléments cibles. La grammaire d'une correspondance de relations est spécifiée ci-dessous.

```
1 CORRESPONDANCE_RELATIONS = EXPRESSION_SELECTION (","  EXPRESSION_SELECTION)*
2                            "->" EXPRESSION_SELECTION (","  INTERVALLE)?
```

La formule 5.12 définit le typage d'une correspondance de relations où le type des expressions sources et de l'expression cible est le type « classe ». Nous parlons ici de « classe » et non de « relation » car le type d'une expression telle que `gg[i]` est une classe de type `Element` et non une relation.

$$\frac{exp_1, \ldots, exp_n \longrightarrow exp \quad \Gamma \vdash exp : \kappa}{\Gamma \vdash exp_1 : \kappa_1 \ldots \Gamma \vdash exp_n : \kappa_n} \tag{5.12}$$

Concernant la définition de la cardinalité d'une relation, cette étape est facultative mais est cependant conseillée lorsque la correspondance de relations devient complexe.

```
1 |billets| -> |gg|
2
3 billets[i]            -> gg[i]              , i=[1..|billets|/2]
4 billets[|billets|-i] -> gg[|billets|/2+i], i=[1..|billets|/2]
```

L'exemple ci-dessus montre comment définir une correspondance de relations en plusieurs étapes : la première, ligne 3, spécifie que pour tout billet à la position i, $i \in [1, |billets|/2]$, il existe un élément à la même position i ; la seconde, ligne 4, stipule que pour tout billet à la position i, i de $|billets|$ à $|billets|/2$ inclus, il existe un élément à la position j, $j \in [|billets|/2, |billets|]$. L'instruction `|billets| -> |gg|`, qui stipule que la cardinalité de `gg` est égale à celle de `billets`, est facultative mais apporte plus de clarté sur le nombre d'éléments de la relation `gg`.

Dans certains cas, il est nécessaire d'utiliser une classe cible comme source d'une instruction de relations. Ce type d'instructions, appelé *correspondance de relations cible-cible*, s'applique dans une seule situation : lorsque la relation à définir fait référence à un objet cible déjà spécifié. Par exemple, la classe `Noeud` possède une relation fils et un relation parent. Lorsqu'un nœud $n_1$définit ses fils, ces derniers doivent spécifier que leur parent est le nœud $n1$. Le code suivant illustre ce principe en décrivant une correspondance de classes dont la classe cible est `Noeud`. La ligne 3 parcourt, à l'aide d'un regroupement d'instructions, les instances `Noeud` de la relation fils de n_1. La ligne 4 définit que le parent de chacune de ces instances est n_1.

```
1 ... -> Noeud n1 {
2   //...
3   fils[i], i=1..|fils| {
4     n1 -> parent
5   }
6 }
```

Pour une correspondance de relations cible-cible, il ne doit pas exister de correspondance de classes associée. La formule 5.13 définit formellement ce type de correspondance qui ne peut utiliser en tant que « source », qu'une seule classe provenant du schéma cible.

$$\mathcal{CRCC}(c_s, c_c) \implies \mathcal{C}(c_s) \wedge \mathcal{C}(c_c) \wedge \{c_s, c_c\} \in \mathcal{T} \tag{5.13}$$

Correspondance d'attributs. Une correspondance d'attributs lie des constantes ou des éléments de ces classes sources à un élément de la classe cible. Par exemple, l'instruction précédente met en relation le nombre de billets du blog avec la position horizontale du titre de la vignette SVG cible :

```
1 -|billets|*140 -> titre.x
```

Le type de l'attribut cible de cet exemple est un type primitif. Dans un tel cas, la correspondance d'attributs possède la sémantique décrite par la formule 5.14 spécifiant que lorsque le type de l'attribut cible est un type primitif, celui de l'expression source est soit un type primitif, soit un attribut dont le type est compatible avec celui de la cible.

$$\frac{exp \longrightarrow exp' \quad \Gamma \vdash exp' : \upsilon \quad i' \leq \#\{\upsilon \,|\, \mathcal{A}(exp', \upsilon)\} \leq j'}{\Gamma \vdash exp : \omega \quad \omega \preceq \upsilon} \tag{5.14}$$

La grammaire ci-dessus définit le format d'une correspondance d'attributs, où l'expression de sélection spécifie l'attribut de la classe cible concernée mis en relation avec l'expression de gauche. Lorsque la cardinalité maximale de l'attribut cible est supérieure à 1, il est alors possible de définir un intervalle de parcours de la même manière que pour les listes et les relations.

```
1 CORRESPONDANCE_ATTRIBUTS = EXPRESSION "->"
2                            EXPRESSION_SELECTION (",", INTERVALLE)?
```

Lorsque le type d'un attribut est une classe, la correspondance d'attributs doit respecter la même grammaire et la même sémantique (*cf.* formule 5.12) que celles définies pour une correspondance de relations.

5.6.3 Instruction conditionnelle

Il est possible de définir un branchement conditionnel dans une correspondance de classes de manière identique à celle du langage Java. Le code suivant définit, par exemple, la couleur de fond d'un rectangle SVG en fonction de l'auteur du billet d'un blog.

```
1 if(billet.auteur=="Arnaud Blouin") {
2   "red"  -> rect.fill
3 }else if(billet.auteur=="Olivier Beaudoux")
4   "blue" -> rect.fill
5 else
6   "#e6e6e6" -> rect.fill
```

La grammaire d'une instruction conditionnelle est la suivante.

```
1 EXPRESSION_CONDITIONNELLE =
2                 "if" "(" EXPRESSION_BOOLEENNE ")" INSTRUCTIONS_SI_SINON
3                 ("else" INSTRUCTIONS_SI_SINON)?
4 INSTRUCTIONS_SI_SINON      = ("{" INSTRUCTION* "}") | INSTRUCTION
```

5.7 Fonction

Le but d'une fonction Malan est de réaliser un calcul destiné à être utilisé dans une instruction d'une correspondance de classes ou dans une autre fonction, cela afin de simplifier le code d'une correspondance de schémas. Malan fournit un ensemble de fonctions de base prédéfinies comme les fonctions `max`, `min` et `abs` pour le calcul arithmétique ; `sort`, `invert` et `sub` pour la manipulation d'ensembles ; ou encore `concat`, `toLowerCase` et `length` pour la manipulation de chaînes de caractères. De plus, il est possible de définir ses propres fonctions en parallèle des correspondances de classes dans une correspondance de schémas. Les fonctions sont entièrement impératives et leur syntaxe est proche de celle des langages Java et C#. La grammaire de l'en-tête d'une fonction est la suivante :

```
1 "function" TYPE ID
2 "(" TYPE ID ("," TYPE ID)* ")"
3 "{" FCT_INSTRUCTION* "}"
```

où le `TYPE` et le `ID` de la ligne 1 correspondent respectivement au type de l'objet retourné et au nom de la fonction. La ligne 2 définit les paramètres de la fonction où `TYPE ID` définissent respectivement le type et le nom du paramètre. Le corps d'une fonction, ligne 3, se compose d'un ensemble fini d'instructions impératives dont la grammaire est définie dans l'annexe A.1. La formule 5.15 décrit le typage d'une fonction.

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash o : \tau' \quad \tau' \preceq \tau}{\Gamma \vdash t\ function\ ID(\ldots)\{\ldots\ return\ o;\} : \tau} \tag{5.15}$$

Pour illustrer la définition d'une fonction, le code suivant correspond à une implémentation possible de la fonction prédéfinie `sub`, qui crée un sous-ensemble de la liste `l` donnée en paramètre, dont `début` et `fin` en sont respectivement les bornes inférieure et supérieure.

```
1  function list sub(list l, int début, int fin){
2    if(début>fin || début<1 || fin>|l|)
3      error("Paramètre(s) invalide(s)");
4
5    list sub = nil;
6
7    for(int i=début; i<=fin; i++)
8      sub = sub + l[i];
9
10   return sub;
11 }
```

La ligne 1 du code ci-dessus déclare l'en-tête de la fonction `sub` qui retourne une liste non typée. Cette fonction possède comme paramètres une liste non typée `l`, ainsi que les positions de départ et de fin des éléments à considérer. Les lignes 2 et 3 vérifient la validité des paramètres, tandis que la ligne 5 initialise la liste `sub` comme étant une liste vide. Cette nouvelle liste est remplie aux lignes 7 et 8 avant d'être retournée (ligne 10).

```
1 billet2G : Billet -> G {
2   //...
3   calculerYRect(Billet) -> rect.y
```

```
4 }
5
6 function int calculerYRect(Billet b) {
7   return (position(b)-1)*130+30 ;
8 }
9 //...
```

Le code précédent illustre l'organisation des fonctions par rapport aux correspondances de classes : la fonction `calculerYRect` est spécifiée dans la correspondance de schémas au même titre que la correspondance de classes `billet2G` qui l'utilise. La formule 5.16 définit le typage de l'appel d'une fonction.

$$\frac{\{\tau_1 \times \ldots \times \tau_n \rightarrow \tau\} \quad \Gamma \vdash p_1 : \tau'_1 \ldots \Gamma \vdash p_n : \tau'_n \quad \tau'_1 \preceq \tau_1 \ldots \tau'_n \preceq \tau_n \quad \Gamma \vdash exp : \delta}{\Gamma \vdash exp(p_1 \ldots, p_n) : \tau} \tag{5.16}$$

5.8 Synthèse

La figure 5.5 présente, sous la forme d'un métamodèle, la synthèse des éléments du langage Malan décrits dans les sections précédentes. L'élément racine de ce métamodèle est la correspondance de schémas utilisant un diagramme de classes source et un autre cible, et possédant un ensemble de correspondances de classes, de fonctions et de paramètres. De même, une correspondance de classes se compose de classes sources et d'une classe cible issues, respectivement, du diagramme source et cible ; elle contient également un ensemble d'instructions destinées à mettre en relation les éléments des classes sources avec ceux de la classe cible.

5.9 Contraintes de validité d'une correspondance de schémas

Dans cette section, nous décrivons, les différentes règles qu'une correspondance de schémas doit respecter.

Contrainte 1 : présence d'une correspondance de classes. La figure 5.6 précise la première contrainte pour notre exemple du blog. La correspondance de classes $\mathcal{CC}(Blog, SVG, \Omega)$ possède la correspondance de relations $\mathcal{CR}(Billet, G)$ parmi ses instructions Ω : une correspondance de classes $\mathcal{CC}(Billet, G, \Omega')$ fondée sur ces mêmes classes *Billet* et G doit alors être déclarée.

De manière générale, la formule 5.17 formalise cette contrainte en stipulant que pour toute correspondance de relations incluse dans les instructions d'une correspondance de classes, il existe une autre correspondance de classes conforme aux classes utilisées dans la première :

$$\mathcal{CC}(s_1, \ldots, s_n, c, \Omega) \wedge \mathcal{CR}(s'_1, \ldots, s'_n, c') \in \Omega \implies \exists \mathcal{CC}(s'_1, \ldots, s'_n, c', \Omega') \tag{5.17}$$

Contrainte 2 : présence d'une correspondance de relations. La figure 5.7 introduit le principe de la seconde contrainte où $\mathcal{CC}(s_1, \ldots, s_n, Livre, \Omega)$ est une correspondance de classes dont la classe cible est *Livre*, et $\mathcal{R}(Livre, Page)$ une composition dont la cardinalité minimale est supérieure à 0 et définissant qu'un livre se compose d'un moins une page.

Puisqu'il existe une correspondance de classes $\mathcal{CC}(s_1, \ldots, s_n, Livre, \Omega)$ utilisant *Livre* comme classe cible, il doit alors exister une correspondance de relations parmi les instructions Ω et,

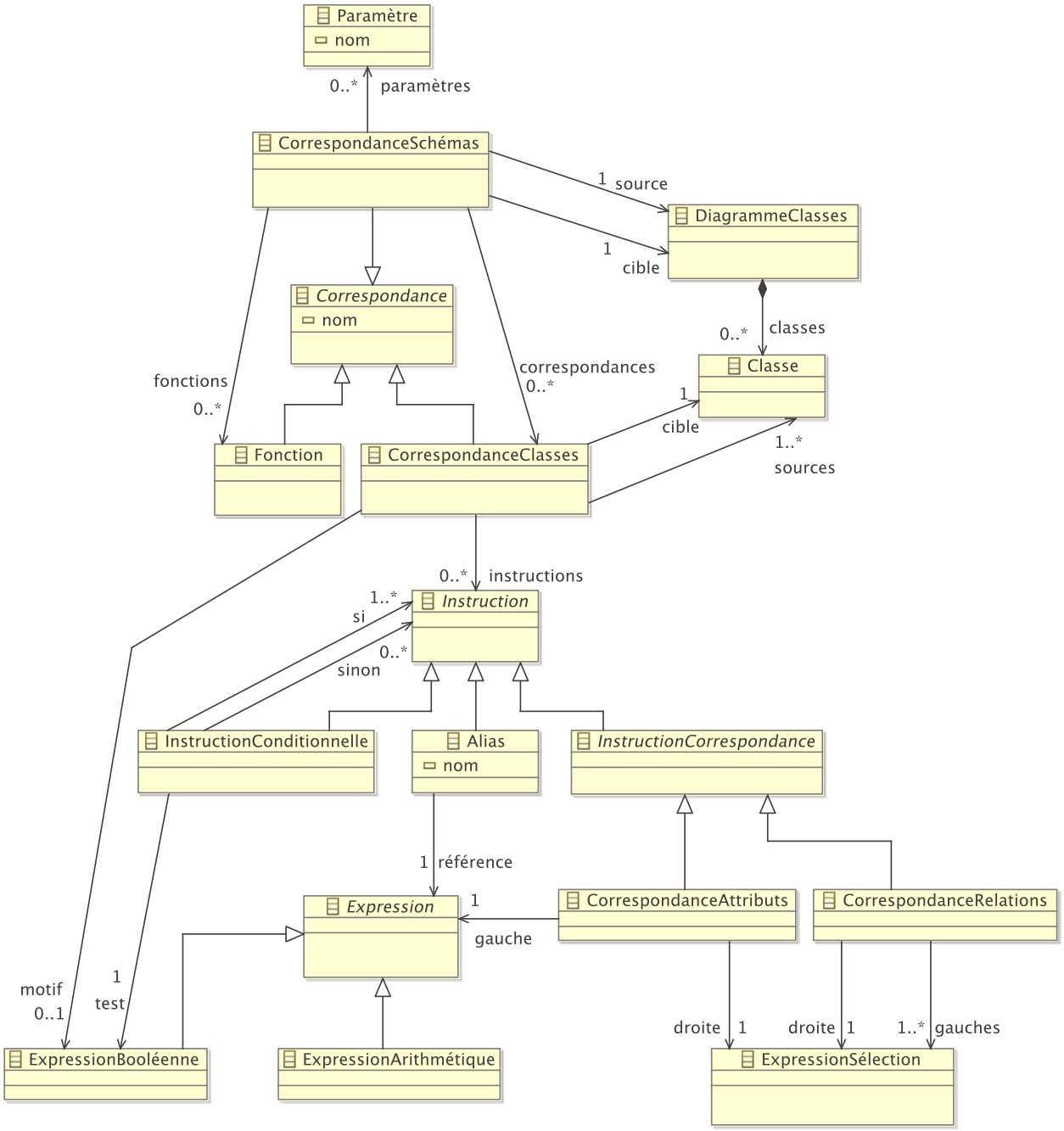


Fig. 5.5: Les principaux éléments du métamodèle de Malan

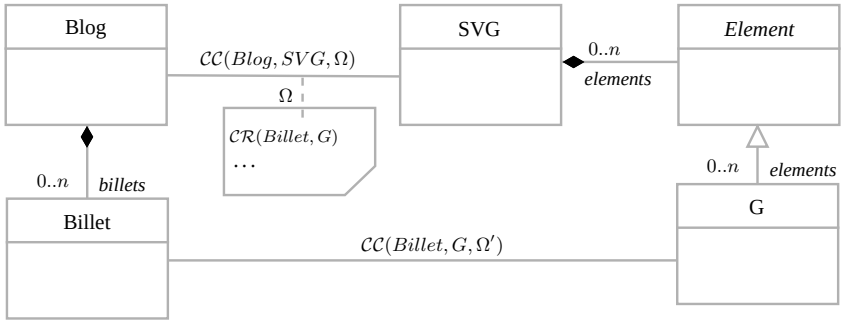


Fig. 5.6: Illustration de la contrainte de validité sur les correspondances de relations

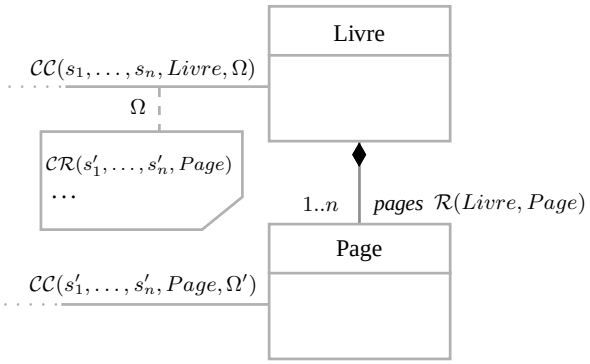


Fig. 5.7: Illustration de la contrainte de validité sur les relations

d'après la contrainte 1, une correspondance de classes $\mathcal{CC}(s'_1, \ldots, s'_n, Page, \Omega')$ utilisant $Page$ comme classe cible. La formule 5.18 formalise en remplaçant respectivement $Livre$ et $Page$ par c et c'.

$$\mathcal{CC}(s_1, \ldots, s_n, c, \Omega) \wedge \mathcal{R}(c, c') \,|\, \#_{min}\{\mathcal{R}(c, c')\} \geq 1 \implies \exists \mathcal{CC}(s'_1, \ldots, s'_n, c', \Omega') \wedge \exists \mathcal{CR}(s'_1, \ldots, s'_n, c') \in \Omega \tag{5.18}$$

La règle 5.18 ne tient cependant pas compte des classes abstraites : la classe $Element$ de la figure 5.6 étant abstraite, il doit alors exister une correspondance de classes $\mathcal{CC}(s''_1, \ldots, s''_n, c'', \Omega'')$ telle que la classe c'' non abstraite (en l'occurrence la classe G) hérite d'$Element$, comme le définit la formule 5.19.

$$\mathcal{CR}(s_1, \ldots, s_n, c, \Omega) \,|\, abs(c) \implies \exists \mathcal{CC}(s'_1, \ldots, s'_n, c, \Omega') \vee \exists \mathcal{CC}(s''_1, \ldots, s''_n, c'', \Omega'') \,|\, \neg abs(c'') \wedge (\mathcal{C}(c'') \implies \mathcal{C}(c)) \tag{5.19}$$

Contrainte 3 : unicité des correspondances de classes. La contrainte suivante (formule 5.20) stipule qu'il ne peut exister, dans une même correspondance de schémas, qu'une seule correspondance de classes possédant le même en-tête. Par exemple, dans l'exemple de la figure

5.7 il n'est pas possible de déclarer deux correspondances de classes utilisant les mêmes classes sources $s_1, \ldots, s_n$ et la même classe cible *Livre*.

$$\mathcal{CC}(s_1, \ldots, s_n, c, \Omega) \wedge \mathcal{SM}(s_1, \ldots, s_n, c', \Omega') \implies c' \neq c \tag{5.20}$$

5.10 Discussion sur différents critères d'évaluation

L'évaluation d'un langage formel se focalise généralement sur deux critères : l'expressivité et l'utilisabilité. Le premier évalue la capacité d'un langage formel à pouvoir décrire différents problèmes. Le second cherche à déterminer la capacité d'un langage formel à satisfaire l'utilisateur. Puisque ces deux critères ne sont pas formalisables mathématiquement, cette section consiste en une discussion scindée en deux parties dédiées chacune à un critère.

Concernant l'*expressivité*, d'après les objectifs présentés dans l'introduction de ce chapitre (*cf.* section 5.1, page 63), Malan permet le traitement des éléments sources, le calcul de « layout », la manipulation des relations (et d'ensembles) et la sélection d'éléments sources. Pour cela, la définition de fonctions est un outil primordial (notamment en ce qui concerne les calculs) rendant le langage Malan *turing-complet* (une machine de Turing est décrite dans l'annexe A.4, page 211). Malan est par conséquent plus expressif que le modèle de correspondance UML défini par Hausmann et Kent (2003) et que l'environnement Clio qui ne permettent que des calculs et des opérations ensemblistes de bases. Bien que turing-complet, XSLT est un langage dédié uniquement à la transformation de documents XML et ne peut s'appliquer à notre problème plus spécifique. Les langages répondant le mieux à nos besoins sont ceux de l'IDM, comme le langage ATL. Cependant, ATL n'a pas été conçu pour être utilisé dans le contexte des IHM et la possibilité de rendre ce langage incrémental n'a jamais été démontrée. Concernant OCL, ce langage et Malan partagent un point commun : ils apportent de la sémantique à des objets, à savoir des objets UML pour OCL et des correspondances pour Malan. Cependant, l'opération d'itération rend OCL proche d'un langage de programmation impératif. Malan, au contraire, définit les correspondances et navigue dans les relations de manière exclusivement déclarative, favorisant ainsi son utilisation de manière active. L'expression OCL suivante crée une chaîne de caractère pour chaque classe `e` de l'ensemble `classes` contenant le nom de la classe ainsi que celui des classes dont elle hérite :

```
classes->collect(e | e.name + " extends " + e.supertypes->iterates(
      e;acc:String=''|acc+if acc='' then '' else ' and ' endif + e.name))
```

Cette expression donne par exemple « *Bag {'Homme extends Humain', 'Licorne extends Cheval and Oiseau'}* », si `classes` contient les classes *Homme* et *Licorne*. L'expression Malan équivalente est la suivante :

```
1 classes {
2   name + " extends " + (i>1 ? " and ": "") + supertypes[i].name, i=1..|supertypes|
3 }
```

En plus d'une syntaxe plus concise, l'expression en `Malan` est facilement utilisable dans un contexte actif contrairement à celle en OCL du fait de l'itération.

A propos de l'*utilisabilité*, une correspondance de schémas ou de classes est une contrainte de la forme « $\forall P_1 \exists P_2 \,|\, P_3$ » [Popa *et al.*, 2002]. Nous estimons que le format « *source* → *cible* » est

Algorithme 5.3 : Pseudo-code de l'ordonnancement des correspondances de classes

Procédure ordonnancerCorrespondancesClasses
Entrées : $S, C, \Sigma \,|\, \mathcal{CS}(S, C, \Sigma)$// *La correspondance de schéma.*
début
  // *Sélection de la correspondance de classes racine.*
  Soient $s_1, \ldots, s_n \in S, c \in C, \Omega \,|\, \mathcal{CC}(s_1, \ldots, s_n, c, \Omega) \wedge \nexists c' \in C \,|\, (\mathcal{R}(c', c) \vee \mathcal{C}(c) \implies \mathcal{C}(c'))$
  parcourirClasse(c)
  parcourirRelations(c)
fin

Procédure parcourirRelations
Entrées : $c \,|\, \mathcal{C}(c)$ // *La classe possédant les relations à parcourir.*
début
  pour chaque $c' \,|\, \exists \mathcal{R}(c, c')$ **faire**
    // *Pour chaque relation de la classe c, on étudie la classe cible de cette relation.*
    parcourirClasse(c')
  fin
fin

Procédure parcourirClasse
Entrées : $c \,|\, \mathcal{C}(c)$ // *La classe à étudier.*
début
  si $abs(c)$ **alors**
    // *Si la classe c est abstraite, sont étudiées les classes qui en héritent.*
    pour chaque $c' \,|\, \mathcal{C}(c') \implies \mathcal{C}(c)$ **faire**
      parcourirClasse(c')
    fin
  sinon
    si $\exists s'_1, \ldots, s'_n, \Omega' \,|\, \mathcal{CC}(s'_1, \ldots, s'_n, c, \Omega') \wedge \mathcal{CR}(s'_1, \ldots, s'_n, c)$ **alors**
      /* *S'il existe une correspondance de classes ayant comme classe cible c, on étudie les instructions des correspondances de classes dont la classe cible est abstraite et est une parente de c, ainsi que ses instructions* Ω'.*/
      étudier Ω'
      pour *i de 1 à m par pas de -1* **faire**
        Soit $c_{p_i} \,|\, abs(c_p) \wedge \mathcal{CC}(s_{p_n}, \ldots, s_{p_n}, c_p, \Omega_p) \wedge \mathcal{C}(c) \implies \mathcal{C}(c_{p_m}) \implies \ldots \implies \mathcal{C}(c_{p_1})$
        étudier Ω_p
      fin
    fin
  fin
fin

celui qui s'adapte le mieux à cette situation et est notamment utilisé dans un certain nombre de langages de correspondance et de transformation, tels que Clio et ATL. Nous pensons également que la définition d'une correspondance de schémas doit s'effectuer en partie graphiquement. Ce principe, que nous avons commencé à utiliser dans notre prototype, a pour but de simplifier la spécification de la structure globale d'une correspondance de schémas en traçant des lignes, symbolisant les correspondances de classes, entre les différentes classes concernées. Le développeur ajouterait ensuite les instructions textuelles de chaque correspondance de classes dans un éditeur dédié.

5.11 Exécution des correspondances de schémas

Le langage Malan peut être utilisé de deux manières différentes. Soit une transformation est générée à partir d'une correspondance de schémas pour être exécutée dans un SI, soit une correspondance de schémas est directement implémentée dans un SI. Dans les deux cas, les correspondances de classes sont exécutées dans un certain ordre comme l'explique la section suivante.

5.11.1 Ordonnancement des correspondances de classes

Bien que la définition des correspondances de classes se fasse de manière déclarative, il existe un ordre dans lequel ces dernières sont exécutées. Un diagramme de classes UML étant un graphe, l'ordonnancement s'effectue selon un algorithme de parcours de graphe en profondeur ou en largeur. L'algorithme 5.3 effectue l'ordonnancement des correspondances de classes en profondeur. La difficulté de cet algorithme réside dans le fait qu'une correspondance de classes ayant sa classe cible c abstraite doit être exécutée après chaque correspondance de classes possédant la classe cible c' non abstraite héritant de c. Par exemple, dans le cadre de la création d'un document instance SVG *via* la correspondance de schémas de la figure 5.6, la correspondance de classes racine est $\mathcal{CC}(Blog, SVG, \Omega)$. Supposons qu'il existe une correspondance de classes ayant pour classe cible *Element* tel que $\mathcal{CC}(s_1, \ldots, c_n, Element, \Omega'')$, étant donné qu'*Element* est abstraite, elle ne peut être instanciée. Il est donc nécessaire de parcourir tout d'abord $\mathcal{CC}(Billet, G, \Omega')$ pour créer une instance de G qui hérite d'*Element*, pour ensuite remonter le graphe UML et appliquer les instructions des classes abstraites parentes de G, en l'occurrence Ω'', sur G.

L'ordonnancement des correspondances de classes est déterministe et nécessite l'absence de cycle dans la correspondance de schéma pour être terminable.

5.11.2 Génération de transformations *via* Malan

La génération de transformations, dont le principe est décrit par la figure 5.8, a pour principal avantage de clairement séparer la correspondance du processus de transformation : une transformation est générée à partir d'une correspondance de schémas Malan (étape ①), laquelle est ensuite appliquée sur des données sources pour créer des données cibles, conformément à ce qui est défini dans la correspondance de schémas (étape ②).

Notre prototype permet actuellement de générer des transformation XSLT. Bien que ce langage ne soit pas actif, son utilisation a permis la validation empirique de ce principe. Le but

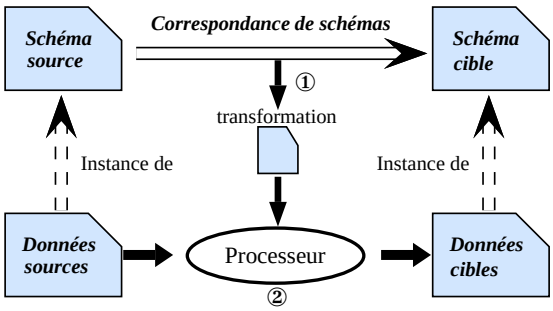


Fig. 5.8: Principe de l'utilisation de Malan pour générer des transformations

final est de générer des transformations actives AcT dont les spécifications et l'implémentation sont en cours de développement [Beaudoux *et al.*, 2009].

Un tel processus doit respecter deux propriétés : la validité des transformations générées ; la complétude du passage d'une correspondance de schémas à une transformation. La validité a deux sens : il faut que les transformations ainsi que les données qu'elles génèrent soient conformes à leur schéma respectif. Le fait de travailler au niveau des schémas assure que les données cibles créées à partir d'une transformation, elle-même générée *via* une correspondance de schémas, sont théoriquement valides dés lors que l'on considère le processus de passage d'une correspondance de schémas à une transformation XSLT comme étant fiable et que donc la transformation est conforme à son schéma. La complétude est une propriété certifiant que toute correspondance de schémas Malan doit être exprimable dans le langage de transformation cible choisi. La complétude du passage de correspondances de schémas Malan à des transformations AcT n'a pas été encore démontrée étant donné que AcT est en cours de développement. Notre implémentation[1] assure cependant de manière empirique la complétude du passage Malan-XSLT à une fonctionnalité près : étant impératives, les fonctions Malan ne sont pas toujours traduisibles en XSLT.

5.11.3 Implémentation des correspondances de schémas Malan

Afin d'éviter le problème de complétude, une implémentation des correspondances de schémas Malan est possible. Cette implémentation se fonde sur le principe de la liaison active qui, à partir d'une correspondance de schémas établie entre deux diagrammes de classes UML, crée un ensemble de liaisons actives entre des données instances de ces deux diagrammes.

Définition : liaison active. *Une liaison active $\mathcal{LA}(i_{s_1}, \ldots, i_{s_n}, i_t)$, relative à une correspondance de classes $\mathcal{CC}(s_1, \ldots, s_n, t, \Omega)$, est un lien entre des instances $i_{s_1}, \ldots, i_{s_n}$ des classes $s_1, \ldots, s_n$ et une instance i_t de la classe t, tel que :*

$$\mathcal{CC}(s_1, \ldots, s_n, t, \Omega) \wedge \mathcal{LA}(i_{s_1}, \ldots, i_{s_n}, i_t) \implies \bigwedge_{j=1}^{n} \mathcal{I}(i_{s_j}, s_j) \wedge \mathcal{I}(i_t, t) \tag{5.21}$$

où le prédicat $\mathcal{I}(i, c)$ stipule que i est une instance de la classe c. Pour une même correspondance de classes, il peut ainsi exister plusieurs liaisons actives en fonction des données sources.

[1] Téléchargeable à l'adresse suivante : http ://gri.eseo.fr/software/malan/

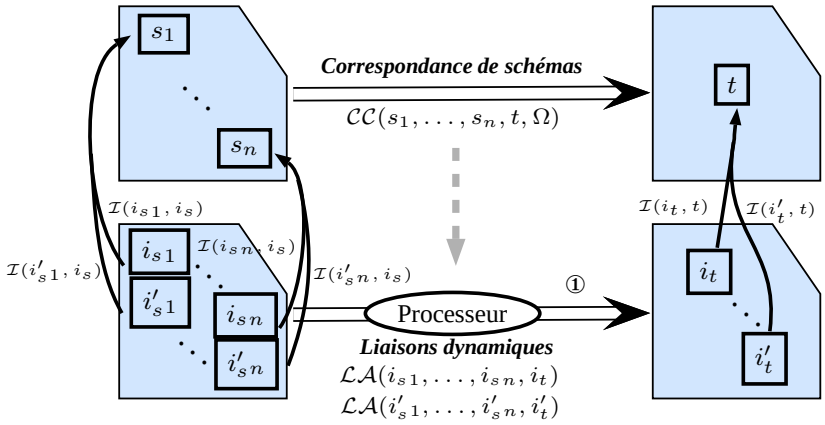


Fig. 5.9: Principe de l'utilisation de Malan pour la liaison « données-présentation »

Une liaison active est, au sens de Akehurst (2000), une implémentation active d'une correspondance de classes : lorsque les données sources $i_{s_1}, \ldots, i_{s_n}$, utilisées par un liaison active $\mathcal{LA}(i_{s_1}, \ldots, i_{s_n}, i_t)$, sont modifiées, cette dernière réapplique les instructions Ω de la correspondance de classes $\mathcal{CC}(s_1, \ldots, s_n, t, \Omega)$ concernées par les changements des données sources. Il est possible d'implémenter une correspondance de schémas de manière dynamique ou statique. L'implémentation dynamique consiste à interpréter la correspondance de schémas lors de l'établissement de la liaison active entre des données sources et cibles. Cette utilisation a pour avantage de pouvoir modifier la correspondance de schémas lors de l'exécution, mais requière que cette dernière soit interprétée à chaque initialisation de la liaison active. L'implémentation statique consiste à générer une transformation active Malan décrite dans un langage de programmation choisi, par exemple Java, à partir d'une correspondance de schémas Malan. Cette transformation active peut ainsi être directement intégrée, par exemple, dans un programme Java évitant l'inconvénient de l'initialisation de l'implémentation dynamique. De plus, le problème de complétude n'a plus lieu d'être puisque le langage de correspondance de celui de transformation sont les mêmes, à savoir le langage Malan.

Notre implémentation de Malan utilise le patron de conception « *Observateur* » : chaque attribut ou relation d'une classe est « observé » par les liaisons actives concernées. Lorsque l'attribut ou la relation est modifié, les liaisons actives mettent à jour les instances cibles impliquées.

Il existe quatre opérations fondamentales qu'un processeur gérant les liaisons actives doit pouvoir appliquer (étape ① de la figure 5.9) : l'*initialisation* des liaisons actives, et par conséquent l'instanciation des classes cibles, au chargement des données sources ; la *création*, la *suppression* et la *modification* d'une donnée source. Ces trois dernières opérations sont provoquées par les actions réalisées par l'utilisateur dans le cas des IHM, ou par le système. Elles entraînent la mise à jour, la suppression ou la création de données cibles.

5.12 Conclusion

Dans le contexte des SI, toute modification appliquée sur des données sources doit être répercutée sur leurs présentations cibles. Les environnements de développement modernes répondent

à cet aspect en proposant des modèles de « *data binding* ». Ces modèles sont cependant dépendants de la plate-forme d'IHM et non-nécessairement interopérables. De plus, ils sont prévus pour être définis dans l'interface. Les langages de transformations peuvent créer des présentations cibles à partir de données sources. Ils ne permettent cependant pas d'établir un lien durable entre eux.

En réponse à cet aspect, nous proposons un langage de correspondance, appelé Malan, dédié à la spécification de correspondances de schémas entre des données sources et une présentation cible d'un SI. Malan a pour principaux avantages de disposer d'une expressivité élevée tout en visant à garder une syntaxe la plus simple possible et d'être indépendant de toute plate-forme d'IHM. A la vue des exemples développés, cette expressivité s'avère adaptée à la définition des correspondances de schémas même complexes. Une correspondance de schémas Malan peut s'exécuter soit en générant une transformation, soit en interprétant directement la correspondance de schémas par un processeur Malan.

Notre implémentation actuelle du langage Malan comporte, tout d'abord, un générateur de transformations XSLT dont le but était de valider le principe de la génération de transformations. Elle contient également un interpréteur dédié à la gestion de liaisons actives, utilisé dans nos études de cas (*cf.* partie IV). L'objectif final de l'utilisation de Malan est de générer des transformations AcT dont la spécification et l'implémentation sont en cours de développement. Cette génération devra assurer la conformité et la validité des transformations AcT générées.

Troisième partie

Définition d'actions, d'interactions et d'instruments avec Malai

Introduction

Cette partie est dédiée à une présentation de l'état de l'art des modèles d'interactions et d'actions, et de nos travaux, fondés sur ces modèles, concernant la conception des parties statique et dynamique des actions, des interactions et des instruments. Le premier chapitre porte sur les différents modèles d'interaction et d'action. Les modèles conceptuels en rapport avec nos travaux, comme la manipulation directe et l'interaction instrumentale, sont tout d'abord introduits. Les principaux modèles de description d'interactions et d'actions, fondés sur ces modèles conceptuels, sont ensuite détaillés.

Le deuxième et le troisième chapitres présentent nos travaux sur la conception de la facette interactive d'un SI. Les composants de cette facette, à savoir l'interface, les actions, les interactions et les instruments, se composent de deux parties : la partie statique, présentée dans le deuxième chapitre, décrit les caractéristiques des composants sans s'intéresser à leur fonctionnement ; la partie dynamique, détaillée dans le troisième chapitre, décrit le fonctionnement des composants. Un même exemple d'un éditeur de documents XML est utilisé tout au long des chapitres 7 et 8. Cet éditeur a pour fonctionnalités l'ajout, la suppression, le déplacement et le renommage de nœuds au travers d'un composant graphique arbre.

Chapitre 6

Modèles d'interaction et d'action : un état de l'art

Sommaire

6.1 Introduction

En IHM, un modèle conceptuel apporte un ensemble de principes à suivre lors de la conception de SI sans toutefois donner de détails techniques concernant la description d'interactions ou d'actions. Par exemple, la manipulation directe (*cf.* section 6.2.2, page 94) prône la conception d'interfaces dans lesquelles l'utilisateur visualise et manipule les objets de manière naturelle et directe, nonobstant la façon dont le développeur puisse les implémenter. Les modèles de description d'interactions et d'actions proposent quant à eux des formalismes et des principes pour mettre en place ces modèles conceptuels dans des boîtes à outils. Nous les avons classés selon trois groupes : les modèles à flots de données, ceux fondés sur les machines à états et enfin ceux fondés sur les réseaux de Petri.

Parmi les modèles conceptuels majeurs, nous présentons dans la section 6.2 ceux nous ayant directement inspirés lors de la conception de notre modèle d'interaction, à savoir : la théorie de l'action de Norman, la manipulation directe, les modèles centrés sur les objets d'intérêt ainsi que ceux centrés sur l'interaction. La section 6.3.1 est dédiée aux principaux modèles de description d'interactions sur lesquels se fondent nos travaux. La section 6.3.2 porte sur les modèles de description d'actions. A partir de cet état de l'art, nous concluons ce chapitre en mettant en exergue les principales idées à la base des interfaces utilisateur modernes et que nous utilisons dans notre modèle d'interaction.

6.2 Modèles conceptuels

La manipulation directe et la théorie de l'action de Norman apportent des principes pour la conception de SI. Les approches mettant l'objet d'intérêt au centre des IHM, telles que IUOO (« Interface Utilisateur Orientée Objet ») et les trois principes de Beaudouin-Lafon (réification, polymorphisme et réutilisabilité), complètent le concept de manipulation directe. Enfin, les concepts d'interacteur et d'instrument précisent la façon dont les objets d'intérêt peuvent être manipulés par le biais d'actions.

6.2.1 La théorie de l'action

La théorie de l'action de Norman décompose le processus mental que suit un utilisateur lors de la résolution d'un problème, en sept étapes ordonnancées selon la figure 6.1 [Norman et Draper, 1986; Norman, 1988].

Une fois le but spécifié (p. ex. « créer un diagramme représentant les sept étapes de la théorie de Norman ») il doit être transformé en intentions (p. ex. « réaliser le diagramme à l'aide d'un éditeur de dessins »). Des actions sont déduites à partir de ces intentions (p. ex. « créer un encadré "*Formulation du but*" », *etc.*) pour être ensuite exécutées sur le système. L'utilisateur doit percevoir le résultat des actions exécutées (p. ex. « affichage de l'encadré "*Formulation du but*" », *etc.*) pour en interpréter leurs effets sur le système et en évaluer la réussite (p. ex. « l'encadré a bien été crée »).

Pour que l'utilisateur puisse enchaîner ces sept étapes, il doit à tout instant pouvoir connaître l'état du système et des différentes actions qu'il peut réaliser. Le feed-back est également nécessaire pour qu'il puisse évaluer le résultat de ses actions.

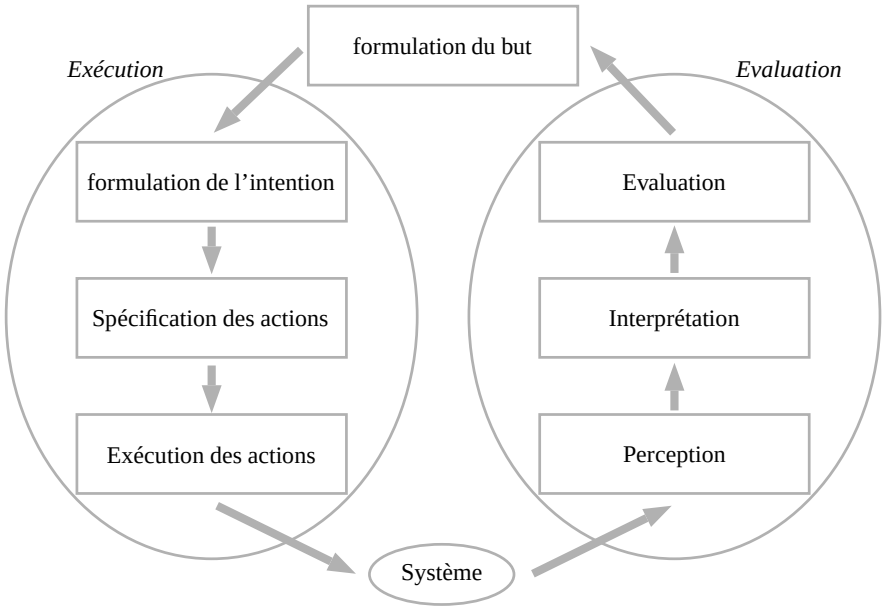


Fig. 6.1: Organisation des sept étapes de la théorie de Norman, tiré de [Norman et Draper, 1986]

Cette théorie met en avant des concepts majeurs, comme le feed-back des actions et la visibilité de l'état du système, que nous reprenons et complétons dans nos travaux.

6.2.2 La manipulation directe

La manipulation directe est l'un des modèles conceptuels à la base des interfaces modernes [Shneiderman, 1982; Shneiderman, 1983]. Il a pour but de permettre à l'utilisateur d'avoir la sensation de manipuler et de visualiser directement les objets du système, de manière analogue à ce qu'il ferait dans le monde réel. De plus, il préconise que les actions pouvant être appliquées sur les objets doivent s'inspirer du monde physique, permettant ainsi :

- au novice, d'apprendre rapidement les fonctionnalités de base, en étant par exemple aidé par un utilisateur expérimenté ;
- à l'expert, de réaliser rapidement un grand nombre d'actions, même si cela implique la définition de nouvelles fonctionnalités ;
- aux utilisateurs intermittents bien informés, de mémoriser les principes d'utilisation ;
- aux utilisateurs, de percevoir immédiatement si les actions qu'ils réalisent correspondent à leur but et, par conséquent, de pouvoir facilement changer la direction de leurs activités.

Dans cette approche, les messages d'erreur sont rarement nécessaires. D'un point de vue conceptuel, la manipulation directe prône les propriétés suivantes :

- les objets d'intérêt sont représentés de manière continue ;
- les actions physiques remplacent les commandes à syntaxe complexe ;
- les actions, incrémentales et réversibles, ayant un impact sur l'objet d'intérêt doivent avoir un résultat immédiatement visible.

Hutchins *et al.* (1985) distinguent deux aspects du caractère direct de la manipulation : la distance et l'engagement. La *distance* correspond à l'écart entre le modèle mental de l'utilisateur, décrivant la manière dont l'utilisateur pense que le système fonctionne [Norman, 1988], et le modèle physique du système (comment le système fonctionne réellement). L'*engagement* concerne la sensation de satisfaction que ressent l'utilisateur à manipuler de manière directe les objets d'intérêt.

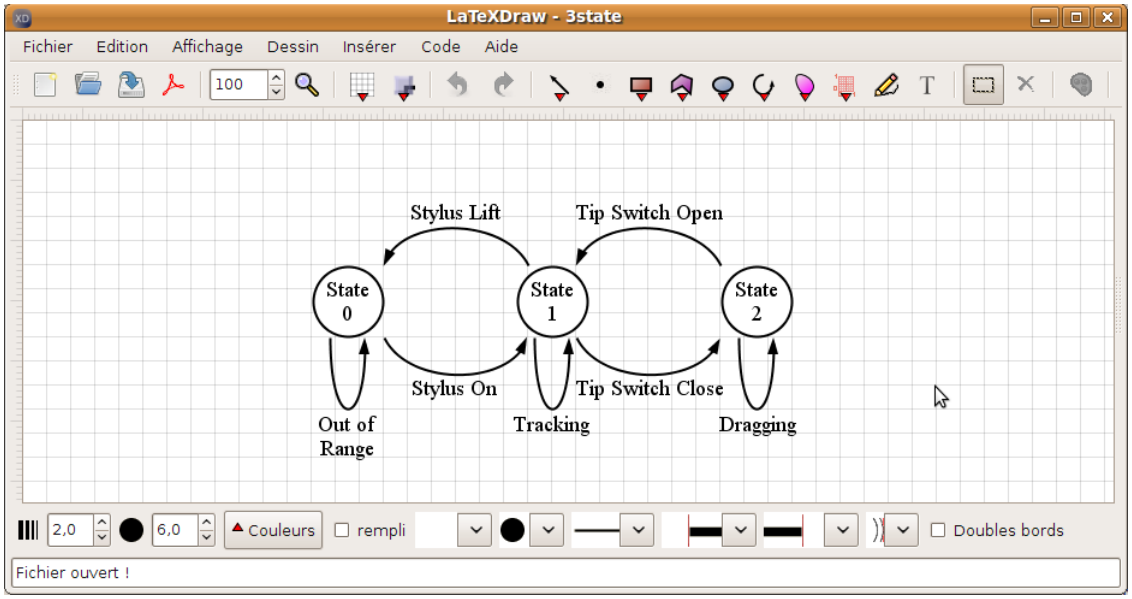


Fig. 6.2: Exemple d'utilisation de la manipulation directe : un éditeur de dessins vectoriels

La figure 6.2 présente un éditeur de dessins vectoriels. Il applique le principe de la manipulation directe pour créer et déplacer des formes, ce qui implique une distance nulle. Il utilise également des composants graphiques, comme ceux de la barre d'outils, pour éditer des formes, augmentant ainsi la distance entre l'utilisateur et les objets manipulés. Le principe de l'interface centrée sur les objets du domaine rejoint le principe de la manipulation directe.

6.2.3 Modèles centrés sur les objets du domaine

Les modèles centrés sur les objets du domaine focalisent la conception d'un SI sur les données sources, appelées « objets du domaine », qu'un utilisateur doit pouvoir manipuler. Nous présentons les deux modèles de ce type que nous considérons prédominants : l'interface utilisateur orientée objet et les trois principes de Beaudouin-Lafon (réification, polymorphisme et réutilisabilité).

Interface utilisateur orientée objet (IUOO)

IUOO est un modèle développé par IBM dans les années 1990 [Collins, 1995]. Son principe fondamental se divise en deux parties : du côté utilisateur, il prône le fait que le modèle mental de l'utilisateur doit correspondre le plus possible à l'interface et aux données sous-jacentes ; du

côté développeur, ce modèle stipule qu'une interface et un système bien conçus doivent être le plus proche possible du modèle de données, comme l'illustre la figure 6.3.

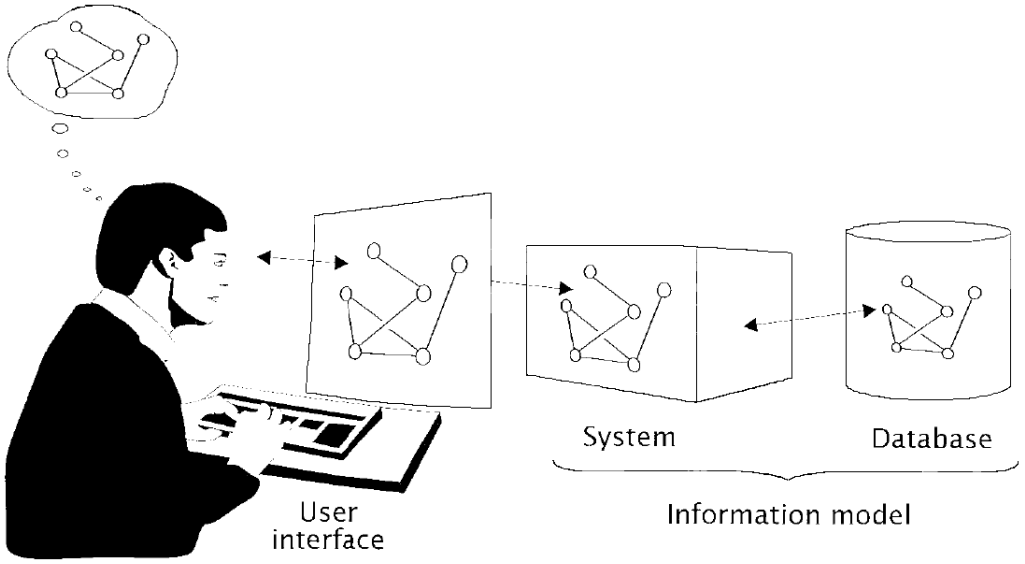


Fig. 6.3: Principe des interfaces utilisateur orientées objets, extrait de [Collins, 1995]

Réification, polymorphisme et réutilisabilité

Le modèle « réification, polymorphisme et réutilisabilité », également axé autour de la notion d'objet du domaine, définit trois principes pour la conception des interfaces graphiques : la *réification*, le *polymorphisme* et la *réutilisabilité* [Beaudouin-Lafon et Mackay, 2000].

La *réification* est le processus transformant les concepts en objets, lesquels deviennent alors manipulables par l'utilisateur. Par exemple, la gomme d'un éditeur de dessins est une réification de l'action consistant à effacer une portion du dessin. De même, la barre de défilement est la réification de l'action de défilement d'un document. La réification concerne également le regroupement d'un ensemble d'objets en un groupe d'objets : une action appliquée au groupe s'applique à chacun de ses objets.

Le *polymorphisme* consiste à pouvoir appliquer une même action sur des objets de différents types. Ce principe permet de factoriser le nombre d'actions fournies par un système. Il devient alors possible de construire une interface simple tout en augmentant ses fonctionnalités. Par exemple, la plupart des interfaces fournissent les actions couper, copier et coller pouvant être exécutées sur différents types d'objet. Par ailleurs, le polymorphisme peut également s'appliquer à un ensemble d'objets de même type ; dans ce cas, l'action sera appliquée sur chacun des objets du groupe.

La *réutilisabilité* se compose de deux facettes. La première concerne la réutilisation d'une action préalablement exécutée par l'utilisateur, afin qu'il ne réalise pas une nouvelle fois tout le processus lié à la création de l'action. Par exemple, l'action *refaire*, généralement associée à l'action *défaire*, met en œuvre ce principe. Deuxièmement, les objets créés par l'utilisateur

peuvent être réutilisés. Par exemple, un objet peut être copié plusieurs fois par une action « copier » ou « coller » suivie de plusieurs actions « coller ».

6.2.4 Modèles centrés sur les outils

Les modèles centrés sur l'interaction s'appuient sur ceux centrés sur les objets du domaine en apportant une contribution importante : pour manipuler les données sources, l'utilisateur doit avoir à sa disposition un ensemble d'*outils*, comme dans le monde réel. Ces outils établissent le lien entre les interactions utilisateur et les objets du domaine. Nous présentons dans cette section le modèle de l'interacteur, l'interaction instrumentale et le modèle DPI.

L'interacteur

La notion d'interacteur a été introduite par Myers lors du développement du système Garnet [Myers, 1990; Myers *et al.*, 1990]. Un interacteur peut utiliser les interactions usuelles fondées sur l'usage du clavier et de la souris, ou des interactions spécifiques à un système donné. Par exemple, les poignées de contrôle, permettant de déplacer ou de transformer une forme d'un dessin vectoriel, sont des interacteurs. Pour l'utilisateur, les interacteurs apportent les avantages suivants :

- plusieurs actions peuvent être associées à un même objet d'intérêt ;
- plusieurs instruments physiques peuvent être utilisés en parallèle ;
- ils permettent le feed-back sémantique ; c.-à-d. qu'un évènement doit être analysé en fonction de son contexte et de paramètres pour savoir quel feed-back doit être utilisé.

L'utilisation d'interacteurs par le développeur de SI présente les bénéfices suivants :

- un interacteur est indépendant du système et de l'interface, et peut donc être développé séparément ;
- le détail de la gestion des évènements en entrée n'est pas visible par le développeur d'un interacteur ;
- ils favorisent la réutilisation de code puisqu'un même interacteur peut être employé pour différents systèmes. Cette propriété permet également un rapide prototypage d'un système.

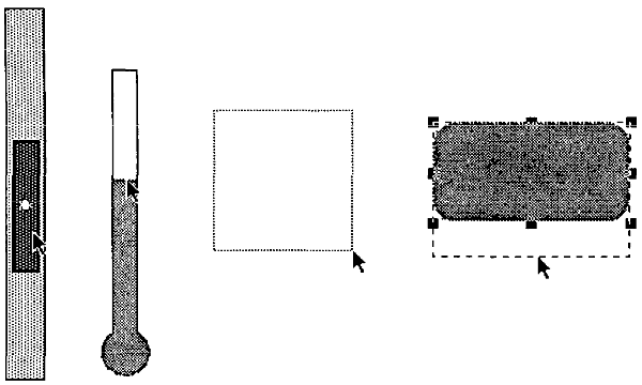


Fig. 6.4: Exemples d'interacteurs, extrait de [Myers, 1990]

La figure 6.4 présente des exemples d'interacteurs, tels que l'ascenseur et les poignées de contrôle.

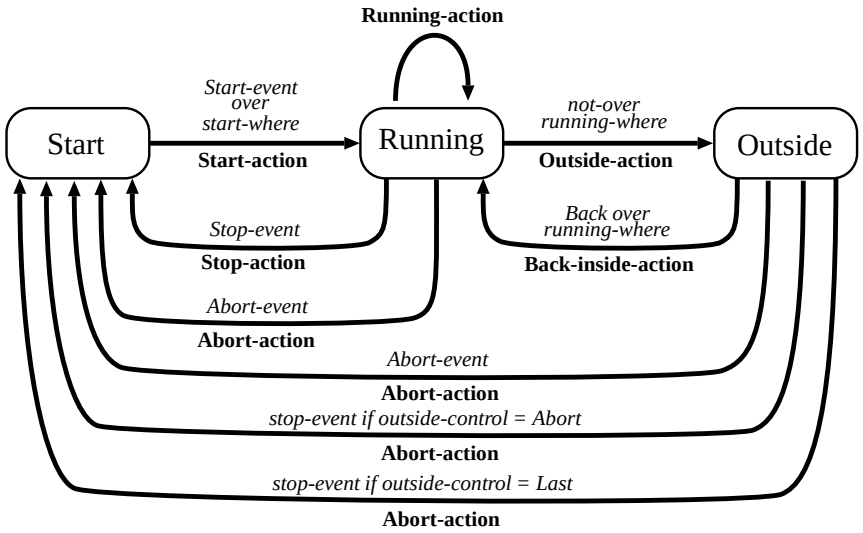


Fig. 6.5: Machine à états décrivant le fonctionnement d'un interacteur, extrait de [Myers, 1990]

Le fonctionnement d'un interacteur se base sur la machine à états décrite par la figure 6.5. Elle définit notamment qu'un instrument démarre lorsqu'un évènement initial survient, et s'arrête lorsqu'un évènement terminal ou annulateur se produit. Ce processus est à la base du fonctionnement de notre modèle d'interaction.

L'interaction instrumentale

L'interaction instrumentale est un modèle fondé sur la notion d'instrument, qui prolonge et généralise le principe de l'interacteur [Beaudouin-Lafon, 2000]. Beaudouin-Lafon (1997) définit un instrument de la manière suivante :

> « [Un instrument joue] *un rôle de médiateur entre l'action de l'utilisateur et les objets de l'application.* [...] *Il est composé d'une partie physique et d'une partie logique. La partie physique comprend les transducteurs d'entrée-sortie utilisés par l'instrument, en entrée pour capter l'action physique de l'utilisateur et en sortie pour lui présenter un retour d'information.* [...] *La partie logique de l'instrument comprend en entrée la méthode de transformation des actions de l'utilisateur sur l'instrument logique et en sortie la représentation de l'instrument.* »

La figure 6.6 illustre un exemple d'interaction instrumentale : l'utilisateur effectue un glisser-déposer sur la barre de défilement à l'aide d'une souris (*action*). L'instrument retourne un feedback permettant à l'utilisateur de visualiser le déplacement de la barre. L'instrument transforme ensuite l'interaction de l'utilisateur en une action (*command*) de navigation. Lorsque l'action est appliquée sur la présentation, ces dernières répondent en faisant défiler le document.

L'origine de l'interaction instrumentale résulte du constat selon lequel *« les applications graphiques interactives présentent une grande complexité visuelle de leur interface et une faible*

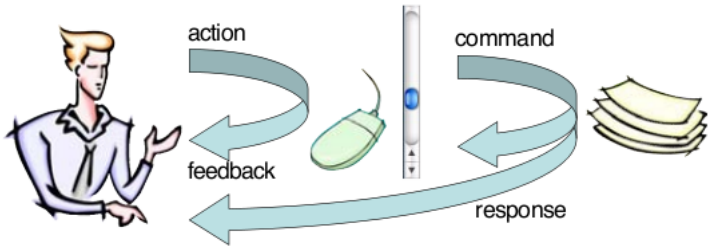


Fig. 6.6: Principe de l'interaction instrumentale, extrait de [Beaudouin-Lafon, 2004]

efficacité de l'interaction » [Beaudouin-Lafon, 1997]. Ce modèle vise ainsi à recentrer le développement des systèmes sur l'interaction *via* les instruments, au lieu de se focaliser sur le développement des interfaces [Beaudouin-Lafon, 2004]. Ce modèle d'interaction complète parfaitement celui de l'interface orientée objet : l'OOUI se focalise sur les objets d'intérêt tandis que l'instrument se concentre sur l'interaction que l'utilisateur peut réaliser sur ces objets. De plus, l'interaction instrumentale intègre le polymorphisme par le biais des instruments génériques réalisant différentes actions en fonction du contexte. Ce modèle utilise également le principe de la réification dans le sens où un instrument est la représentation d'un concept sous forme d'objet. Ce modèle ne propose cependant pas un modèle d'action qui permettrait aux instruments de générer des actions considérées comme des objets à part entière et, ainsi, de faire le lien entre les interactions (partie physique) et les actions (partie logique). Nos travaux sur l'interaction visent notamment à combler ce manque.

DPI

DPI (*Document, Présentation, Instrument*) est un modèle conceptuel fondé sur l'interaction instrumentale mais centré sur les documents [Beaudoux et Beaudouin-Lafon, 2001; Beaudoux, 2004b]. Le principe de ce modèle est décrit dans la figure 6.7 : l'*instrument* transforme les *gestes* réalisés par l'utilisateur (étape ①) en *actions*, ces dernières étant alors consommées par le document (étape ②). Beaudoux (2004c), page 81, classe les instruments en trois catégories :

> « *Les* instruments directs*, métaphore de l'outil, sont manipulés en respectant une certaine forme de continuité entre la main et l'instrument : l'utilisateur agit directement sur l'instrument pour agir directement sur le document. Les* instruments indirects*, métaphore de l'appareil, sont manipulés sans continuité directe avec l'objet d'intérêt : l'utilisateur (inter)agit directement sur l'instrument et indirectement sur le document. Les instruments de perception, métaphore de la lentille optique, sont manipulés comme les instruments directs puisque l'utilisateur agit directement sur l'instrument (déplacement), mais agissent comme les instruments indirects puisqu'ils ne visent pas directement les documents.* »

Le *document* est « *un objet persistant et perceptible que l'utilisateur peut lire, écrire et annoter* » [Beaudoux, 2004b]. Il se compose de deux facettes : la *facette persistance* correspond au support physique permettant le stockage de documents ; la *facette présentation* confère au document une ou plusieurs apparences concrètes pour l'utilisateur. Les *présentations* multiples permettent de représenter un document sous différents points de vue et autorisent l'édition d'un même document au travers des plusieurs présentations.

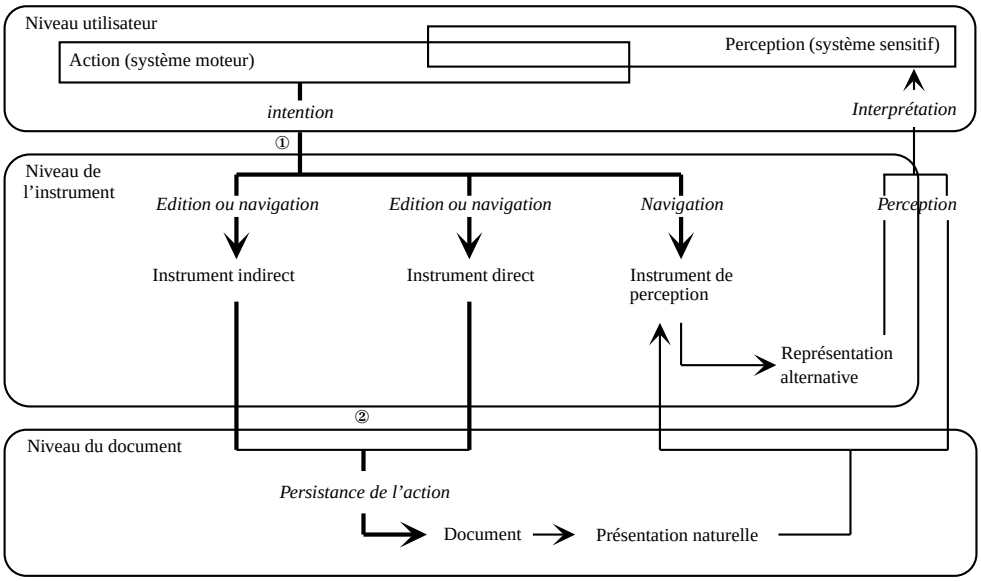


Fig. 6.7: Le modèle conceptuel de DPI, inspiré de [Renouard, 2007]

Nos travaux reprennent le principe de séparation entre les interactions et les actions, ainsi que le concept d'instrument jouant le rôle de médiateur entre les interactions de l'utilisateur et les actions. Nous apportons cependant un modèle d'action plus complet. Notre modèle renforce également la position de l'instrument en tant que médiateur entre les interactions et les actions. L'instrument a pour rôle de créer, mettre à jour, exécuter, avorter ou recycler une action en une autre, en fonction des interactions utilisateur. Cette gestion des actions est facilitée par le feedback intérimaire fourni par l'instrument, informant l'utilisateur à tout instant des différentes actions qu'il peut réaliser.

6.3 Modèles de description

Les modèles de description d'interactions et d'actions proposent quant à eux des formalismes et des principes pour mettre en place les modèles conceptuels introduits précédemment dans des boîtes à outils. Cette section se scinde en deux parties : la première est consacrée aux modèles de description d'interactions ; la seconde se focalise sur ceux dédiés à la description d'actions.

6.3.1 Modèles de description d'interactions

Nous avons classé les différents modèles de description d'interactions selon trois catégories : les modèles à flot de données, ceux fondés sur les machines à états et ceux fondés sur les réseaux de Petri.

6.3.1.1 Modèles à flot de données

Un modèle à flot de données se caractérise par un ensemble d'objets possédant une ou plusieurs portes d'entrée et de sortie par lesquelles les données arrivent et sortent. Ces objets

sont interconnectés par l'intermédiaire de leurs portes, et traitent ainsi le flot de données depuis l'interaction jusqu'aux actions.

ICon (*Input CONfiguration*) est une boîte à outils destinée à la conception d'interactions [Dragicevic et Fekete, 2001]. Elle permet de décrire, à l'aide d'un langage graphique dédié, toute la chaîne de gestion des entrées/sorties de manière explicite. Son principe est de pouvoir connecter des périphériques physiques (souris, clavier, *etc.*) à des actions, des filtres, *etc.*, en utilisant un formalisme fondé sur les flots de données ; ces connections sont établies entre les attributs des périphériques et des actions.

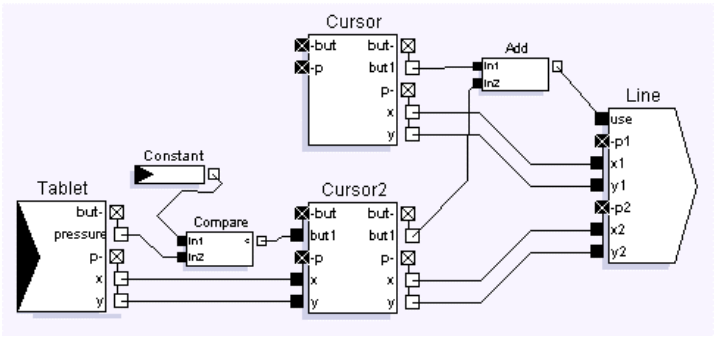


Fig. 6.8: Interaction bimanuelle avec ICon, extrait de [Dragicevic et Fekete, 2001]

La figure 6.8 présente un exemple d'utilisation d'ICon : un stylet (objet `Cursor2`) et une souris (objet `Cursor`) sont configurés pour composer une interaction bimanuelle permettant l'ajout d'une ligne (objet `Line`) à une application de dessin.

Le formalisme de ce modèle permet la configuration d'interactions complexes et novatrices. Il peut être associé au principe de correspondance de schémas (voir section 3.4.2) puisque les composants sont mis en correspondance *via* leurs attributs. Cependant, les notions d'interaction, d'instrument, d'action et d'interface ne sont pas clairement séparées, limitant ainsi l'application de certaines propriétés comme le polymorphisme et la réutilisabilité.

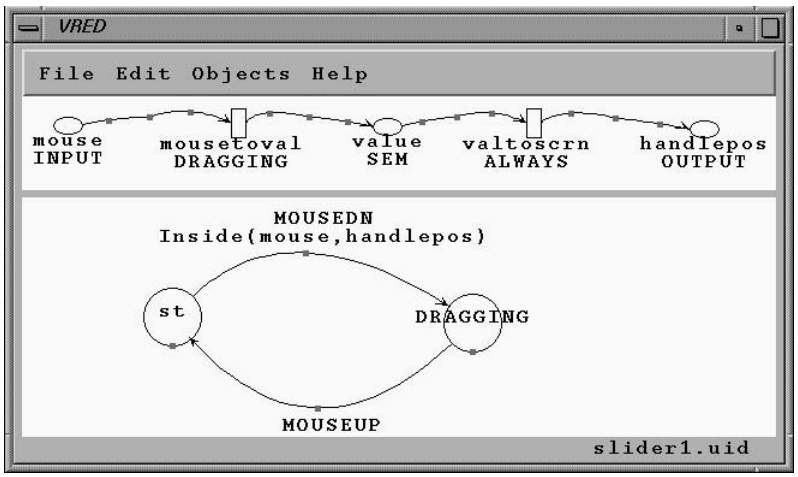


Fig. 6.9: L'éditeur VRED, extrait de [Jacob *et al.*, 1999]

VRED est un éditeur d'interactions fondé sur un modèle mixte à flot de données et à flot de contrôle, dédié principalement à la réalité virtuelle [Jacob *et al.*, 1999]. Le principe du flot de contrôle est de représenter graphiquement les structures de contrôle des langages de programmation textuels afin de faciliter la description des interactions « non-WIMP ». La figure 6.9 présente l'éditeur VRED décrivant le déplacement d'un objet à l'aide d'un curseur. La partie haute correspond à l'éditeur de flot de données dans laquelle les variables sont représentées par des ellipses, les liens par des rectangles et les flux de données par des flèches. Il décrit le passage d'un mouvement de la souris en position sur l'écran. La partie basse concerne la machine à états décrivant, en l'occurrence, le fonctionnement d'un « *slider* ».

6.3.1.2 Modèles fondés sur les machines à états

Les modèles fondés sur les machines à états, présentés dans cette section, utilisent la notion d'automate déterministe fini. Dans le domaine de l'interaction, ce principe a été employé par Buxton (1990) dans le contexte de la manipulation directe. Son modèle Three-state permet de décrire le fonctionnement de périphériques en utilisant seulement trois états : l'état 0, dénommé *hors de portée*, correspond à un évènement n'ayant aucun effet sur le système ; l'état 1, *recherche*, se rapporte au déplacement du curseur ; l'état 2, *déplacement*, concerne quant à lui le déplacement du curseur et d'un objet. La figure 6.10 présente la description d'un stylet avec le modèle Three-state. On y distingue bien les trois états qui catégorisent le statut de l'interaction à un instant donné. Dans l'état 0, le stylet n'est pas posé sur l'écran tactile ; une fois le stylet au contact de l'écran, la machine à états passe à l'état 1 ; le passage à l'état 2 s'effectue lors d'une pression du stylet sur l'écran.

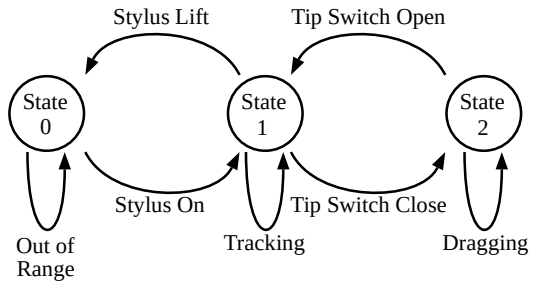
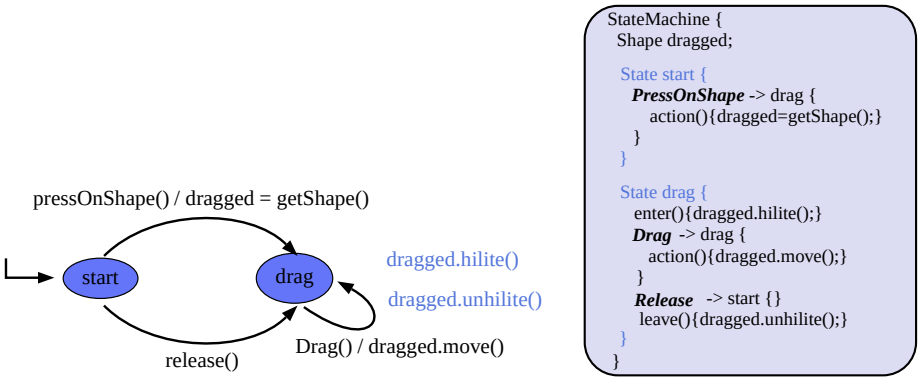


Fig. 6.10: Description d'un stylet avec le modèle Three-state, extrait de [Buxton, 1990]

Cependant, le modèle Three-state a un pouvoir d'expression limité par rapport aux modèles généraux de machines à états : par exemple, la librairie SwingStates permet de spécifier des interactions à l'aide de machine à états [Appert et Beaudouin-Lafon, 2008]. La figure 6.11a montre une représentation conceptuelle d'une machine à états décrivant une interaction *glisser-déposer*, dont le code SwingStates est présenté dans la figure 6.11b.

Les machines à états ont pour principal défaut l'explosion combinatoire du nombre d'états et de transitions lorsque la complexité du problème à traiter augmente. Pour éviter ce problème, SwingStates permet d'activer en parallèle plusieurs machines à états. Une autre solution consiste à hiérarchiser les états, comme le propose le modèle StateCharts [Harel, 1987], puis plus tard le modèle HsmTk [Blanch et Beaudouin-Lafon, 2006]. Ces modèles définissent le principe



(a) Machine à états (b) Code de la machine à états

Fig. 6.11: Machine à états SwingStates pour le *glisser-déposer*, extrait de [Appert, 2007]

qu'un état peut contenir une machine à états, limitant ainsi l'explosion du nombre d'états et de transitions pour un niveau donné, mais augmentant alors la complexité du modèle.

6.3.1.3 Modèles fondés sur les réseaux de Petri

Le réseau de Petri est une généralisation des machines à états [Petri, 1963]. Il permet d'augmenter l'expressivité des automates en représentant explicitement l'état du système *via* des jetons évoluant au cours de l'exécution, le déplacement de ces jetons étant conditionné par un ensemble de règles.

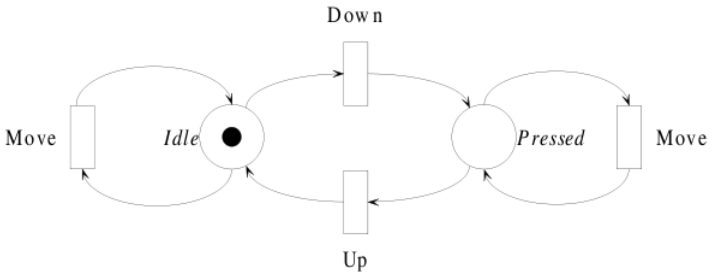


Fig. 6.12: Description d'une souris à un bouton, extrait de [Accot *et al.*, 1996]

Les réseaux de Petri limitent l'augmentation du nombre d'états et de transitions dont souffrent les machines à états. Le modèle décrit dans [Accot *et al.*, 1996], suivi de celui de transducteurs formels [Accot *et al.*, 1997], permettent de décrire des interactions tout comme SwingStates mais en se basant sur les réseaux de Petri. La figure 6.12 décrit le comportement d'une souris à un bouton. On y trouve les places (représentées par des cercles) *Idle* et *Pressed* reliées par des transitions (les rectangles) représentant des évènements (*Move*, *Down* et *Up*). La complexité des réseaux de Petri est cependant supérieure à celle des machines à états.

6.3.2 Modèles de description d'actions

Nous présentons, dans cette section, différents modèles de description d'actions dont notre modèle s'inspire en partie. Nous ne traitons pas ici des modèles de tâche puisqu'ils sont dédiés à une description de granularité plus faible : dans le contexte des SI, nous définissons une *action*, suivant la théorie de l'action de Norman et Draper (1986), comme le résultat d'une manipulation *atomique* réalisée par l'utilisateur. De manière plus abstraite, une *tâche*, qui peut contenir des sous-tâches, décrit comment un utilisateur peut atteindre un but [Paternò *et al.*, 1997].

6.3.2.1 La classification de Foley

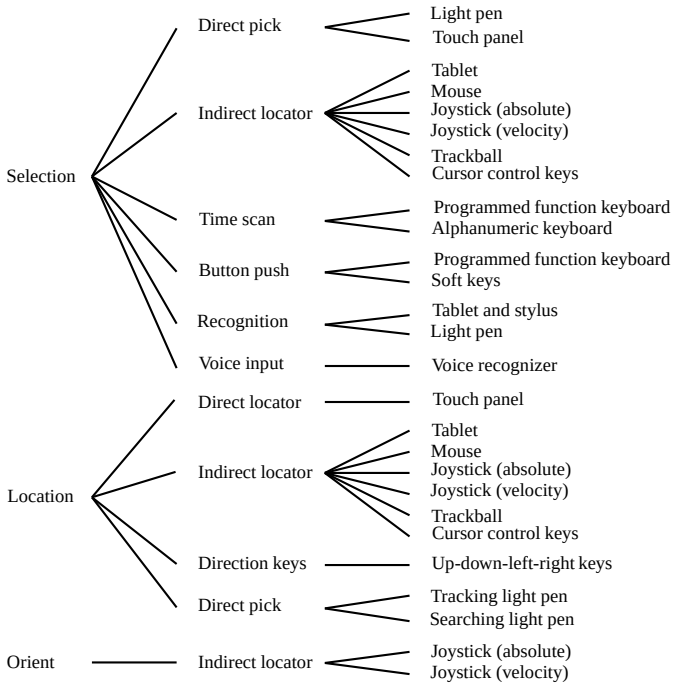


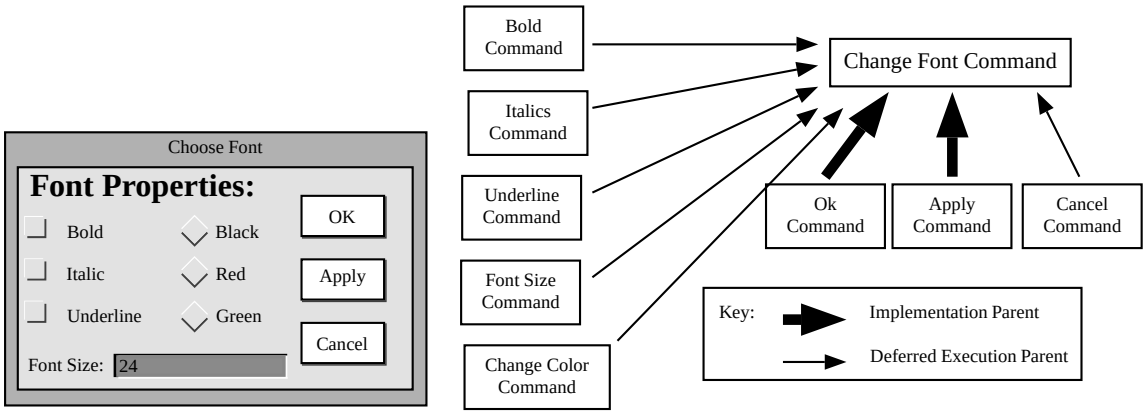
Fig. 6.13: Extrait de la classification de périphériques selon des actions génériques, extrait de [Foley *et al.*, 1984]

Foley *et al.* (1984) classent les principaux périphériques d'entrée en fonction des différentes actions qu'ils permettent de réaliser, elles-mêmes classées selon six actions génériques : la *sélection*, le *positionnement* et l'*orientation* d'un objet, le *dessin*, la *saisie de texte* et la *définition d'une valeur* par un utilisateur.

La figure 6.13 présente la classification pour la sélection, le positionnement et l'orientation : par exemple, le « joystick » est un positionneur indirect permettant de réaliser ces trois actions génériques. Cette classification a pour inconvéniant de ne pas pouvoir prendre en compte des actions utilisateur et des périphériques spécifiques à un problème donné.

6.3.2.2 Les objets de commande

La notion d'objet de commande (OC) est utilisée dans les travaux de Myers et Kosbie (1996), notamment au travers du système Amulet [Myers *et al.*, 1997]. Ce terme est également présent dans la librairie MacApp [Wilson, 1990], développée par Apple pour la création d'applications graphiques pour Macintosh. Pour chaque opération du système, un OC, possédant les méthodes `do` et `undo` d'exécution et d'annulation, est défini.



(a) Interface (b) Hiérarchie de OC de l'interface

Fig. 6.14: Exemple d'objets de commande d'une interface, extrait de [Myers et Kosbie, 1996]

Contrairement à MacApp, le modèle de Myers permet de hiérarchiser les OC, autorisant ainsi la décomposition d'OC en plusieurs autres. La figure 6.14a présente une boîte de dialogue pour laquelle la figure 6.14b contient sa hiérarchie de OC. La fragmentation d'un OC est réalisée par les liens « *Implementation Parent* ». Par exemple, l'OC « *Changement de Police* » se divise entre plusieurs OC plus atomiques tels que « *Modifier la taille de la police* » et « *Modifier la couleur de la police* », *etc.*

Les OC possèdent un cycle de vie relativement simple qui ne permet pas, par exemple, de fournir un feed-back intérimaire, ni d'être avorté puis recyclé en d'autres OC lors de leur exécution.

6.4 Conclusion

A partir des modèles conceptuels présentés dans ce chapitre, trois idées principales ressortent. Premièrement, l'utilisateur doit pouvoir manipuler de manière directe les objets d'intérêt comme il le ferait dans le monde réel ; il s'agit de la manipulation directe. Ensuite, la notion d'objet rejoint la manipulation directe dans le sens où l'utilisateur doit avoir la sensation qu'il manipule des objets à part entière. Troisièmement, l'instrument est l'élément central permettant de la manipulation directe et indirecte des objets d'intérêt. Il permet d'établir le lien entre les interactions et les actions, ainsi que de fournir le feed-back intérimaire.

Les méthodes de description d'interactions permettent de préciser finement comment les HID peuvent être utilisés pour produire les interactions. Par exemple un glisser-déposer peut être

réalisé à l'aide d'une souris ou d'un stylet. Les boîtes à outils telles que ICon et SwingStates vont dans ce sens : elles permettent de relier différents périphériques à différentes interactions. Cependant, aucune des méthodes présentées ne définit clairement la notion d'action telle que spécifiée dans la théorie de Norman, et la relation entre ces actions, les interactions et les périphériques.

Notre modèle, présenté dans le chapitre suivant, vise à retenir les avantages des principes du modèle d'action de Norman, de l'interaction instrumentale, de la manipulation directe, du concept d'interacteur et du modèle DPI, tout en apportant un ensemble d'améliorations parmi lesquelles : la possibilité de définir le feed-back intérimaire de l'instrument grâce auquel l'utilisateur est tenu au courant de l'état du système et des différentes actions qu'il peut accomplir ; considérer les actions comme des objets à part entière possédant leur propre cycle de vie ; modéliser les interactions qu'un utilisateur réalise à l'aide de périphériques sous la forme de machines à états ; utiliser l'instrument comme lien entre le monde physique (HID et interactions) et le monde logique (actions, interface et données).

Chapitre 7

MALAI : partie statique des actions, des interactions et des instruments

Sommaire

7.1 Introduction

Malai est un modèle conceptuel d'interaction qui réunit les principes du modèle d'action de Norman, de la manipulation directe, du concept d'interacteur, de l'interaction instrumentale et du modèle DPI présentés dans le chapitre précédent. Malai vise à retenir les avantages de ces modèles tout en y apportant les contributions suivantes : une librairie d'interactions prédéfinies pouvant être utilisées tel quel dans différents SI ; la définition du feed-back intérimaire [Myers, 1990] des instruments ; la réutilisation d'une même action pour différentes interfaces qui utilisent la même présentation abstraite ; un processus d'annulation et de ré-exécution des actions ; la mise en relation des interactions et des actions par le biais des instruments.

Ce chapitre se consacre à la présentation de la partie statique du modèle d'interaction Malai. La partie statique se charge de décrire les *informations* qui caractérisent l'interface et les éléments qui la composent, à savoir les présentations, les actions, les interactions et les instruments. Elle ne décrit pas leur comportement lors de l'exécution d'un SI.

Ce chapitre s'organise de la manière suivante : la section 7.2 présente la vue d'ensemble du modèle Malai ; la section 7.3 décrit l'interface qui se compose de présentations et d'instruments ; la section 7.4 présente la présentation ; la section 7.5 détaille la partie statique des interactions, des actions et des instruments.

7.2 Vue d'ensemble du modèle conceptuel

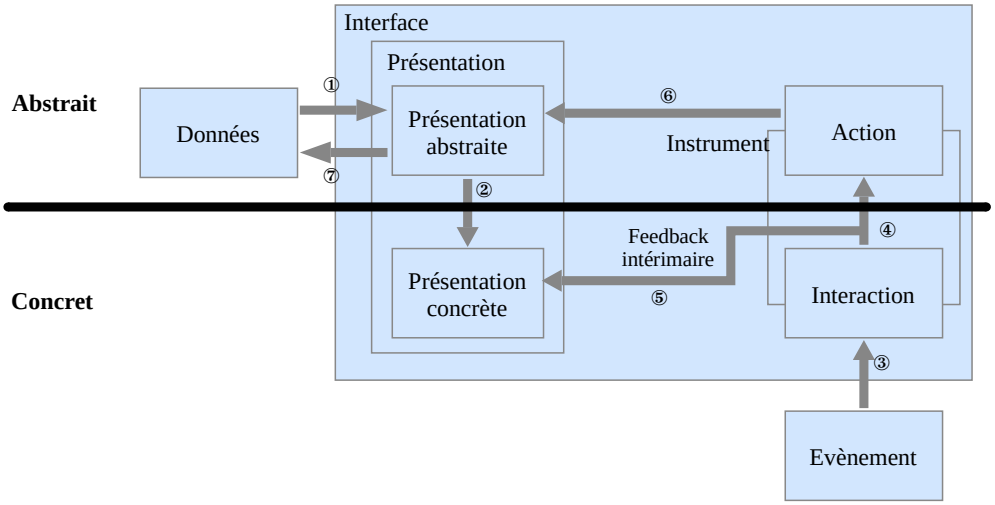


Fig. 7.1: Organisation du modèle Malai

La figure 7.1 décrit l'organisation du modèle Malai, laquelle divise le SI en plusieurs éléments fondamentaux : les données, l'interface, les présentations (abstraites et concrètes) et les instruments (évènements, interactions et actions). Chacun de ces éléments est défini indépendamment des autres par la partie statique du modèle Malai. Cette partie statique fait l'objet du présent chapitre. Les liens entre chaque élément (flèches ① à ⑦) sont, quant à eux, définis par la partie dynamique du modèle abordée dans le chapitre suivant.

L'*interface* regroupe les présentations, les instruments et les composants graphiques du SI. La *présentation* se compose d'une *présentation abstraite* et d'une *présentation concrète*. La première est construite à partir des *données sources* par une correspondance de schémas Malan (①). La présentation concrète est elle-même créée puis mise à jour par une autre correspondance de schémas Malan, établie de la présentation abstraite vers celle concrète (②). Les *interactions* se fondent sur des *évènements* produits par des HID lorsque l'utilisateur interagit avec le SI (③). L'*instrument*, pivot entre les parties abstraite et concrète d'un SI, transforme des *interactions* de l'utilisateur en *actions* (④). Le *feed-back intérimaire* de l'instrument précise l'état de l'interaction et de l'action en cours au niveau de la présentation concrète (⑤). L'action s'exécute sur la présentation abstraite (⑥), laquelle met alors à jour les données sources (⑦) *via* la correspondance de schémas inverse à la ①. Si une seconde présentation est liée à ces mêmes données sources, le lien ① entre à nouveau en jeu de manière à synchroniser les deux présentations.

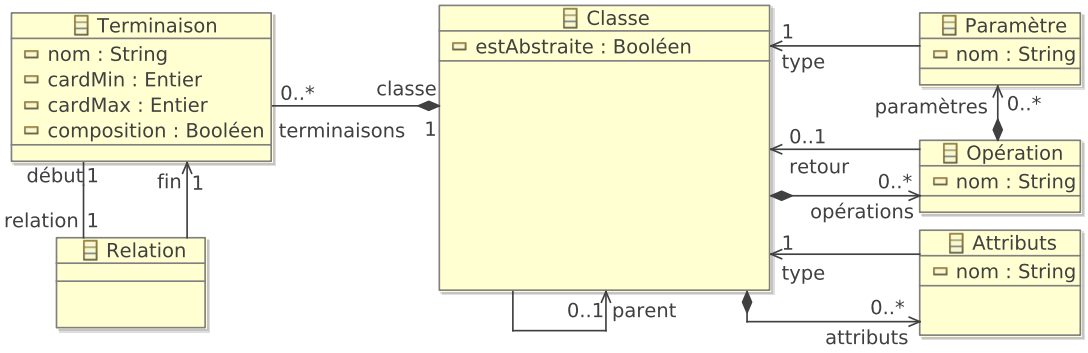


Fig. 7.2: Métamodèle de classe

La partie statique utilise, pour chaque élément, le métamodèle de classe de la figure 7.2. Il stipule qu'une classe peut être abstraite et posséder une classe parente, des attributs, des opérations et des terminaisons. Un attribut se compose d'un nom et d'un type correspondant à une classe. Une opération possède un nom, des paramètres et peut définir le type de la valeur retournée. Un paramètre se caractérise par un nom et un type. Une relation se compose d'une terminaison de début et d'une autre de fin. Une terminaison possède un nom, une cardinalité maximale et une autre maximale, et définit s'il s'agit d'une composition.

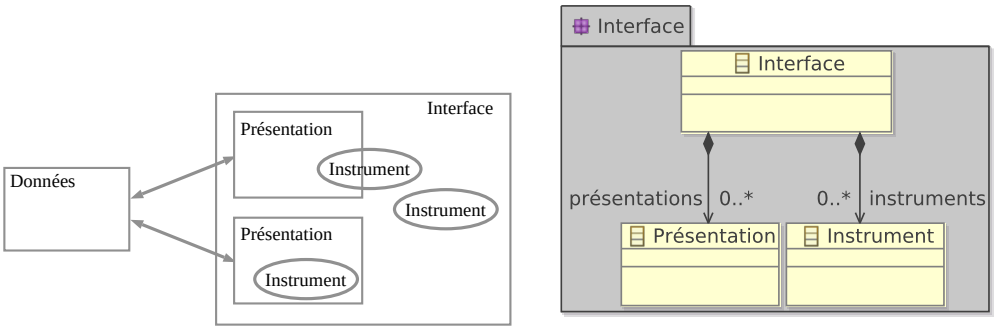
7.3 Interface

Cette section détaille les caractéristiques de l'interface. La définition et le principe de cet élément sont tout d'abord introduits. Un exemple est ensuite décrit.

7.3.1 Définition et principe

Dans notre modèle conceptuel, une interface se compose de présentations et d'instruments (*cf.* figure 7.3b). Une présentation permet à un utilisateur de visualiser des données sources d'un certain point de vue. Les instruments sont manipulés par l'utilisateur pour réaliser des actions

sur les présentations. Comme l'illustre la figure 7.3a, un instrument peut faire partie intégrante d'une présentation.



(a) Différences entre une interface, une présentation et un instrument

(b) Métamodèle d'interface

Fig. 7.3: Principe et métamodèle de l'interface

L'interface possède également un ensemble de composants graphiques, tels que les fenêtres et les panneaux. Pour décrire ces composants graphiques, nous utilisons un modèle générique de composants graphiques défini en annexe B.1, page 216, qui complète celui plus général de la figure 7.3b. Celui-ci est applicable pour différentes plates-formes d'IHM, telles que la plate-forme Swing, utilisée pour nos prototypes.

7.3.2 Exemple

Notre éditeur de documents XML possède l'interface décrite par le diagramme de classes de la figure 7.4a. Elle se compose d'une fenêtre contenant la présentation de l'arbre XML (`ArbreUI`) et d'un panneau possédant des composants graphiques, en l'occurrence de type « bouton ». L'interface possède les instruments suivants : les déplieurs de nœuds ; l'annulateur d'actions ; le copieur ; la main ; le renommeur ; la gomme ; le sélectionneur d'instruments. Ce dernier permet de sélectionner la main, la gomme ou le renommeur.

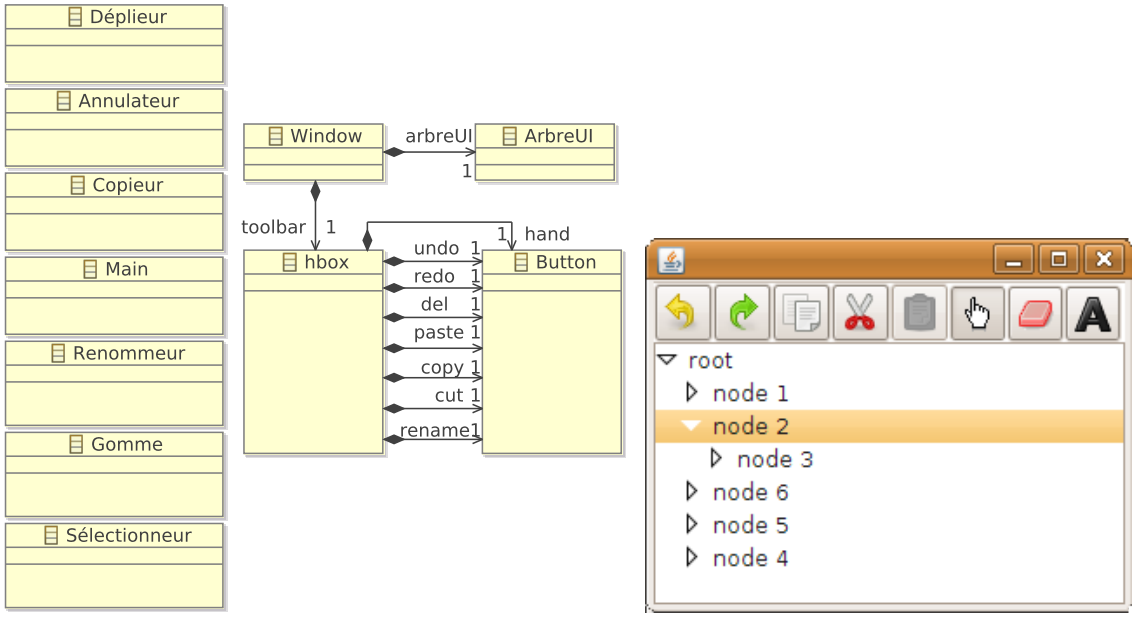
La figure 7.4b présente l'interface finale s'exécutant sur la plate-forme Swing. Les déplieurs sont présents à coté de chaque nœud sous la forme de triangles. Ces instruments, qui gèrent le pliage et le dépliage des nœuds, illustrent le cas d'un instrument faisant partie intégrante d'une présentation.

7.4 Présentation

Cette section détaille la présentation. La définition et le principe de cet élément sont tout d'abord présentés. Un exemple est ensuite décrit.

7.4.1 Définition et principe

Dans les chapitres 5 et 6, nous avons présenté le langage de correspondance Malan dédié à la définition de correspondances de schémas entre des données sources et leurs présentations cibles ;



(a) Description de l'interface (b) Interface en Swing

Fig. 7.4: L'interface de l'exemple de l'arbre

Malai utilise Malan dans ce but, comme l'illustre la flèche ① de la figure 7.5. Une présentation se divise en deux parties : la présentation abstraite et la présentation concrète. La présentation abstraite définit le modèle abstrait de la présentation. Elle ne contient aucune information à caractère graphique (p. ex. les coordonnées) des éléments qui la composent, contrairement à la présentation concrète. Par conséquent, il est également nécessaire d'établir une correspondance de schémas Malan entre ces deux types de présentations (flèche ②).

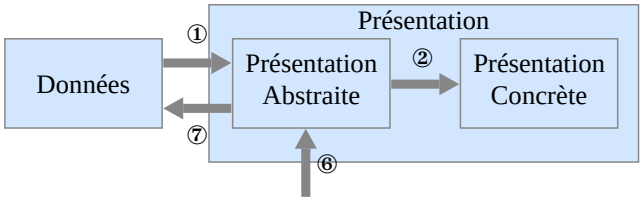


Fig. 7.5: Principes de la modification des données dans Malai

Les actions que réalise l'utilisateur modifient la présentation abstraite (flèche ⑥) et non les données sources. Ce choix se justifie par la sémantique des données sources qui n'est généralement pas la même que celle de la présentation abstraite. L'exemple de l'agenda, présenté dans le chapitre 10, page 165, illustre cette différence : les données sources définissent un emploi du temps composé d'enseignements scolaires, tandis que la présentation abstraite est un agenda standard contenant des évènements. Dans un tel cas, les actions (p. ex. l'ajout un évènement dans l'agenda) doivent être appliquées sur la présentation abstraite (c.-à-d. l'agenda) qui répercutera les changements sur la présentation concrète (flèche ②) et qui pourra synchroniser les données

sources (flèche ⑦). Cette synchronisation s'effectue grâce à une correspondance de schémas inverse à celle de la flèche ① qui doit être définie par le développeur[1].

Il est envisageable de définir des actions modifiant directement les données sources. Dans le cadre de l'agenda, par exemple, une action `AjouterCours` pourrait être définie. Cependant, ce principe ne permet pas la *réutilisation* de l'agenda dans différents SI, du fait de la dépendance envers le modèle de données.

Pour modifier la présentation abstraite, une action peut : faire appel aux opérations définies dans la classe de l'objet visé par l'action ; modifier directement les attributs de l'objet ; ajouter ou supprimer un autre objet dans l'une des relations de l'objet. Dans le pseudo-code de ce chapitre, la modification, l'ajout et le retrait sont symbolisés, respectivement, par les méthodes `#modifier`, `#ajouter` et `#supprimer`, définies de la manière suivante :

- `#modifier(Attribut attr, Objet valeur)`. Cette méthode associe la valeur donnée en paramètre à l'attribut `attr`.
- `#modifier(Paramètre param, Objet valeur)`. Cette méthode remplace la valeur du paramètre d'une correspondance de schémas par celle fournie.
- `#ajouter(Relation rel, Objet obj)`. Cette méthode ajoute l'objet `obj` à la fin de la relation ordonnée `rel`.
- `#ajouter(Relation rel, Objet obj, Entier pos)`. Cette méthode ajoute l'objet `obj` à la position `pos` de la relation ordonnée `rel`.
- `#supprimer(Relation rel, Objet obj)`. Cette méthode supprime l'objet `obj` de la relation `rel`.
- `#supprimer(Relation rel, Objet pos)`. Cette méthode supprime l'objet à la position `pos` de la relation ordonnée `rel`.

7.4.2 Exemple

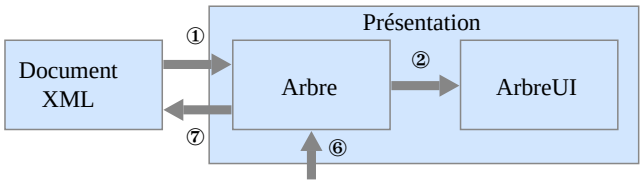
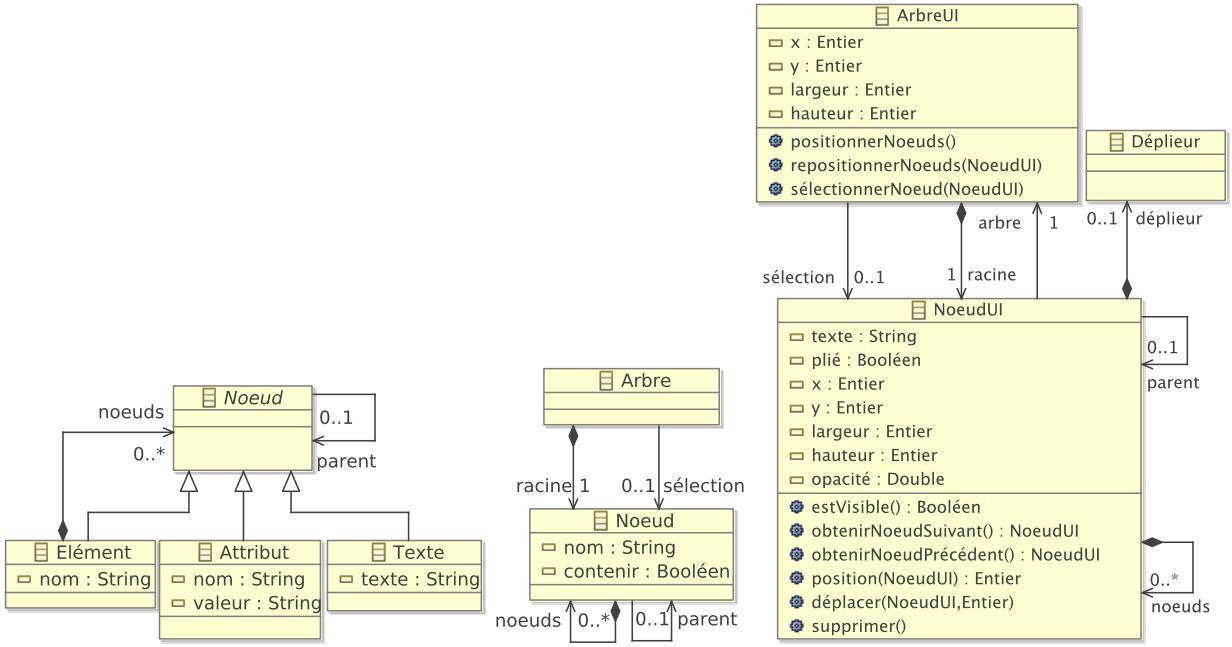


Fig. 7.6: Organisation de la présentation pour l'exemple de l'arbre XML

La figure 7.6 décrit la présentation de notre éditeur simplifié de documents XML, ainsi que les liens entre les présentations et les données sources. Les données sources correspondent à un document XML, la présentation abstraite à un arbre abstrait (`Arbre`), et la présentation concrète à un composant graphique arbre (`ArbreUI`) au travers duquel s'effectue l'édition.

La figure 7.7a décrit un métamodèle simplifié des documents XML considérés. Un document XML contient un ensemble de nœuds (classe `Noeud`), dont la racine est une instance de la classe `Elément`. Un élément est un nœud doté d'un nom et d'un ensemble de nœuds fils. Ces nœuds

[1] Une correspondance de schéma ne peut être automatiquement bidirectionnelle dans tous les cas [Fagin, 2007]. Un outil pourrait cependant générer une partie de la correspondance de schémas ⑥ à partir de la ①.



(a) Métamodèle XML (b) Présentation abstraite de l'arbre (c) Présentation concrète de l'arbre

Fig. 7.7: Métamodèle XML et présentations de l'éditeur XML

fils sont soit d'autres éléments, soit des attributs, soit des nœuds textuels. Un attribut possède un nom et une valeur. Un nœud textuel possède une valeur.

La figure 7.7b décrit la présentation abstraite de l'éditeur qui est un arbre ayant pour racine un nœud. Un nœud de cet arbre possède un nom, un éventuel nœud parent et un ensemble de nœuds fils. L'attribut `contenir` définit si un nœud peut posséder des nœuds fils.

La présentation concrète de l'éditeur (*cf.* figure 7.7c) est décrite par les classes `ArbreUI`, `NoeudUI` et `DéplieurUI`. La classe `ArbreUI` représente un arbre possédant des coordonnées, des dimensions et un nœud racine. La classe `NoeudUI` définit le texte, les coordonnées et les dimensions d'un nœud de l'arbre. L'attribut `plié` précise si le nœud affiche ses nœuds fils ; à cet effet, chaque nœud possède un composant graphique `DéplieurUI` représentant l'instrument de pliage-dépliage des nœuds.

7.5 Instrument

Un instrument se divise en deux parties, comme l'illustre la figure 7.8 : la partie abstraite définit ses données et les actions produites par l'instrument ; la partie concrète décrit la partie visible de l'instrument et les interactions qu'il utilise[2]. L'instrument joue ainsi le rôle de pivot entre ces deux parties qu'il met en relation par le biais de liaisons.

[2]Les parties concrète et abstraite sont, respectivement, appelées partie physique et partie logique par Beaudouin-Lafon (2000).

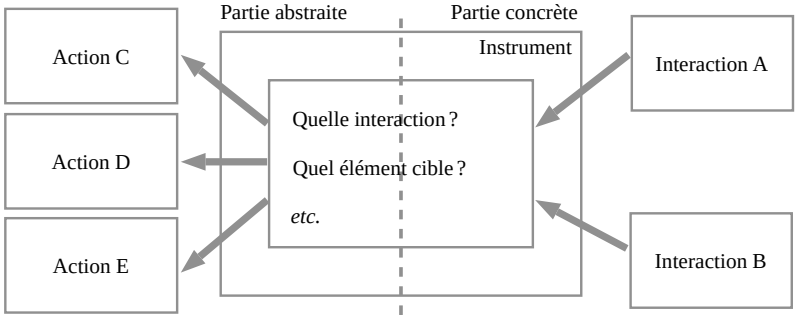


Fig. 7.8: Le passage d'une interaction à une action dans un instrument

Cette section présente tout d'abord le partie statique des interactions et des actions. Elle présente ensuite la partie statique de l'instrument.

7.5.1 Interaction

Nous définissons dans cette section la partie statique d'une interaction et donnons un exemple.

Définition et principe

La partie statique d'une interaction se rapporte aux données de l'interaction sans se préoccuper de la manière dont elle se déroule. Elle est définie par une classe et un ensemble d'évènements (*cf.* figure 7.9). Un évènement est également décrit par une classe précisant ses données.

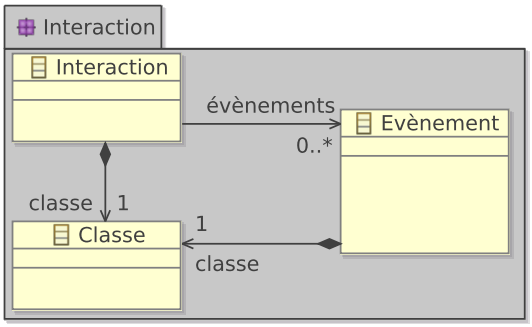


Fig. 7.9: Métamodèle d'interaction

Exemple

Certains instruments de l'éditeur XML, tels que la main, utilisent une interaction de type « glisser-déposer ». Cette instrument consiste à prendre un objet, en l'occurrence un nœud, puis à le déplacer pour ensuite le déposer dans un autre nœud parent. La partie statique de cette interaction se définit par un point initial et un point final correspondant aux positions de la prise et du dépôt de l'objet (*cf.* figure 7.10). L'attribut `source` spécifie l'objet visé, tandis que l'attribut

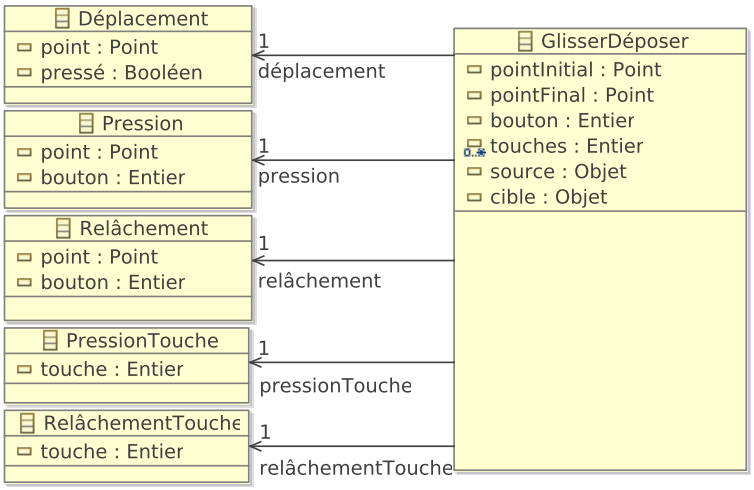


Fig. 7.10: Exemple de description statique d'une interaction

`cible` correspond à l'objet dans lequel la source est déposée. Les attributs `bouton` et `touches` définissent respectivement le bouton du HID employé et les éventuelles touches du clavier utilisées lors de l'interaction. Cette interaction est liée aux évènements suivants : `Déplacement` correspond à un déplacement du dispositif de pointage (p. ex. la souris), dont les coordonnées sont définies par l'attribut `point` et où l'attribut `pressé` définit si le dispositif exerce une pression ; `Pression` et `Relâchement` correspondent respectivement à une pression et un relâchement du dispositif de pointage, et définissent la position de la pression ou du relâchement, ainsi que le numéro du bouton utilisé ; `PressionTouche` et `RelâchementTouche` correspondent respectivement à une pression et un relâchement d'une touche du clavier, et possèdent un attribut `touche` définissant le code ASCII de la touche concernée.

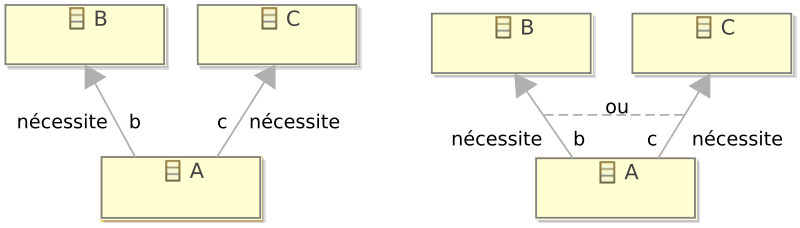
7.5.2 Action

Cette section décrit la partie statique d'une action qui est définie par une classe, dont le métamodèle a été augmenté par l'ajout de la relation de « nécessité ». Un exemple est ensuite détaillé pour illustrer cette partie de l'action.

Définition et principe

Une action résulte du besoin pour un utilisateur de réaliser une tâche ou une partie d'une tâche. Une action s'applique sur une présentation abstraite ; elle est exécutée par un instrument par le biais d'interactions. La définition d'une action s'effectue cependant indépendamment des instruments pouvant l'exécuter.

La création d'une action peut *nécessiter* l'exécution préalable d'autres actions. Dans notre éditeur, l'action `SupprimerNoeud` de l'instrument de suppression ne peut être exécutée que si une action `SélectionNoeud` a précédemment été exécutée. La figure 7.11a présente le formalisme utilisé pour représenter cette relation : l'action A nécessite l'exécution préalable d'une action



(a) Relations de nécessité (b) Relations de nécessité excluvises

Fig. 7.11: Dépendances entre actions

B et d'une action C pour être elle-même exécutée, ce qui se traduit par le prédicat suivant : $A \implies B \wedge C$. Un nom est associé à chaque relation de nécessité. Par exemple, l'action A connaît les actions B et C respectivement sous le nom de b et c (*cf.* figure 7.11a). Une action peut également nécessiter telle ou telle action. Par exemple, pour coller un nœud dans l'arbre, il est nécessaire d'avoir, au préalable, un nœud copié *ou* coupé. Ce type de contrainte est représenté par une ligne entre les deux relations de nécessité concernées. La figure 7.11b définit que l'action A nécessite l'exécution préalable d'une action B *ou* d'une action C pour être elle-même exécutée, correspondant cette fois au prédicat $A \implies B \vee C$.

La définition de relations de nécessité entre un ensemble d'actions peut cependant entraîner l'apparition de dépendances circulaires empêchant la création des actions concernées. Par exemple, une action A ne peut se nécessiter elle-même ($A \implies A$) ; de même, si $A \implies B$ et $B \implies A$, alors ni l'action A, ni l'action B ne pourront être créées. Toute implémentation de Malai devra donc permettre la vérification de ce type d'erreur.

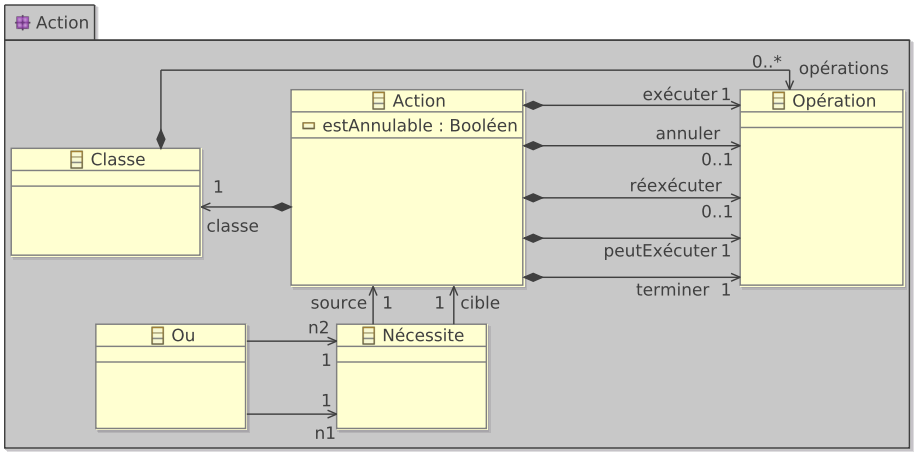


Fig. 7.12: Métamodèle de la partie statique d'une action

Le métamodèle de la partie statique d'une action (*cf.* figure 7.12) décrit qu'une action est définie par une classe dont les attributs décrivent ses données. Une action peut être annulable (attribut `estAnnulable`). Les relations de « nécessité » entre les actions sont modélisées par la

classe `Nécessite` possédant une action source et une autre cible. De même, la classe `Ou` symbolise le principe d'alternative entre des actions nécessaires : `n1` et `n2` représentent les deux relations entrant en jeu dans le `Ou`. Une action possède les opérations suivantes : `exécuter` correspond à l'exécution de l'action ; `peutExécuter` retourne « vrai » si l'action peut être exécutée ; `terminer` clôt l'action ; `annuler` et `réexécuter` annule et ré-exécute l'action si celle-ci est annulable. Ces opérations seront utiles lors de la définition de la partie dynamique (*cf.* 8).

Exemple

Le diagramme de la figure 7.13 décrit la partie statique des actions proposées par l'éditeur de documents XML. L'action à la base de toutes les autres est la sélection d'un nœud (`SélectionnerNoeud`) possédant un attribut `sélection` relatif au nœud sélectionné, ainsi qu'un attribut `arbre` correspondant à la présentation abstraite de l'arbre. La suppression (`SupprimerNoeud`), la duplication (`DupliquerNoeud`), la copie (`CopierNoeud`), la coupe (`CouperNoeud`) et le déplacement d'un nœud (`DéplacerNoeud`) sont des actions nécessitant la sélection préalable du nœud concerné. Les attributs sont utilisés lors de leur exécution et lors de leur éventuelle annulation. Ce diagramme définit également que les actions `CollerNoeud`, `CouperNoeud`, `SupprimerNoeud` et `DupliquerNoeud` sont annulables, c.-à-d. que l'utilisateur a la possibilité d'annuler leur exécution.

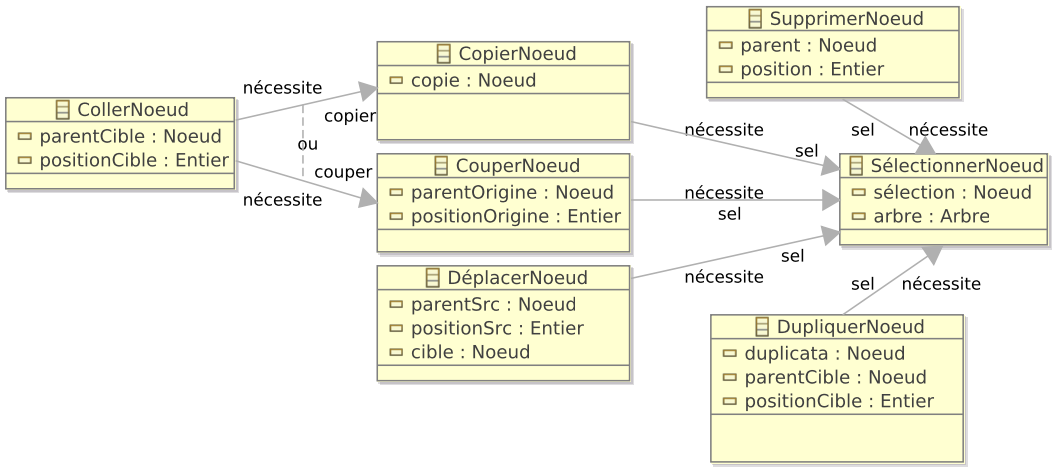


Fig. 7.13: Les actions de l'exemple de l'arbre

7.5.3 Liaison entre les interactions et les actions

La partie statique d'un instrument se compose d'une classe dotée d'un ensemble d'attributs le décrivant (*cf.* figure 7.14). La classe `Instrument` possède un attribut `activé` dont le but est de définir si l'instrument est actif ou non. Lorsqu'un instrument n'est pas actif, il ne peut être utilisé dans le SI et ses éventuels composants graphiques ne sont plus visibles ou accessibles par l'utilisateur. Dans notre éditeur XML, si un nœud ne possède pas de fils, son déplieur est désactivé et n'est alors plus visible. Un instrument possède un ensemble de liaisons établies entre

des actions et des interactions. Une liaison possède un attribut `exécution` définissant si l'action doit être exécutée au cours de l'interaction ou lorsqu'elle se termine.

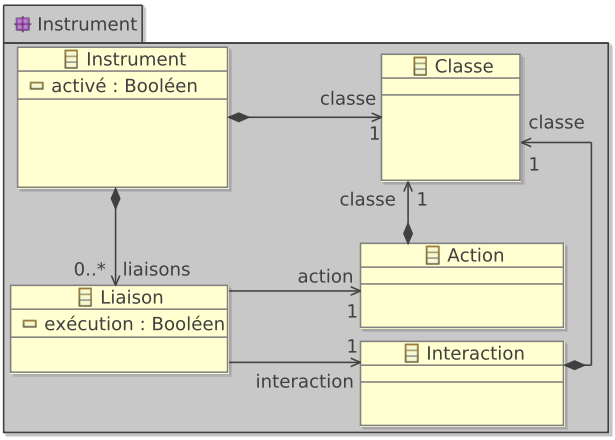
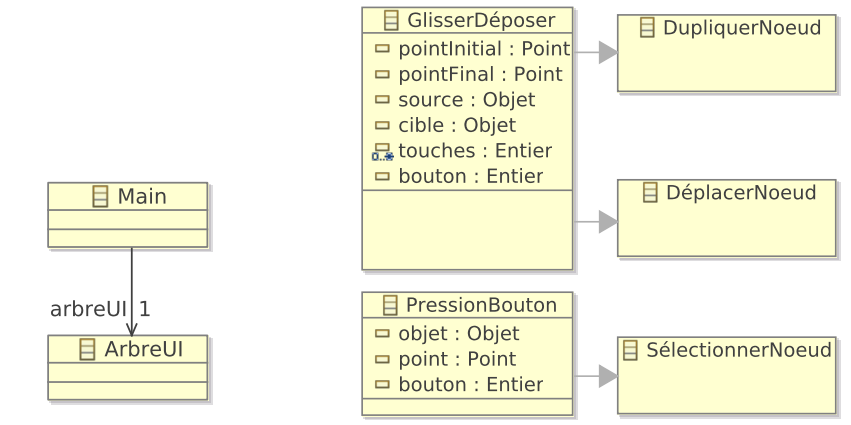


Fig. 7.14: Métamodèle d'instrument

L'instrument `Main` de l'éditeur de documents XML a pour but de sélectionner, déplacer et dupliquer les nœuds de l'arbre. La partie statique de cet instrument est définie par le diagramme de classes de la figure 7.15a. L'instrument `Main` possède un unique paramètre : la présentation concrète de l'arbre (`ArbreUI`) dans laquelle la « main » opère. La figure 7.15b définit les liaisons établies entre les interactions et les actions de l'instrument `Main`. Les deux premières liaisons stipulent que l'interaction `GlisserDéposer` crée une action `DupliquerNoeud` ou `DéplacerNoeud`. L'interaction `PressionBouton` engendre, quant à elle, une action `SélectionnerNoeud`. La partie dynamique de l'instrument décrit la manière dont ces interactions et ces actions sont liées.



(a) Diagramme de classes

(b) Liaisons entre les interactions et les actions

Fig. 7.15: Partie statique de l'instrument `Main`

7.6 Conclusion

Dans ce chapitre, nous avons présenté les parties statiques des interactions, actions, instruments et interfaces, lesquelles définissent leurs données et non leur fonctionnement. Le diagramme de classes UML est le principal formalisme utilisé dans ce cadre. Il a pour avantage de travailler à un niveau abstrait, indépendamment de toutes plates-formes d'IHM et de données. Nous avons également étendu ce type de diagramme en proposant : une relation de *nécessité* qui s'établit entre différentes actions ; une relation de *liaison* établissant le pont entre interactions et actions.

Le chapitre suivant complète la présentation du modèle Malai en décrivant la partie dynamique des interactions, des actions et des instruments. Elle a pour but de définir leur comportement en se fondant sur la partie statique.

Chapitre 8

Malai : partie dynamique des actions, des interactions et des instruments

Sommaire

8.1 Introduction

Nous présentons dans ce chapitre la partie dynamique de Malai, laquelle décrit le comportement des actions, des interactions et des instruments ainsi que le comportement des liaisons établies par les instruments entre ces dernières. Ce chapitre complète ainsi la partie statique décrite dans le chapitre précédent. Etant entièrement définies par la partie statique, l'interface et ses présentations ne sont pas présentes ici.

La figure 8.1 reprend la vue d'ensemble de Malai en mettant en exergue les parties dynamiques des interactions, des instruments et des actions. La machine à états de l'interaction (flèche ③) décrit le comportement de l'interaction à partir d'évènements. La liaison entre une interaction et une action se compose (flèche ④) : d'une *condition* liant l'interaction à l'action ; d'un éventuel *feed-back intérimaire* de l'instrument pour une présentation concrète donnée (flèche ⑤). L'action peut être exécutée, annulée ou réexécutée sur la présentation abstraite (flèche ⑥) ; ces différentes étapes sont spécifiées à l'aide d'un pseudo-langage.

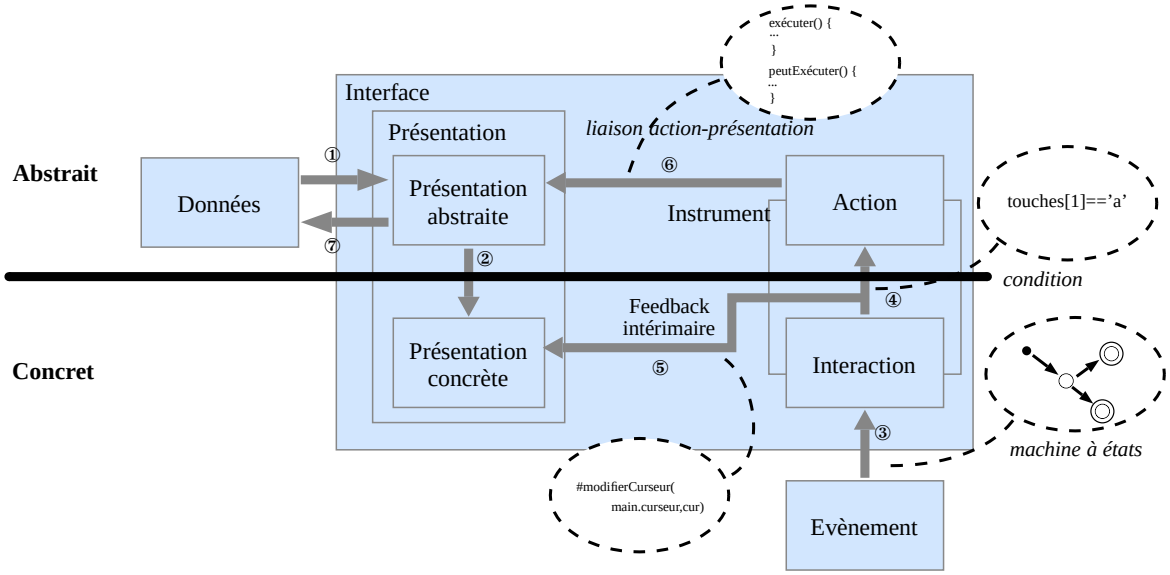


Fig. 8.1: Organisation des parties dynamiques

Ce chapitre s'organise de la manière suivante : la section 8.2 présente la partie dynamique des interactions ; la section 8.3 décrit la partie dynamique des actions ; la section 8.4 est dédiée à la partie dynamique de l'instrument ; la section 8.5 conclut la présentation du modèle conceptuel Malai.

8.2 Interaction

Cette section se consacre à la présentation de la partie dynamique d'une interaction, c.-à-d. au formalisme utilisé pour décrire son cycle de vie.

8.2.1 Définition et principe

La partie dynamique d'une interaction est définie par une machine déterministe à nombre fini d'états. Les transitions de cette dernière assurent le passage d'un état à un autre si une condition donnée est respectée. Dans notre contexte de modélisation d'une interaction, une condition est un prédicat composé de deux parties. La première se réfère au nom d'un évènement produit par un HID, tel que l'évènement `pressionTouche` produit par un clavier. La seconde, optionnelle, utilise les paramètres de l'évènement pour servir de filtre. Par exemple, la condition `pressionTouche | touche=='a'` stipule que le changement d'état s'effectue lorsqu'un évènement `pressionTouche` survient et que la touche « a » est pressée, `touche` étant un paramètre de l'évènement. Malai n'est pas pourvu d'un modèle de HID dont le but serait de décrire les HID et les évènements qu'ils produisent.

La modélisation à l'aide de machines à états fournit un moyen simple et efficace de décrire des comportements, en particulier des interactions homme-machine [Appert et Beaudouin-Lafon, 2008]. Ce principe permet également la spécification d'interactions complexes indépendamment des actions et des instruments des SI. Un développeur dispose ainsi d'une librairie d'interactions prédéfinies facilement intégrables dans différents SI, comme les interactions relatives à la souris et au clavier. Il peut également définir de nouvelles interactions venant enrichir cette librairie.

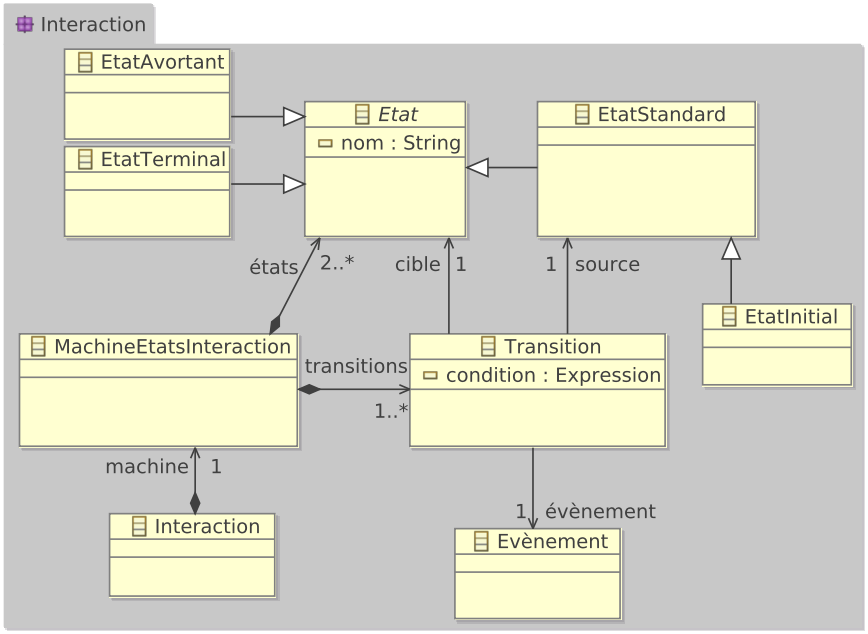


Fig. 8.2: Métamodèle d'interaction

Le métamodèle de la figure 7.14 décrit la machine à états de l'interaction. Elle se compose d'états et de transitions : chaque transition est établie entre un état source et un état cible. La classe abstraite `Etat` représente un état qui peut être :

- Un `Etat Terminal` : ce type d'état ne peut être la source d'une transition puisqu'il n'est plus possible de changer d'état lorsque l'interaction arrive à son terme. Une machine à états doit disposer d'au moins un état terminal.
- Un `Etat Initial` : une machine à états ne doit posséder qu'un seul état initial à partir duquel débute l'interaction.
- Un `Etat Avortant` : de même qu'un état terminal, un état avortant ne peut être la source d'une transition. Le but d'un état avortant est de préciser qu'un évènement termine prématurément l'interaction.
- Un `Etat Standard` : il s'agit d'un état intermédiaire entre l'état initial et les états terminaux et avortants.

8.2.2 Cycle de vie

La figure 8.3 présente le cycle de vie d'une interaction. Les transitions de ce cycle correspondent à des évènements produits par des HID auxquels peut être associée une condition. Une condition est un prédicat qui utilise des paramètres de l'évènement. Par exemple, la transition `pressionTouche | touche=='a'` fait référence à un évènement `pressionTouche` adjointe de la condition `touche=='a'`. Il existe trois types d'évènements :

1. un *évènement terminal* est un évènement dont l'état cible de la transition est un état terminal ;
2. un *évènement avortant* est un évènement dont l'état cible de la transition est un état avortant ;
3. un *évènement non terminal et non avortant* est un évènement liant un état source à un état cible non terminal et non avortant.

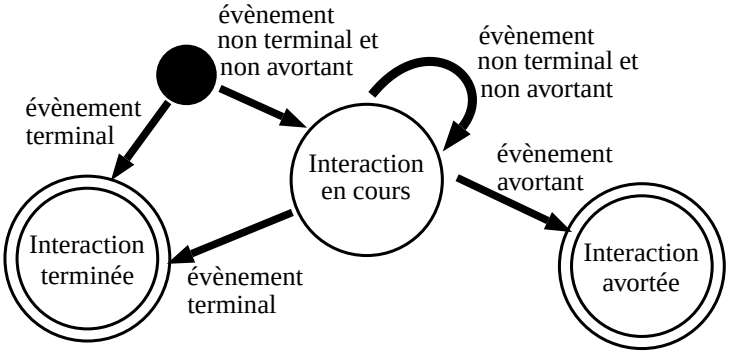


Fig. 8.3: Le cycle de vie d'une interaction

Une interaction démarre lorsqu'un évènement survient. S'il s'agit d'un évènement terminal, l'interaction se termine. Dans le cas contraire, l'interaction entre dans l'état « Interaction en cours ». L'interaction ne change pas d'état tant que des évènements non terminaux et non avortants surviennent. Lorsqu'un évènement terminal se produit, l'interaction se termine. De même, lorsqu'un évènement avortant survient, l'interaction est alors avortée (p. ex. l'appui de la touche « Echap. » lors d'un « glisser-déposer »). Ce processus suit la recommandation de la

manipulation directe stipulant qu'un utilisateur doit pouvoir arrêter toute interaction effectuée sur un système [Shneiderman, 1983].

8.2.3 Exemple

La figure 8.4 décrit la partie dynamique d'une interaction « glisser-déposer » et rappelle également sa partie statique. La figure 8.4b décrit une machine à états de la figure 8.3 appliquée à une interaction « glisser-déposer ». Certains instruments de l'éditeur de documents XML, tels que la main, l'utilisent pour déplacer ou dupliquer un nœud. Les évènements utilisés dans cette machine à états sont ceux définis dans la partie statique de l'interaction que rappelle la figure 8.4a. Le nom de chaque évènement de la machine à états d'une interaction correspond au nom de la relation établie entre la classe de l'interaction et celle de l'évènement. Cette interaction démarre avec un évènement `pression`, s'exécute tant que des évènements `déplacement` surviennent, et se termine avec un évènement `relâchement`. Elle peut être avortée si l'utilisateur appuie sur la touche « Echap. », ou si un évènement `relâchement` apparaît alors que l'état courant est l'état `pressé`. Les évènements `pressionTouche` et `relâchementTouche` des états `pressé` et `glissé` permettent l'utilisation du clavier pendant l'interaction.

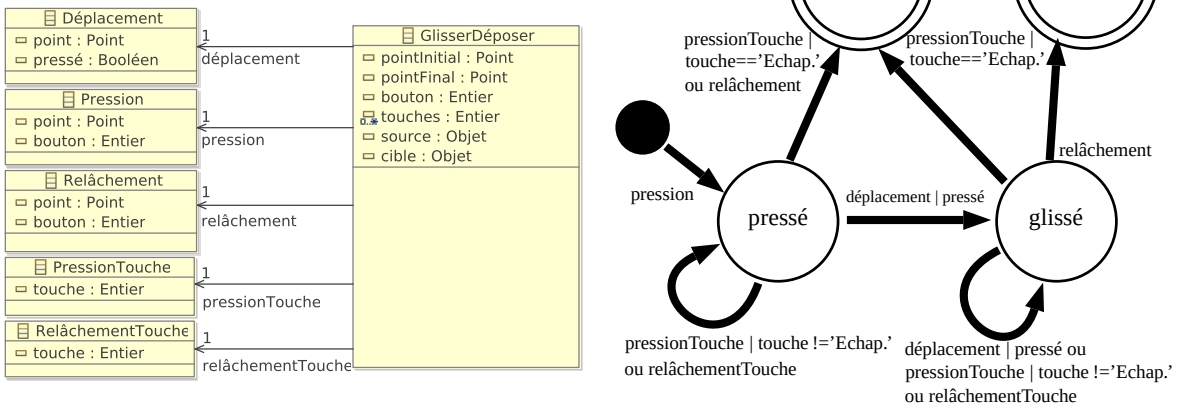
Le pseudo-code de la figure 8.4c décrit deux des cinq états de l'interaction `GlisserDéposer` : l'état initial et l'état `pressé`. L'unique transition de l'état initial (ligne 2) établit le point de pression comme étant le point initial de l'interaction. L'objet source visé lors de la pression est ensuite récupéré. Après que le bouton utilisé ait été enregistré, les instruments susceptibles d'utiliser cette interaction sont notifiés de son démarrage. Concernant l'état `pressé`, la transition, dont la condition et l'état cible sont respectivement `déplacement | pressé` et `glissé` (ligne 10), déduit le point final du « glisser-déposer » à partir du point de l'évènement `déplacement` ; l'objet cible est alors récupéré. Les instruments liés à l'interaction sont notifiés de l'évolution de l'interaction ; cette notification est valable pour tous les états non terminaux et non avortant. Les transitions (lignes 23 et 26) stipulent qu'une pression de la touche « Echap. » ou un relâchement du bouton avortent l'interaction. Les transitions (lignes 15 et 19) sont dédiées à l'utilisation du clavier. Lorsqu'une touche est pressée et qu'il ne s'agit pas de la touche « Echap. », elle est enregistrée dans l'ensemble `touches`. Si une touche est relâchée, elle est alors retirée de l'ensemble `touches`.

8.3 Action

Dans cette section, nous décrivons la partie dynamique d'une action, c'est-à-dire le formalisme utilisé pour décrire son cycle de vie.

8.3.1 Définition et cycle de vie

Le cycle de vie d'une action est donné figure 8.5. Il étend les cycles de vie des actions que l'on trouve usuellement dans la littérature et dans lesquels les actions sont créées et exécutées *après* les interactions. Les instruments mettent en relation la machine à états des interactions avec le cycle de vie des actions, comme expliqué précédemment. Ce processus permet ainsi de gérer les différentes étapes d'une action en fonction de l'interaction à laquelle elle est liée.



(a) Diagramme de classes

(b) Machine à états

```
1   Etat initial {
2     Transition pression -> pressé {
3       pointInitial = pression.point
4       source       = obtenirObjet(pointInitial)
5       bouton       = pression.bouton
6     }
7   }
8   Etat pressé {
9     Transition glissement -> glissé {
10      pointFinal = glissement.point
11      cible      = obtenirObjet(pointFinal)
12    }
13    Transition pressionTouche[touche!='Echap.'] -> pressé {
14      touches.ajouter(pressionTouche.touche)
15    }
16    Transition relâchementTouche -> pressé {
17      touches.supprimer(pressionTouche.touche)
18    }
19    Transition pressionTouche[touche=='Echap.'] -> avorté {
20    }
21    Transition relâchement -> avorté {
22    }
23  }
24  //...
```

(c) Pseudo-code

Fig. 8.4: Exemple d’une interaction « glisser-déposer »

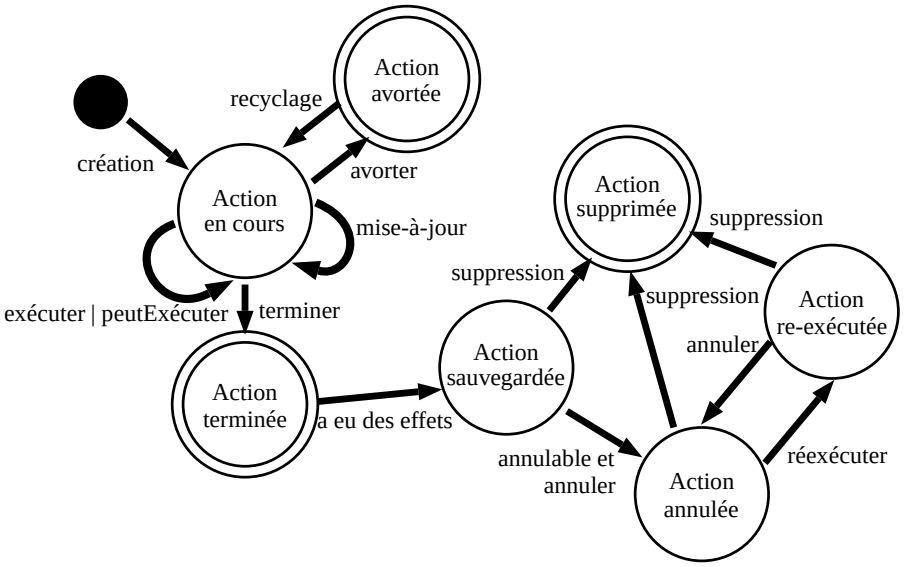


Fig. 8.5: Le cycle de vie d'une action

Une fois créée, une action en cours peut être *exécutée* et *mise à jour* plusieurs fois. La fonction *peutExécuter* retourne la valeur « vrai » si l'action peut être exécutée ; l'exécution d'une action s'effectue uniquement dans ce cas. Une action en cours peut également être *avortée* pour être éventuellement *recyclée* en une autre action qui devient la nouvelle action en cours. Par exemple, l'action `DéplacerNoeud` est tout d'abord avortée puis ensuite recyclée en une action `DupliquerNoeud` lorsque la touche « ctrl » est pressée. Une fois l'état terminal « Action terminée » atteint, si l'exécution de l'action a eu des effets sur la présentation abstraite, elle est mémorisée pour être éventuellement défaite et refaite ; sinon, le cycle de vie de l'action se termine. Une action mémorisée peut être supprimée du système (l'état terminal « Action supprimée »). Ce cas de figure peut par exemple survenir lorsque que la taille de la mémoire dédiée aux opérations d'annulation et de ré-exécution a été volontairement limitée.

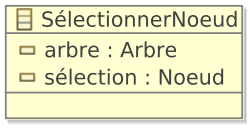
Les sections suivantes détaillent les principales étapes de ce cycle de vie.

8.3.2 Exécution, annulation et ré-exécution

L'exécution d'une action modifie une présentation abstraite. Les modifications sont ensuite propagées vers la présentation concrète, puis vers les données sources si nécessaire.

Par exemple, La sélection d'un nœud d'un arbre ne modifie pas le document source XML mais la présentation abstraite et, plus précisement, la relation `sélection` de la classe `Arbre`. Le pseudo-code de l'action `SélectionnerNoeud` (*cf.* figure 8.6b) utilise sa partie statique, rappelée dans la figure 8.6a. La méthode `exécuter` détaille l'exécution de l'action : le nœud déjà sélectionné est remplacé par le nœud `sélection`. Elle utilise pour cela la méthode `#supprimer( Relation rel, Entier pos)` qui supprime l'instance à la position `pos` dans la relation `rel`. La méthode `#ajouter(Relation rel, Objet obj)` ajoute, si possible, l'objet `obj` à la fin de la relation `rel`. La fonction `peutExécuter`, qui conditionne l'exécution de l'action, vérifie la validité des deux attributs de l'action. Lorsqu'une action `SélectionnerNoeud` est exécutée, la

correspondance de schémas Malan, établie entre les présentations abstraite et concrète, met à jour la relation `sélection` de la classe `NoeudUI`.

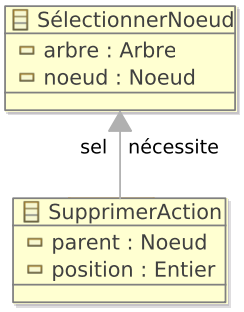


(a) Partie statique

```
1  exécuter() {
2    si(|arbre.sélection|==1)
3      #supprimer(arbre.sélection, 1)
4    #ajouter(arbre.sélection, sélection)
5  }
6  booléen peutExécuter() {
7    retourner sélection≠null et arbre≠null
8  }
```

(b) Pseudo-code

Fig. 8.6: Description de l'action `SélectionnerNoeud`



(a) Partie statique

```
1  exécuter() {
2    parent      = sel.sélection.parent
3    position    = parent.position(sel.sélection)
4    #supprimer(parent.noeuds, sel.sélection)
5  }
6  peutExécuter() { retourner sel.estExécutée() }
7  annuler() {
8    #ajouter(parent.noeuds, sel.sélection, position)
9  }
10 réexécuter() {
11   #supprimer(parent.noeuds, sel.sélection)
12 }
```

(b) Pseudo-code

Fig. 8.7: Description de l'action `SupprimerNoeud`

L'action `SupprimerNoeud`, qui supprime un nœud de la présentation abstraite, nécessite l'exécution au préalable d'une action `SélectionnerNoeud` pour être créée. Dans le pseudo-code de l'action `SupprimerNoeud` (*cf.* figure 8.7b), l'action `SélectionnerNoeud` est utilisée *via* l'identifiant `sel`, défini dans la partie statique de l'action (*cf.* figure 8.7a). L'exécution de cette action sauvegarde le parent et la position du nœud à supprimer, puis l'efface de la présentation abstraite *via* la méthode `#supprimer`. Le premier paramètre de cette méthode est la relation `parent.noeuds` concernée par la suppression ; le second est le nœud à supprimer. Cette action est à même d'être défaite et refaite : la méthode `annuler` réinsère le nœud supprimé à son ancienne position dans la relation `parent.noeud` par le biais de la méthode `#ajouter`. La méthode `réexécuter` supprime, de la même manière que la méthode `exécuter`, le nœud sélectionné.

8.3.3 Avortement et recyclage

Au cours d'une interaction, l'utilisateur peut vouloir changer l'action à réaliser. Par exemple, il peut commencer à déplacer un nœud, puis appuyer sur la touche « ctrl » pour le dupliquer. Notre modèle conceptuel prend en considération cet aspect en permettant d'avorter une action

en cours. Dans ce cas, elle peut être définitivement avortée ou bien recyclée en une action différente si certaines conditions sont respectées. Le recyclage est géré par l'instrument qui lie les interactions et aux actions comme le détaille la section suivante.

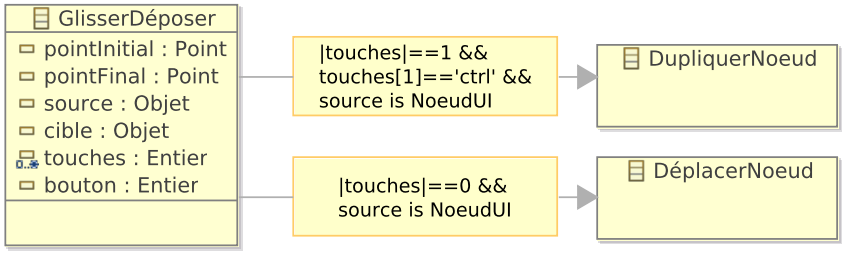


Fig. 8.8: Exemple de recyclage d'actions

La figure 8.8 décrit les conditions entre l'interaction `GlisserDéposer` de la figure 8.4b et les actions `DéplacerNoeud` et `DupliquerNoeud`. Lorsqu'un utilisateur fait glisser un nœud, une action `DéplacerNoeud` est créée, étant donné que la condition « */touches/==0 && source is NoeudUI* » est respectée. Si l'utilisateur appuie sur la touche « ctrl », l'action en cours `DéplacerNoeud` est alors avortée et recyclée en une action `DupliquerNoeud` puisque les paramètres de l'interaction correspondent maintenant à la condition « */touches/==1 && touches[1]=='ctrl' && source is NoeudUI* ». Ce processus continue jusqu'à ce que l'interaction se termine ou soit avortée.

8.4 Instrument

Nous décrivons dans cette section la partie dynamique de l'instrument en en détaillant tout d'abord les principes pour ensuite présenter son feed-back intérimaire. Enfin, un exemple d'instrument est présenté.

8.4.1 Définition et principe

Un instrument se compose d'un ensemble de liaisons chacune établie entre une interaction et une action (*cf.* figure 8.9). La partie dynamique de l'instrument décrit le fonctionnement de ces liaisons qui se composent :

1. d'une *condition* devant être respectée pour lier l'interaction à l'action ;
2. d'une *machine à états* définie à partir de celle de l'interaction et de l'action.

Condition

Un instrument décrit à quelle condition il peut créer une action d'un certain type lorsqu'une interaction donnée est effectuée. De manière formelle, soit le prédicat $\sigma(a, t, c, s)$ définissant que l'instrument s peut produire, si la condition c est respectée (c étant décrit dans une logique $\mathcal{L}$), une action a grâce à l'interaction t. Nous définissons les conditions établies entre les interactions et les actions d'un instrument s comme étant un ensemble fini Σ_s tel que :

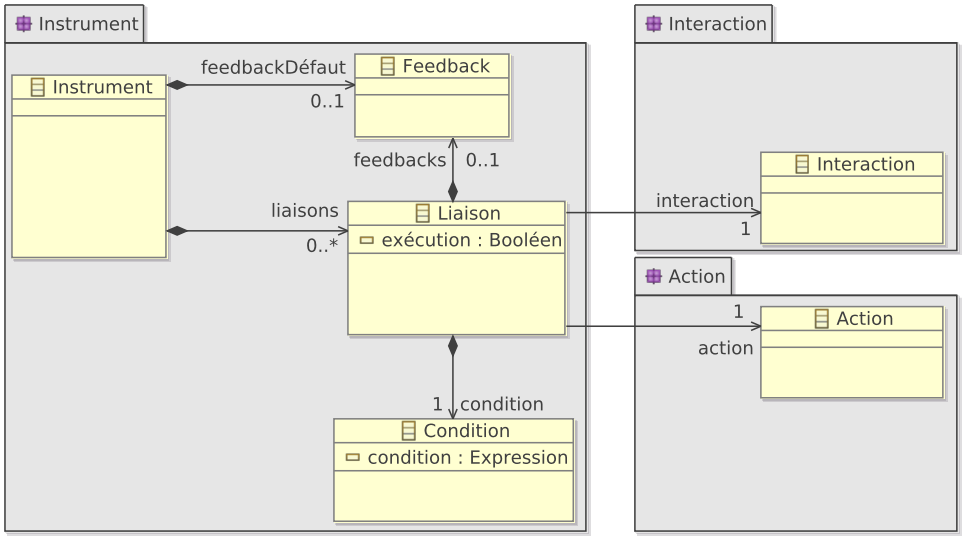


Fig. 8.9: Métamodèle d'instrument

$$\Sigma_s = \{a_1, \ldots, a_n, t_1, \ldots, t_i, c_1, \ldots, c_j, s\} \,|\, \sigma(a_1, t_1, c_1, s) \wedge \ldots \wedge \sigma(a_n, t_i, c_j, s) \tag{8.1}$$

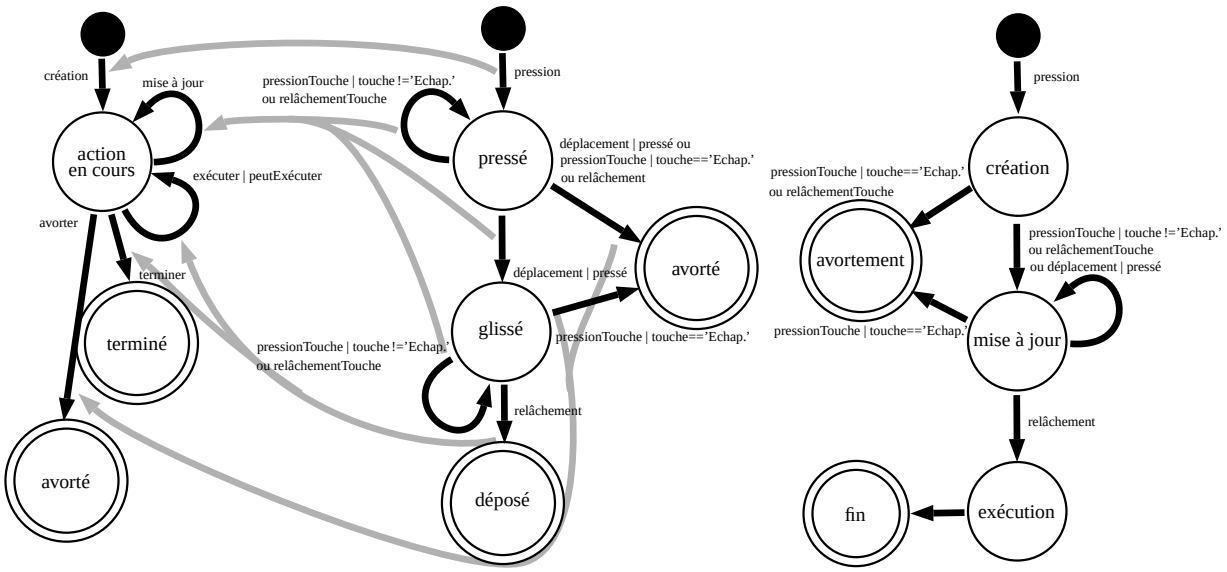
Au regard de la formule 8.1, une question se pose : une même interaction t et une même condition c peuvent-elles conduire à plusieurs actions $a_1, \ldots, a_n$, c.-à-d. $\sigma(a_1, t, c, s) \wedge \ldots \wedge \sigma(a_n, t, c, s)$? Cette situation est généralement à éviter du fait de la complexité qu'elle engendre au niveau de la perception du système du point de vue de l'utilisateur. Cependant, il existe des cas de figure dans lesquels cette fonctionnalité est nécessaire. Prenons l'exemple de l'instrument « gomme » de notre l'éditeur XML permettant de supprimer des nœuds de manière directe en cliquant sur le nœud visé. L'action `SupprimerNoeud` nécessite l'exécution au préalable d'une action `SélectionnerNoeud`. Lorsque l'utilisateur clique sur le nœud à supprimer, le nœud visé doit être sélectionné avant d'être supprimé. Ainsi, pour une même interaction (le simple clic) et une même condition (la cible de l'interaction doit être un nœud), deux actions sont créées : une pour sélectionner le nœud et une autre pour le supprimer.

La formule 8.1 se traduit dans le métamodèle de la partie dynamique de l'instrument (*cf.* figure 8.9) par le quintuplet composé des classes `Instrument`, `Liaison`, `Condition`, `Action` et `Interaction` : un instrument définit un ensemble de liaisons associées chacune à une interaction et à une action. Une liaison possède également une condition déclarée à l'aide du langage Malan.

Machine à états

La partie dynamique d'un instrument se charge de lier la machine à états de l'interaction avec celle du cycle de vie des actions. Elle définit ainsi à quel moment de l'interaction l'action doit être créée, mise à jour, exécutée, avortée et terminée en fonction des changements d'état de l'interaction. Par exemple, l'interaction `GlisserDéposer` et l'action `DéplacerNoeud` sont mises en relation par des liens représentés en gris dans la figure 8.10a. Tout d'abord, le lien entre les transitions `pression` et `création` indique que le démarrage de l'interaction avec l'évènement

`pression`, crée une action `DéplacerNoeud`. Ensuite, les transitions `déplacement | pressé`, `pressionTouche | touche=='Echap.'` et `relâchementTouche`, sont reliées à la transition `mise à jour` du cycle de vie de l'action, signifiant que ces évènements provoquent la mise à jour de l'action en cours. De même, l'évènement `relâchement` de l'état `glissé` déclenche l'exécution puis la fin de l'action. Enfin, l'avortement de l'action a lieu lorsque la touche « Echap » est pressée et lorsqu'un évènement `relâchement` survient dans l'état `pressé`.



(a) Liens entre les deux machines à états

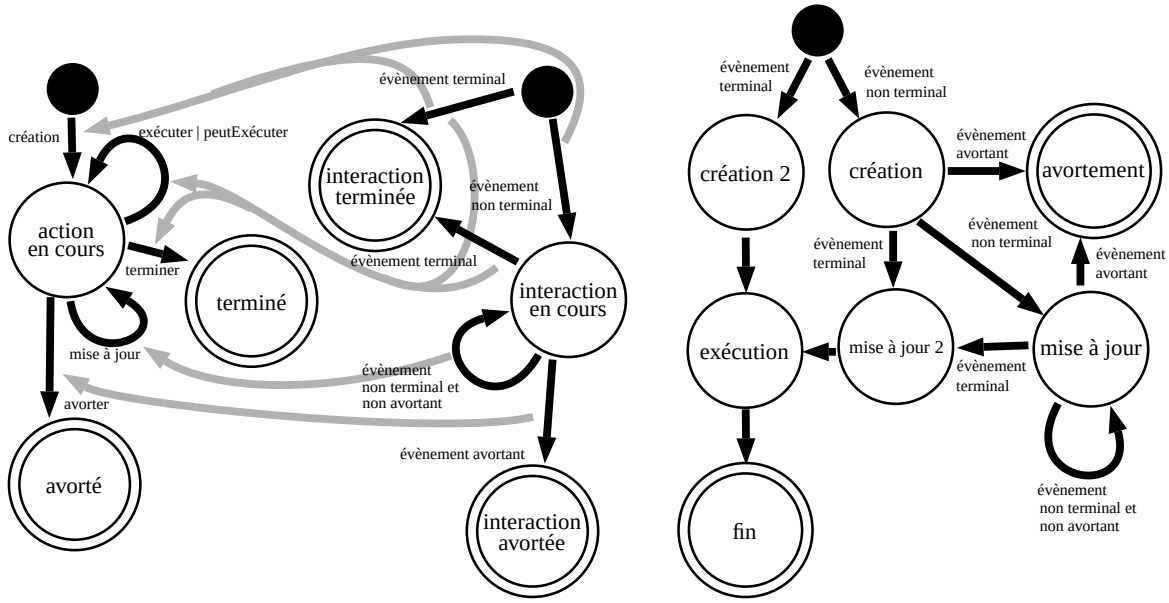
(b) Machine à états des liens

Fig. 8.10: Machine à états de la liaison entre l'interaction `GlisserDéposer` et l'action `DéplacerNoeud`

Les liens établis entre les deux machines à états de la figure 8.10a peuvent être représentés sous la forme d'une autre *machine à états*, comme l'illustre la figure 8.10b pour l'interaction `GlisserDéposer` et l'action `DéplacerNoeud`. Dans cette machine à états, les transitions correspondent aux évènements de l'interaction provoquant un changement dans le cycle de vie de l'action, et les états aux transitions du cycle de vie de l'action.

Après avoir lié différentes interactions et actions, un modèle standard de liaison a été défini. Ce modèle spécifie les liens, valables dans tous les cas étudiés, entre le cycle de vie d'une interaction et celui d'une action (*cf.* figure 8.11a). La figure 8.11b correspond à la machine à états réalisée à partir des liens établis entre les deux cycles de vie. Tout d'abord, le premier lien stipule qu'une action est créée lorsque le premier évènement conforme de l'interaction survient. Ensuite, tout évènement conforme lance la mise à jour de l'action. Si un évènement avortant intervient, l'action est avortée. De même, si un évènement terminal apparaît, l'action est exécutée puis déclarée comme terminée. A noter que la transition `évènement terminal` de l'état initial provoque tout d'abord la création de l'action.

Une condition fait partie de la machine à états établie entre l'interaction et l'action. Elle prend la forme d'un lien partant de chaque état de l'interaction vers la transition `avorter`. Le prédicat de ce lien, `évènement | !condition`, signifie que lorsqu'un évènement survient et que



(a) Liens entre les deux cycles de vie (b) Machine à états des liens

Fig. 8.11: Modèle standard de liaison

la condition n'est plus respectée, l'action A_1 doit être avortée. Si une autre condition concernant l'action A_2 est respectée, A_1 est alors recyclée en une action A_2. Pour faciliter la lecture, ces liens ne sont pas représentés dans les figures 8.11a et 8.11b.

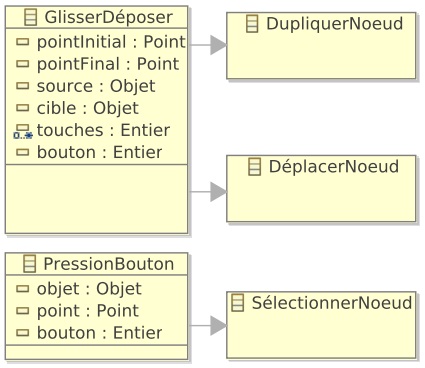
8.4.2 Feed-back intérimaire

Le feed-back intérimaire de l'instrument donne un aperçu à l'utilisateur de l'état de l'instrument, de l'action et de l'instrument en cours. Lorsque l'action est exécutée au fur et à mesure de l'interaction, l'instrument n'a pas à fournir de feed-back relatif à l'action : c'est l'exécution directe de l'action qui le joue. Dans le cas contraire, l'instrument doit fournir à l'utilisateur une réponse continue aux actions qu'il effectue sur le SI [Shneiderman, 2004]. Le métamodèle de l'instrument définit qu'une liaison peut spécifier un feed-back intérimaire (*cf.* figure 8.9). Lorsqu'au moins une liaison définit un feed-back intérimaire, l'instrument doit en spécifier un à son tour dont le but est d'annuler les effets des autres.

8.4.3 Exemple

La figure 8.12a rappelle les trois liaisons définies dans la partie statique de l'instrument `Main`. La figure 8.12b décrit la partie dynamique de cet instrument sous la forme d'un pseudo-code. Chaque liaison est représentée de la manière suivante :

```
1 Interaction "->" Action "{"
2   "condition" : condition
```



(a) Partie statique de l'instrument

```
1  Liaisons Main main {
2    var NoeudUI noeudTemp
3
4    GlisserDéposer gd -> DupliquerNoeud action {
5      condition : |gd.touches|==1 && gd.touches[1]=='ctrl'
6                  && gd.source is NoeudUI
7      feedback :
8        changerCurseur(Curseur.DUPLIQUER_CURSEUR, action)
9        mettreAJourNoeudTemp(gd)
10   }
11
12   GlisserDéposer gd -> DéplacerNoeud action {
13     condition : |gd.touches|==0 && gd.source is NoeudUI
14     feedback :
15       changerCurseur(Curseur.DEPLACEMENT_CURSEUR, action)
16       mettreAJourNoeudTemp(gd)
17   }
18
19   PressionBouton pb -> SélectionnerNoeud action {
20     condition :
21       pb.objet is NoeudUI || pb.objet is ArbreUI
22   }
23
24   défaut {
25     #modifier(main.curseur, Curseur.DEFAUT_CURSEUR)
26     main.dissocierNoeudCurseur()
27     noeudTemp = null
28   }
29
30   changerCurseur(Curseur curseur, Action action){
31     si(action.peutFaire())
32       #modifier(main.curseur, curseur)
33     sinon
34       #modifier(main.curseur, Curseur.ERREUR_CURSEUR)
35   }
36   mettreAJourNoeudTemp(GlisserDéposer gd) {
37     si(noeudTmp==null {
38       noeudTemp = nouveau NoeudUI(gd.source)
39       noeudTemp.opacité = 0.5
40       main.associerNoeudCurseur(noeudTemp)
41     }
42     noeudTemp.x = gd.pointFinal.x
43     noeudTemp.x = gd.pointFinal.y
44   }
45 }
```

(b) Partie dynamique de l'instrument

Fig. 8.12: Parties statique et dynamique de l'instrument `Main`

```
3   ("feedback"  : feedback)?
4 "}"
```

où `Interaction` et `Action` spécifient respectivement l'interaction et l'action concernées par la liaison ; `condition` correspond au prédicat de la condition à respecter ; `feedback` définit l'éventuel feed-back intérimaire de la liaison.

La première liaison concerne l'interaction `GlisserDéposer` et l'action `DupliquerNoeud` (lignes 4 à 10). Elle stipule qu'un `GlisserDéposer` provoque la création d'une action visant à dupliquer un nœud (`DupliquerNoeud`) si la source de l'interaction est le nœud visé et si l'unique touche du clavier pressée est la touche « ctrl ». Lorsque l'action est exécutable, le feed-back intérimaire de cette liaison consiste à modifier le curseur pour qu'il symbolise la duplication. Lorsque l'action n'est pas exécutable le curseur suivant est utilisé :  . De plus, une copie du nœud à copier accompagne le curseur lors du glisser-déposer. La seconde liaison concerne cette même interaction et l'action `DéplacerNoeud` (lignes 12 à 17). Elle définit que la création d'une action `DéplacerNoeud` est possible lorsqu'aucune touche du clavier n'est pressée et que l'objet source de l'interaction est le nœud visé. Le feed-back intérimaire est identique au précédent à la différence qu'un curseur symbolisant le déplacement est utilisé. La troisième et dernière liaison (lignes 19 à 22) précise que la pression d'un bouton engendre la sélection d'un nœud (action `SélectionNoeud`) lorsque l'objet ciblé par la pression est une instance de `NoeudUI` (dans le cas d'une sélection) ou de `ArbreUI` (dans le cas d'une déselection). Cette liaison ne définit pas de feed-back intérimaire.

Le feed-back des deux liaisons utilisent les méthodes `changerCurseur` et `mettreAJourNoeudTmp`. La méthode `changerCurseur` remplace le curseur courant par celui passé en paramètre si l'action est exécutable. Dans le cas contraire, un autre curseur est utilisé. La méthode `mettreAJourNoeudTmp` crée un nœud temporaire et l'associe au curseur.

8.5 Conclusion

Nous avons présenté dans ce chapitre la partie dynamique des interactions, des actions et des instruments. Les machines déterministes à nombre fini d'états sont au centre de cette partie de par leur capacité à décrire le comportement ainsi que l'état, à un instant donné, des interactions, des actions et des instruments. Ce formalisme a été préféré à celui des réseaux de Petri, tel que ICO [Navarre *et al.*, 2001], principalement pour une raison de simplicité ; malgré les interactions et les instruments complexes décrits dans les études de cas de la partie suivante, nous n'avons pas rencontré de cas pour lesquels l'expressivité des machines à états est insuffisante. C'est pourquoi les bénéfices qu'apporterait l'utilisation de réseaux de Petri nous paraissent faibles par rapport à leur complexité intrinsèque. Des travaux futurs pourraient étudier la possibilité d'utiliser indifféremment des réseaux de Petri ou des machines à états dans Malai.

Une mise en perspective avec le modèle Arch [The UIMS Tool Developers Workshop, 1992] relève des similarités (*cf.* figure 8.13) : le *noyau fonctionnel* (NF), qui représente la partie non interactive de l'application, correspond dans Malai aux données et à la présentation abstraite. Le *contrôleur de dialogue* correspond à la présentation concrète et aux actions. L'*adaptateur de NF* à la correspondance de schéma Malan liant les présentations abstraite et concrète. Parmi les différences entre Arch et Malai, notons que le modèle Arch est centré sur la notion de « widget » tandis que le modèle d'interaction Malai est fondé sur la notion d'instrument. Dans Arch, les

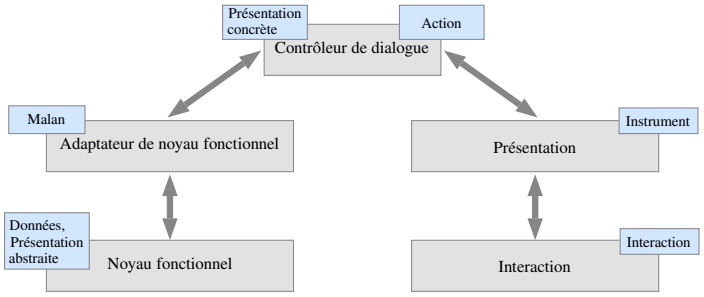


Fig. 8.13: Comparaison entre Malai et Arch

composants *Interaction* et *Présentation* correspondent respectivement aux « widgets » d'une boîte à outils et à leur représentation logique. Ces deux composants peuvent être respectivement associés aux interactions et aux instruments dans Malai : l'utilisateur n'interagit plus avec des « widgets » mais manipule des instruments. Une autre différence majeure concerne le rôle du contrôleur de dialogue : dans Arch, le contrôleur de dialogue sert de pivot entre la présentation et l'adapteur de NF contrairement à la présentation concrète dans Malai.

Le gain de développement qu'apporterait la modélisation de la partie dynamique avec Malai par rapport à l'implémentation classique de cette partie est difficilement quantifiable. Le formalisme graphique utilisé pour décrire les machines à états permet une prise de recul vis-à-vis du code. Il garantit également une entière indépendance par rapport aux plates-formes d'IHM. De plus, au vu des études de cas présentées dans la partie suivante, la réutilisation d'interactions prédéfinies et la liaison standard établie entre le cycle de vie de l'action et celui de l'interaction semblent faciliter la conception de la facette interactive de SI.

Concernant la généralité et la flexibilité de la solution, critères d'évaluation définis par Olsen (2007), Malai permet la définition de SI mais n'aborde pas les notions d'application distribuée, de réalité mixte ou d'interaction multimodale. La flexibilité de Malai réside dans la possibilité de réutiliser : des actions pour différentes interfaces utilisant des mêmes données ; des interactions pour différents instruments.

Le pseudo-code du feed-back intérimaire, des interactions sera utilisé pour concevoir un langage dédié. Ce langage viendra compléter le formalisme graphique utilisé dans ce chapitre pour représenter les machines à états des actions, des interactions et des instruments. Un développeur pourra ainsi concevoir les actions, les interactions, les instruments et les interfaces d'un SI au travers d'un éditeur dédié que nous développerons. Cet éditeur permettra ensuite de générer le code pour une plate-forme d'IHM. Ce processus nécessitera le développement de transformations de modèles entre les modèles conçus et différentes plates-formes cibles (Java, .Net, *etc.*). A l'heure actuelle, une implémentation de Malai existe à partir de laquelle différents SI ont été développés : l'éditeur XML utilisé comme exemple dans ce chapitre et le précèdent ; l'éditeur de dessins vectoriels et l'agenda présentés dans la partie suivante. Pour développer les interactions, l'implémentation de Malai se repose sur la boîte à outils SwingStates [Appert et Beaudouin-Lafon, 2008].

Quatrième partie

Utilisation de Malai et de Malan : études de cas

Introduction

Dans les parties précédentes, nous avons introduit le langage de correspondance Malan, dédié au lien entre les données sources et les présentations d'un SI, ainsi que le modèle Malai, s'occupant de la facette interactive d'un SI. Cette partie propose deux études de cas mettant en pratique Malan et Malai lors de la conception de deux SI aux caractéristiques différentes.

Le premier chapitre de cette partie présente une étude de cas concernant la conception d'un éditeur de dessins vectoriels. L'importance et la possibilité de concevoir des interactions complexes pour ce type de SI est la raison de ce choix. L'objectif principal de cette étude est d'évaluer la capacité de Malai à modéliser des actions, des interactions et des instruments basiques (interfaces WIMP) et complexes (interfaces post-WIMP).

Le second chapitre décrit une étude de cas s'intéressant à la conception d'un agenda standard dans lequel des évènements peuvent être édités. Les données utilisées dans ce SI représentent un emploi du temps d'un étudiant. Cette différence entre la sémantique des données sources et celle des présentations du SI rend la définition de leurs liaisons complexe. C'est pourquoi ce SI a été choisi au travers duquel le langage Malan est évalué.

Chapitre 9

L'éditeur de dessins vectoriels

Sommaire

9.1 Introduction

Les éditeurs graphiques sont des SI propices à l'utilisation de techniques d'interaction post-WIMP ainsi qu'à la mise en place des modèles conceptuels présentés dans le chapitre 6, page 91. L'éditeur graphique de réseaux de Petri CPN2000 colorés en est un bon exemple [Beaudouin-Lafon et Lassen, 2000; Beaudouin-Lafon et Mackay, 2000] ; cet éditeur post-WIMP met en œuvre les trois principes énoncés par Beaudouin-Lafon (*cf.* section 6.2.3, page 95), la manipulation directe (*cf.* section 6.2.2, page 94) ainsi que l'interaction instrumentale (*cf.* section 6.2.4, page 97).

Cette première étude de cas est ainsi consacrée à la conception d'un éditeur de dessins vectoriels fondé sur Malan et Malai. Du fait de l'utilisation d'interactions post-WIMP et de HID peu utilisés, cette étude de cas se focalise principalement sur la facette « interaction » de la conception, c.-à-d. sur l'évaluation du modèle Malai dans le cadre d'un SI fortement interactif. Au travers de l'éditeur, un utilisateur doit pouvoir : créer et supprimer des formes simples (rectangle, ellipse et polygone) ; transformer des formes (redimensionnement, pivotement et déplacement) ; éditer des formes (changement de l'épaisseur de la bordure, de la couleur de fond et de la bordure, affichage de la bordure, déplacement d'un point d'un polygone). Les données sources de l'éditeur sont représentées dans le format SVG, comme l'illustre la figure 9.1. A partir de ces données, la présentation abstraite `ModèleCanvas` est créée (flèche ①). Celle-ci permet de créer la présentation concrète `CanvasUI` (flèche ②). Les actions modifient l'instance `ModèleCanvas` (flèche ⑥) qui répercute les modifications sur la présentation concrète (flèche ②). Les données sources peuvent être synchronisées par la correspondance de schémas ⑦ inverse à la ②.

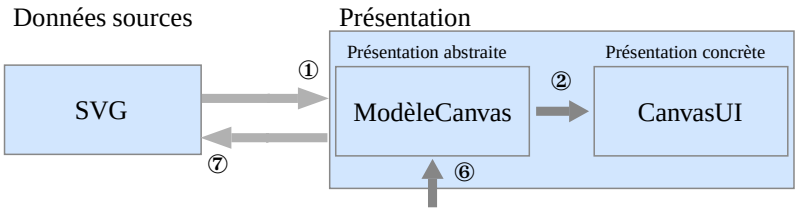


Fig. 9.1: Liens entre les données et la présentation

Ce chapitre s'organise de la manière suivante : la section 9.2 décrit le sous-ensemble du format SVG utilisé comme schéma source ; la section 9.3 présente l'interface de l'éditeur ; la section 9.4 détaille les présentations abstraite et concrète ; la section 9.5 définit les correspondances de schémas Malan entre les données sources, la présentation abstraite et la présentation concrète ; les sections 9.6, 9.7 et 9.8 sont respectivement dédiées à la définition des actions qu'un utilisateur peut réaliser, aux interactions et aux instruments utilisés, tous décrits selon le modèle Malai ; la section 9.9 conclut ce chapitre en analysant les avantages et les limites de nos travaux par rapport à l'étude de cas proposée.

9.2 Données sources

Le schéma des données sources de l'éditeur est un sous-ensemble du format SVG [W3C, 2003] contenant les classes nécessaires à la représentation d'un rectangle, d'une ellipse, d'un polygone et de leurs attributs de base comme la couleur de fond et l'épaisseur.

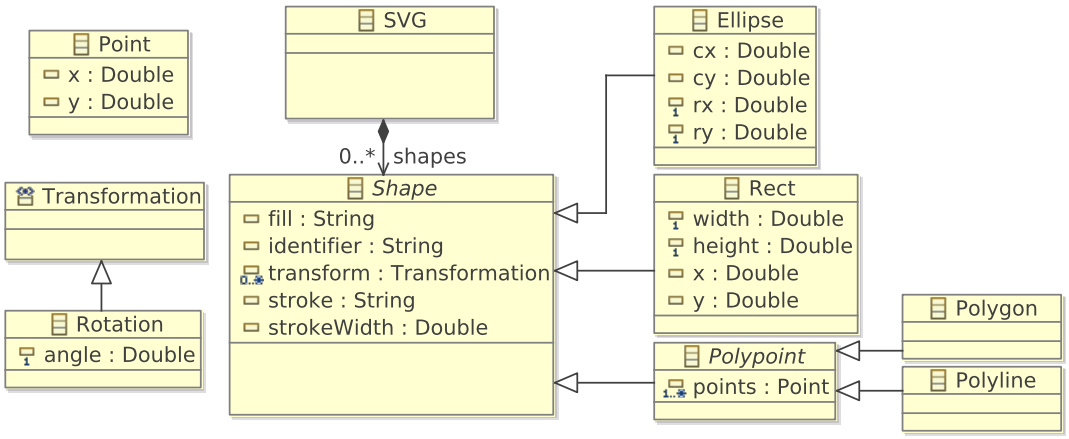
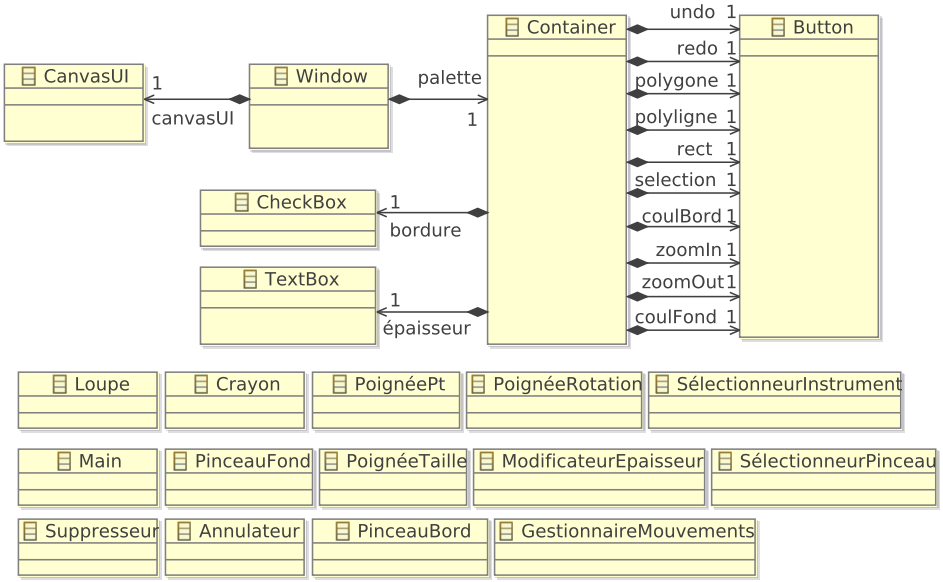


Fig. 9.2: Schéma des données de l'éditeur de dessins vectoriels

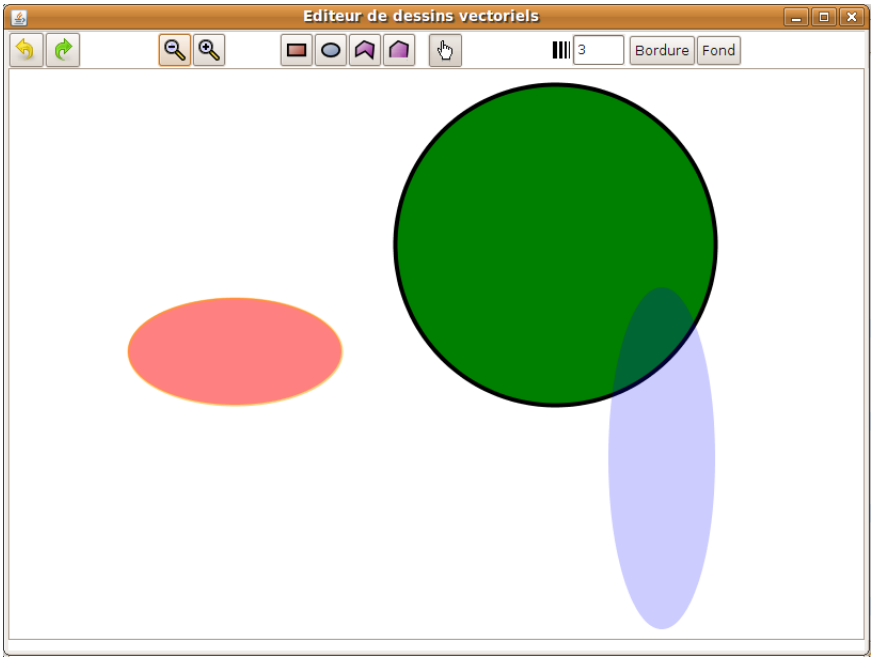
La figure 9.2 décrit le schéma des données sources dans lequel la classe `SVG` correspond à la classe racine contenant toutes les formes du dessin (classe abstraite `Shape`). La classe `Shape` possède les attributs de base nécessaires à la description de formes : `fill` définit la couleur intérieure de l'élément ; `identifier` correspond à son identifiant ; `transform` à une liste de rotations à réaliser sur les formes ; `stroke` indique la couleur de contour. Les différentes formes prises en compte sont : l'ellipse où (`cx`, `cy`) définit les coordonnées de son centre et (`rx`, `ry`) ses rayons ; le rectangle où (`width`, `height`) définit ses dimensions, et (`x`, `y`) les coordonnées de son origine ; le polypoint ouvert (polyligne) ou fermé (polygone) est défini par un ensemble de points.

9.3 Interface

L'interface de l'éditeur de dessins vectoriels est spécifiée par le diagramme de classes de la figure 9.3a. Elle se compose d'une fenêtre (`Window`) composée de la présentation concrète (`CanvasUI`) et d'un conteneur (`Container`). Ce dernier contient les instruments suivants dédiés à la manipulation des formes : la loupe modifie le zoom ; le crayon permet de dessiner des formes ; les poignées de déplacement de points, de rotation et de redimensionnement sont dédiées à la transformation des formes ; le sélectionneur des instruments d'édition permet de choisir entre la main et le crayon ; la main permet de déplacer et de sélectionner des formes ; l'annulateur annule et réexécute les actions ; le suppresseur supprime des formes ; le modificateur d'épaisseur modifie l'épaisseur des formes ; l'activateur de palettes permet d'activer les palettes de couleurs ; les palettes de couleurs de fond et de contour modifient respectivement la couleur de fond et



(a) Modèle de l'interface



(b) Interface en Swing

Fig. 9.3: Interface de l'éditeur de dessins

de contour des formes ; le gestionnaire de mouvements permet de pivoter ou redimensionner des formes.

La figure 9.3b présente l'interface finale s'exécutant sur la plate-forme Swing. La présentation concrète, au centre, contient des formes manipulables par l'utilisateur. La sélection de formes entraîne l'affichage de leurs poignées de contrôle. Ce principe illustre le fait qu'un instrument, en l'occurrence une poignée de contrôle, peut faire partie intégrante de la présentation concrète.

9.4 Présentation

La présentation abstraite de l'éditeur définit une zone de dessin sans prendre en compte les paramètres graphiques de la présentation. La présentation concrète spécifie le rendu graphique de la zone de dessin, des formes qui la composent et des instruments présents dans la présentation.

9.4.1 Présentation abstraite

La figure 9.4 décrit le schéma de la présentation abstraite. Cette dernière comprend une zone de dessin (classe `ModèleCanvas`) contenant un ensemble de formes, plus précisément des polygones, des rectangles et des ellipses. Une forme se compose d'un ensemble de points, d'une épaisseur, d'une couleur de contour, d'une couleur de fond et d'un angle de rotation. La classe `Forme` possède des opérations visant à simplifier la modification et l'accès de ses attributs. Les opérations `obtenirXMin`, `obtenirXMax`, `obtenirYMin` et `obtenirYMax` retournent respectivement la coordonnée X la plus petite parmi les points de la forme, celle la plus grande, la coordonnée Y la plus petite et celle maximale. Ces coordonnées sont respectivement modifiables par les opérations `définirPtGauche`, `définirPtDroite`, `définirPtHaut` et `définirPtBas`. Un polygone possède un attribut `ouvert` spécifiant si la ligne entre son premier et son dernier point doit être affichée.

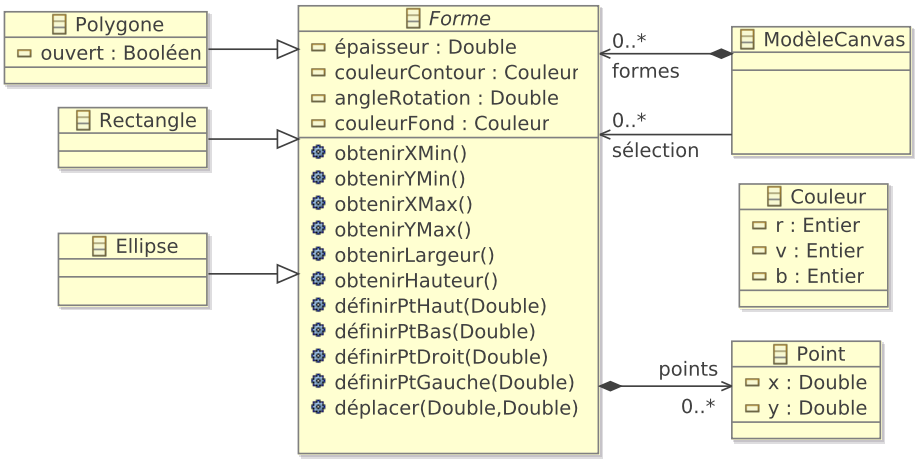


Fig. 9.4: Schéma de la présentation abstraite

9.4.2 Présentation concrète

La présentation concrète, dont le schéma est représenté par la figure 9.5, a pour racine la classe `CanvasUI`. Elle contient des instances `FormeUI` formant le dessin (relation `formesUI`) ; des instances `FormeUI` correspondant aux formes sélectionnées (relation `sélection`) ; une éventuelle instance `FormeUI` relative à une forme temporairement ajoutée à la présentation (relation `formeTmp`). La classe `FormeUI` possède : un ensemble de points ; un angle de rotation ; une épaisseur du contour ; une couleur de contour ; une couleur d'intérieur ; une bordure (classe `Bord`) ; un attribut `sélectionné` définissant si la forme est sélectionnée ou non. Une bordure contient des poignées de contrôle modifiant les dimensions (`PoignéeTailleUI`) et faisant pivoter la forme (`PoignéeRotationUI`). Une instance `FormeUI` est soit un rectangle (`RectangleUI`), soit une ellipse (`EllipseUI`), soit un polygone (`PolygoneUI`), soit un ensemble de lignes jointes (`PolyligneUI`). Le polygone et les lignes jointes possèdent des poignées de contrôle destinées à modifier l'emplacement de leurs points (classe `PoignéePoint`).

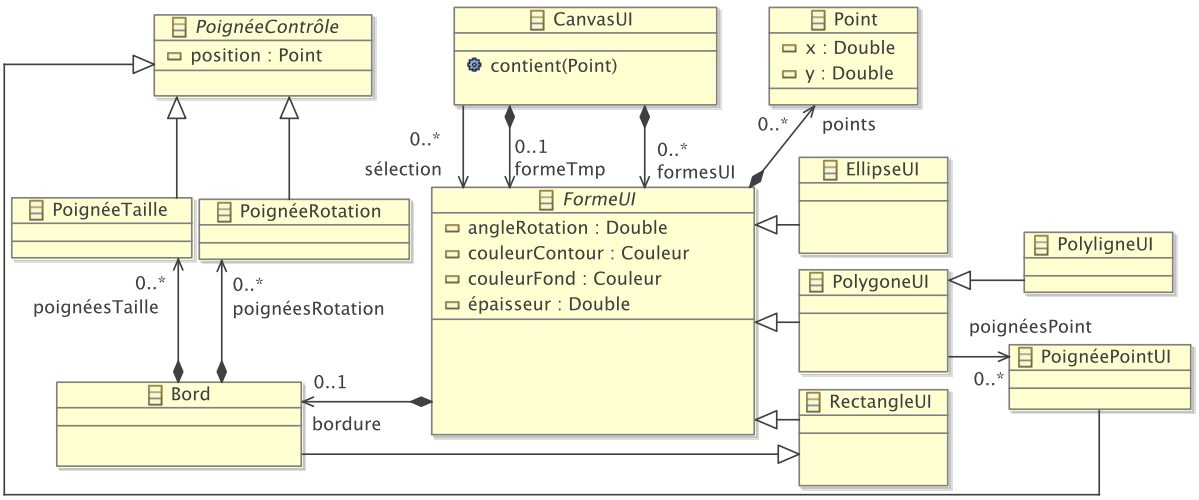


Fig. 9.5: Schéma de la présentation concrète

9.5 Correspondances de schémas

Cette section détaille tout d'abord la correspondance de schémas liant le schéma des données sources à celui de la présentation abstraite. Celle créant un pont entre les présentations abstraite et concrète est ensuite présentée.

9.5.1 Données sources vers présentation abstraite

La sémantique des données sources et celle de la présentation abstraite sont identiques puisqu'elles décrivent toutes les deux une scène graphique 2D de manière vectorielle. La correspondance de schémas présentée dans cette section est donc une translation de schémas (*cf.* section 3.2.1, page 35).

```
1 "canvas.uml/svg" -> "canvas.uml/canvasModèle" {
```

```
2   SVG -> ModèleCanvas { shapes -> formes }
3   Shape2Forme : Shape shape -> Forme {
4     strokeWidth              -> épaisseur
5     getRotationAngle(shape) -> angleRotation
6     alias f getFill(shape)
7     alias s getStroke(shape)
8     couleurFond     { f[1]->r   f[2]->v   f[3]->b }
9     couleurContour { f[1]->r   f[2]->v   f[3]->b }
10  }
11  Ellipse2Ellipse : Ellipse[rx>0 && ry>0] -> Ellipse {
12    alias cx2 |cx|==1 ? cx : 0
13    alias cy2 |cy|==1 ? cy : 0
14    4 -> |points|
15    points[1] { cx2 - rx -> x          cy2 - ry -> y }
16    points[2] { cx2 + rx -> x          cy2 - ry -> y }
17    points[3] { cx2 - rx -> x          cy2 + ry -> y }
18    points[4] { cx2 + rx -> x          cy2 + ry -> y }
19  }
20  Rect2Rectangle : Rect[width>0 && height>0] -> Rectangle {
21    alias w width/2
22    alias h height/2
23    alias x2 |x|==1 ? x : 0
24    alias y2 |y|==1 ? y : 0
25    4 -> |points|
26    points[1] { x2 - w -> x          y2 - h -> y }
27    points[2] { x2 + w -> x          y2 - h -> y }
28    points[3] { x2 - w -> x          y2 + h -> y }
29    points[4] { x2 + w -> x          y2 + h -> y }
30  }
31  Polypoint2Polygone : Polypoint[|points|>0] -> Polygone { points -> points}
32  Polygon2Polygone   : Polygon   -> Polygone          { false -> ouvert }
33  Polyline2Polygone  : Polyline  -> Polygone          { true  -> ouvert }
34  Point2Point        : Point     -> Point             { x -> x   y -> y }
35  }
```

Cette correspondance de schémas débute par la mise en relation, ligne 2, des classes `SVG` et `ModèleCanvas` représentant toutes les deux la racine du dessin. Ces classes contiennent respectivement des instances de la classe `Shape` et de la classe `Forme` liées par la correspondance de relations de la ligne 2 se rapportant à la correspondance de classes `Shape2Forme` (ligne 3). Cette dernière spécifie la relation entre les attributs de la classe `Forme` et ceux de la classe `Shape` par le biais de fonctions, définies dans l'annexe C.1, page 217, transformant la valeur source dans le format cible. Par exemple, la fonction `getFill` (ligne 11) retourne une liste de trois entiers correspondant aux trois composantes d'une couleur (rouge, vert et bleu), lesquelles sont liées à l'attribut `couleurFond`. Etant donné que les classes sources `Ellipse`, `Rect` et `Polypoints` héritent de `Shape` et que les classes cibles `Ellipse`, `Rectangle` et `Polygone` héritent, quant à elles, de `Forme`, la correspondance de relations de la ligne 2 fait également référence aux correspondances de classes `Ellipse2Ellipse` (ligne 11), `Rect2Rectangle` (ligne 20) et `Polypoint2Polygone` (ligne 31). `Ellipse2Ellipse` et `Rect2Rectangle` définissent les quatre points qui composent l'ellipse et le rectangle en utilisant les attributs des classes sources `Ellipse` et `Rect`, à condition que les rayons x et y de l'ellipse et que la largeur et la hauteur du rectangle soient supérieurs à 0. Puisque les attributs `cx`, `cy`, `x`, et `y` de ces dernières sont facultatifs, il est nécessaire de

tester leur cardinalité et de leur donner une valeur par défaut (en l'occurrence 0) si elle est nulle. Les alias `cx2` (ligne 12), `cy2` (ligne 13), `x2` (ligne 23) et `y2` (ligne 24) réalisent respectivement ces tests. La correspondance de schémas `Polypoint2Polygone` met en relation les points d'un `Polypoint` source avec ceux d'un `Polygone` cible *via* la correspondance de classes `Point2Point` (ligne 34), à condition que le nombre de points soit supérieur à 0. Les correspondances de classes `Polygon2Polygone` (ligne 32) et `Polyline2Polygone` (ligne 33) définissent si le polygone cible doit être ouvert ou non.

Bien qu'il s'agisse d'une translation de schémas, il serait difficile d'automatiser un processus de correspondance (un *« schema matching »*) entre ces deux schémas. En effet, hormis la définition des points qui est identique, les formes sont décrites dans des formalismes trop éloignés et nécessitent la définition de fonctions pour passer d'un schéma à l'autre.

9.5.2 Présentation abstraite vers présentation concrète

Le passage de la présentation abstraite vers la présentation concrète, défini dans la correspondance de schémas suivante, est relativement simple étant donnée la proximité entre la définition des deux schémas. La présentation concrète apporte des poignées de contrôle visant à transformer des formes, ainsi que la gestion de leur bord.

La correspondance de schémas possède un paramètre `zoom` (ligne 2) spécifiant le zoom à appliquer sur la présentation concrète. Le zoom aurait pu être défini dans la classe `CanvasUI`. Nous avons cependant choisi de le déclarer en tant que paramètre de la correspondance de classes afin d'illustrer l'utilisation des paramètres. Les formes abstraites et concrètes sont reliées par la correspondance de relations de la ligne 4 se référant à la correspondance de classes `Forme2FormeUI` (ligne 7). Cette dernière lie les attributs sources et cibles et en particulier l'attribut cible `bordure` (ligne 13) relatif au contour d'une forme. La spécification de ce dernier s'effectue à l'aide de la classe `Forme` (ligne 21) dont les coordonnées sont calculées grâce au zoom ainsi qu'aux opérations `obtenirXMin`, `obtenirYMin`, `obtenirLargeur` et `obtenirHauteur`. Une bordure possède également une poignée de rotation et une de redimensionnement pour chacun de ses quatre points (lignes 33 à 40). Une instance `PolylineUI`, respectivement `PolygoneUI`, est mise en relation avec une instance `Polygone` uniquement si l'attribut `ouvert` est vrai, respectivement faux (lignes 18 et 19). De plus, pour chaque point d'un `PolylineUI` et d'un `PolygoneUI`, il existe une poignée de contrôle destinée à déplacer ce point.

```
1  Modele2vueCanvas : "canvas.uml/canvasModèle" -> "canvas.uml/canvasVue" {
2    param zoom
3    ModèleCanvas -> CanvasUI {
4      formes     -> formesUI
5      sélection  -> sélection
6    }
7    Forme2FormeUI : Forme f -> FormeUI {
8      angleRotation     -> angleRotation
9      épaisseur         -> stroke.épaisseur
10     numéro            -> numéro
11     couleurContour    -> couleurContour
12     couleurFond       -> couleurFond
13     f                 -> bordure
14     points            -> points
15   }
16   Ellipse           -> EllipseUI     { }
```

```
17    Rectangle          -> RectangleUI  { }
18    Polygone[!ouvert] -> PolygoneUI   { points -> poignéesPoint }
19    Polygone[ouvert]  -> PolylineUI   { points -> poignéesPoint }
20    Point p           -> PoignéePoint { p -> position }
21    Forme -> Bord {
22      alias x obtenirXMin()*zoom
23      alias y obtenirYMin()*zoom
24      alias l obtenirLargeur()*zoom
25      alias h obtenirHauteur()*zoom
26      4 -> |points|
27      4 -> |poignéesRotation|
28      4 -> |poignéesTaille|
29      points[1] {  x   -> x   y   -> y  }
30      points[2] {  x+l -> x   y   -> y  }
31      points[3] {  x   -> x   y+h -> y  }
32      points[4] {  x+l -> x   y+h -> y  }
33      poignéesRotation[1] {  x   -> position.x   y   -> position.y  }
34      poignéesRotation[2] {  x+l -> position.x   y   -> position.y  }
35      poignéesRotation[3] {  x   -> position.x   y+h -> position.y  }
36      poignéesRotation[4] {  x+l -> position.x   y+h -> position.y  }
37      poignéesTaille[1]   {  x   -> position.x   y   -> position.y  }
38      poignéesTaille[2]   {  x+l -> position.x   y   -> position.y  }
39      poignéesTaille[3]   {  x   -> position.x   y+h -> position.y  }
40      poignéesTaille[4]   {  x+l -> position.x   y+h -> position.y  }
41    }
42    Point2Point : Point -> Point { x*zoom -> x   y*zoom -> y }
43  }
```

9.6 Actions

Cette section décrit les différentes actions que peut réaliser l'utilisateur. Le diagramme de classes UML de la figure 9.6 définit les actions, leurs paramètres et leur hiérarchie. On y trouve les actions spécifiques aux instruments et aux correspondances de schémas (classes `ModifierZoom`, `SélectionnerInsEdition` et `ModifierRotationRedim`), celles dédiées à la gestion des formes (`SélectionnerFormes`, `SupprimerFormes` et `AjouterForme`), celles visant à transformer des formes (`DéplacerPoints`, `RedimensionnerFormes`, `DéplacerFormes` et `PivoterFormes`), et celles destinées à modifier des attributs des formes (`ModifierEpaisseur`, `ModifierCoulIntérieur`, `ModifierCoulContour`). Cette dernière catégorie d'actions est décrite dans l'annexe C.3, page 222.

9.6.1 Actions spécifiques aux instruments et aux correspondances de schémas

Les actions relatives aux instruments et aux correspondances de schémas ne s'appliquent pas sur la présentation abstraite mais modifient des paramètres d'un instrument ou d'une correspondance de schémas. Cette section décrit la sélection des instruments pour un type d'édition et la modification du zoom appliqué sur les formes de la présentation concrète. L'action `ModifierRotationRedim` est détaillée dans l'annexe C.3, page 222.

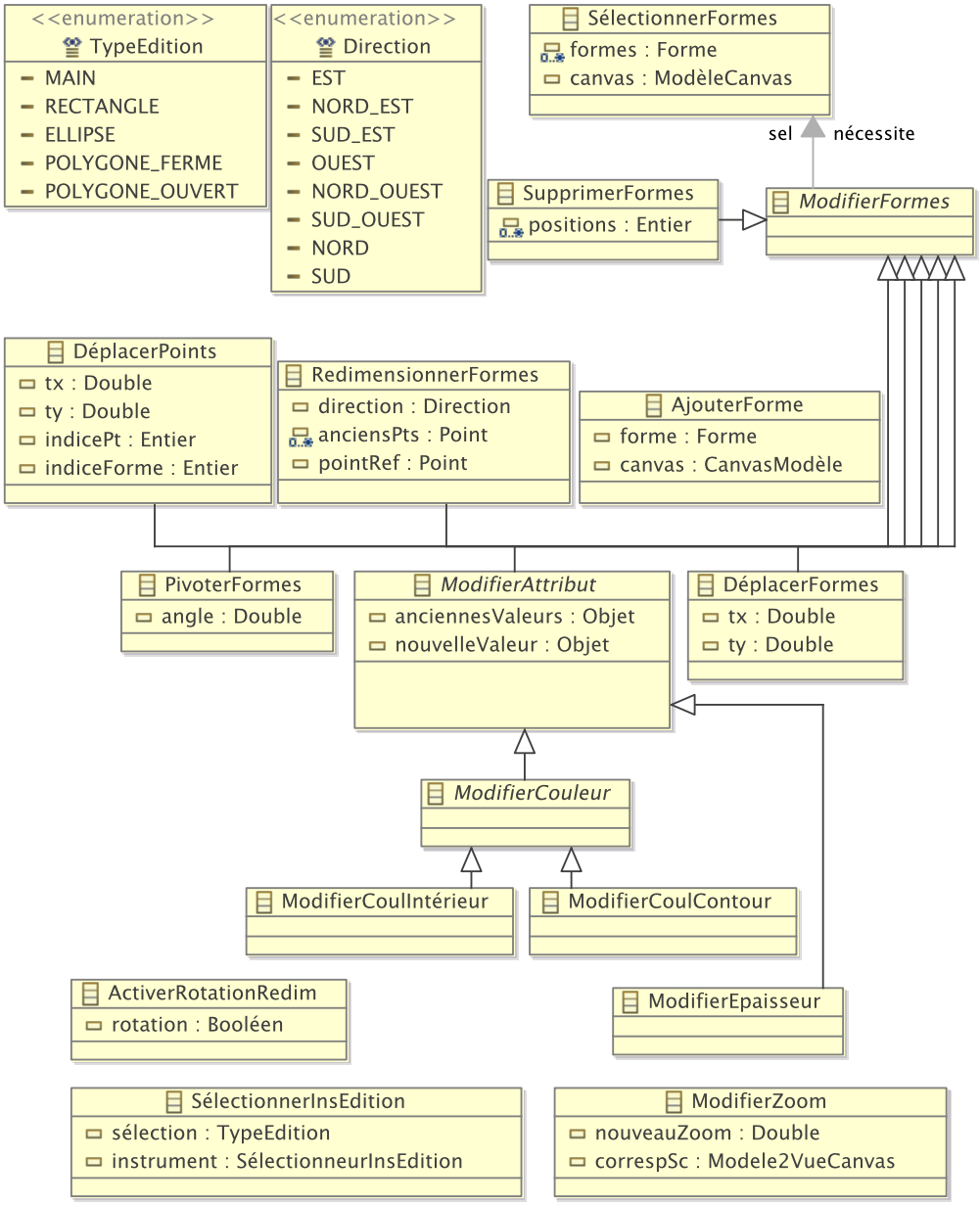
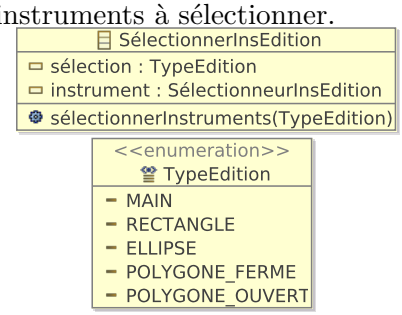


Fig. 9.6: Actions de l'éditeur

Sélectionner les instruments pour un type d'édition

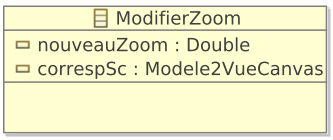
L'éditeur de dessin possède deux instruments dédiés à l'édition des formes : le crayon et la main. La gestion de ces instruments est réalisée par un instrument de type `Sélectionneur-InsEdition`. L'action `SélectionnerInsEdition` possède un attribut `sélection` spécifiant le nouveau type d'instruments à sélectionner. Les différents types sont définis au travers de l'énumération `TypeInstrument`. L'attribut `instrument` correspond à l'instrument à paramétrer. La fonction `peutExécutuer` vérifie que l'instrument et le nouveau type d'instrument à sélectionner sont bien définis. La méthode `exécuter` informe l'instrument de sélection du nouveau type d'instruments à sélectionner.




```
1  booléen peutExécuter() {
2    retourner instrument≠null et sélection≠null
3  }
4  exécuter() {
5    instrument.sélectionnerInstruments(sélection)
6  }
```

Modifier le zoom

Le pseudo-code suivant définit l'action modifiant le zoom à appliquer sur la présentation concrète. Pour cela, cette action doit attribuer au paramètre `zoom` de la correspondance de schémas liant la présentation abstraite à celle concrète (attribut `correspSc`), la nouvelle valeur du zoom (attribut `nouveauZoom`). La méthode `#modifier`, utilisée dans la méthode `exécuter` (ligne 6) réalise cette tâche. La fonction `peutExécuter` vérifie que la nouvelle valeur du zoom appartient à l'intervalle $]0;5]$, où 5 correspond à un zoom de 500%.



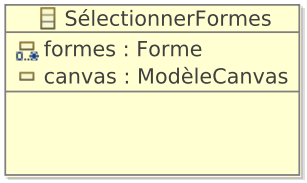

```
1  booléen peutExécuter(){
2    retourner correspSc≠null et nouveauZoom>0 et
3              nouveauZoom≤5
4  }
5  exécuter() {
6    #modifier(correspSc.zoom, nouveauZoom) }
7  }
```

9.6.2 Gestion des formes

Cette section porte sur la définition des actions dédiées à la gestion des formes, c.-à-d. aux actions visant à sélectionner, ajouter et supprimer des formes.

Sélectionner des formes

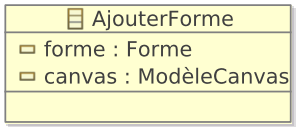
La sélection de formes, dont le pseudo-code est déclaré ci-dessous, modifie la relation `sélection` de la classe `Forme`. Cette action possède deux paramètres : `sélection` correspond aux formes à sélectionner ; `canvas` est l'instance `ModèleCanvas` contenant les formes. La méthode `exécuter` (ligne 4) supprime de la sélection les formes déjà sélectionnées (lignes 5 à 6). Elles sont ensuite remplacées par les formes de l'ensemble `sélection` (ligne 9).




```
1  booléen peutExécuter() {
2    retourner canvas≠null
3  }
4  exécuter() {
5    tant que |canvas.sélection|>0 faire
6      #supprimer(canvas.sélection, canvas.sélection[1])
7    si(sélection≠null)
8      pour chaque Forme f de sélection faire
9        #ajouter(canvasUI.sélection, selectionUI)
10 }
```

Ajouter une forme

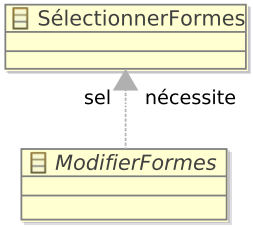
L'ajout d'une forme (attribut `forme`) dans un dessin (attribut `canvas`) s'effectue par le biais de la méthode `#ajouter` (ligne 4). Celle-ci ajoute la forme dans la relation `formes` de la classe `Forme`. L'ajout d'une forme étant annulable, les méthodes `annuler` et `réexécuter` doivent être précisées. La première (ligne 5) utilise la méthode `#supprimer` pour réaliser le travail inverse de celui de la méthode `#ajouter`. La méthode `réexécuter` (ligne 6) se charge de réappliquer l'ajout de la forme.




```
1  booléen peutExécuter() {
2    retourner forme≠null et canvas≠null
3  }
4  exécuter()      { #ajouter(canvas.formes, forme))    }
5  annuler()       { #supprimer(canvas.formes, forme)) }
6  réexécuter()    { exécuter() }
```

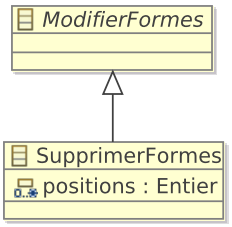
Supprimer des formes

L'action `SupprimerFormes` étend la classe abstraite `ModifierFormes` définie ci-dessous. Cette dernière stipule qu'il est nécessaire de sélectionner au préalable les formes avant de les modifier. La fonction `peutExécuter` se charge de vérifier la validité de la sélection.




```
1  booléen peutExécuter() {
2    retourner sel.estExécutée() et |sel.sélection|>0
3  }
```

L'action `SupprimerFormes` hérite de l'action `ModifierFormes` ; elle possède un unique attribut `positions` utilisé pour sauvegarder la position des formes supprimées dans la relation `formes`. La méthode `exécuter` supprime chaque forme sélectionnée de la relation `sélection` puis de la relation `formes` de la classe `Forme` (lignes 2 à 4). La position de chaque forme est sauvegardée dans l'ensemble `positions` qui est utilisé pour l'annulation de l'action (lignes 7 à 9). La méthode `#ajouter` utilise la position de l'ajout dans la relation. La méthode `réexécuter` (ligne 11) effectue le même travail que la méthode `exécuter` sans toutefois sauvegarder la position des formes.



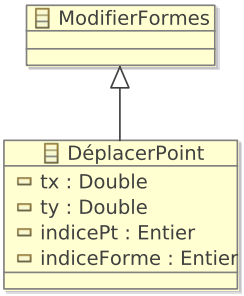

```
1  exécuter() {
2    pour chaque Forme f de sel.sélection faire
3      #supprimer(sel.canvas.sélection, f)
4      positions.ajouter(#supprimer(sel.canvas.formes, f))
5  }
6  annuler() {
7    pour i de |sel.sélection| à 1 par pas de −1 faire
8      #ajouter(sel.canvas.formes, sel.sélection[i],
9               positions[i])
10 }
11 réexécuter() {
12   pour chaque Forme f de sel.sélection
13   faire #supprimer(sel.canvas.formes, f)
14 }
```

9.6.3 Transformer des formes

Cette section se focalise sur les actions dédiées à la transformation de formes, c.-à-d. à celles visant à déplacer un point d'un polygone, à redimensionner, à faire pivoter ou à déplacer des formes. Cette dernière, étant relativement proche de l'action `DéplacerPoint`, est définie en annexe C.3, page 222.

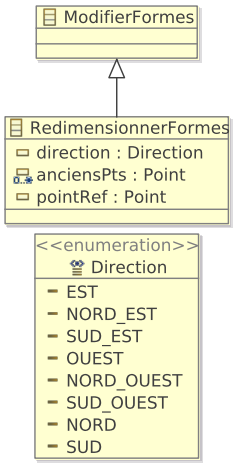
Déplacer un point d'un polygone

Le déplacement d'un point nécessite : les paramètres de translation `tx` et `ty` ; la position du polygone dans la sélection (attribut `indiceForme`) et l'indice du point du polygone (attribut `indicePt`). La fonction `peutExécuter` (ligne 1) vérifie que la valeur des deux indices est cohérente. Les méthodes `exécuter` (ligne 9) et `réexécuter` (ligne 11) utilise la méthode `translater` (ligne 12) pour effectuer le déplacement. Cette méthode récupère le point visé pour en modifier les coordonnées *via* la méthode `#modifier`. La méthode `annuler` (ligne 10) procède à la translation inverse de celle déjà réalisée.




```
1  booléen peutExécuter() {
2    si(sel.estExécutée() et indiceForme<|sel.sélection| et
3       indiceForme≥0)
4      Forme forme = sel.sélection[indiceForme]
5      retourner forme est Polygone et
6                indicePt≥0 et indicePt<|forme.points|
7    sinon retourner faux
8  }
9  exécuter()   { translater(tx, ty)   }
10 annuler()    { translater(−tx, −ty) }
11 réexécuter() { translater(tx, ty)   }
12 translater(Double translationX, Double translationY) {
13   Point pt = sel.sélection[indiceForme].points[indicePt]
14   #modifier(pt.x, pt.x + translationX)
15   #modifier(pt.y, pt.y + translationY)
16 }
```

Redimensionner des formes




```
1  booléen peutExécuter() {
2    retourner sel.estExécutée() et direction≠null et
3              pointRef≠null
4  }
5  exécuter() {
6    pour chaque Forme f de sel.sélection
7    faire pour chaque Point p de f.points
8          faire anciensPts += nouveau Point(p.x, p.y)
9    redimensionner()
10 }
11 annuler() {
12   Entier i = 1
13   pour chaque Forme f de sel.sélection
14   faire pour chaque Point p de f.points
15         faire p.x = anciensPts[i].x
16               p.y = anciensPts[i].y
17               i = i+1
18 }
19 réexécuter() { redimensionner() }
20 redimensionner() {
21   pour chaque Forme f de sel.sélection faire
22     selon(direction)
23       cas EST:
24         f.définirPtDroit(pointRef.x)
25       cas NORD_EST:
26         f.définirPtHaut(pointRef.y)
27         f.définirPtDroit(pointRef.x)
28       cas NORD:
29         f.définirPtHaut(pointRef.y)
30       cas NORD_OUEST:
31         f.définirPtHaut(pointRef.y)
32         f.définirPtGauche(pointRef.x)
33       cas SUD_EST:
34         f.définirPtBas(pointRef.y)
35         f.définirPtDroit(pointRef.x)
36       cas SUD:
37         f.définirPtBas(pointRef.y)
38       cas SUD_OUEST:
39         f.définirPtBas(pointRef.y)
40         f.définirPtGauche(pointRef.x)
41       cas OUEST:
42         f.définirPtGauche(pointRef.x)
43 }
```

Le redimensionnement de formes se réalise à l'aide d'un point de référence (attribut `pointRef`) et d'une direction (attribut `direction`) indiquant, respectivement, la position de référence et le style du redimensionnement. Après avoir sauvegardé les points à modifier dans l'attribut `anciensPts`, la méthode `exécuter` (ligne 5), au même titre que la méthode `réexécuter` (ligne 19), appelle la méthode `redimensionner` (ligne 20). Celle-ci redimensionne les formes en fonction de la direction et à l'aide des fonctions `définirPtDroit`, `définirPtGauche`, `définirPtHaut` et `définirPtBas` de la classe `Forme`. La méthode `annuler` (ligne 11) consiste à réattribuer les anciennes valeurs des points *via* l'attribut `anciensPts`. Cette action illustre le fait que les opé-

rations déclarées dans les classes sources sont utilisables au sein des correspondances de schémas et des actions, simplifiant ainsi leur développement.

9.7 Interactions

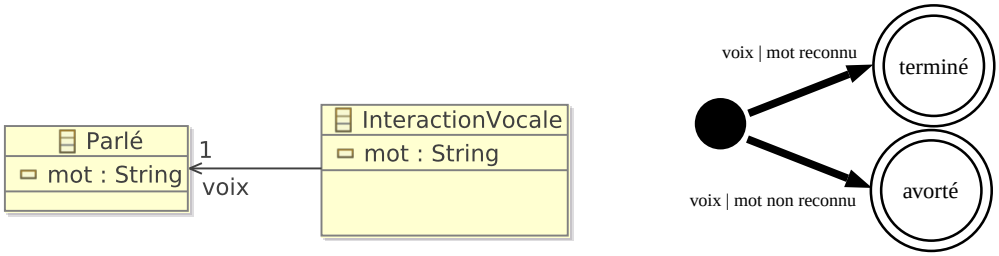
Cette section décrit les parties statique et dynamique des interactions utilisées dans l'éditeur. Parmi ces interactions se trouvent des interactions classiques, comme la pression d'un bouton, mais également des interactions post-WIMP et des interactions spécifiques à l'éditeur. Les interactions classiques sont décrites dans l'annexe C.2, page 220. Etant donné que le pseudo-code d'une interaction a déjà été expliqué (*cf.* figure 7.10, page 116), celui des interactions de cette étude de cas n'est pas détaillé.

9.7.1 Interactions post-WIMP

Afin d'illustrer la possibilité de définir et d'utiliser des interactions complexes, des interactions multimodales fondées sur la voix et le gyroscope, ainsi qu'une interaction bimanuelle sont présentées dans cette section.

Interaction vocale

L'interaction vocale, décrite par la machine à états de la figure 9.7b, débute lorsqu'un évènement `voix` se produit et se termine ou est avortée si le mot prononcé est reconnu ou non. La classe `InteractionVocale` (figure 9.7a) possède un unique attribut `mot` décrivant le mot prononcé.



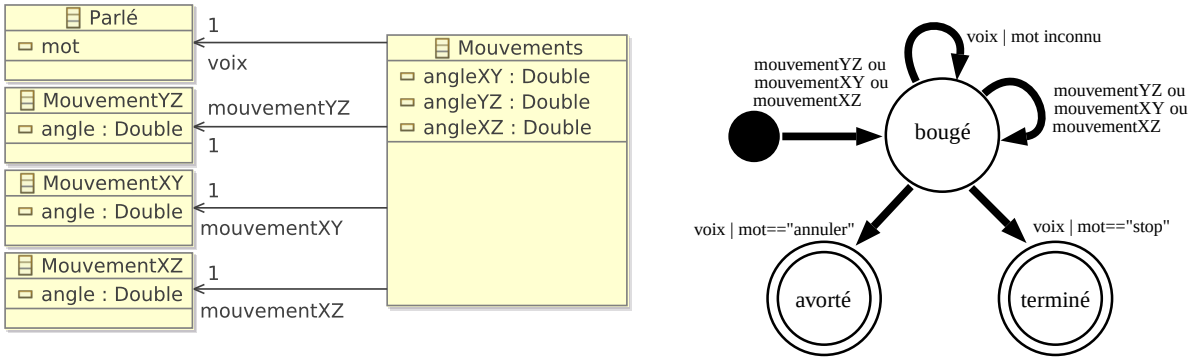
(a) Diagramme de classes

(b) Machine à états

Fig. 9.7: Interaction Vocale

Mouvements contrôlés par la voix

La figure 9.8b décrit cette interaction multimodale dans laquelle des mouvements, selon les trois plans XY, YZ et XZ, sont captés par un gyroscope jusqu'à ce que l'interaction se termine ou soit avortée. Lorsqu'un évènement `Parlé` survient, plusieurs cas de figure sont possibles en fonction du mot prononcé : pour « stop », l'interaction se termine ; pour « annuler », l'interaction est avortée ; si le mot n'est pas valide, l'interaction ne change pas d'état. La classe décrivant cette interaction (figure 9.8a) se compose des attributs `angleXY`, `angleYZ` et `angleXZ` correspondant aux angles XY, YZ et XZ produits par le gyroscope.

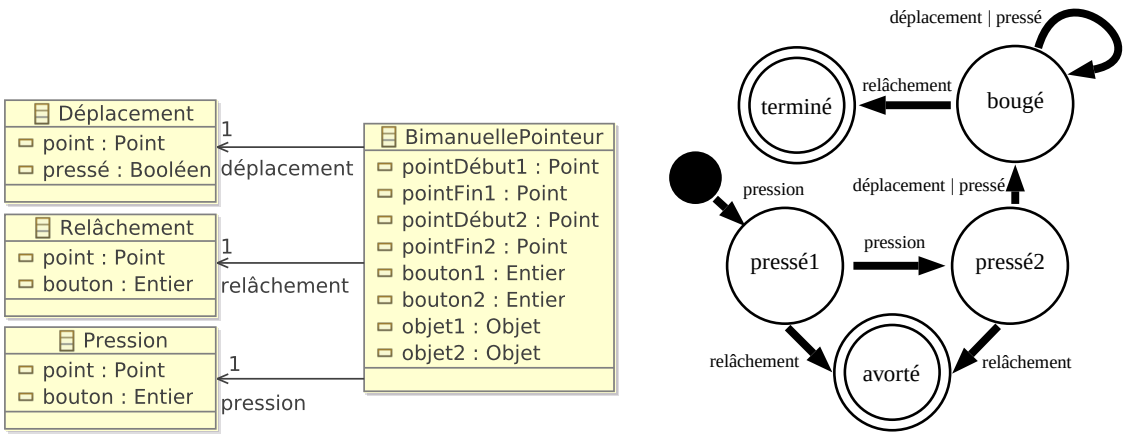


(a) Diagramme de classes (b) Machine à états

Fig. 9.8: Mouvements contrôlés par la voix

Interaction bimanuelle

L'interaction bimanuelle est utilisée pour redimensionner des formes à l'aide de deux dispositifs de pointage. La classe `BimanuellePointeur`, décrite par la figure 9.9a, se caractérise par la position de départ, d'arrivée, le bouton pressé et l'objet ciblé des deux HID. Elle utilise également trois évènements : la pression d'un bouton ; le déplacement du dispostif de pointage ; le relâchement d'un bouton. La machine à états de cette interaction (*cf.* figure 9.9b) débute par la pression d'un bouton du premier HID (état `pressé1`). S'il est relâché, l'interaction est avortée, mais si un bouton du deuxième HID est pressé, l'état `pressé2` est alors atteint. La machine à états est alors en attente d'un déplacement d'un des deux HID qui la conduirait à l'état `bougé`. Dans ce dernier, des déplacements sont possibles et ne changent pas l'état courant. Cependant, si l'un des deux boutons pressés est relâché, l'interaction se termine.



(a) Diagramme de classes (b) Machine à états

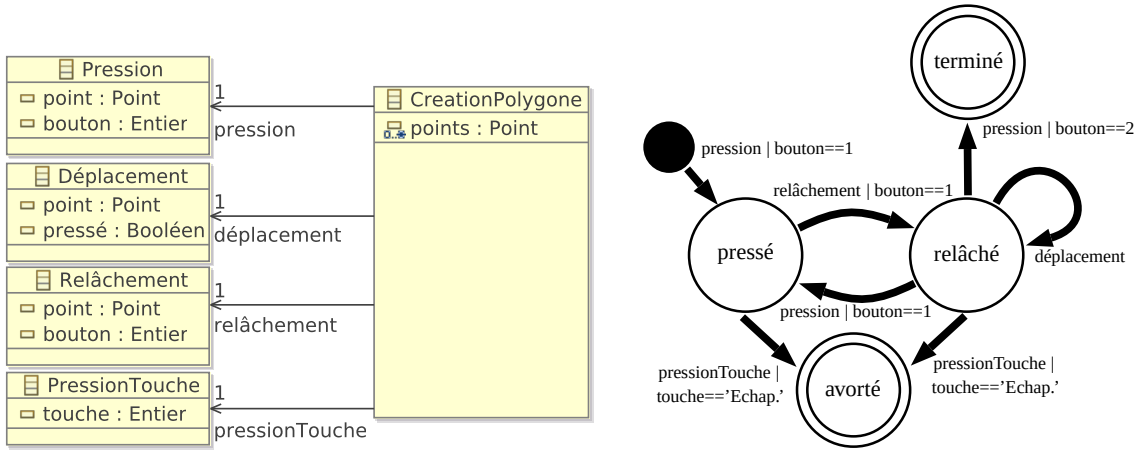
Fig. 9.9: Interaction bimanuelle

9.7.2 Interaction spécifique à l'éditeur

Cette section décrit une interaction conçue spécifiquement pour l'éditeur de dessins.

Interaction pour la création de polygones

Une interaction spéciale est utilisée pour la création de polygones. Son fonctionnement, décrit par la figure 9.10b, consiste en une succession de clics réalisés par le bouton numéro 1. Le déplacement du HID est possible entre chaque clic afin de définir le point suivant. Lors de ce processus, une pression de la touche « Echap. » avorte l'interaction. Elle se termine lorsque le bouton numéro 2 est pressé. La partie statique de cette interaction (figure 9.10a) se compose de la liste des points visés par les clics. Elle définit également les évènements utilisés, à savoir : la pression et le relâchement d'un bouton ; le déplacement du dispositif de pointage ; la pression d'une touche.



(a) Diagramme de classes

(b) Machine à états

Fig. 9.10: Interaction pour la création de polygones

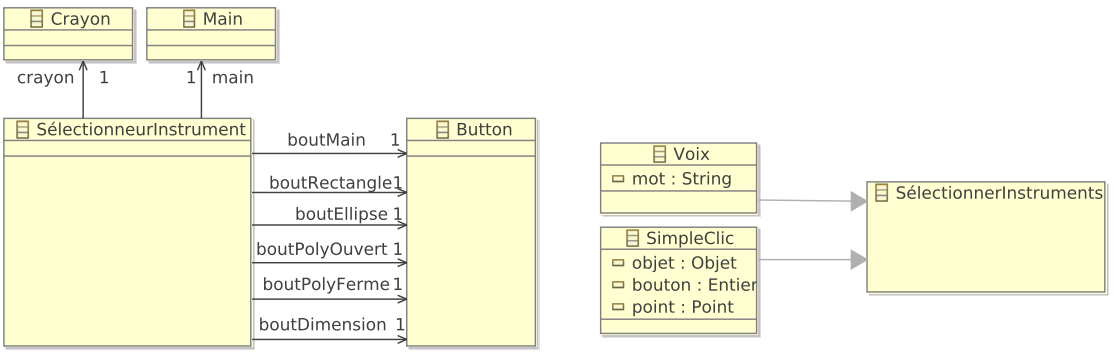
9.8 Instruments

Cette section est consacrée à la description des parties statique et dynamique des différents instruments dont dispose l'utilisateur. Les instruments transformant les formes sont tout d'abord décrits. Ceux permettant de modifier des attributs des formes (couleurs, épaisseur, *etc.*) sont ensuite présentés.

9.8.1 Les instruments dédiés à la transformation des formes

Sélectionneur d'instruments d'édition

Le but de l'instrument `SélectionneurInstrument`, dont le diagramme de classes est défini dans la figure 9.11a, est d'activer ou désactiver les instruments dédiés à différents types d'édi-



(a) Diagramme de classes (b) Liaisons entre les interactions et l'action

Fig. 9.11: Partie statique du sélectionneur d'instruments d'édition

tion, à savoir la main et le crayon. Cet instrument possède un bouton pour chacun des types disponibles.

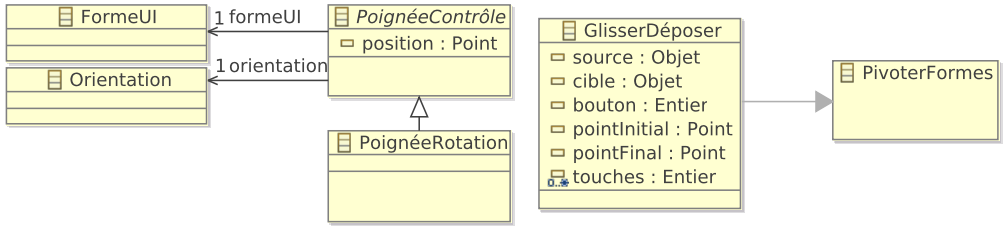
La figure 9.11b définit les liaisons entre l'action et les interactions gérées par l'instrument. Le pseudo-code ci-dessous détaille la partie dynamique de ces liaisons. Tout d'abord, le simple clic, dont la cible doit être un des boutons de l'instrument, est utilisé pour créer une action `SélectionnerInstrument` (lignes 2 à 6). La voix est également employée dans ce but : la sélection des instruments est effectuée en fonction du mot prononcé : « main », « rectangle », « ellipse », « polygone » ou « ligne » (lignes 7 à 13).

```
1  Liaisons SélectionneurInstrument ins {
2    Voix voix -> SélectionnerInstrument action {
3      condition :
4        voix.mot=="main" || voix.mot=="rectangle" ||
5        voix.mot=="ellipse" || voix.mot=="polygone" || voix.mot=="ligne"
6    }
7    SimpleClic clic -> SélectionnerInstrument action {
8      condition :
9        clic.objet==boutSelection || clic.objet==boutRectangle ||
10       clic.objet==boutEllipse || clic.objet==boutPolyOuvert ||
11       clic.objet==boutPolyFerme
12   }
13 }
```

Poignées de contrôle

Les poignées de contrôle sont des instruments faisant partie intégrante de la présentation concrète. Trois types de poignées de contrôle sont définis : celles visant à faire pivoter des formes ; celles dédiées au redimensionnement de formes ; celles déplaçant les points de polygones. Etant similaires d'un point de vue conceptuel, seule la poignée de rotation est décrite dans cette section ; les poignées de redimensionnement et de déplacement de points sont présentées en annexe C.4, page 226.

Comme le décrit le diagramme de classes de la figure 9.12a, une poignée de rotation possède : une position ; la forme à laquelle la poignée appartient ; une orientation (nord, sud, sud-ouest, *etc.*). Une action `PivoterFormes` peut être exécutée par le biais d'un glisser-déposer (*cf.* figure



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 9.12: Partie statique de la poignée de rotation

9.12b) ; la condition de cette liaison, définie dans le pseudo-code suivant, stipule que l'objet source de ce dernier doit correspondre à l'instrument (ligne 3).

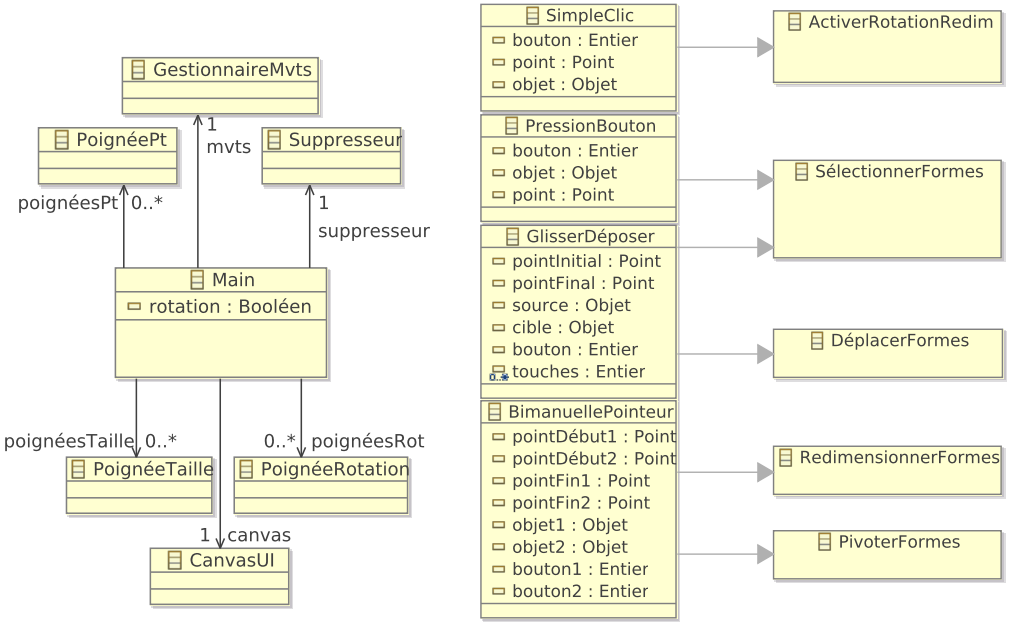
Le feed-back intérimaire de cet instrument se rapporte à un changement de curseur lorsque l'action en cours est une action de type `PivoterFormes` (lignes 4 et 7).

```
1 Liaisons PoignéeRotation poignée {
2   GlisserDéposer gd -> PivoterFormes action {
3     condition : gd.source==poignée
4     feedback  : remplacer curseur courant par curseur rotation
5   }
6   défaut {
7     remplacer curseur courant par curseur par défaut
8   }
9 }
```

Main

La main est un instrument permettant la sélection, le déplacement, le redimensionnement et la rotation de formes.

L'attribut `rotation` que possède la classe `Main` (*cf.* figure 9.13a) définit si la main doit permettre la rotation ou le redimensionnement. Lorsque `rotation` prend la valeur « vrai », les instruments liés à la rotation sont activés et ceux liés au redimensionnement désactivés. Les liaisons entres les actions et les interactions gérées par cet instrument sont représentées dans la figure 9.13b et décrites dans le pseudo-code ci-dessous. Un simple clic sur une forme déjà sélectionnée exécute une action `ActiverRotationRedim` qui modifie l'attribut `rotation` (lignes 2 à 4). Une interaction de type `PressionBouton` crée une action `SélectionnerFormes` à condition que l'objet concerné par la pression soit du type `FormeUI` pour la sélection d'une forme ou du type `CanvasUI` pour la désélection de formes (lignes 5 à 7). L'interaction `GlisserDéposer` permet de sélectionner plusieurs formes si l'objet ciblé par le point initial de cette interaction est le `CanvasUI` de l'éditeur (lignes 8 à 10) : les formes à sélectionner sont celles contenues ou en intersection avec le rectangle formé par les points initial et final du « glisser-déposer ». Cette interaction est également utilisée pour déplacer des formes (lignes 11 à 13). Dans ce cas la condition à respecter stipule que l'objet source de l'interaction doit être une instance de `FormeUI`. L'interaction bimanuelle permet de redimensionner (lignes 14 à 16) ou de faire pivoter des formes (lignes 17 à 19) en fonction de la valeur de l'attribut `rotation` et si l'objet de la première pression est une instance de `FormeUI`. La seconde pression définit le second point de



(a) Diagramme de classes (b) Liaisons entre les interactions et les actions

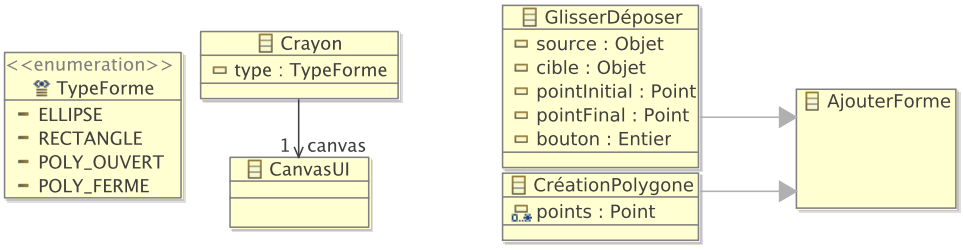
Fig. 9.13: Partie statique de la `Main`

référence du redimensionnement ou de la rotation ; ainsi, lorsque l'un des deux points bouge, le redimensionnement ou la rotation est recalculé en fonction des deux points.

```
1  Liaisons Main main {
2    SimpleClic clic -> ActiverRotationRedim action {
3      condition : clic.objet is FormeUI &&
4                  main.canvas.sélection.contains(clic.objet)
5    }
6    PressionBouton pression -> SélectionnerFormes action {
7      condition : pression.objet is FormeUI || pression.objet is CanvasUI
8    }
9    GlisserDéposer gd -> SélectionnerFormes action {
10     condition : gd.objet is CanvasUI
11   }
12   GlisserDéposer gd -> DéplacerFormes action {
13     condition : gd.objet is FormeUI
14   }
15   BimanuellePointeur bi -> RedimensionnerFormes action {
16     condition : !main.rotation &&  bi.objet1 is FormeUI
17   }
18   BimanuellePointeur bi -> PivoterFormes action {
19     condition : main.rotation &&  bi.objet1 is FormeUI
20   }
21 }
```

Crayon

Le crayon, réification du concept de crayon du dessinateur, permet la création de formes, à savoir de rectangles, d'ellipses et de polygones ouverts ou fermés. La partie statique de l'instrument (*cf.* figure 9.14a) définit un unique attribut `type` correspondant au type de formes à dessiner. Le crayon connaît également le `CanvasUI` dans lequel il dessine.



(a) Diagramme de classes (b) Liaisons entre les interactions et l'action

Fig. 9.14: Partie statique du crayon

Comme l'illustre la figure 9.14b, le crayon utilise l'interaction `GlisserDéposer` pour créer rectangles et ellipses. La condition de cette liaison, définie dans le pseudo-code ci-dessous, stipule que le type de formes à dessiner est soit « rectangle », soit « ellipse », et que le point de pression de l'interaction se situe dans le `CanvasUI` (lignes 2 à 7). Ce point, ainsi que le point final, forment un rectangle à partir duquel l'ellipse ou le rectangle est créé. La création de polygones demande une interaction spéciale : l'interaction `CréationPolygone`. L'utilisateur clique, avec le même bouton, dans le `CanvasUI` pour définir les points du polygone ; un clic avec un autre bouton termine la création. Pour créer un polygone, il faut que le type de formes à dessiner soit « polygone ouvert » ou « polygone fermé » et que le premier point se situe dans le `CanvasUI` (lignes 8 à 13).

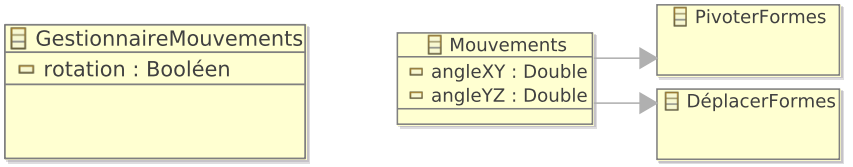
Le feed-back intérimaire de ces deux liaisons est identique (lignes 6 et 12). Il consiste à ajouter temporairement au `CanvasUI` la forme en cours de création (ligne 21) afin que l'utilisateur puisse suivre l'évolution de l'action qui réalise. Cela à condition que l'action en cours soit réalisable. Cette forme est supprimée du `CanvasUI` lorsque l'action est exécutée ou avortée (ligne 15).

```
1  Liaisons Crayon crayon {
2    GlisserDéposer gd -> AjouterForme action {
3      condition :
4        (crayon.type==TypeForme.ELLIPSE || crayon.type==TypeForme.RECTANGLE)
5        && crayon.canvas.contains(crayon.pointInitial)
6      feedback  : feedback(action)
7    }
8    CréationPolygone pol -> AjouterForme action {
9      condition :
10       (crayon.type==TypeForme.POLY_OUVERT || crayon.type==TypeForme.POLY_FERME)
11       && crayon.canvas.contains(pol.points[1])
12     feedback  : feedback(action)
13   }
14   défaut {
15     #supprimer(canvas.formeTmp, 1)
16   }
```

```
17    feedback(AjouterForme a) {
18      si(a.peutExécuter())
19      alors formeUITemp = nouveau FormeUI(a.forme)
20            #supprimer(crayon.canvas.formeTmp, 1)
21            #ajouter(crayon.canvas.formeTmp, formeUITemp)
22    }
23  }
```

Gestionnaire de mouvements

Le gestionnaire de mouvements est un instrument permettant de déplacer ou pivoter des formes en fonction de mouvements selon l'axe XY ou YZ. L'unique attribut `rotation` que possède la partie statique de cet instrument (*cf.* figure 9.15a) définit si l'instrument doit réaliser des rotations ou des redimensionnements (*cf.* figure 9.15b). Lors d'une rotation, seuls les mouvements XY sont pris en compte pour un déduire un angle de rotation. Une fois la rotation effectuée, l'utilisateur prononce le mot « stop » pour conclure l'action (voir l'interaction `Mouvements` de la figure 9.8b, page 155). Le déplacement fonctionne de manière identique à la différence que seul l'axe YZ est pris en compte.



(a) Diagramme de classes

(b) Liaisons entre l'interaction et les actions

Fig. 9.15: Partie statique du gestionnaire de mouvements

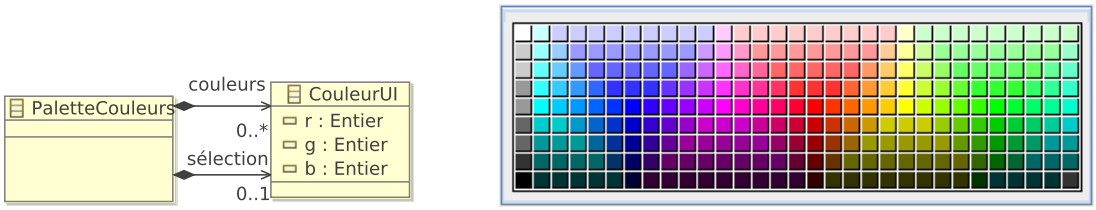
Le pseudo-code suivant décrit la partie dynamique de l'instrument. L'action `PivoterFormes` est créée si la rotation est activée (ligne 3). De même, L'action `DéplacerFormes` est créée si la rotation est désactivée (ligne 6).

```
1  Liaisons GestionnaireMouvements gest {
2    Mouvements mvt -> PivoterFormes action {
3      condition : gest.rotation
4    }
5    Mouvements mvt -> DéplacerFormes action {
6      condition : !gest.rotation
7    }
8  }
```

9.8.2 Instruments modifiant des paramètres de formes

Nous présentons dans cette section les instruments modifiant des paramètres des formes tels que la couleur de fond, l'épaisseur ou le paramètre stipulant si la bordure d'une forme doit être affichée. N'apportant que peu d'éléments intéressant, l'instrument supprimant les formes et la loupe sont définis dans l'annexe C.4, page 226.

Pinceau

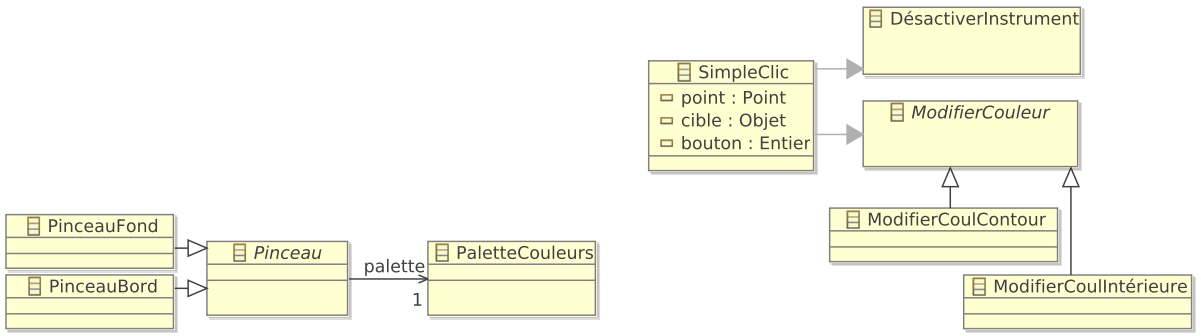


(a) Diagramme de classes

(b) Composant graphique Swing

Fig. 9.16: Composant graphique de la palette de couleurs

Le pinceau est un instrument permettant de modifier la couleur d'un objet. Cet instrument possède une palette de couleurs permettant se sélectionner la couleur à appliquer choisir une couleur parmi un ensemble proposé. La figure 9.16b est un exemple d'une palette de couleurs qui consiste en un panneau composé d'un ensemble de rectangles colorés sélectionnables symbolisant les couleurs proposées. Le diagramme de classes de la figure 9.16a décrit cette palette de couleurs (classe `PaletteCouleurs`) qui possède : un ensemble de `CouleurUI` correspondant aux différentes couleurs de la palette (relation `couleurs`) ; l'éventuelle couleur sélectionnée (relation `sélection`).



(a) Diagramme de classes

(b) Liaison entre l'interaction et l'action

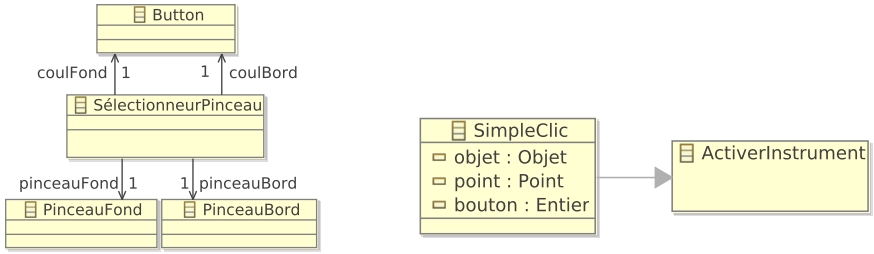
Fig. 9.17: Partie statique du pinceau

Comme le décrit le diagramme de classes de la figure 9.17a, les classes `PinceauBord` et `PinceauFond`, utilisées dans l'éditeur pour modifier les couleurs de la bordure et de fond de formes, héritent de la classe abstraite `Pinceau`. Celle-ci possède un composant graphique de type `PaletteCouleurs`. Le pseudo-code ci-dessous décrit les liaisons du pinceau. Un pinceau prend en entrée une interaction de type `SimpleClic` pour : modifier la couleur des formes sélectionnées (action `ModifierCouleur`) lorsque la cible du simple clic est du type `CouleurUI` (ligne 4) ; se désactiver lorsque le point du simple clic est en-dehors de la palette (ligne 7). Pour les `PinceauFond` et `PinceauBord`, les actions générées sont respectivement `ModifierCoulIntérieure` et `ModifierCoulBord`.

```
1 Liaisons Pinceau pinceau {
```

```
2     SimpleClic sc -> ModifierCouleur action {
3       condition :
4         pal.objet is CouleurUI && pinceau.palette.couleurs.contains(sc.cible)
5     }
6     SimpleClic sc -> DésactiverInstrument action {
7       condition :!pinceau.palette.couleurs.contains(sc.cible)
8     }
9   }
```

Sélectionneur de pinceau



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 9.18: Partie statique du sélectionneur de pinceau

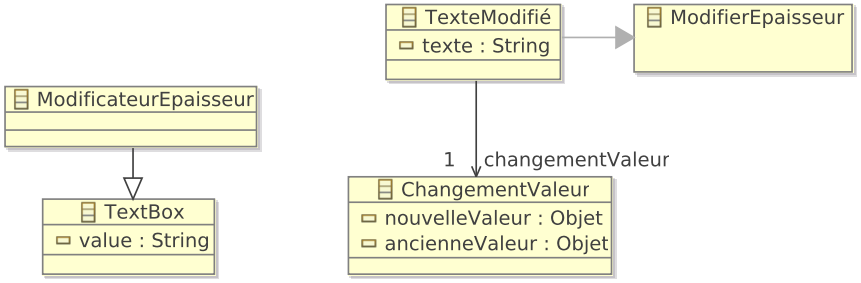
Cet instrument a pour but d'activer le pinceau `PaletteCouleurBord` ou `PaletteCouleurFond` *via*, respectivement, les boutons `coulBord` et `coulFond` (*cf.* figure 9.18a).

L'unique liaison définie pour cet instrument est établie entre l'interaction `SimpleClic` et l'action `ActiverInstrument` (*cf.* figure 9.18b). La condition de cette liaison, définie dans le pseudo-code suivant, stipule que l'activation du pinceau ne peut s'effectuer qui si le clic simple est réalisé sur un des deux boutons de l'instrument. L'instrument paramètre l'action `ActiverInstrument` pour qu'elle active la palette correspondant au bouton cliqué.

```
1   Liaisons SélectionneurPinceau select {
2     SimpleClic clic -> ActiverInstrument action {
3       condition : clic.objet==select.coulBord || clic.objet==select.coulFond
4     }
5   }
```

Instrument pour la modification de l'épaisseur

L'instrument `ModificateurEpaisseur` est dédié à la modification de l'épaisseur de formes. Cet instrument se fonde sur un instrument de type `TextBox`, comme l'illustre le diagramme de classes de la figure 9.19a. Cet exemple d'instrument met en avant la difficulté d'utiliser un modèle instrumental avec des composants graphiques déjà existants : le `TextBox`, tiré de notre modèle de composants graphiques (*cf.* figure B.1, page 216), est un composant graphique standard disponible sur la plupart des plates-formes d'IHM. Celles-ci définissent déjà un modèle d'interaction pour ce composant graphique qui peut différer d'une plate-forme à une autre. Il n'est donc pas nécessaire de redéfinir une interaction pour l'utiliser ; c'est le passage de l'interface concrète à l'interface finale qui doit définir le lien entre le `TextBox` et le composant graphique



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 9.19: Parties statique de l'instrument `ModificateurEpaisseur`

correspondant de la plate-forme d'IHM sélectionnée. Les liaisons ne s'établissent donc pas en utilisant les interactions du composant graphique qui dépendent de la plate-forme d'IHM choisie, mais avec une interaction fondée sur l'évènement engendré par ces interactions : par exemple, la liaison de l'instrument `ModificateurEpaisseur` stipule que l'interaction `TexteModifié` crée une action `ModifierEpaisseur` (*cf.* figure 9.19b, page 164). `TexteModifié` ne définit pas la manière dont l'utilisateur interagit avec le `TextBox`, mais se fonde sur l'évènement `ChangementValeur` correspondant au résultat de cette interaction.

La liaison de la figure 9.19b, page 164 est décrite par le pseudo-code ci-dessous. La condition de cette liaison stipule que la nouvelle valeur du champ de texte doit être un nombre supérieur à 0 (ligne 3).

```
1 Liaisons ModificateurEpaisseur ep {
2   TexteModifié tm -> ModifierEpaisseur action {
3     condition : isInteger(tm.texte) && toInteger(tm.texte)>0
4   }
5 }
```

9.9 Conclusion

Nous avons présenté dans ce chapitre une étude de cas dédiée à la conception d'un éditeur de dessins vectoriels. L'étude se focalise principalement sur la facette interactive (actions, instruments et interactions) définie avec le modèle conceptuel Malai. Ce modèle a notamment montré qu'il pouvait décrire aussi bien des interactions et des instruments classiques (double clic, glisser-déposer, *etc.*) que complexes (mouvements contrôlés par la voix, interaction bimanuelle, *etc.*).

Cette étude de cas a mis en avant le fait que le modèle Malai ne décrit pas en détail certaines informations, comme celles concernant la mise à jour d'une action par un instrument. Une des prochaines étapes pour compléter Malai consiste à définir un langage dédié à la description des actions, de la mise à jour d'une action par un instrument, du feed-back intérimaire, *etc.* Ce langage sera défini à partir du pseudo-code utilisé dans ce chapitre et le suivant.

Chapitre 10

L'agenda

Sommaire

10.1 Introduction

Les applications gérant les emplois du temps, comme Google calendar[1]et Lightning[2], sont des SI largement utilisés se fondant, pour la plupart, sur le principe de la manipulation directe : un calendrier contient des évènements éditables de manière directe et indirecte. Ce type de SI est notamment au centre de l'étude de cas utilisée dans [Beaudoux *et al.*, 2009] pour présenter l'environnement AcT (*cf.* section 3.6).

Cette seconde étude de cas se concentre sur la conception d'un agenda usuel. Au travers de ce SI, un utilisateur doit pouvoir : créer et supprimer des évènements ; changer leurs horaires ; modifier la description d'évènements ; naviguer dans l'agenda. Les données manipulées par ce SI sont multiples : il peut s'agir par exemple d'un agenda personnel ou d'emplois du temps d'étudiants qu'un responsable des études doit organiser. Cette dernière proposition est celle visée par cette étude de cas. Cette dernière se focalise sur les liens entre les données et les présentations abstraite et concrète, qui nécessitent des calculs non triviaux pour notamment définir la position des évènements de l'agenda. La figure 10.1 décrit les liens entre les données et leurs présentations. La présentation abstraite (`AgendaCanvas`) est créée à partir d'emplois du temps d'étudiants (flèche ①). De même, la présentation concrète (`AgendaCanvasUI`) est créée à partir de la présentation abstraite (flèche ②). Les actions modifient la présentation abstraite (flèche ⑥) qui répercute ensuite les modifications à la présentation concrète (flèche ②). Les données sources peuvent être synchronisées à partir de la présentation abstraite (flèche ⑦) par une correspondance de schémas inverse à la numéro ①. L'agenda personnel est donné à titre indicatif pour illustrer le fait que plusieurs modèles de données sources peuvent être utilisés pour un même modèle de présentation et d'instrument.

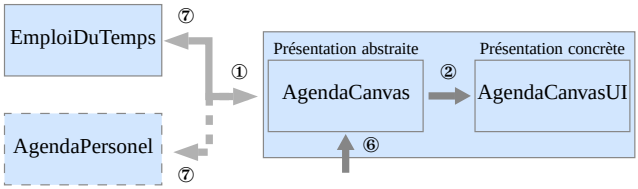


Fig. 10.1: Liens entre les données sources et les présentations cibles

Ce chapitre s'organise de la manière suivante : la section 10.2 est consacrée à la présentation du schéma des données sources définissant un emploi du temps d'un étudiant ; la section 10.3 présente l'interface de l'agenda ; la section 10.4 détaille les présentations abstraite et concrète ; la section 10.5 définit les correspondances de schémas Malan entre les données sources et la présentation abstraite, et entre cette dernière et la présentation concrète ; les sections 10.6 et 10.7 sont dédiées respectivement à la définition des actions et des instruments, tous décrits *via* le modèle Malai ; la section 10.9 conclut ce chapitre en analysant les avantages et les limites de notre approche pour la réalisation de cette étude de cas.

[1]http ://www.google.com/calendar
[2]http ://www.mozilla.org/projects/calendar/lightning/

10.2 Données sources

Les données sources de l'agenda, dont le schéma est défini dans la figure 10.2, décrivent un emploi du temps d'un étudiant. Un emploi du temps (classe `EmploiDuTps`) se compose de ressources et de semaines. Une ressource désigne : un enseignant décrit par son nom et son prénom ; une matière possédant un intitulé et faisant référence aux enseignants concernés ; une plage horaire possédant une date de début et une date de fin ; une salle (amphithéâtre ou laboratoire) pourvue d'une capacité et d'un nom. Une semaine est décrite par : l'année à laquelle elle se réfère ; le numéro de division et le nom de la promotion de l'étudiant ; le numéro de la semaine ; les cinq jours de cours (du lundi au vendredi). Une journée se compose d'enseignements, c.-à-d. de cours magistraux (classe `Cours`), de travaux pratiques (classe `TP`), et de travaux dirigés (classe `TD`). Un enseignement concerne une matière, est assuré par un enseignant, a lieu dans une salle, et se déroule sur une ou deux plages horaires consécutives. Si aucun enseignant n'est précisé pour un enseignement, le premier enseignant de la matière de l'enseignement (association `Matiere.enseignants`) est considéré.

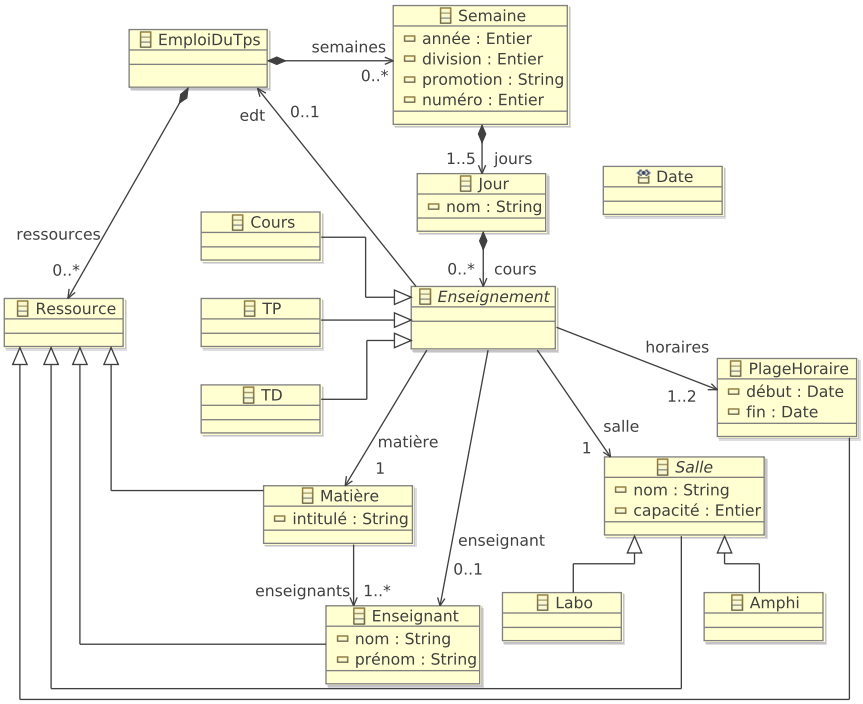
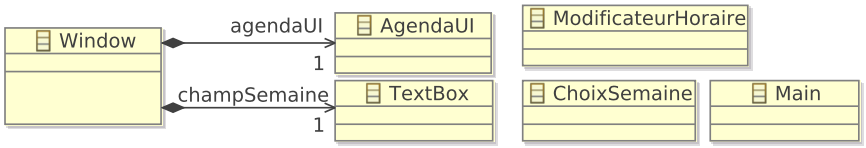


Fig. 10.2: Schéma des données sources de l'agenda

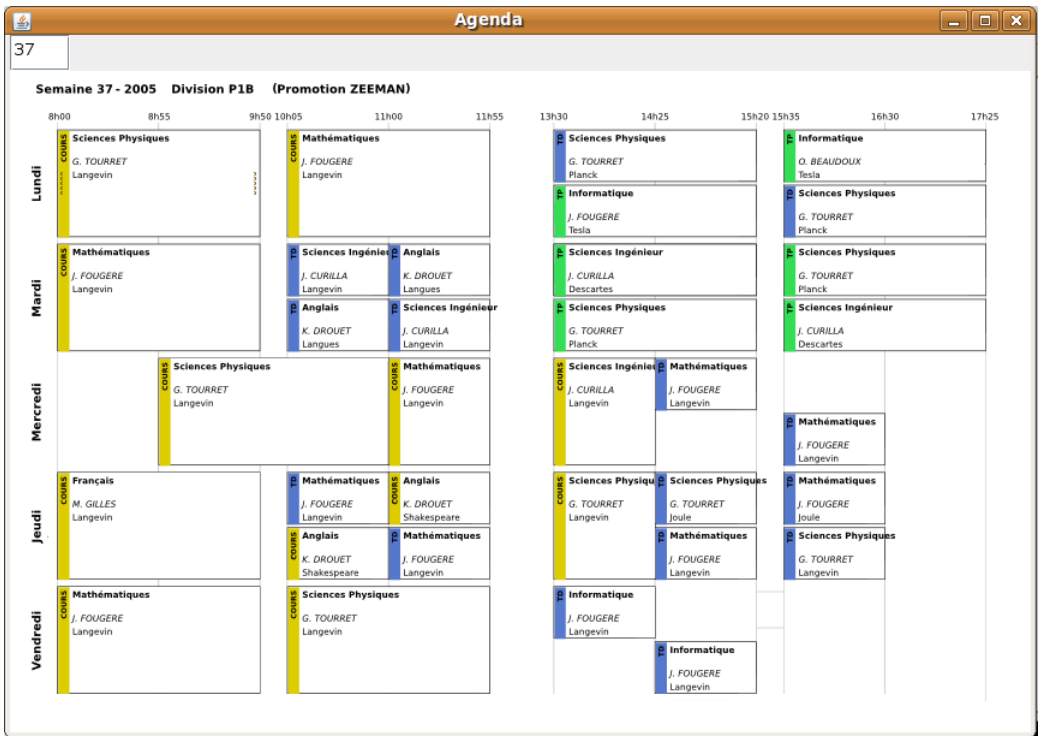
10.3 Interface

L'interface de l'agenda, dont la structure est représentée par le diagramme de classes de la figure 10.3a, consiste en une fenêtre (classe `Window`) contenant une instance `AgendaUI` et un champ de texte pour le changement de semaine (`TextBox`). Les classes `Window` et `TextBox` sont issues du modèle de composants graphiques de la figure B.1, page 216. L'agenda dispose des

instruments suivant : `ModificateurHoraire` permet de modifier les horaires des évènements ; `ChoixSemaine` permet de sélectionner la semaine à présenter ; `Main` est un instrument dédié à la manipulation directe des évènements.



(a) Diagramme de classes de l'interface



(b) Interface en Swing

Fig. 10.3: Interface de l'agenda

La figure 10.3b présente l'interface finale s'exécutant sur la plate-forme Swing. Elle montre la présence d'un champ de texte au nord et de la présentation concrète de l'agenda au centre. Cette dernière contient des évènements présentant leurs descriptions, leur type à un horaire donné. Les évènements de chaque jour sont disposés de manière horizontale. L'évènement sélectionné dispose de deux poignées, visibles aux extrémités droite et gauche sous la forme de deux ou quatre points alignés, permettant de modifier sa date de début et de fin.

10.4 Présentations

Dans cette section sont présentées les présentations abstraite et concrète de l'agenda. Elles se différencient des données sources par le fait qu'elles décrivent un agenda usuel et non un emploi du temps d'un étudiant.

10.4.1 Présentation abstraite

La présentation abstraite de l'agenda, décrite dans la figure 10.4, a pour racine la classe `AgendaCanvas`. Cette dernière possède un titre, des évènements, des séparateurs des différents horaires (classe `LigneHeure`), et différents types d'évènements. Un type d'évènements (p. ex. les anniversaires, les cours, les TD, les réunions, *etc.*) possède des types de descriptions. Par exemple, pour un anniversaire les types de descriptions peuvent être le nom, le prénom, l'année de naissance de la personne concernée. Un type de descriptions dispose d'un nom et d'une liste des différents choix possibles. Un évènement a lieu à une date d'un jour donné et possède une durée, des descriptions et un type.

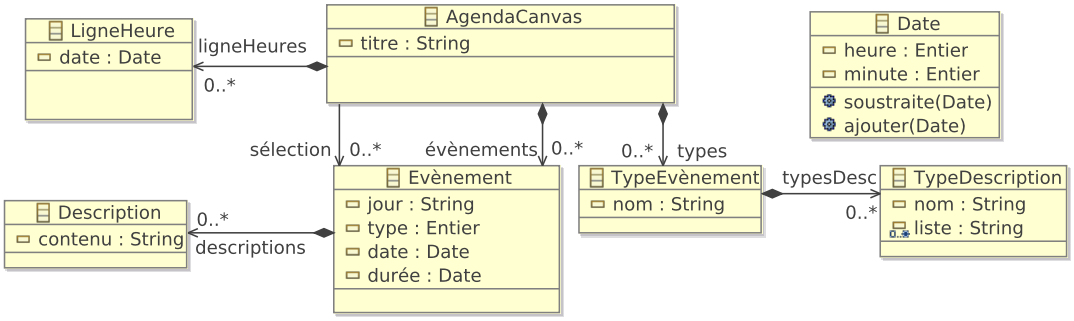


Fig. 10.4: Présentation abstraite de l'agenda

10.4.2 Présentation concrète

La présentation concrète de l'agenda, décrite par la figure 10.5, suit la même organisation que la présentation abstraite (*cf.* figure 10.4). La classe principale, `AgendaCanvasUI`, hérite de la classe `Component`, tirée du modèle de composants graphiques (*cf.* figure B.1, page 216). Elle possède un titre, des évènements (classe `EvènementUI`), des jours (`JourUI`), un instrument dédié à la modification des horaires des évènements (`ModificateurHoraire`) et des séparateurs des horaires (`LigneHeureUI`). La classe `EvènementUI` se caractérise par une couleur, un type, des coordonnées X et Y, une largeur, une hauteur, et un ensemble de descriptions (`DescriptionUI`). Une instance `EvènementUI` peut être associée à l'instrument `ModificateurHoraire` de l'agenda afin de pouvoir modifier ses horaires. Une instance `DescriptionUI` peut posséder une `ListBox`, issue du modèle de composants graphiques, correspondant aux différents choix de description possibles. La classe `JourUI` possède un nom et des coordonnées X et Y. La classe `LigneHeureUI`, utilisée pour séparer verticalement les horaires, dispose également de coordonnées X et Y pour chacun de ses deux points et d'une date (attributs `heure` et `minute`).

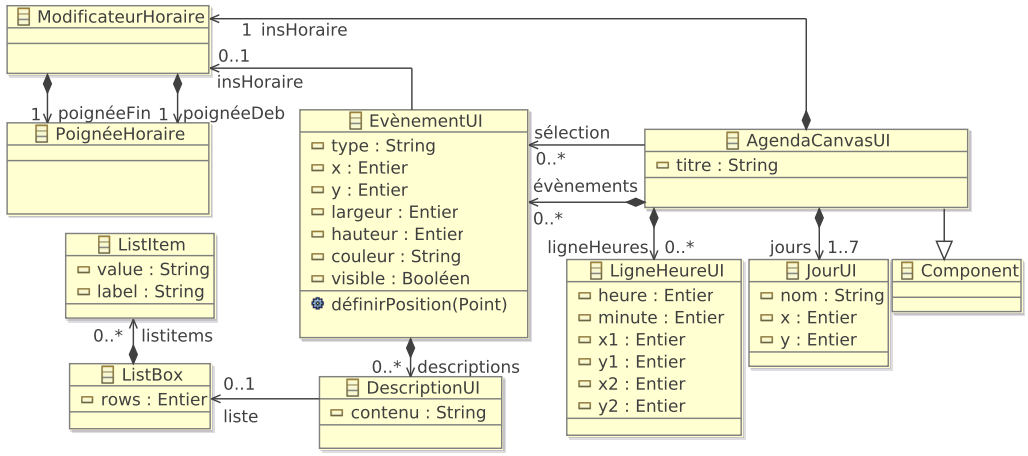


Fig. 10.5: Présentation concrète de l'agenda

10.5 Correspondances de schémas

Cette section présente la correspondance de schémas liant l'emploi du temps à la présentation abstraite de l'agenda, puis celle établie entre cette dernière avec la présentation concrète.

10.5.1 Données sources vers présentation abstraite

La correspondance de schémas Malan établie entre les données sources et la présentation abstraite, dont le code est donné ci-dessous, a pour particularité d'utiliser un paramètre `indexSemaine` (ligne 2). Ce dernier définit quelle semaine de l'emploi du temps doit être présentée. La correspondance de classes racine `EdT2AC` (ligne 4) met en relation la semaine concernée avec l'agenda cible (ligne 7). De plus, une sélection est effectuée, ligne 6, sur les différents horaires existants pour en déduire les instances `LigneHeure` à définir (lignes 8 et 9) : l'instruction de la ligne 5 sélectionne les créneaux horaires parmi les ressources existantes pour en déduire les heures de l'emploi de temps. Les lignes 10 à 21 sont consacrées à la création des trois types d'évènements : TD, TP et cours. L'alias `typesEdt` fait référence aux trois noms de ces types. Les alias `ms`, `es` et `ss` font respectivement référence aux matières, aux enseignants et aux salles de l'emploi du temps. Les trois types d'évènements sont ensuite créés de manière identique (lignes 15 à 21). Chacun de ces types possède les trois mêmes types de description relatifs aux matières, aux enseignants et aux salles. Par exemple, pour le premier de ces types, les intitulés de toutes les matières forment la liste des contenus possibles pour ce type de description.

La correspondance de classes `Semaine2Agenda` spécifie tout d'abord le titre de l'agenda (ligne 26) pour ensuite mettre en relation chaque enseignement et son jour correspondant, avec un évènement (ligne 28). Cette dernière étape utilise deux indices, `i` et `j`, de telle manière que l'instruction `(jours[i].enseignements)[j]` retourne le $j^{ième}$ enseignement de la $i^{ème}$ journée de la semaine.

La correspondance de classes `Ens2Evt` (ligne 31), qui lie un enseignement et un jour avec un évènement, commence par spécifier le jour, l'heure et la durée de l'évènement. La calcul de la durée (ligne 35) utilise l'opération `soustraire` de la classe `Date` calculant la durée à partir

des deux horaires de l'enseignement. La matière, l'enseignant et la salle complètent également la description de l'évènement (lignes 37 à 39).

La correspondance de classes entre une matière et une description, ligne 42, stipule que le contenu de cette dernière se rapporte à l'intitulé de la matière. De la même manière, le nom et le prénom de l'enseignant, ainsi que le nom de la salle sont associés à des descriptions (respectivement lignes 42, 43 et 44).

Etant donné que la classe source `Enseignement` est abstraite, il est nécessaire de définir les correspondances de classes utilisant les classes qui en héritent. `Cours2Evt` (ligne 47), `TD2Evt` (ligne 48) et `TP2Evt` (ligne 49) sont les correspondances de classes liant respectivement un cours, un TD et un TP à un évènement. Elles définissent toutes les trois le type de l'évènement.

```
1  EDT2Agenda : "edt.uml/edt" -> "edt.uml/agendaCanvas" {
2    param indexSemaine
3
4    EdT2AC : EmploiDuTps -> AgendaCanvas ac {
5      alias horaires resources[is Horaire]
6      alias heures   unique(horaires.début + horaires.fin)
7      semaines[indexSemaine] -> ac
8      |heures|               -> lignesHeures
9      heures[i]              -> lignesHeures[i].heure, i=[1..|heures|]
10     alias typesEdT typesEdT()
11     alias ms       (List<Matière>)resources[is Matière]
12     alias es       (List<Enseignant>)resources[is Enseignant]
13     alias ss       (List<Salle>)resources[is Salle]
14     3 -> |types|
15     types[i], i=1..|types| {
16       typesEdT[i] -> nom
17       true        -> restructif
18       typesDesc[1]{ "Matière"     -> nom  ms.intitulé          -> liste}
19       typesDesc[2]{ "Enseignant"  -> nom  es.(prénom+" "+nom) -> liste}
20       typesDesc[3]{ "Salle"       -> nom  ss.nom               -> liste}
21     }
22   }
23   Semaine2Agenda : Semaine -> AgendaCanvas {
24     alias ens |jours.enseignements|
25     "Semaine " + numéro + "-" + année + "\tDivision " + division + "\t(" +
26                                         promotion + ")" -> titre
27     ens -> |évènements|
28     (jours[i].enseignements)[j], jours[i] ->
29                                         évènements[j], i=[1..|jours|], j=[1..ens]
30   }
31   Ens2Evt : Enseignement e, Jour j -> Evènement {
32     alias ens |e.enseignant|==0 ? e.matière.enseignant[1] : e.enseignant
33     j.nom          -> jour
34     e.horaires[1] -> date
35     e.horaires[|e.horaires|].fin.soustraire(e.horaires[1].début) -> durée
36     4          -> |descriptions|
37     e.matière -> descriptions[1]
38     ens       -> descriptions[2]
39     e.salle   -> descriptions[3]
40   }
41   Date -> Date { heure  -> heure        minute -> minute }
42   Matière              -> Description { intitulé -> contenu          }
43   Enseignant           -> Description { prénom + " " + nom -> contenu }
```

```
44    Salle                -> Description { nom       -> contenu          }
45    Labo                 -> Description { }
46    Amphi                -> Description { }
47    Cours2Evt : Cours, Jour -> Evènement  { 3 -> type }
48    TD2Evt    : TD, Jour    -> Evènement  { 2 -> type }
49    TP2Evt    : TP, Jour    -> Evènement  { 1 -> type }
50    function List<String> typesEdT() { return { "TP", "TD", "Cours" }; }
51  }
```

10.5.2 Présentation abstraite vers présentation concrète

La complexité du lien entre les présentations abstraite et concrète, dont le code est donné ci-dessous, est due au calcul du « layout » des différents composants de l'agenda. La correspondance de schémas débute par la mise en relation des classes `AgendaCanvas` et `AgendaCanvasUI` (ligne 2). Cette correspondance de classes définit tout d'abord les instances de la classe cible `JourUI` qui composent la relation `jours`. Pour cela, la fonction `jours`, déclarée ligne 47, retourne sous la forme de chaînes de caractères la liste des jours de la semaine. Cette liste est utilisée pour spécifier la taille de la relation `jours` (ligne 4) ainsi que les instances qui la composent (lignes 5 à 9). Cette dernière étape a recours à un regroupement d'instructions utilisant un itérateur `i`. Celui-ci parcourt successivement les instances de la relation `jours` pour en définir le nom et les coordonnées. Les relations cibles `lignesHeures` et `évènements` sont ensuite définies par les correspondances de relations des lignes 11 et 12 à l'aide des relations sources homonymes et de l'agenda source.

La correspondance de classes `Ligne2Ligne` (ligne 16) se charge de calculer les coordonnées de la cible `LigneHeureUI`. Pour cela, la fonction `dateVersPixels` (ligne 48), qui retourne une valeur calculée par rapport à la date de la `LigneHeure` donnée en paramètre, est utilisée. Elle calcule la coordonnée `x` à partir de la date de la `LigneHeure` source (instruction `dateVersPixels(lh.date)`) pondérée par la coordonnée `x` de la première `LigneHeure` de l'agenda source (instruction `dateVersPixels(ac.ligneHeures[1].date)`).

De la même manière, la correspondance de classes `Evt2Evt`, ligne 25, définit la coordonnée `x` d'un évènement. Le calcul de la coordonnée `y` d'un évènement est l'étape la plus complexe de la correspondance de schémas puisqu'elle requière des algorithmes d'ordonnancement. Celui que nous avons développé utilise les fonctions `layout` (ligne 49) et `superpose` (ligne 61) pour regrouper sur différentes lignes les évènements d'une même journée qui ne se superposent pas. La fonction `layout` crée un ensemble vide de listes d'évènements (ligne 50). L'algorithme vérifie ensuite si l'évènement courant `e` se superpose à des évènements de la première liste d'évènements par le biais de la fonction `superpose` (ligne 53). Si c'est le cas, la deuxième liste est vérifiée ainsi de suite jusqu'à la fin de l'ensemble. Si l'évènement superpose toutes les listes existantes, une nouvelle liste contenant l'évènement courant est alors créée et ajoutée à l'ensemble (lignes 55 et 56). Dans le cas contraire, l'évènement est ajouté à la liste qu'il ne superpose pas (ligne 57). Prenons par exemple les quatre évènements suivants :

Nom	Début	Fin
A	10h	12h
B	8h	9h
C	9h	21h
D	21h	22h

L'évènement `A` est tout d'abord ajouté dans une première liste. L'évènement `B`, ne superposant pas `A`, est ajouté à la même liste que `A`. De même, l'évènement `C` ne superpose ni `A`, ni `B` et est donc ajouté à leur liste. Ce qui n'est pas la cas de l'évènement `D` qui est alors ajouté dans une seconde liste. Au final, la fonction `layout` retourne l'ensemble des listes d'évènements suivant : `{ {A, B, D}, {C} }`. Ainsi, l'instruction `position(e, layout(ac.évènements))[1]` de la ligne 27 calcule le « layout » des évènements puis y recherche la position de l'évènement `e` courant *via* la fonction prédéfinie `position`. Etant donné que la fonction `layout` retourne une double liste, `position` retourne un tableau de deux éléments correspondant à la position de `e` dans cette double liste. Par exemple pour `D`, `position` retourne `{2, 1}` où 2 correspond à la deuxième liste et 1 au premier élément de celle-ci. L'indice de la liste et le jour de l'évènement sont alors utilisés pour calculer la position y de ce dernier (ligne 27). Plus simplement, la largeur d'un évènement est calculée à partir de sa durée (ligne 28). Les différentes descriptions de l'évènement sont ensuite misent en relation (lignes 30 à 35). Cette étape crée pour chaque `DescriptionUI`, une liste contenant les choix possibles de contenu. Pour cela, le type d'évènement correspondant à la valeur de l'attribut `type` d'un évènement est associé à l'alias `type` (ligne 31). Ensuite, le contenu des descriptions est mis en relation (ligne 33). Enfin, la liste d'un `EvènementUI` est définie à partir du type de la description (ligne 34). Le nom du type et une couleur sont associés aux attributs cibles `type` et `couleur`.

La correspondance de classes `TypeDesc2List`, ligne 41, définit le contenu de la liste cible à partir des valeurs possibles du type de descriptions source.

```
1  "edt.uml/agendaCanvas" -> "edt.uml/agendaCanvasUI" {
2    AgendaCanvas ag -> AgendaCanvasUI acUI {
3      alias joursSemaine jours()
4      |joursSemaine| -> |jours|
5      jours[i], i=[1..|jours|] {
6        joursSemaine[i] -> nom
7        5               -> x
8        130 + (i-1)*155 -> y
9      }
10     titre                    -> titre
11     ligneHeures, ag -> lignesHeures
12     evènements,  ag -> evènements
13     sélection,   ag -> sélection
14     if(|sélection|>0) acUI.insHoraire -> acUI.sélection[1].insHoraire
15   }
16   Ligne2Ligne : LigneHeure lh, AgendaCanvas ac -> LigneHeureUI {
17     alias x dateVersPixels(lh.date) - dateVersPixels(ac.ligneHeures[1].date)
18     x   -> x1
19     x   -> x2
20     5   -> y1
21     805 -> y2
22     lh.date.heure  -> heure
23     lh.date.minute -> minute
24   }
25   Evt2Evt : Evènement e, AgendaCanvas ac -> EvènementUI {
26     dateVersPixels(e.date) - dateVersPixels(ac.ligneHeures[1].date) -> x
27     130 + (jourVersInt(e.jour))*155 + (position(e, layout(ac.évènements))[1]-1)*60-> y
```

```
28      dateVersPixels(e.durée)      -> largeur
29      60/|lignes|                  -> hauteur
30      |descriptions|              -> |descriptions|
31      alias type ac.types[e.type]
32      descriptions[i], i=1..|descriptions| {
33        descriptions[i].contenu -> contenu
34        type.typesDesc[i]       -> liste
35      }
36      type.nom                -> type
37      obtenirCouleur(e.type) -> couleur
38      false                  -> sélectionné
39      true                   -> visible
40    }
41    TypeDesc2List : TypeDescription -> ListBox {
42      1 -> rows
43      |liste| -> |listitems|
44      listitems[i], i=1..|listitems| { liste[i] -> value   liste[i] -> label }
45    }
46    function int jourVersInt(string j){ return jours().getIndex(j); }
47    function list<string> jours(){ return{"Lundi","Mardi","Mercredi","Jeudi","Vendredi"}; }
48    function int dateVersPixels(Date d) { return 5 * (60 * d.heure + d.minute) / 2; }
49    function list<list<Evènement>> layout(list<Evènement> evts) {
50      list<list<Evènement>> lignes = nil;
51      for(Evènement e : evts) {
52        int i=1;
53        while(i<=|lignes| && superpose(e, lignes[i])) i++;
54        if(i>|lignes|) {
55          lignes = lignes + nil;
56          lignes[|lignes|] = lignes[|lignes|] + e;
57        }else lignes[i] = lignes[i] + e;
58      }
59      return lignes;
60    }
61    function bool superpose(Evènement e, list<Evènement> evts) {
62      for(Evènement evt : evts)
63        if((evt.fin>e.début && evt.fin<=e.fin) || (evt.début<e.fin && evt.début>=e.début)
64           || (evt.début<=e.début && evt.fin>=e.fin))
65          return true;
66      return false;
67    }
68  }
```

L'algorithme utilisé pour calculer la disposition des évènements est optimisable. L'évènement `C` de l'exemple précédent se termine à 21h, heure à laquelle l'évènement `D` débute. Il serait logique de placer `C` dans la même liste que celle de `D` et non dans celle de `A` et de `B` afin de juxtaposer les évènements les plus proches optimisant ainsi l'espace de la présentation concrète.

Concernant l'implémentation du langage Malan, l'appel de la fonction `layout` à la ligne 27 pourrait laisser penser que cette fonction est appelée pour chaque instance de la classe source `Evènement` existante, ce qui serait néfaste en terme de performance. Une optimisation du processeur Malan viserait à mettre en cache le résultat de cette fonction lors de son premier appel dans cette correspondance de classes, pour être ensuite réutilisé.

10.6 Actions

Cette section décrit les différentes actions que peut faire l'utilisateur sur l'agenda. Le diagramme de classes de la figure 10.6 présente l'organisation et les paramètres de ces actions permettant de : changer la semaine visualisée (`ChangerSemaine`) ; gérer les évènements (`AjouterEvt`,

`SupprimerEvts` et `SélectionnerEvts`) ; modifier les horaires et les descriptions d'évènements (`HorairesEvts`, `ModifierDébutEvts`, `ModifierDuréeEvts`, `DéplacerEvts` et `ModifierDescriptionEvts`).

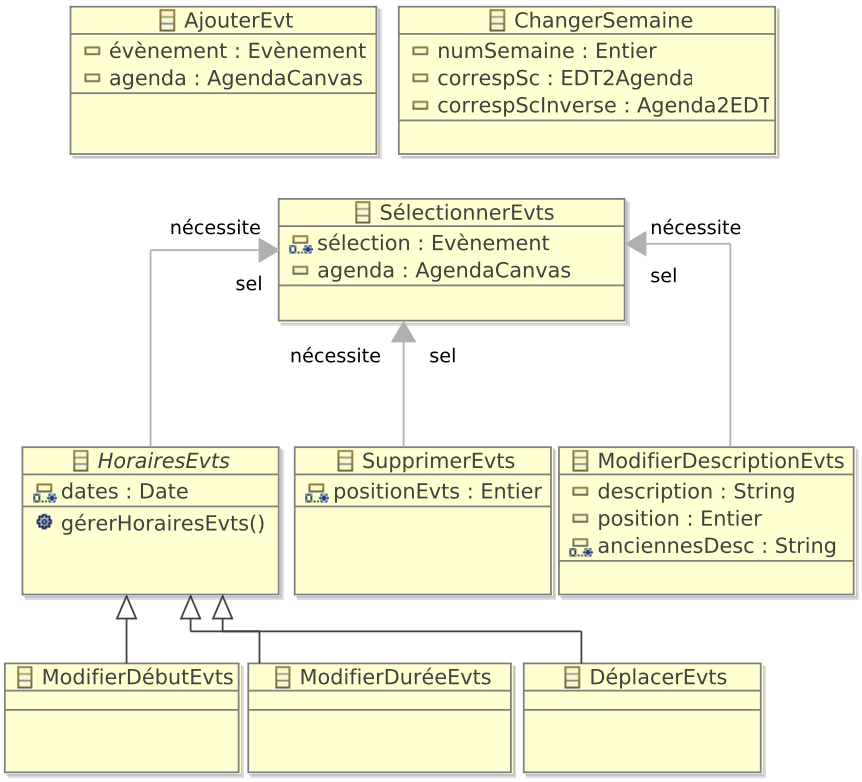
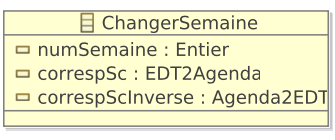


Fig. 10.6: Actions de l'agenda

10.6.1 Action spécifique aux correspondances de schémas

L'unique action spécifique aux correspondances de schémas a pour but de changer la semaine présentée par l'agenda. Le choix de la semaine à afficher s'effectue *via* le paramètre `indexSemaine` de la correspondance de schémas entre l'emploi du temps et la présentation abstraite (*cf.* section 10.5) et de son inverse. Le pseudo-code ci-dessous décrit l'action `ChangerSemaine` dont la méthode `exécuter` attribue une nouvelle valeur à ce paramètre. La fonction `peutExécuter` vérifie, quant à elle, que la nouvelle valeur du paramètre est bien un numéro de semaine valide.




```
1  booléen peutExécuter() {
2    retourner correspSc≠null et correspScInverse≠null
3              et numSemaine>0 et numSemaine<54
4  }
5  exécuter() {
6    #modifier(correspSc.indexSemaine, numSemaine)
7    #modifier(correspScInverse.indexSemaine,
8              numSemaine)
9  }
```

10.6.2 Gestion des évènements

Cette section décrit les actions dédiées à la gestion des évènements, c.-à-d. à leur sélection, leur ajout et leur suppression.

Sélectionner des évènements

L'action de sélection d'évènements, dont le pseudo-code est donné ci-dessous, possède deux paramètres : les évènements à sélectionner (attribut `sélection`) ; l'instance `AgendaCanvas` concernée (attribut `agendaUI`). La méthode `exécuter` (ligne 4) vide tout d'abord la sélection courante (ligne 5). Chaque évènement `evt` de `sélection` est ensuite ajouté à la relation sélection de la classe `AgendaCanvas` (ligne 7).


```
1  booléen peutExécuter() {
2    retourner sélection≠null et agenda≠null
3  }
4  exécuter() {
5    tant que(|agenda.sélection|>0)
6      #supprimer(agenda.sélection, agenda.sélection[1])
7    pour chaque Evènement evt de sélection faire
8      #ajouter(agenda.sélection, evt)
9  }
```

Ajouter un évènement

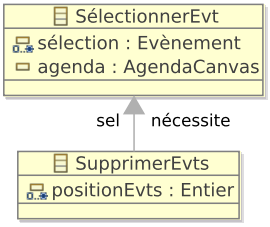
Le pseudo-code suivant décrit l'ajout d'un évènement dans un agenda. La fonction `peutExécuter` vérifie que l'évènement à ajouter et l'agenda concerné sont valides (ligne 1). La méthode `exécuter` ajoute l'évènement en utilisant la méthode `#ajouter` (ligne 4). L'ajout d'un évènement est une action annulable et doit donc définir les méthodes `réexécuter` et `annuler`. La première réalise le même travail que la méthode `exécuter` (ligne 10). La seconde se charge de supprimer l'évènement préalablement ajouté (ligne 7).


```
1  booléen peutExécuter() {
2    retourner évènement≠null et agenda≠null
3  }
4  exécuter() {
5    #ajouter(agenda.évènements, évènement)
6  }
7  annuler() {
8    #supprimer(agenda.évènements, évènement)
9  }
10 réexécuter() { exécuter() }
```

Supprimer des évènements

L'action `SupprimerEvts`, décrite ci-dessous, possède un attribut `positionEvts` utilisé dans la méthode `exécuter` pour sauvegarder la position dans la relation `évènements` des évènements supprimés (ligne 5). La suppression est réalisée par la fonction `#supprimer` qui prend en paramètres la relation `évènements` concernée par la suppression et l'évènement à supprimer. Cette fonction retourne la position de l'élément supprimé. La méthode `réexécuter` réapplique

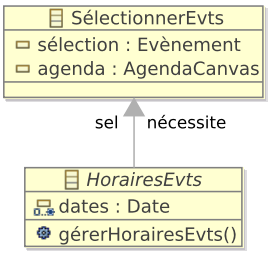
la suppression des évènements sans toutefois sauvegarder leur position à nouveau (ligne 13). La méthode `annuler` effectue le travail inverse (ligne 7) : chaque évènement supprimé est ajouté à son ancienne position dans la relation `évènements` par la biais de la méthode `#ajouter`.




```
1  booléen peutExécuter() { retourner sel.estExécutée() }
2  exécuter() {
3    pour chaque Evènement evt de sel.sélection
4    faire  positionEvts.ajouter(
5             #supprimer(sel.agenda.évènements, evt))
6  }
7  annuler() {
8    pour i de |sel.sélection| à 1 par pas de -1 faire
9      Evènement evt      = sélection[i]
10     entier    position = positionEvts[i]
11     #ajouter(sel.agenda.évènements, evt, position)
12 }
13 réexécuter() {
14   pour chaque Evènement evt de sel.sélection
15   faire  #supprimer(sel.agenda.évènements, evt)
16 }
```

10.6.3 Gestion des horaires d'évènements

Les actions dédiées à la gestion des horaires d'évènements héritent toutes de l'action abstraite `HoraireEvts` décrite ci-dessous. Celle-ci possède un attribut `dates` concernant le nouvel horaire de chaque évènement sélectionné. La méthode `exécuter` appelle ensuite la méthode abstraite `gérerHorairesEvts`, définie dans chaque action qui étend `HoraireEvts`. Le but de cette méthode est de modifier les horaires en fonction des dates contenues dans l'attribut `dates` et de remplacer ces dernières par celles modifiées. Ainsi, la méthode `annuler` (ligne 10) se contente d'appeler cette méthode pour remettre en place les anciennes dates et sauvegarder à nouveau les nouvelles qui sont remises en place dans la méthode `réexécuter` (ligne 11). La fonction `peutExécuter` (ligne 1) vérifie que le nombre de nouvelles positions correspond bien au nombre d'évènements sélectionnés.

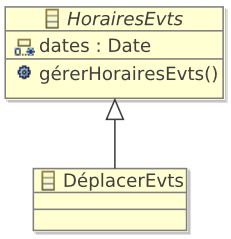



```
1  booléen peutExécuter() {
2    retourner sel.estExécutée() et
3           dates≠null et |dates|>0
4  }
5  exécuter() {
6    pour chaque Point p de positions faire
7      dates.ajouter(déduireDate(p))
8    gérerHorairesEvts()
9  }
10 annuler()    { gérerHorairesEvts() }
11 réexécuter() { gérerHorairesEvts() }
12 abstraite gérerHorairesEvts()
```

Les sections suivantes présentent les actions fondées sur l'action `HoraireEvts` qui ont pour unique rôle de définir la méthode `gérerHorairesEvts`. Etant semblable à l'action visant à déplacer des évènements, l'action modifiant la durée d'évènements est présentée en annexe C.3, page 222.

Déplacer des évènements

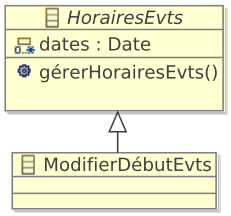
Le déplacement d'évènements modifie la date de début des évènements sélectionnés. Ainsi, la méthode `gérerHorairesEvts`, dont le pseudo-code est fourni ci-dessous, modifie la date de chaque évènement sélectionné, *via* la méthode `#modifier`, en utilisant la date contenue dans l'ensemble `dates` (ligne 6). Cette nouvelle date est ensuite remplacée par l'ancienne date de l'évènement (ligne 7).




```
1  gérerHorairesEvts() {
2    pour i de 1 à |sel.sélection| inclus faire
3      Evènement evt = sel.sélection[i]
4      Date début    = evt.obtenirDate()
5      si(dates[i]≠null) alors
6        #modifier(evt.date, dates[i])
7        dates[i] = début
8  }
```

Modifier la date de début d'évènements

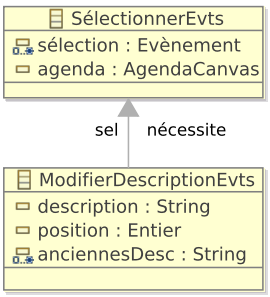
Pour modifier la date de début d'évènements sans en changer la date de fin, il est nécessaire de calculer la nouvelle durée comme le réalise la ligne 8 du code ci-dessous. La nouvelle date de début et la nouvelle durée sont ensuite attribuées à l'évènement courant (lignes 9 et 10). Pour terminer, l'ancienne date de chaque évènement remplace la nouvelle dans l'ensemble `dates` (ligne 11).




```
1  gérerHorairesEvts() {
2   pour i de 1 à |sélection| inclus faire
3     Evènement evt       = sélection[i]
4     Date durée          = evt.durée
5     Date début          = evt.date
6     Date nouveauDébut   = dates[i]
7     Date nouvelleDurée  = début.soustraire(nouveauDébut)
8                           .additionner(durée)
9     #modifier(evt.date,   nouveauDébut)
10    #modifier(evt.durée,  nouvelleDurée)
11    dates[i] = début
12 }
```

10.6.4 Gestion des descriptions d'évènements

Pour qu'une description d'évènements soit modifiée, il est nécessaire d'en connaître la position dans la relation `evt.descriptions` (attribut `position`), ainsi que sa nouvelle valeur (attribut `description`). La méthode `exécuter` se charge de sauvegarder chaque description à modifier puis les replace par la nouvelle (ligne 5). La méthode `annuler` remet en place les anciennes descriptions (ligne 11), tandis que la méthode `réexécuter` réapplique les nouvelles descriptions (ligne 19).




```
1  booléen peutExécuter() {
2   retourner sel.estExécutée() et description≠null et
3             position>0
4  }
5  exécuter() {
6   pour chaque Evènement evt de sel.sélection faire
7    si(|evt.descriptions|>=position) alors
8      anciennesDesc.ajouter(evt.descriptions[position])
9     #modifier(evt.descriptions, description, position)
10 }
11 annuler() {
12  entier i = 1
13  pour chaque Evènement evt de sel.sélection faire
14   si(|evt.descriptions|>=position) alors
15    #modifier(evt.descriptions, anciennesDesc[i],
16              position)
17    i++
18 }
19 réexécuter() {
20  pour chaque Evènement evt de sel.sélection faire
21   si(|evt.descriptions|>=position) alors
22    #modifier(evt.descriptions, description, position)
23 }
```

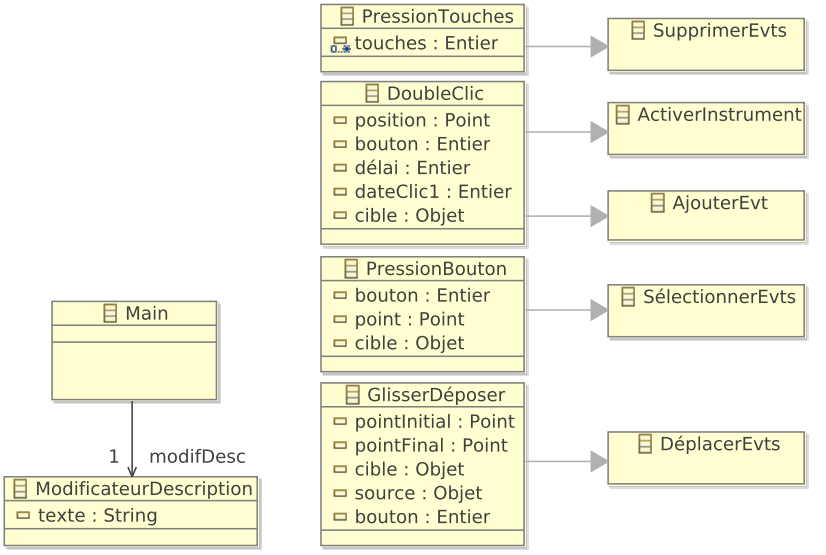
10.7 Instruments

Nous décrivons dans cette section les parties statiques et dynamiques des instruments dont dispose l'utilisateur pour manipuler l'agenda.

10.7.1 Main

Cet instrument a pour but de manipuler de manière directe les évènements de l'agenda. La classe `Main` possède un instrument de type `ModificateurDescription` que la main active lors d'un double-clic sur une description (*cf.* figure 10.7a).

Les liaisons établies entre les interactions et les actions de cet instrument sont représentées dans la figure 10.7b. Le pseudo-code défini ci-dessous décrit les conditions de ces liaisons. L'action de suppression d'évènements (`SupprimerEvts`) est engendrée par la pression de touches (`PressionTouches`) à la condition que l'unique touche pressée soit la touche « Supp. ». Dans le but de modifier la description d'un évènement, le double-clic est utilisé pour activer l'instrument `ModificateurDescription` *via* l'action `ActiverInstrument`. La condition à respecter stipule que l'objet cible du double clic doit être une instance `DescriptionUI`. Le double-clic permet également de créer des évènements si la cible est un `AgendaUI`. Leur type et leurs descriptions sont des valeurs par défaut tirées du premier type d'évènement de la relation `AgendaCanvas.type`. La pression d'un bouton sélectionne un évènement de l'agenda à condition que la cible de la pression soit l'évènement concerné. Pour finir, le déplacement d'évènements s'effectue par le biais d'un glisser-déposer si la clause suivante est respectée : la source de l'interaction doit être une instance `EvènementUI`, tandis que la cible doit être un composant de la présentation concrète de l'agenda (`AgendaUI`, `EvènementUI` ou `PoignéeHoraire`).



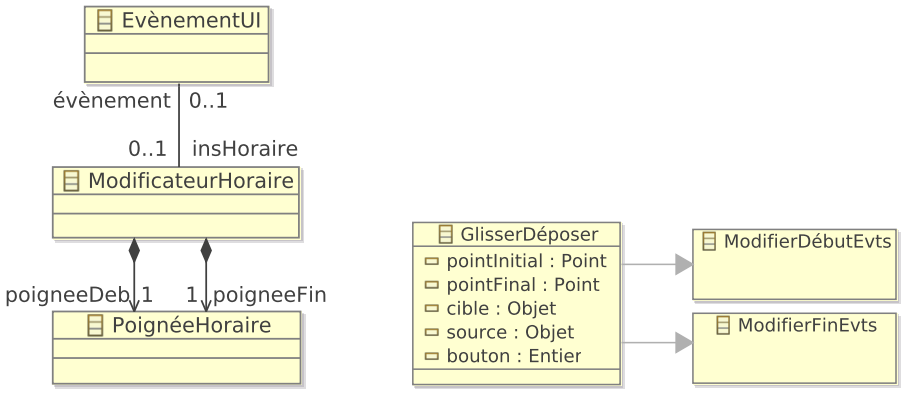
(a) Diagramme de classes (b) Liaisons entre les interactions et les actions

Fig. 10.7: Partie statique de la main

Le pseudo-code suivant définit également le feed-back intérimaire de la liaison entre l'interaction `GlisserDéposer` et l'action `DéplacerEvts`. S'il existe une action en cours de type `DéplacerEvts`, alors le curseur courant est remplacé par un autre symbolisant le mouvement d'un objet (ligne 17). Dans le cas contraire, le curseur par défaut est attribué au SI (ligne 21).

```
1  Liaisons Main main {
2    PressionTouche p -> SupprimerEvts action {
3      condition : |p.touches|>0 && p.touches[1]=='Supp.'
4    }
5    DoubleClic clic -> ActiverInstrument action {
6      condition : clic.cible is DescriptionUI
7    }
8    DoubleClic clic -> AjouterEvt action {
9      condition : clic.cible is AgendaUI
10   }
11   PressionBouton pb -> SélectionnerEvts action {
12     condition : pb.cible is EvènementUI
13   }
14   GlisserDéposer gd -> DéplacerEvts action {
15     condition : gd.source is EvènementUI && (cible is AgendaUI ||
16                 gd.cible is EvènementUI || gd.cible is PoignéeHoraire)
17     feedback  :
18       remplacer curseur courant par curseur déplacement
19   }
20   défaut {
21     remplacer curseur courant par curseur par défaut
22   }
23 }
```

10.7.2 Le modificateur d'horaires



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 10.8: Partie statique du modificateur d'horaires

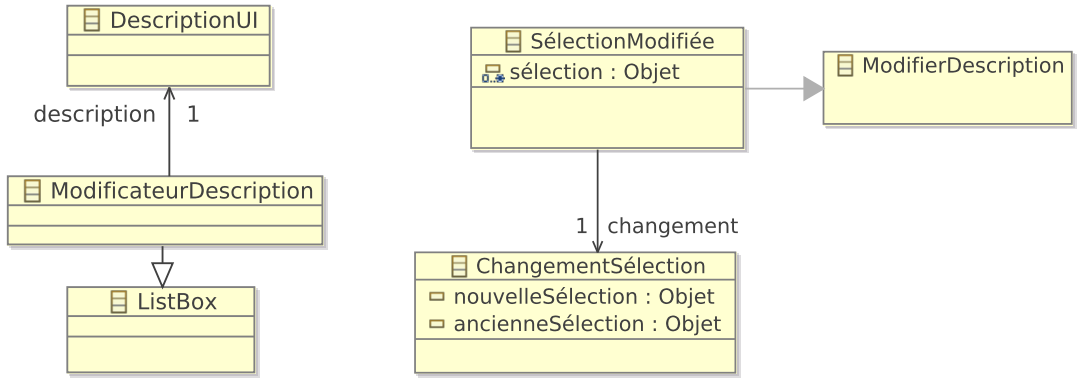
Le modificateur d'horaires se compose de deux poignées modifiant le début et la fin d'évènements (*cf.* figure 10.8a). Elles ont la particularité d'être des composants graphiques de la présentation concrète. Leur position et leurs dimensions sont calculées en fonction de la position de l'évènement. Lorsqu'un évènement est sélectionné, le modificateur d'horaires lui est associé. La figure 10.8b présente les liaisons, décrites dans le pseudo-code ci-dessous, entre l'interaction `GlisserDéposer` et les actions `ModifierDébutEvts` et `ModifierFinEvts`. Lorsque la source de cette interaction est la poignée `poigneeDeb`, l'action `ModifierDébutEvts` est créée. De manière identique, si la source est la poignée `poigneeFin`, l'action `ModifierFinEvts` est créée.

Le feed-back intérimaire de cet instrument consiste à changer le curseur du SI lorsque une action de type `ModifierHoraire` est en cours (lignes 4 et 7).

```
1  Liaisons PoignéeHoraire ph {
2    GlisserDéposer gd -> ModifierDébutEvts action {
3      condition : gd.source==ph.poigneeDeb
4      feedback  : remplacer curseur courant par curseur redimensionnement
5    }
6    GlisserDéposer gd -> ModifierFinEvts action {
7      condition : gd.source==ph.poigneeFin
8      feedback  : remplacer curseur courant par curseur redimensionnement
9    }
10   défaut {
11     remplacer curseur courant par curseur par défaut
12   }
13 }
```

10.7.3 Modificateur de descriptions

L'instrument `ModificateurDescription` utilisé pour modifier les descriptions des évènements est une liste (classe `ListBox`) associée à la description concernée (*cf.* figure 10.9a). De même que pour le champ texte de l'éditeur de dessins vectoriels (*cf.* section 9.8, page 156), une `ListBox` est un composant graphique disponible sur la plupart des plates-formes d'IHM. La



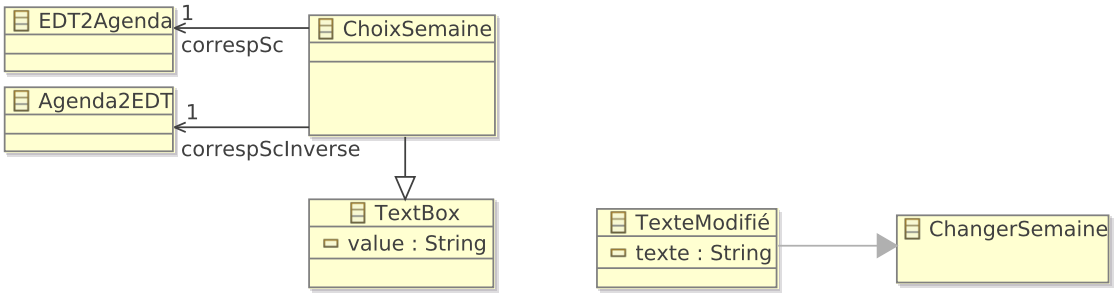
(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 10.9: Partie statique du modificateur de descriptions

manière dont l'utilisateur interagit avec une `ListBox` n'est donc pas redéfinie mais est laissée à la charge de la plate-forme d'IHM choisie. La liaison de la figure 9.19b, page 164, stipule uniquement que l'interaction `SélectionModifiée`, fondée sur l'évènement `ChangementSélection`, crée une action `ModifierDescription`. Un évènement `ChangementSélection` est provoqué lorsque la sélection de la liste change. Etant donné qu'une interaction `SélectionModifiée` crée dans tous les cas une action `ModifierDescription`, cette liaison n'est pas décrite en pseudo-code.

10.7.4 Sélectionneur de la semaine à afficher

L'instrument `ChoixSemaine` est un champ de texte (classe `TextBox`) dédié à la sélection de la semaine à présenter dans l'agenda (*cf.* figure 10.10a). Cet instrument connaît les correspondances de schémas liant l'emploi du temps à l'agenda (`EDT2Agenda`) et l'agenda à l'emploi du temps (`Agenda2EDT`).



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. 10.10: Partie statique du sélectionneur de la semaine à afficher

La liaison établie entre l'interaction `ModifierTexte` et l'action `ChangerSemaine` présentée dans la figure 9.19b, page 164 est décrite par le pseudo-code ci-dessous. Il définit que l'interaction `ModifierTexte` crée une action `ChangerSemaine` si la nouvelle valeur du champ de texte est un nombre compris entre 1 et 53 inclus (lignes 3 et 4).

```
1 Liaisons ChoixSemaine sem {
2   TexteModifié tm -> ChangerSemaine action {
3     condition : isInteger(tm.texte) && toInteger(tm.texte)>=1 &&
4                 toInteger(tm.texte)<=53
5   }
6 }
```

10.8 Autre présentation concrète

L'agenda présenté dans ce chapitre est optimisé pour une exécution sur un ordinateur standard. Dans cette section, nous modifions différentes parties qui composent cet agenda pour obtenir un autre agenda adapté à un autre support, à savoir un téléphone portable possédant un écran tactile de taille réduite. La figure 10.11 présente l'interface finale de ce nouvel agenda dans laquelle les évènements d'une journée sont affichés de manière verticale afin de profiter de la hauteur de l'écran. Cette interface possède un instrument en plus destiné à sélectionner le jour à afficher.

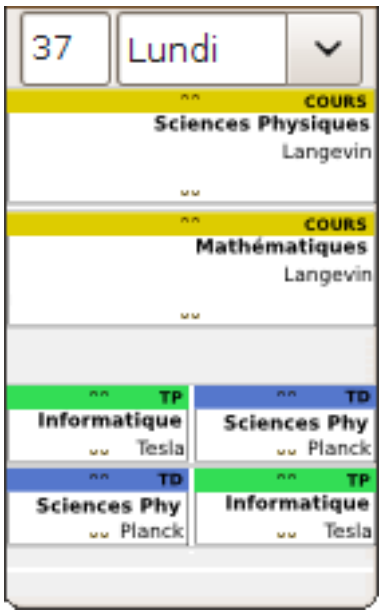


Fig. 10.11: Interface du second agenda

Cette adaptation ne modifie pas les données et les présentations abstraite et concrète. Les correspondances de schémas reliant ces trois éléments doivent être cependant adaptées. Concernant la correspondance de schémas entre les données et la présentation abstraite, un nouveau paramètre `indexJour` doit être déclaré pour spécifier quel jour doit être présenté. De plus, la correspondance de classes `Semaine2Agenda` doit être remplacée par la suivante, dans laquelle seuls les enseignements d'un jour précis, indiqué par le paramètre `indexJour`, sont utilisés (lignes 3 et 4).

```
1 Semaine2Agenda : Semaine -> AgendaCanvas {
```

```
2       ""     -> titre
3       |jours[indexJour].enseignements| -> |évènements|
4       jours[indexJour].enseignements, jours[indexJour] -> évènements
5     }
```

La correspondance de classes établie entre les présentations abstraite et concrète doit également être partiellement modifiée. La correspondance de classes entre `AgendaCanvas` et `AgendaCanvasUI` doit notamment être remaniée pour ne pas créer d'étiquette correspondant aux jours (ligne 2). Ensuite, les lignes (`LigneHeureUI`) ne doivent pas être affichées horizontalement mais verticalement (lignes 6 à 10). Pour finir, les coordonnées et les dimensions d'une instance `EvènementUI` doivent être adaptées à un écran de taille réduite (lignes 14 à 17).

```
1   AgendaCanvas -> AgendaCanvasUI {
2       0 -> |jours|
3       ...
4     }
5     Ligne2Ligne : LigneHeure lh, AgendaCanvas ac -> LigneHeureUI {
6       alias y (dateVersPixels(lh.date) - dateVersPixels(ac.ligneHeures[1].date))/10
7       y   -> y1
8       y   -> y2
9       0   -> x1
10      180 -> x2
11      ...
12    }
13    Evt2Evt : Evènement e, AgendaCanvas ac -> EvènementUI {
14      (dateVersPixels(e.date)-dateVersPixels(ac.ligneHeures[1].date))/10 -> y
15      0 -> x
16      dateVersPixels(e.durée)/10  -> hauteur
17      180                         -> largeur
18      ...
19    }
```

Les actions ne nécessitent pas de révision. Le choix des interactions dépend des HID gérés par la plate-forme d'exécution cible. En l'occurrence, les interactions utilisées par l'agenda peuvent être gérées par un téléphone disposant d'un écran tactile et d'un clavier. Par conséquent, l'unique modification à apporter au modèle d'instrument est l'ajout d'un instrument dédié à la sélection du jour.

10.9 Conclusion

Nous avons présenté dans ce chapitre une étude de cas dédiée à la conception d'un agenda standard. L'étude se focalise essentiellement sur la correspondance de schémas Malan établie entre les données et la présentation abstraite et sur celle liant les présentations abstraite et concrète.

Le calcul de la disposition des évènements de l'agenda était notamment problématique étant donné la complexité des algorithmes d'ordonnancement. Celui que nous avons développé fonctionne avec le langage Malan ; néanmoins un autre algorithme plus sophistiqué nécessiterait l'utilisation des éléments cibles comme source d'une instruction de correspondance, ce qui n'est pas possible avec la spécification actuelle de Malan. Cette fonctionnalité, compatible avec le modèle théorique de Malan, en constitue une possible amélioration.

Le développement d'un SI devant s'exécuter sur différents types de support est également abordé dans cette étude au travers de l'adaptation d'un agenda s'exécutant sur un ordinateur standard, en un agenda pour téléphone portable. La division en différentes parties (données, interface, présentation, action, instrument et interaction) et la conception à différents niveaux d'abstraction (interface abstraite et interface concrète) facilite cette adaptation en factorisant les étapes de conception. Dans l'exemple présenté, hormis la nouvelle interface qui a dû être entièrement adaptée, seules quelques modifications ont dû être apportées aux correspondances de schémas et aux instruments.

Conclusion

Les travaux présentés dans cette thèse sont motivés par les différentes évolutions du web, des données et des appareils sur lesquels s'exécutent les SI, ainsi que les nouvelles techniques d'interaction. Ces évolutions ont amené à revoir le traitement des trois aspects suivants relatifs au développement de SI : la liaison entre les données sources et leurs présentations *;* la conception de la facette interactive des SI *;* l'exécution d'un SI sur différentes plates-formes d'exécution.

Cette conclusion rappelle les contributions de nos travaux, et discute des perspectives de recherche auxquelles ils devraient mener.

Contributions

Nos travaux se divisent en deux parties complémentaires correspondant, d'une part, aux liens entre les données sources d'un SI et leurs présentations et, d'autre part, à la facette interactive des SI.

Dans la première partie, nous avons développé un langage de correspondance appelé Malan (*a MApping LANguage*). Le but de ce langage est d'établir des ponts, appelés correspondances de schémas, entre les schémas des données sources et de leurs présentations cibles. Les schémas utilisés sont représentés sous la forme de diagrammes de classe UML. Ce choix s'explique par la capacité dont dispose UML à s'abstraire de la plate-forme de données. Par rapport aux modèles de « data binding » des plates-formes RIA, Malan a l'avantage d'être indépendant de toute plate-forme de données et d'IHM. De plus, ce langage a été conçu dans le cadre d'une utilisation active : tout changement des données sources doit provoquer une mise à jour des présentations cibles. Les langages de transformations classiques ne permettent pas ce type d'utilisation. Malan peut être utilisé de deux manières différentes. Premièrement, un processeur dédié peut directement utiliser une correspondance de schémas lors de l'exécution d'un SI afin de créer des liaisons actives entres les instances sources et cibles qu'elles gèrent. Deuxièmement, une transformation active peut être générée à partir d'une correspondance de schémas.

La seconde contribution consiste en un modèle conceptuel d'interaction, appelé Malai, rassemblant les principaux avantages de la manipulation directe, de l'interaction instrumentale, du modèle d'action de Norman, du modèle DPI et du concept d'interacteur. Malai vise à améliorer : la définition et la réutilisation des interactions dans différents SI ; la définition du feed-back intérimaire des instruments ; la modélisation des actions en les considérant comme des objets disposant de leur propre cycle de vie et pouvant être facilement annulable. Malai divise un SI en plusieurs éléments fondamentaux : les données sources ; l'interface ; les présentations abstraites et concrètes ; les instruments, pivots entre les interactions et les actions.

Ces deux contributions ont été conçues pour être employées dans le cadre d'une approche IDM travaillant dans différents niveaux d'abstractions. Le premier niveau considéré dans cette thèse est l'interface abstraite. Celle-ci se réfère aux présentations abstraites et aux actions. L'interface concrète enrichît l'interface abstraite en y apportant les présentations concrètes, les interactions, les instruments et les correspondances de schémas entre les présentations abstraites et concrètes.

Perspectives

Certaines étapes du processus de développement de SI en différents niveaux d'abstraction n'ont pas été abordées dans cette thèse. Tout d'abord, le passage de l'interface concrète à l'interface finale, c.-à-d. au code source d'un SI pour une plate-forme d'IHM donnée, nécessite le développement de transformations de modèles. A partir du pseudo-code utilisé dans les études de cas des chapitres 9 et 10, nous envisageons de définir un DSL dédié à la définition du feedback intérimaire, de la mise à jour des actions dans les instruments, ou encore du code contenu dans les états des interactions visant à mettre à jour leurs paramètres. De plus, l'utilisation du modèle de tâche n'a pas été abordée dans nos travaux. Il pourrait pourtant servir à générer une partie du modèle d'action qu'un développeur devra compléter en y définissant le fonctionnement des actions *via* le DSL. L'évaluation empirique de nos travaux consiste également en une étape importante à effectuer sur le court terme. Elle permettra de confirmer ou d'infirmer les hypothèses que nous avons émises sur les avantages de notre langage de correspondance Malan, et sur les caractéristiques de notre modèle Malai.

A plus long terme, une des pistes consiste à mettre en place un métamodèle de plate-forme d'exécution. Il permettrait de décrire les caractéristiques des plates-formes d'exécution (p. ex. dimensions de l'écran) ainsi que les HID qu'elles supportent. Par exemple, lors d'un changement de plate-forme d'exécution, le modèle de cette dernière permettrait de vérifier si les interactions utilisées sont possibles pour cette nouvelle plate-forme. Nous n'abordons pas non plus, dans cette thèse, les notions d'utilisateur et d'environnement relatives au contexte d'usage d'un SI. Ces deux notions se réfèrent respectivement aux différents utilisateurs qu'un SI doit gérer, et à la description physique de l'environnement dans lequel les interactions sont réalisées. La prise en compte de ces deux notions permettrait d'utiliser Malai pour des problèmes d'adaptation au contexte. De même, le travail en collaboration n'est pas abordé.

Liste des figures

Liste des publications personnelles

Revues nationales avec comité de sélection

[1] Arnaud Blouin, Olivier Beaudoux, et Stéphane Loiseau. Un tour d'horizon des approches pour la manipulation des données du web. *Document Numérique*, 11(1-2/2008) :63–83, 2008.

Conférences internationales avec comité de sélection

[2] Arnaud Blouin et Olivier Beaudoux. Mapping paradigm for document transformation. Dans *DocEng '07 : Proceedings of the 2007 ACM symposium on Document engineering*, pages 219–221. ACM Press, 2007.

[3] Arnaud Blouin, Olivier Beaudoux, et Stéphane Loiseau. Malan : A mapping language for the data manipulation. Dans *DocEng '08 : Proceedings of the 2008 ACM symposium on Document engineering*, pages 66–75. ACM Press, 2008.

Conférences nationales avec comité de sélection

[4] Olivier Beaudoux, Arnaud Blouin, et Slimmane Hammoudi. Transformations actives dédiées aux IHM. Dans *Cinquièmes Journées sur l'Ingénierie Dirigée par les Modèles*. 2009.

[5] Arnaud Blouin et Olivier Beaudoux. Malai : un modèle conceptuel d'interaction pour les systèmes interactifs. Dans *IHM'09 : Proceedings of the 21th international conference on Association Francophone d'Interaction Homme-Machine*. 2009.

Références bibliographiques

[Abiteboul *et al.*, 2000] Serge Abiteboul, Peter Buneman, et Dan Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[Abiteboul *et al.*, 2002] Serge Abiteboul, Sophie Cluet, et Tova Milo. Correspondence and translation for heterogeneous data. *Theoretical Computer Science*, 275(1-2) :179–213, 2002.

[Abiteboul, 1997] Serge Abiteboul. Querying semi-structured data. Dans *ICDT '97 : Proceedings of the 6th International Conference on Database Theory*, pages 1–18. Springer-Verlag, 1997.

[Abiteboul, 1999] Serge Abiteboul. On views and XML. Dans *PODS '99 : Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–9, 1999.

[Accot *et al.*, 1996] Johnny Accot, Stéphane Chatty, et Philippe Palanque. A formal description of low level interaction and its application to multimodal interactive systems. Dans *Proceedings of the 3th Eurographics workshop on the design, specification and verification of interactive software*, 1996.

[Accot *et al.*, 1997] Johnny Accot, Stéphane Chatty, Sébastien Maury, et Philippe Palanque. Formal transducers : models of devices and building bricks for the design of highly interactive systems. Dans *Proceedings of the 4th Eurographics workshop on the design, specification and verification of interactive software*, 1997.

[Akehurst, 2000] David H Akehurst. *Model Translation : A UML-based specification technique and active implementation approach*. PhD thesis, The University of Kent, 2000.

[Almendros-Jiménez et Iribarne, 2008] Jesús M. Almendros-Jiménez et Luis Iribarne. An extension of UML for the modeling of WIMP user interfaces. *J. Vis. Lang. Comput.*, 19(6) :695–720, 2008.

[Appert et Beaudouin-Lafon, 2008] C. Appert et M. Beaudouin-Lafon. SwingStates : adding state machines to Java and the Swing toolkit. *Software, Practice and Experience*, 38(11) :1149–1182, 2008.

[Appert, 2007] Caroline Appert. *Modélisation, Évaluation et Génération de Techniques d'Interaction*. PhD thesis, Laboratoire de Recherche en Informatique, Université Paris-Sud, 2007.

[Atzeni, 2006] Paolo Atzeni. Schema and data translation. Dans *ICDE '06 : Proceedings of the 22nd International Conference on Data Engineering*, page 103. IEEE Computer Society, 2006.

[Balzert *et al.*, 1996] H. Balzert, F. Hofmann, V. Kruschinski, et C. Niemann. The JANUS application development environment – generating more than the user interface. Dans *Computer-Aided Design of User Interfaces*, 1996.

[Batini *et al.*, 1986] C. Batini, M. Lenzerini, et S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4) :323–364, 1986.

[Beaudouin-Lafon et Lassen, 2000] M. Beaudouin-Lafon et H. M. Lassen. The architecture and implementation of CPN2000, a post-WIMP graphical application. Dans *Proc. of UIST'00*, volume 2, pages 181–190, 2000.

[Beaudouin-Lafon et Mackay, 2000] M. Beaudouin-Lafon et W.E. Mackay. Reification, polymorphism and reuse : Three principles for designing visual interfaces. Dans *In Proceedings of Advanced Visual Interfaces (AVI'00)*, pages 102–109, 2000.

[Beaudouin-Lafon, 1997] M. Beaudouin-Lafon. Interaction instrumentale : de la manipulation directe à la réalité augmentée. Dans *Actes IHM (IHM'97)*. Cépaduès Éditions, 1997.

[Beaudouin-Lafon, 2000] M. Beaudouin-Lafon. Instrumental interaction : An interaction model for designing post-WIMP interfaces. Dans *CHI '00 : Proceedings of the 18th international conference on Human factors in computing systems*, volume 2, pages 446–453, 2000.

[Beaudouin-Lafon, 2004] Michel Beaudouin-Lafon. Designing interaction, not interfaces. Dans *AVI '04 : Proceedings of the working conference on Advanced visual interfaces*, 2004.

[Beaudoux *et al.*, 2009] Olivier Beaudoux, Arnaud Blouin, et Slimmane Hammoudi. Transformations actives dédiées aux IHM. Dans *Cinquièmes Journées sur l'Ingénierie Dirigée par les Modèles*, 2009.

[Beaudoux et Beaudouin-Lafon, 2001] Olivier Beaudoux et Michel Beaudouin-Lafon. DPI : A conceptual model based on documents and interaction instruments. Dans *People and Computers XV - Interaction without frontiers*, pages 247–263. Springer Verlag, 2001.

[Beaudoux, 2004b] Olivier Beaudoux. *Espaces de travail interactifs et collaboratifs : vers un modèle centré sur les documents et les instruments d'interaction.* PhD thesis, Université Patris XI, UFR scientifique d'Orsay, June 2004.

[Beaudoux, 2004c] Olivier Beaudoux. Un modèle de composants (inter)actifs centré sur les documents. *INFORMATION-INTERACTION-INTELLIGENCE*, 4 :41–58, 2004.

[Beaudoux, 2005c] Olivier Beaudoux. XML active transformation (eXAcT) : transforming documents within interactive systems. Dans *DocEng '05 : Proceedings of the 2005 ACM symposium on Document engineering*, pages 146–148. ACM Press, 2005.

[Benzaken *et al.*, 2003] Véronique Benzaken, Giuseppe Castagna, et Alain Frisch. CDuce : an XML-centric general-purpose language. Dans *ICFP '03 : Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM Press, 2003.

[Berardi *et al.*, 2005] Daniela Berardi, Diego Calvanese, et Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1) :70–118, 2005.

[Berger *et al.*, 2003] Sacha Berger, François Bry, Sebastian Schaffert, et Christoph Wieser. Xcerpt and visXcerpt : from pattern-based to visual querying of XML and semistructured data. Dans *VLDB'2003 : Proceedings of the 29th international conference on Very large data bases*, pages 1053–1056. VLDB Endowment, 2003.

[Bézivin, 2004] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. *RSTI-L'Objet*, 10(4) :45–157, 2004.

[Bézivin, 2005] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2) :171–188, 2005.

[Blanch et Beaudouin-Lafon, 2006] Renaud Blanch et Michel Beaudouin-Lafon. Programming rich interactions using the hierarchical state machine toolkit. Dans *AVI '06 : Proceedings of the working conference on Advanced visual interfaces*, pages 51–58. ACM, 2006.

[Blouin *et al.*, 2008b] Arnaud Blouin, Olivier Beaudoux, et Stéphane Loiseau. Un tour d'horizon des approches pour la manipulation des données du web. *Document Numérique*, 11(1-2/2008) :63–83, 2008.

[Blumendorf *et al.*, 2008] Marco Blumendorf, Grzegorz Lehmann, Sebastian Feuerstack, et Sahin Albayrak. Executable models for human-computer interaction. Dans *Proceedings of DSV-IS2008*, 2008.

[Buxton, 1990] William Buxton. A three-state model of graphical input. Dans *INTERACT '90 : Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, pages 449–456. North-Holland Publishing Co., 1990.

[Calvary *et al.*, 2003] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Q. Limbourg, L. Bouillon, et Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting With Computers*, 15(3) :289–308, 2003.

[Codd, 1970] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[Coelho et Florido, 2007] Jorge Coelho et Mario Florido. XCentric : logic programming for XML processing. Dans *WIDM '07 : Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 1–8. ACM Press, 2007.

[Collins, 1995] Dave Collins. *Designing Object-Oriented User Interfaces*. Benjamin Cummings, January 1995.

[Combemale, 2008] Benoît Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle*. PhD thesis, Université de Toulouse, 2008.

[Czarnecki et Helsen, 2003] Krzysztof Czarnecki et Simon Helsen. Classification of model transformation approaches. Dans *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Czarnecki et Helsen, 2006] K. Czarnecki et S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3) :621–645, 2006.

[Déry-Pinna *et al.*, 2009] Anne-Marie Déry-Pinna, Cédric Joffroy, Audrey Occello, Philippe Renevier, et Michel Riveill. L'ingénierie dirigée par les modëles au coeur d'un framework d'aide á la composition d'interfaces utilisateurs. Dans *Cinquièmes Journées sur l'Ingénierie Dirigée par les Modèles*, 2009.

[Dragicevic et Fekete, 2001] P. Dragicevic et J.D. Fekete. Input device selection and interaction configuration with ICON. Dans *Proceedings of IHM-HCI 01*, pages 543–448. Springer Verlag, 2001.

[Dragicevic, 2004] Pierre Dragicevic. *Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables*. PhD thesis, Université de Nantes, 2004.

[Drix, 2002] Philippe Drix. *XSLT fondamental*. Eyrolles, april 2002.

[Elliott *et al.*, 2002] James Elliott, Robert Eckstein, Marc Loy, David Wood, et Brian Cole. *Java Swing*. O'Reilly, 2002.

[Elwert et Schlungbaum, 1995] Thomas Elwert et Egbert Schlungbaum. Modelling and generation of graphical user interfaces in the TADEUS approach. Dans *DSVIS'95, Proceedings of the Eurographics Workshop*, 1995.

[Fabro et Valduriez, 2007] Marcos Didonet Del Fabro et Patrick Valduriez. Semi-automatic model integration using matching transformations and weaving models. Dans *The 22nd Annual ACM SAC*, pages 963–970, 2007.

[Fagin *et al.*, 2003] Ronald Fagin, Phokion G. Kolaitis, et Lucian Popa. Data exchange : getting to the core. *ACM Transactions on Database Systems*, 30(1) :174–210, 2003.

[Fagin, 2007] Ronald Fagin. Inverting schema mappings. *ACM Trans. Database Syst.*, 32(4) :25, 2007.

[Favre *et al.*, 2006] Jean-Marie Favre, Jacky Estublier, et Mireille Blay. *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*. lavoisier, 2006.

[Fleurey, 2006] Franck Fleurey. *Langage et méthode pour une ingénierie des modèles fiable*. PhD thesis, Université de Rennes 1, 2006.

[Foley *et al.*, 1984] James D. Foley, Victor L. Wallace, et Peggy Chan. The human factors of computer graphics interaction techniques. *IEEE Comput. Graph. Appl.*, 4(11) :13–48, 1984.

[Foley *et al.*, 1988] J. Foley, C. Gibbs, et S. Kovacevic. A knowledge-based user interface management system. Dans *CHI '88 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 67–72. ACM, 1988.

[Frisch, 2004] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Université Paris 7, 2004.

[Frisch, 2006] Alain Frisch. OCaml + XDuce. Dans *ICFP '06 : Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 192–200. ACM, 2006.

[Gapeyev *et al.*, 2006] Vladimir Gapeyev, François Garillot, et Benjamin C. Pierce. Statically typed document transformation : An Xtatic experience. Dans *In Workshop on Programming Language Technologies for XML (PLAN-X)*, pages 2–13, 2006.

[Gauffre *et al.*, 2008] Guillaume Gauffre, Emmanuel Dubois, et Remi Bastide. Domain-specific methods and tools for the design of advanced interactive techniques. *Lecture Notes in Computer Science*, 5002/2008 :65–76, 2008.

[Gorny, 1995] Peter Gorny. Expose : an hci-counseling tool for user interface designers. *SIGCHI Bull.*, 27(2) :35–37, 1995.

[Griffiths *et al.*, 1998] T. Griffiths, J. Mckirdy, N. Paton, J. Kennedy, R. Cooper, P. Barclay, C. Goble, P. Gray, M. Smyth, A. West, et A. Dinn. An open model-based interface development system : The teallach approach. Dans *In Supplementary Proceedings of DS-VIS'98*, pages 32–49. Eurographics, 1998.

[Haas *et al.*, 2005] Laura M. Haas, Mauricio A. Hernandez, Howard Ho, Lucian Popa, et Mary Roth. Clio grows up : from research prototype to industrial tool. Dans *SIGMOD '05 : Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 805–810. ACM Press, 2005.

[Harel, 1987] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[Harren *et al.*, 2005] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, et Vivek Sarkar. XJ : facilitating XML processing in java. Dans *WWW '05 : Proceedings of the 14th international conference on World Wide Web*, pages 278–287. ACM, 2005.

[Hausmann et Kent, 2003] Jan Hendrik Hausmann et Stuart Kent. Visualizing model mapping in UML. Dans *SoftVis '03 : Proceedings of the 2003 ACM symposium on Software visualization*, pages 169–178. ACM Press, 2003.

[Hayes *et al.*, 1985] Philip J. Hayes, Pedro A. Szekely, et Richard A. Lerner. Design alternatives for user interface management sytems based on experience with cousin. Dans *CHI '85 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 169–175. ACM, 1985.

[Hernandez et Kambhampati, 2004] Thomas Hernandez et Subbarao Kambhampati. Integration of biological sources : current systems and challenges ahead. *SIGMOD Rec.*, 33(3) :51–60, 2004.

[Hosoya et Pierce, 2003] Haruo Hosoya et Benjamin C. Pierce. XDuce : A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2) :117–148, 2003.

[Hudak, 1989] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3) :359–411, 1989.

[Hutchins *et al.*, 1985] E. L. Hutchins, J. D. Hollan, et D. A. Norman. Direct manipulation interfaces. Dans *Human-computer interaction*, pages 311–338, 1985.

[ISO8879, 1986] ISO8879. Information processing–text and office systems–standard generalized markup language (sgml), 1986.

[Jacob *et al.*, 1999] Robert J. K. Jacob, Leonidas Deligiannidis, et Stephen Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transaction Computer-Human Interaction*, 6(1) :1–46, 1999.

[Joffroy, 2009] Cédric Joffroy. Composition d'interfaces homme-machine dirigée par la composition du noyau fonctionnel. Dans *IHM '09 : Proceedings of the 21th international conference on Association Francophone d'Interaction Homme-Machine*, 2009.

[Jouault, 2006] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles.* PhD thesis, Université de Nantes, 2006.

[Kay, 2004a] Michael Kay. *XPath 2.0.* Wiley Publishing Inc., 2004.

[Kay, 2004b] Michael Kay. *XSLT 2.0.* Wiley Publishing Inc., 2004.

[Kazoun et Lott, 2007] Chafic Kazoun et Joey Lott. *Programming Flex 2.0.* O'Reilly Media, 2007.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, et Anurag Mendhekar. Aspect-oriented programming. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[Kirkegaard *et al.*, 2004] Christian Kirkegaard, Anders Møller, et Michael I. Schwartzbach. Static analysis of XML transformations in java. *IEEE Transactions on Software Engineering*, 30(3) :181–192, 2004.

[Kolaitis, 2005] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. Dans *PODS '05 : Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 61–75. ACM Press, 2005.

[Kurtev *et al.*, 2002] Ivan Kurtev, Jean Bézivin, et Mehmet Aksit. Technological spaces : An initial appraisal. Dans *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[Lee et Chu, 2000] Dongwon Lee et Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Rec.*, 29(3) :76–87, 2000.

[Lenzerini, 2002] Maurizio Lenzerini. Data integration : a theoretical perspective. Dans *PODS '02 : Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM Press, 2002.

[Lepreux *et al.*, 2007] Sophie Lepreux, Anas Hariri, José Rouillard, Dimitri Tabary, Jean-Claude Tarby, et Christophe Kolski. Towards multimodal user interfaces composition based on usixml and mbd principles. Dans *HCII 2007*, 2007.

[Lonczewski et Schreiber, 1996] Frank Lonczewski et Siegfried Schreiber. The fuse-system : an integrated user interface design environment. Dans *In proceedings of CADUI'96*, 1996.

[MacDonald, 2007] Matthew MacDonald. *Pro WPF : Windows Presentation Foundation in .NET 3.0*. Apress, 2007.

[Märtin, 1996] Christian Märtin. Software life cycle automation for interactive applications : The ame design environment. Dans *Computer-Aided Design of User Interfaces*, pages 57–74, 1996.

[Miller *et al.*, 2000] Renée J. Miller, Laura M. Haas, et Mauricio A. Hernández. Schema mapping as query discovery. Dans *VLDB '00 : Proceedings of the 26th International Conference on Very Large Data Bases*, pages 77–88. Morgan Kaufmann Publishers Inc., 2000.

[Miller *et al.*, 2001] Renée J. Miller, Mauricio A. Hernandez, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, et Lucian Popa. The clio project : Managing heterogeneity. *SIGMOD Record*, 30(1) :78–83, 2001.

[Minsky, 1968] Marvin L. Minsky. *Semantic Information Processing*. MIT Press, 1968.

[Moore *et al.*, 2004] William Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, et Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004.

[Morris, 2009] Simon Morris. *JavaFX in Action*. Manning Publication Co., 2009.

[Muller *et al.*, 2005] Pierre-Alain Muller, Franck Fleurey, et Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. Dans *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[Muller *et al.*, 2006] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, et Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. Dans *Proceedings of the MoDELS/UML 2006*, 2006.

[Myers *et al.*, 1990] B.A. Myers, D. Giuse, R. B. Dannenberg, B. Vander Zanden, D.Kosbie, P. Marchal, et E. Pervin. Garnet : Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11) :71–85, 1990.

[Myers *et al.*, 1997] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, et Patrick Doane. The amulet environment : New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23 :347–365, 1997.

[Myers *et al.*, 2000] Brad Myers, Scott E. Hudson, et Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7(1) :3–28, 2000.

[Myers et Kosbie, 1996] B. A. Myers et D. S. Kosbie. Reusable hierarchical command objects. Dans *Proc. of CHI'96*, pages 260–267, 1996.

[Myers, 1990] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3) :289–320, 1990.

[Nakatsuyama *et al.*, 1991] Hisashi Nakatsuyama, Makoto Murata, et Koji Kusumoto. A new framework for separating user interfaces from application programs. *SIGCHI Bull.*, 23(1) :88–91, 1991.

[Navarre *et al.*, 2001] David Navarre, Philippe Palanque, Rémi Bastide, et Ousmane Sy. Structuring interactive systems specifications for executability and prototypability. *Lecture Notes in Computer Science*, 1946 :97–119, 2001.

[Norman et Draper, 1986] Donald A. Norman et Stephen W. Draper. *User-Centered System Design : New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, 1986.

[Norman, 1988] Donald A. Norman. *The Design of Everyday Things*. Doubleday Business, 1988.

[OASIS, 2001] OASIS. RELAX NG specification, 2001.

[OASIS, 2002] OASIS. RELAX NG compact syntax specification, 2002.

[Olsen, 1989] D. R. Olsen, Jr. A programming language basis for user interface. Dans *CHI '89 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 171–176. ACM, 1989.

[Olsen, 2007] Dan R. Olsen, Jr. Evaluating user interface systems research. Dans *UIST '07 : Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 251–258. ACM, 2007.

[OMG, 2001] OMG. MDA specification, 2001.

[OMG, 2005a] OMG. MOF QVT Specification, 2005.

[OMG, 2006] OMG. Meta Object Facility (MOF) 2.0 Core Specification, 2006.

[OMG, 2007] OMG. UML 2.1.1 specification, 2007.

[Onizuka *et al.*, 2005] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, et Takashi Honishi. Incremental maintenance for materialized XPath/XSLT views. Dans *WWW '05 : Proceedings of the 14th international conference on World Wide Web*, pages 671–681. ACM Press, 2005.

[Paternò *et al.*, 1997] Fabio Paternò, Cristiano Mancini, et Silvia Meniconi. ConcurTaskTrees : A diagrammatic notation for specifying task models. Dans *INTERACT '97 : Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369, 1997.

[Paternò *et al.*, 2008] Fabio Paternò, Carmen Santoro, Jani Mäntyjärvi, Giulio Mori, et Sandro Sansone. Authoring pervasive multimodal user interfaces. *Int. J. Web Engineering and Technology*, 4(2) :235–261, 2008.

[Petri, 1963] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. Dans *Proceedings of IFIP Congress 62*, pages 386–390, 1963.

[Pietriga *et al.*, 2001] Emmanuel Pietriga, Jean-Yves Vion-Dury, et Vincent Quint. VXT : a visual approach to XML transformations. Dans *DocEng '01 : Proceedings of the 2001 ACM Symposium on Document engineering*, pages 1–10, 2001.

[Popa *et al.*, 2002] Lucian Popa, Yannis Velegrakis, Renee J. Miller, Mauricio A. Hernandez, et Ronald Fagin. Translating web data. Dans *VLDB '02 : Proceedings of the 28th International Conference on Very Large Data Bases*, pages 598–609, 2002.

[Puerta, 1996] Angel Puerta. The MECANO project : Comprehensive and integrated support for model-based interface development. Dans *Computer-Aided Design of User Interfaces*, 1996.

[Puerta, 1997] Angel R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4) :40–47, 1997.

[Raffio *et al.*, 2008] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, et Mauricio A. Hernandez. Clip : a visual language for explicit schema mappings. Dans *Proceeding of IEEE 24th International Conference on Data Engineering*, pages 30–39, 2008.

[Renouard, 2007] Stéphane Renouard. *Interaction Homme-Environnement : Modèle et Outil*. PhD thesis, Université d'Evry-Val d'Essonne, 2007.

[Roth *et al.*, 2006] M. Roth, M. A. Hernandez, P. Coulthard, L. Yan, L. Popa, H. C.-T. Ho, et C. C. Salter. XML mapping technology : making connections in an XML-centric world. *IBM Systems Journal*, 45(2) :389–409, 2006.

[Rumbaugh *et al.*, 2004b] James Rumbaugh, Ivar Jacobson, et Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Professional, 2004.

[Scarpino *et al.*, 2004] Matthew Scarpino, Stephen Holder, Stanford Ng, et Laurent Mihalkovic. *SWT/JFace in Action : GUI Design with Eclipse 3.0*. Manning Publications, 2004.

[Serrano *et al.*, 2008] Marcos Serrano, Laurence Nigay, Jean-Yves L. Lawson, Andrew Ramsay, Roderick Murray-Smith, et Sebastian Denef. The openinterface framework : a tool for multimodal interaction. Dans *CHI '08 : CHI '08 extended abstracts on Human factors in computing systems*, pages 3501–3506, 2008.

[Shneiderman, 1982] Ben Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behavior and information technology*, 1 :237–256, 1982.

[Shneiderman, 1983] Ben Shneiderman. Direct manipulation : a step beyond programming languages. *IEEE Computer*, 16(8) :57–69, 1983.

[Shneiderman, 2004] Ben Shneiderman. *Designing the User Interface : Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 2004.

[Silva et Paton, 2000] Paulo Pinheiro Da Silva et Norman W. Paton. UMLi : The unified modeling language for interactive applications. Dans *UML 2000 - THE UNIFIED MODELING LANGUAGE*, pages 117–132. Springer, 2000.

[Silva, 2000] Paulo Pinheiro Da Silva. User interface declarative models and development environments : A survey. Dans *Proceedings of DSV-IS2000*, pages 207–226. Springer-Verlag, 2000.

[Sottet *et al.*, 2007a] Jean-Sébastien Sottet, Gaëlle Calvary, Joëlle Coutaz, et Jean-Marie Favre. A model-driven engineering approach for the usability of user interfaces. Dans *Proceedings of Engineering Interactive Systems 2007*, pages 140–157, 2007.

[Souchon et Vanderdonckt, 2003] Nathalie Souchon et Jean Vanderdonckt. A review of XML-compliant user interface description languages. *Lecture notes in computer science*, 2844 :377–391, 2003.

[Soutou, 2007] Christian Soutou. *UML 2 pour les bases de données*. Eyrolles, 2007.

[Szekely *et al.*, 1992] Pedro Szekely, Ping Luo, et Robert Neches. Facilitating the exploration of interface design alternatives : the humanoid model of interface design. Dans *CHI '92 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 507–515. ACM, 1992.

[Szekely, 1996] Pedro Szekely. Retrospective and challenges for model-based interface development. Dans *Design, Specification and Verification of Interactive Systems '96*, pages 1–27. Springer-Verlag, 1996.

[Tarby, 1993] J.C. Tarby. *Gestion Automatique du Dialogue Homme-Machine à partir de Spécifications Conceptuelles*. PhD thesis, Université Toulouse I, 1993.

[The UIMS Tool Developers Workshop, 1992] The UIMS Tool Developers Workshop. A metamodel for the runtime architecture of an interactive system : the uims tool developers workshop. *SIGCHI Bulletin*, 24(1) :32–37, 1992.

[Thevenin et Coutaz, 1999] David Thevenin et Joëlle Coutaz. Plasticity of user interfaces : Framework and research agenda. Dans *Proceedings of Interact'99*, pages 110–117, 1999.

[Ullman, 1988] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems Vol. 1 : Classical Database Systems*. Computer Science Press, 1988.

[Ullman, 1989] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems Volume II : The New Technologies*. Computer Science Press, 1989.

[van Deursen *et al.*, 2000] Arie van Deursen, Paul Klint, et Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, 2000.

[Vanderdonckt et Bodart, 1993] Jean M. Vanderdonckt et François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. Dans *CHI '93 : Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 424–429. ACM, 1993.

[Vanderdonckt, 2005] Jean Vanderdonckt. A MDA-compliant environment for developing user interfaces of information systems. *Lecture Notes in Computer Science*, 3520/2005 :16–31, 2005.

[Vanderdonckt, 2008] Jean Vanderdonckt. Model-driven engineering of user interfaces : Promises, successes, and failures. Dans *Proceedings of 5th Annual Romanian Conference on Human-Computer Interaction ROCHI'2008*, 2008.

[Varró *et al.*, 2002] Dániel Varró, Gergely Varró, et András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2) :205–227, 2002.

[Villard et Layaïda, 2002] Lionel Villard et Nabil Layaïda. An incremental XSLT transformation processor for XML document manipulation. Dans *WWW '02 : Proceedings of the 11th international conference on World Wide Web*, pages 474–485, New York, NY, USA, 2002. ACM Press.

[Vion-Dury *et al.*, 2002] Jean-Yves Vion-Dury, Veronika Lux, et Emmanuel Pietriga. Experimenting with the circus language for xml modeling and transformation. Dans *DocEng '02 : Proceedings of the 2002 ACM symposium on Document engineering*, pages 82–87. ACM Press, 2002.

[Vion-Dury, 1999] Jean-Yves Vion-Dury. *Circus : un générateur de composants pour le traitement des langages visuels et textuels.* PhD thesis, Université Joseph Fourier, Grenoble, 1999.

[W3C, 1998] W3C. W3C XML Specification DTD, 1998.

[W3C, 2003] W3C. Scalable Vector Graphics 1.1 specification, 2003.

[W3C, 2006a] W3C. Extensible markup language 1.1 specification, 2006.

[W3C, 2006b] W3C. XML Path language (XPath) 2.0 proposed recommendation, 2006.

[W3C, 2006c] W3C. XQuery 1.0 : An XML query language, 2006.

[W3C, 2007a] W3C. XML Schema definition language 1.1, 2007.

[W3C, 2007b] W3C. XSL transformations (XSLT) 2.0 recommendation, 2007.

[W3C, 2008] W3C. XQuery update facility recommendation, 2008.

[Wallace et Runciman, 1999] Malcolm Wallace et Colin Runciman. Haskell and XML : generic combinators or type-based translation ? Dans *ICFP '99 : Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 148–159, 1999.

[Wilson *et al.*, 1993] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, et P. Markopoulos. Beyond hacking : a model based approach to user interface design. Dans *In Proceedings of HCI'93*, pages 217–231. University Press, 1993.

[Wilson, 1990] David Wilson. *Programming with MacApp.* Addison-Wesley Publishing Company, 1990.

[Yan *et al.*, 2001] Ling Ling Yan, Renée J. Miller, Laura M. Haas, et Ronald Fagin. Data-driven understanding and refinement of schema mappings. Dans *SIGMOD '01 : Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 485–496. ACM Press, 2001.

[Zanden et Myers, 1990] Brad Vander Zanden et Brad A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. Dans *CHI '90 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34. ACM, 1990.

Annexes

Annexe A

Compléments au langage Malan

A.1 La grammaire

La grammaire des instructions des fonctions Malan est identique à celle des instructions d'une fonction ou d'une méthode Java, à quelques différences près :

- Les seuls types disponibles sont ceux du langage Malan ; les types `double` et `char`, ainsi que les tableaux ne sont pas disponibles.
- L'initialisation d'une liste s'effectue de la manière suivante : `list<int> l = nil` définit une liste vide ; `list<int> l = { 1, 2 }` définit une liste d'entiers composée de deux éléments.
- Il n'est pas possible d'instancier une classe.

La grammaire suivante décrit le langage concret de Malan.

```
1  CORRESP_SCHEMAS = (ID ":")? CHEMIN_UML "->" CHEMIN_UML
2                    "{" (CORRESPONDANCE_CLASSE | FONCTION | PARAMETRE)* "}"
3  CHEMIN_UML = """ CHEMIN SEPARATEUR PACKAGE_UML (";" PACKAGE_UML)* """
4  PARAMETRE  = "param" ID
5  CORRESPONDANCE_CLASSE = (NOM_CORRESP ":")? CLASSE_SRC ALIAS_SRC?
6                   ("," CLASSE_SRC ALIAS_SRC?)* FILTRE? "->"
7                   CLASSE_CIBLE ALIAS_CIBLE? "{" INSTRUCTION* "}"
8  FILTRE          = "[" EXPRESSION_BOOLEENNE "]"
9  INSTUCTION      = CORRESPONDANCE_RELATIONS | CORRESPONDANCE_ATTRIBUTS
10                  INSTRUCTION_CONDITIONNELLE | REGROUPEMENT | ALIAS
11 CHEMIN_CLASSE   = ID ("." ID)*
12 ALIAS_CIBLE     = ID
13 ALIAS_SRC       = ID
14 NOM_CORRESP     = ID
15 CLASSE_SRC      = CHEMIN_CLASSE
16 CLASSE_CIBLE    = CHEMIN_CLASSE
17 ALIAS           = "alias" ID EXPRESSION
18 INTERVALLE      = "," ID "=" EXPRESSION_ARITHMETIQUE ".." EXPRESSION_ARITHMETIQUE
19 CORRESPONDANCE_RELATIONS = EXPRESSION_SELECTION ("," EXPRESSION_SELECTION)*
20                            "->" EXPRESSION_SELECTION ("," INTERVALLE)?
21 CORRESPONDANCE_ATTRIBUTS = EXPRESSION "->"
22                            EXPRESSION_SELECTION ("," INTERVALLE)?
23 REGROUPEMENT    = EXPRESSION_SELECTION ("," INTERVALLE)? "{" INSTRUCTION* "}"
24 EXPRESSION_CONDITIONNELLE =
25                "if" "(" EXPRESSION_BOOLEENNE ")" INSTRUCTIONS_SI_SINON
```

```
26                ("else" INSTRUCTIONS_SI_SINON)?
27  INSTRUCTIONS_SI_SINON     = ("{" INSTRUCTION* "}") | INSTRUCTION
28  EXPRESSION_SELECTION = ( "(" ( ID("."ID)*
29                             ( ")" (EXPRESSION_SELECTION | EXPRESSION_NAVIG) ) |
30                             ( EXPRESSION_NAVIG ")" EXPRESSION_NAVIG? )
31                           ) | EXPRESSION_SELECTION ")" EXPRESSION_NAVIG?
32                         ) | EXPRESSION_ID
33  EXPRESSION_ID     = ID ( EXPRESSION_NAVIG )?
34  EXPRESSION_NAVIG  = EXPRESSION_POINTE | EXPRESSION_INDICE
35  EXPRESSION_POINTE = "." EXPRESSION_ID
36  EXPRESSION_INDICE = "[" EXPRESSION_ARITHMETIQUE "]" EXPRESSION_POINTE?
37  EXPRESSION        = EXPRESSION_ARITHMETIQUE | EXPRESSION_BOOLEENNE
38  FONCTION          = "function" ID ID "(" (ID ID ("," ID ID)*)? ")"
39                      "{" INSTRUCTION_FCT* "return " INSTRUCTION_FCT "}"
40
41  ID      = LETTER (LETTER | DIGIT)*
42  DIGIT   = "0".."9"
43  LETTER  = "A".."Z"|"_"|"a".."z"
```

A.2 La sémantique des opérateurs

Cette section décrit la sémantique des opérateurs Malan.

integer

$$\text{(int-add)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

$$\text{(int-diff)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \neq e_2 : bool}$$

$$\text{(int-minus)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 - e_2 : int}$$

$$\text{(int-gt)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 > e_2 : bool}$$

$$\text{(int-times)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \times e_2 : int}$$

$$\text{(int-ge)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \geq e_2 : bool}$$

$$\text{(int-div)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 / e_2 : int}$$

$$\text{(int-lt)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 < e_2 : bool}$$

$$\text{(int-mod)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \% e_2 : int}$$

$$\text{(int-le)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \leq e_2 : bool}$$

$$\text{(int-eq)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 == e_2 : bool}$$

float

$$\text{(float-add)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 + e_2 : float} \qquad \text{(float-diff)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \neq e_2 : bool}$$

$$\text{(float-minus)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 - e_2 : float} \qquad \text{(float-gt)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 > e_2 : bool}$$

$$\text{(float-times)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \times e_2 : float} \qquad \text{(float-ge)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \geq e_2 : bool}$$

$$\text{(float-div)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 / e_2 : float} \qquad \text{(float-lt)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 < e_2 : bool}$$

$$\text{(float-mod)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \% e_2 : float} \qquad \text{(float-le)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \leq e_2 : bool}$$

$$\text{(float-eq)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 == e_2 : bool}$$

float/integer

$$\text{(IF-add)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 + e_2 : float} \qquad \text{(FI-div)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 / e_2 : float}$$

$$\text{(FI-add)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : float} \qquad \text{(IF-times)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \times e_2 : float}$$

$$\text{(IF-sub)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 - e_2 : float} \qquad \text{(FI-times)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \times e_2 : float}$$

$$\text{(FI-sub)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 - e_2 : float} \qquad \text{(FI-mod)}\ \frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \% e_2 : float}$$

$$\text{(IF-div)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 / e_2 : float} \qquad \text{(IF-mod)}\ \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \% e_2 : float}$$

boolean

$$\text{(bool-and)}\ \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \&\& e_2 : bool} \qquad \text{(bool-eq)}\ \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 == e_2 : bool}$$

$$\text{(bool-or)}\ \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 || e_2 : bool}$$

$$\text{(bool-not)}\ \frac{\Gamma \vdash e : bool}{\Gamma \vdash !e : bool} \qquad \text{(bool-diff)}\ \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \neq e_2 : bool}$$

string

$$\text{(str-concat1)}\ \frac{\Gamma \vdash e_1 : string \quad \Gamma \vdash e_2 : \nu}{\Gamma \vdash e_1 + e_2 : string} \qquad \text{(str-eq2)}\ \frac{\Gamma \vdash e_1 : \nu \quad \Gamma \vdash e_2 : string}{\Gamma \vdash e_1 == e_2 : bool}$$

$$\text{(str-concat2)}\ \frac{\Gamma \vdash e_1 : \nu \quad \Gamma \vdash e_2 : string}{\Gamma \vdash e_1 + e_2 : string} \qquad \text{(str-diff1)}\ \frac{\Gamma \vdash e_1 : string \quad \Gamma \vdash e_2 : \nu}{\Gamma \vdash e_1 \neq e_2 : bool}$$

$$\text{(str-eq1)}\ \frac{\Gamma \vdash e_1 : string \quad \Gamma \vdash e_2 : \nu}{\Gamma \vdash e_1 == e_2 : bool} \qquad \text{(str-diff2)}\ \frac{\Gamma \vdash e_1 : \nu \quad \Gamma \vdash e_2 : string}{\Gamma \vdash e_1 \neq e_2 : bool}$$

list

$$\text{(listes-concat)}\ \frac{\Gamma \vdash e_1 : list < \nu > \quad \Gamma \vdash e_2 : list < \nu >}{\Gamma \vdash e_1 + e_2 : list < \nu >} \qquad \text{(liste-élément-sub)}\ \frac{\Gamma \vdash e_1 : list < \nu > \quad \Gamma \vdash e_2 : \nu}{\Gamma \vdash e_1 - e_2 : list < \nu >}$$

$$\text{(début-concat)}\ \frac{\Gamma \vdash e_1 : \nu \quad \Gamma \vdash e_2 : list < \nu >}{\Gamma \vdash e_1 + e_2 : list < \nu >}$$

$$\text{(fin-concat)}\ \frac{\Gamma \vdash e_1 : list < \nu > \quad \Gamma \vdash e_2 : \nu}{\Gamma \vdash e_1 + e_2 : list < \nu >} \qquad \text{(liste-liste-sub)}\ \frac{\Gamma \vdash e_1 : list < \nu > \quad \Gamma \vdash e_2 : list < \nu >}{\Gamma \vdash e_1 - e_2 : list < \nu >}$$

A.3 Description des fonctions prédéfinies Malan

Cette section décrit les fonctions prédéfinies Malan utilisées dans ce manuscrit.

$abs(int) : int$, $abs(float) : float$. Cette fonction retourne la valeur absolue de celle passée en paramètre.

$abs(list < int >) : list < int >$, $abs(list < float) : list < float >$. Cette fonction retourne les valeurs absolues des celles passées en paramètre.

$invert(list) : list$. Cette fonction retourne une liste composée des objets de la liste donnée en paramètre. La position de chaque objet de la liste cible est inverse à celle dans la liste source.

$length(string) : int$. Cette fonction retourne le nombre de caractères de la chaîne passée en paramètre.

$max(list < int >) : int$, $max(list < float >) : float$. Cette fonction retourne la valeur maximale de la liste donnée en paramètre. La liste doit contenir des entiers ou des flottants.

$min(list < int >) : int$, $min(list < float >) : float$. Cette fonction retourne la valeur minimale de la liste donnée en paramètre. La liste doit contenir des entiers ou des flottants.

$sub(list, int, int) : list$. Cette fonction retourne un sous-ensemble de la liste passée en paramètre. Le deuxième et le troisième paramètre correspondent, respectivement, à la position de départ et de fin des objets à placer dans la nouvelle liste.

$toLowerCase(string) : string$. Cette fonction retourne une chaîne de caractères en minuscules correspondant à celle passée en paramètre.

$toUpperCase(string) : string$. Cette fonction retourne une chaîne de caractères en majuscules correspondant à celle passée en paramètre.

$toNumber(string) : float$. Cette fonction convertie une chaîne de caractères en un flottant.

$unique(list) : list$. Cette fonction retourne les éléments de la liste d'entrée sans doublon.

$position(\kappa) : int$. Cette fonction retourne la position de l'instance donnée en paramètre dans sa relation. L'instance n'appartient pas à une relation, la valeur -1 est retournée.

$position(\tau, list < \tau >) : int$. Cette fonction retourne la position d'un objet dans une liste. La valeur -1 est retournée si l'objet n'est pas présent.

$position(\tau, list < list < \tau >>) : list < int >$. Cette fonction retourne la position d'un objet dans une matrice. La position retournée se compose de deux valeurs : la position dans la première liste et celle dans la seconde.

A.4 Machine de Turing en Malan

Cette section décrit une machine de Turing universelle en Malan (une machine de Turing universelle est une machine de Turing pouvant modéliser n'importe quelle machine de Turing). Une machine de Turing, illustrée par la figure A.1, se compose :

- D'un *ruban* contenant un ensemble de *cases* possédant chacune un symbole d'un alphabet donné. Le *symbole blanc* est un symbole spécial utilisé pour définir la valeur par défaut d'une cellule. Un ruban est indéfiniment extensible de son côté droit tout comme de son côté gauche.
- D'un ensemble fini d'*états* où l'état initial q_0 correspond à l'état de départ. Le processus s'arrête lorsqu'un état final est atteint.
- D'une *tête de lecture/écriture*, qui lit et écrit des symboles dans les cases, et se déplace vers la droite ou la gauche de la case courante.
- D'un ensemble d'*actions* ; une action est exécutée lorsque le symbole et l'état courant, lus par la tête, correspondent respectivement à l'état d'entrée et au symbole de l'action. L'exécution d'une action remplace l'état et le symbole courant par l'état et le symbole de sortie de l'action. Ainsi, en fonction de l'action, la tête se déplace vers la droite ou la gauche de la case courante.

La figure A.2 correspond à une fonction Malan définissant une machine de Turing universelle. Cette fonction prend en entrée (lignes 1 et 2) un ruban initial T, pouvant contenir des cases, un état de départ q_0, et un ensemble d'actions A. Elle retourne le ruban final composé

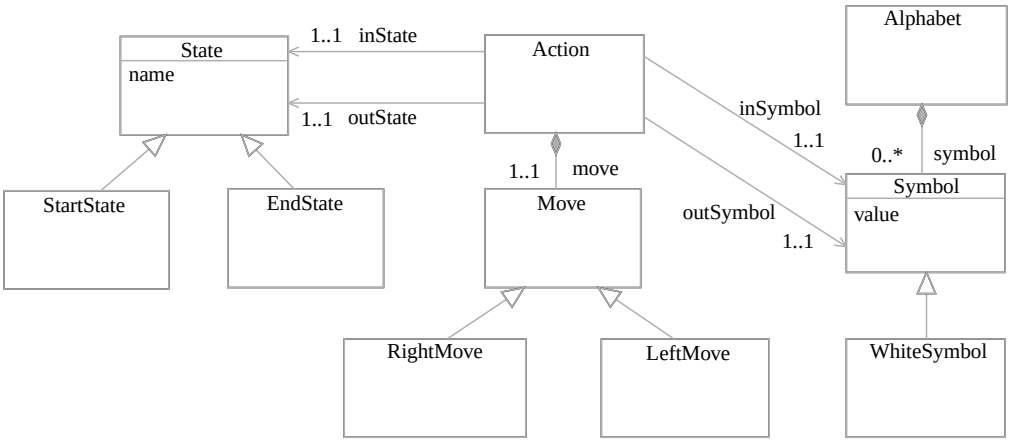


Fig. A.1: Diagramme de classes UML d'une machine de Turing universelle en Malan

d'une liste de symboles. Les lignes 3 et 6 déclarent les variables utilisées dans la fonction, où `tape` correspond au ruban résultant de l'exécution de la machine de Turing, `currState` l'état courant, `position` la position courante de la tête sur le ruban, et `end` un booléen indiquant si le processus doit s'arrêter ou non. La machine se compose d'une boucle « tant que » (de la ligne 8 à la ligne 33) se terminant lorsque l'état courant est un état final (ligne 8), ou si aucune action n'existe pour l'état et le symbole courant (ligne 25). la première étape de la boucle consiste à agrandir le ruban vers la droite ou la gauche en utilisant un symbole blanc dans le cas où la position courante pointe sur une case qui n'existe pas encore (de la ligne 12 à la ligne 17). Une action correspondant aux critères est ensuite recherchée (de la ligne 19 à la ligne 23). Si aucune action n'existe, le processus s'arrête ; sinon, le symbole et l'état courant sont remplacés par le symbole et l'état de sortie de l'action. La tête est alors déplacée vers la droite ou vers la gauche de la case.

```
1  function list<Symbol> turingMachine(list<Symbol> T,
2   StartState q0, Set<Action> A) {
3     int position      = 1;
4     State currState   = q0;
5     bool end          = false;
6     list<Symbol> tape = T;
7
8     while(!end && !(currState is EndState)) {
9       Action a = null;
10      int i    = 1;
11
12      if(position<1) {
13          position = 1;
14          tape     = WhiteSymbol + tape;
15      }
16      else if(position>|tape|)
17              tape = tape + WhiteSymbol;
18
19      while(i<=|A| && a==null)
20          if(A[i].inState.name==currState.name &&
21             tape[position].value==A[i].inSymb.value)
22               a = A[i];
23          else i++;
24
25      if(a==null) end = true;
26      else {
27          tape[position].value = a.outSymb.value;
28          currState = a.outState;
29
30          if(a.move is Right) position++;
31          else position--;
32      }
33    }
34    return tape;
35 }
```

Fig. A.2: Machine de Turing universelle en Malan

Annexe B

Compléments au modèle Malai

Le diagramme de classes de la figure B.1 définit un modèle générique d'interface. La classe `Component` définit de manière générale un composant graphique. Les classes qui héritent de `Component` correspondent à des composants graphiques classiques disponibles sur la majorité des plates-formes d'IHM.

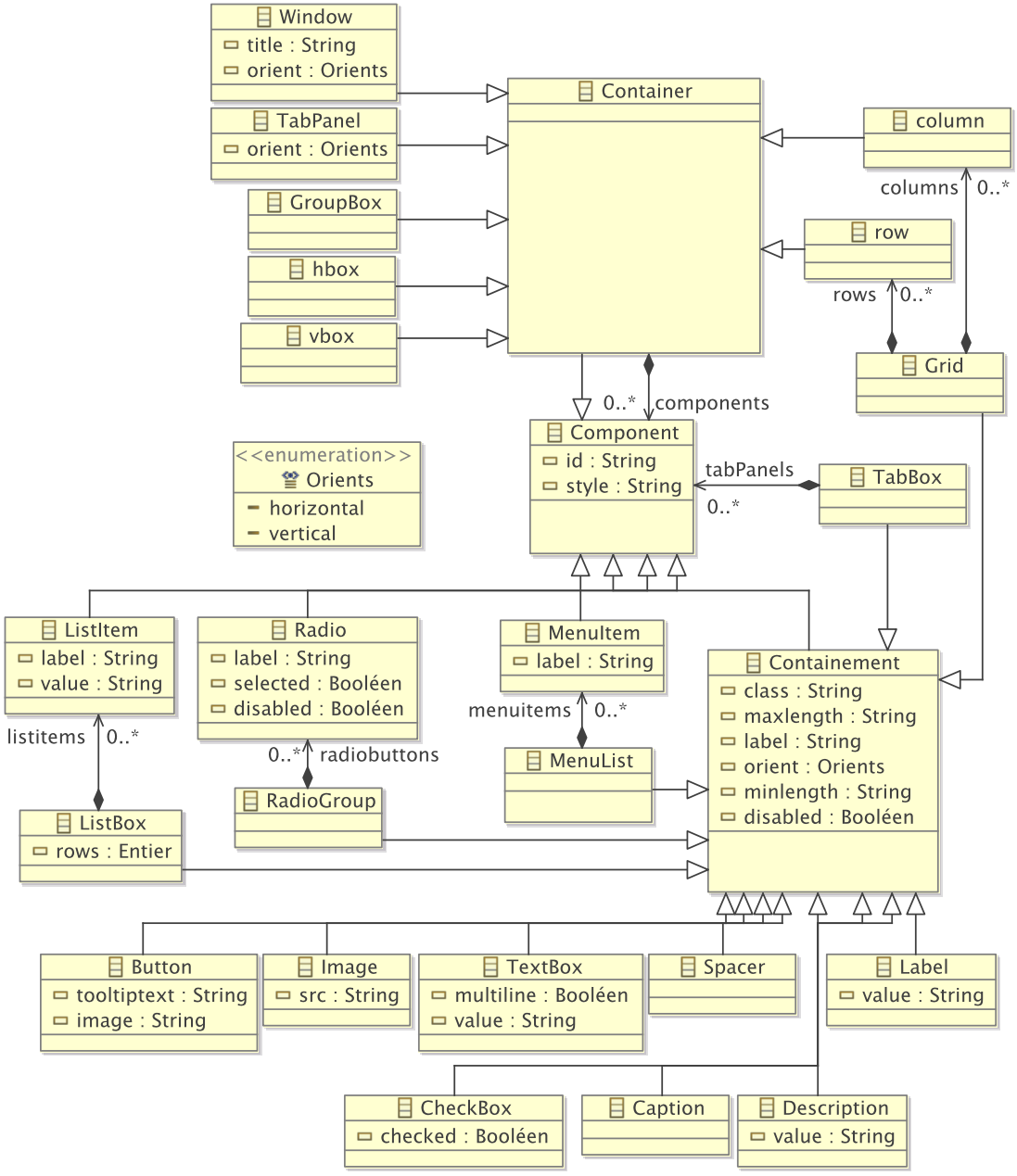


Fig. B.1: Modèle générique d'interface

Annexe C

Compléments des études de cas

C.1 Correspondances de schémas

Cette section définit tout d'abord les correspondances de schémas entre les présentations abstraites et les données sources des deux études de cas. Les fonctions utilisées mais non définies dans les études de cas sont ensuite décrites.

Présentation abstraite vers données sources pour l'éditeur de dessins

Cette correspondance de schémas débute par la mise en relation des classes `ModèleCanvas` et `SVG` (ligne 2). Cette correspondance de classes définit une correspondance de relations stipulant que pour chaque forme, il existe une instance `Shape`. La correspondance de classes `Forme2Shape` relie les attributs de la classe `Forme` avec ceux de la classe `Shape` (ligne 3). Elle définit notamment les attributs `fill` et `stroke` en respectant la grammaire suivante : `"rgb(" ENTIER "," ENTIER "," ENTIER ")"`. Elle définit également l'angle de rotation de la forme en mettant en relation la `Forme` avec une `Transformation`. Cette correspondance de relations se réfère à la correspondance de classes `Forme2Rotation` (ligne 10) qui spécifie l'attribut cible `angle` à partir de l'attribut source `angleRotation`. La correspondance de classes `Ellipse2Ellipse` (ligne 11) calcule les coordonnées du centre de l'ellipse en utilisant les fonctions `obtenirXMin`, `obtenirXMax`, `obtenirYMin` et `obtenirYMax` de la classe `Forme`. De même, les rayons de l'ellipse sont calculés *via* les fonctions `obtenirLargeur` et `obtenirLongueur`. De la même manière, la correspondance de classes `Rectangle2Rect` (ligne 17) calcule les coordonnées et les dimensions d'un rectangle SVG en utilisant des fonctions de la classe `Forme`. La correspondance de classes `Polygone2Polypoint` (ligne 23) met en relation les points des classes `Polygone` et `Polypoint`. Il s'ensuit les correspondances de classes `Polygone2Polygon` (ligne 24) et `Polygone2Polyline` (ligne 25) stipulant qu'un polygone correspond à une instance `Polygon` ou une instance `Polyline` si le polygone est, respectivement, fermé ou non.

```
1  "canvas.uml/canvasModèle" -> "canvas.uml/svg" {
2    ModèleCanvas -> SVG { formes -> shapes }
3    Forme2Shape : Forme f -> Shape {
4      toString(épaisseur)    -> strokeWidth
5      f                      -> transform[1]
6      "rgb("+couleurFond[1]+","+couleurFond[2]+","+couleurFond[3]+")"->fill
7      "rgb("+couleurContour[1]+","+couleurContour[2]+","+couleurContour[3]+")"
```

```
8                          -> stroke
9     }
10    Forme2Rotation : Forme -> Rotation { angleRotation -> angle }
11    Ellipse2Ellipse : Ellipse -> Ellipse {
12      (obtenirXMin()+obtenirXMax)/2 -> cx
13      (obtenirYMin()+obtenirYMax)/2 -> cy
14      obtenirLargeur()/2            -> rx
15      obtenirLongueur()/2           -> ry
16    }
17    Rectangle2Rect : Rectangle -> Rect {
18      obtenirXMin()      -> x
19      obtenirYMin()      -> y
20      obtenirLargeur()   -> width
21      obtenirLongueur()  -> height
22    }
23    Polypone2Polypoint : Polygone -> Polypoint { points -> points}
24    Polygone2Polygon   : Polygone[!ouvert] -> Polygon  {}
25    Polygone2Polyline  : Polygone[ouvert]  -> Polyline {}
26    Point2Point        : Point             -> Point    { x -> x   y -> y }
27  }
```

Présentation abstraite vers données sources pour l'agenda

Cette correspondance de schémas possède un paramètre `indexSemaine` définissant la semaine présentée par l'agenda (ligne 2). La correspondance de classes `Agenda2EmploiTps` (ligne 4) met en relation l'agenda avec la semaine concernée. La correspondance de classes `Agenda2Semaine` lie, quant à elle, les évènements d'un jour donné aux enseignements de cette journée. La correspondance de classes `Evt2Ens` se charge de mettre à jour l'enseignement cible à partir de l'évènement source (ligne 14). Pour cela, la matière, l'enseignant et la salle sont tout d'abord récupérés à partir des descriptions de l'évènement (lignes 18 à 21), pour être ensuite mis en correspondance avec la matière, l'enseignant et la salle de l'enseignement (lignes 22 à 24). Si le type de l'évènement est égal à 1, 2, ou 3, il s'agit respectivement d'un TP (ligne 26), d'un TD (ligne 27) ou d'un cours (ligne 28).

```
1   Agenda2EDT : "edt.uml/agendaCanvas" -> "edt.uml/edt" {
2     param indexSemaine
3
4     Agenda2EmploiTps : AgendaCanvas ag -> EmploiDuTps {
5       ag -> semaines[indexSemaine]
6     }
7     Agenda2Semaine : AgendaCanvas -> Semaine {
8       évènements[jour=="Lundi" && estTypeEdT(type)] -> jours[1].enseignements
9       évènements[jour=="Mardi" && estTypeEdT(type)] -> jours[2].enseignements
10      évènements[jour=="Mercredi"&& estTypeEdT(type)] -> jours[3].enseignements
11      évènements[jour=="Jeudi" && estTypeEdT(type)] -> jours[4].enseignements
12      évènements[jour=="vendredi"&& estTypeEdT(type)] -> jours[5].enseignements
13    }
14    Evt2Ens : Evènement evt -> Enseignement ens {
15      evt.date -> ens.horaires[1]
16      evt.durée.ajouter(evt.date) ->  ens.horaires[2]
17      alias res ens.edt.resources
18      alias m ((list<Matière>)res[is Matière])[intitulé==evt.description[1]]
```

```
19      alias e ((list<Enseignant>)res[is Enseignant])[(prénom+" "+nom)==
20              evt.description[2]]
21      alias s ((list<Salle>)res[is Salle])[nom==evt.description[3]]
22      m[1] -> ens.matière
23      e[1] -> ens.enseignant
24      s[1] -> ens.salle
25    }
26    Evènement[type==1] -> TP     {}
27    Evènement[type==2] -> TD     {}
28    Evènement[type==3] -> Cours {}
29    Date -> Date {
30      heure  -> heure
31      minute -> minute
32    }
33    function bool estTypeEdT(int type) {
34      return type==1 || type==2 || type== 3;
35    }
36  }
```

Définition des fonctions

getRotationAngle

Cette fonction retourne 0 si le paramètre `s` est nul. Dans le cas contraire, elle récupère les transformations de type `Rotation` de la forme pour en déduire son angle de rotation.

```
1  function double getRotationAngle(Shape s) {
2    if(s==null) return 0;
3    List<Rotation> rotations = (Rotation)s.transformations[is Rotation];
4    double angle =  0;
5    for(Rotation r in rotations)
6     angle += r.angleRotation;
7    return angle + getRotationAngle(s.parent);
8  }
```

getFill

La fonction `getFill` retourne la couleur noir si le paramètre `s` est nul ou si l'attribut `fill` ne débute pas par `"rgb("`. Dans le cas contraire, la fonction `SVGRGBtoRGB` est appelée pour convertir la couleur du format source `rgb(r,v,b)` vers une liste de trois entiers. Cette fonction sépare les trois composantes de la couleur *via* la fonction prédéfinie `split`. Elles sont ensuite converties en entiers pour être ajoutées à la liste.

```
1  function list<int> getFill(Shape s) {
2    list<int> fill;
3    if(s!=null)
4      if(|s.fill|==1)
5        if(startsWith(s.fill, "rgb(")) fill = SVGRGBtoRGB(s.fill);
6      else fill = getFill(s.parent);
7    if(|fill|==0) {
8      fill = fill + 0;
9      fill = fill + 0;
```

```
10      fill = fill + 0;
11    }
12    return fill;
13  }
14  function list<int> SVGRGBtoRGB(String str) {
15    String s          = substring(5, str.length(), str);
16    list<String> rgbs = split(",", s);
17    list<int> l       = nil;
18    if(|rgbs|==3) {
19      l = l + toInteger(replaceAll("[\t\r\n ]", "", rgbs[1]));
20      l = l + toInteger(replaceAll("[\t\r\n ]", "", rgbs[2]));
21      l = l + toInteger(replaceAll("[\t\r\n ]", "", rgbs[3]));
22    }
23    return l;
24  }
```

getStroke

La fonction `getStroke` fonctionne de la même manière que la fonction `getFill` à la différence que l'attribut concerné par la fonction est l'attribut `stroke` de la classe `Shape` au lieu de l'attribut `fill`.

```
1   function list<int> getStroke(Shape s) {
2     list<int> stroke;
3     if(s!=null)
4       if(|s.fill|==1)
5         if(startsWith(s.stroke, "rgb(")) stroke = SVGRGBtoRGB(s.stroke);
6       else stroke = getStroke(s.parent);
7     if(|stroke|==0) {
8       stroke = stroke + 0;
9       stroke = stroke + 0;
10      stroke = stroke + 0;
11    }
12    return stroke;
13  }
```

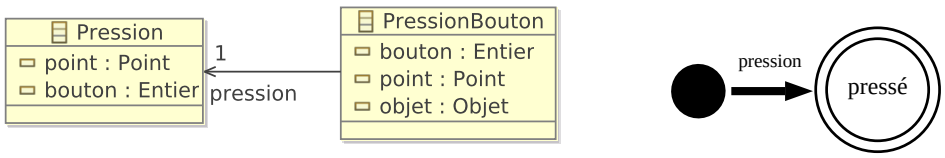
C.2 Interaction

Cette section présente les interactions classiques utilisées dans les études de cas. Elles regroupent les interactions se réalisant à l'aide de HID standards comme le clavier et la souris. La pression d'un bouton, de touches d'un clavier, ainsi que le simple et double clic sont décrits.

Pression d'un bouton

La pression d'un bouton est une interaction relative simple à décrire. Son diagramme de classes (*cf.* figure C.1a) définit la classe `PressionBouton` possédant trois attributs : `bouton` représente le numéro du bouton pressé ; `point` correspond à la position de la pression ; `objet` correspond à l'objet se localisant au point pressé. Cette interaction se fonde sur l'évènement `Pression`. Sa machine à états (*cf.* figure C.1b) démarre lorsque qu'un évènement `pression`,

relatif à la pression d'un bouton d'une souris par exemple, survient. L'état courant devient alors l'état terminal `pressé`.

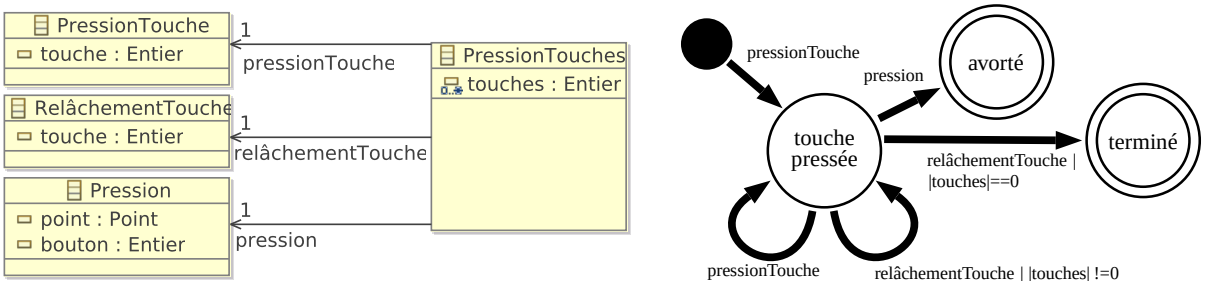


(a) Diagramme de classes (b) Machine à états

Fig. C.1: Pression d'un bouton

Pression de touches

La pression de touches consiste à pouvoir presser plusieurs touches d'un clavier en même temps. La classe définissant la partie statique de l'interaction (*cf.* figure C.2a) contient un unique attribut `touches` contenant les différentes touches pressées lors de l'interaction. Cette interaction se fonde sur les évènements `PressionTouche`, `RelâchementTouche` et `Pression`. La figure C.2b décrit la machine à états de cette interaction qui débute avec la pression d'une première touche ; l'état `touche pressée` est alors atteint. A partir de ce dernier peut survenir d'autres pressions ou des relâchements de touches. Lors d'un relâchement, l'état terminal est alors atteint s'il n'existe plus de touche pressée. La pression d'un bouton pendant l'interaction avorte l'interaction.



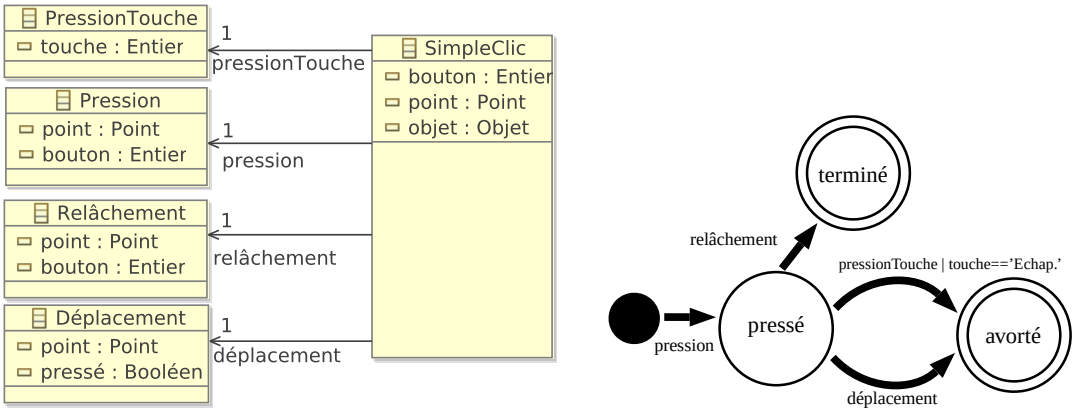
(a) Diagramme de classes (b) Machine à états

Fig. C.2: Pression de touches

Simple et double clic

Le simple et double clic sont des interactions standards utilisées dans tous les SI. Le simple clic débute avec une pression d'un bouton et se termine lorsque ce dernier est relâché (*cf.* figure C.3b). Cette interaction est avortée si un mouvement ou la pression sur la touche « Echap. » survient alors que le bouton est pressé. Un simple clic se caractérise par l'identifiant du bouton, la position du clic et de l'objet de la présentation concrète se situant à cette même position (*cf.* figure C.3a).

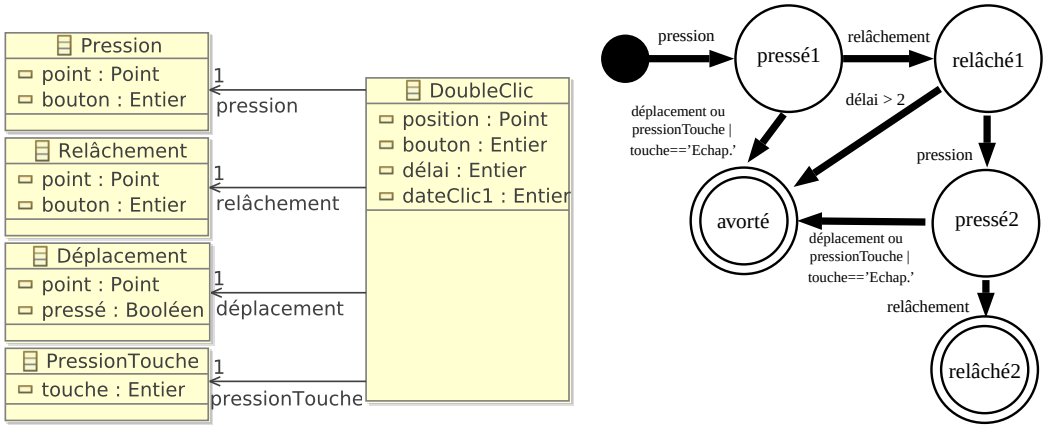
Le double clic se compose de deux simples clics successifs entre lesquels le délai de séparation ne doit pas excéder deux secondes (*cf.* figure C.4b) auquel cas l'interaction est avortée. Le double



(a) Diagramme de classes (b) Machine à états

Fig. C.3: Simple clic

clic possède les mêmes attributs que le simple clic avec en plus la date du premier clic ainsi que le délai autorisé (*cf.* figure C.4a).



(a) Diagramme de classes (b) Machine à états

Fig. C.4: Double clic

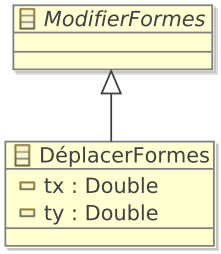
C.3 Actions

Cette section contient la définition des actions triviales ou non essentielles des études de cas des chapitres 9 et 10.

Déplacer des formes

L’action `DéplacerFormes`, qui hérite de l’action abstraite `ModifierFormes`, possède deux attributs, à savoir `tx` et `ty` correspondant respectivement à la translation selon l’axe des X

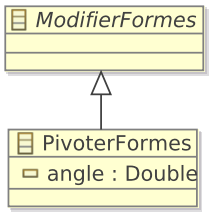
et celui des Y. La fonction `peutExécuter` vérifie que ces paramètres sont valides et appelle la fonction `peutExécuter` de la classe `ModifierFormes`. La méthode `déplacer`, ligne 7, déplacer les formes sélectionnées en utilisant `tx2` et `ty2` donnés en paramètres. Cette méthode est appelée dans les méthodes `exécuter` et `réexécuter` dans lesquels `tx2` et `ty2` correspondent aux attributs `tx` et `ty`. La méthode `annuler` appelle `déplacer` avec les valeurs opposées des deux attributs, c.-à-d. `-tx` et `-ty`.




```
1  booléen peutExécuter() {
2    retourner super.peutExécuter() et (tx≠0 ou ty≠0)
3  }
4  exécuter()    { déplacer(tx, ty)    }
5  annuler()     { déplacer(-tx, -ty) }
6  réexécuter()  { déplacer(tx, ty)    }
7  déplacer(Double tx2, Double ty2) {
8    pour chaque Forme forme de sélection faire
9      forme.déplacer(tx2, ty2)
10 }
```

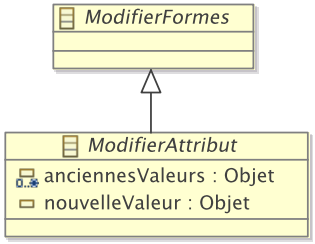
Pivoter des formes

L'action `PivoterFormes`, qui hérite de l'action `ModifierFormes`, possède comme paramètre l'angle de rotation. La fonction `peutExécuter` de cette action (ligne 1) vérifie si l'angle est différent de 0 et si les conditions spécifiées dans la fonction `peutExécuter` de l'action `ModifierFormes` sont respectées. La méthode `pivoter` (ligne 7) ajoute à l'angle de rotation de chaque forme, la valeur du paramètre `angleRotation`. Les méthodes `exécuter`, `annuler` et `réexécuter` (lignes 4 à 6) appelle cette méthode en donnant au paramètre `angleRotation` la valeur de l'attribut `angle` pour les méthodes `exécuter` et `réexécuter`, ou la valeur `angle*-1` pour la méthode `annuler`.




```
1  booléen peutExécuter() {
2    retourner super.peutExécuter() et angle≠0
3  }
4  exécuter()    { pivoter(angle)  }
5  annuler()     { pivoter(-angle) }
6  réexécuter()  { pivoter(angle)  }
7  pivoter(Double angleRotation) {
8    pour chaque Forme forme de sélection
9    faire #modifier(forme.angleRotation,
10           forme.obtenirAngleRotation()+angleRotation)
11 }
```

Modifier un attribut des formes sélectionnées

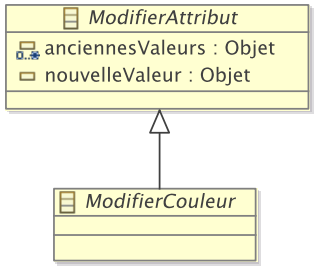



```
1  booléen peutExécuter() {
2    retourner super.peutExécuter() et nouvelleValeur≠null
3  }
```

Les actions modifiant un attribut de formes, se fondent sur l'action `ModifierAttribut`, définie ci-dessous, dont l'attribut `anciennesValeurs` est une sauvegarde des valeurs des attributs à modifier. L'attribut `nouvelleValeurs` correspond à la nouvelle valeur à affecter.

Action `ModifierCouleur`

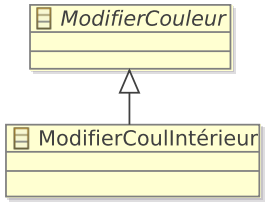
Cette action abstraite, qui hérite de la classe `ModifierAttribut`, regroupe les éléments communs des actions visant à modifier la couleur de certaines parties de formes. La fonction `peutExécuter` teste si la nouvelle valeur est une couleur et appelle la fonction `peutExécuter` de l'action `ModifierAttribut`. La méthode `réexécuter` appelle la méthode `modifierCouleur`. Celle-ci est définie par les actions qui héritent de `ModifierCouleur` et a pour but de remplacer la couleur d'une partie d'une forme par celle donnée en paramètre.




```
1  booléen peutExécuter() {
2    retourner super.peutExécuter et
3              nouvelleValeur est Couleur
4  }
5  réexécuter() {
6    Couleur c = (Couleur) nouvelleValeur
7    pour chaque Forme forme de sel.sélection
8    faire modifierCouleur(forme, c)
9  }
10 abstraite modifierCouleur(Forme f, Couleur c)
```

Modifier la couleur intérieure de formes

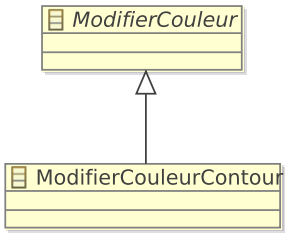
La modification de la couleur de l'intérieur de formes s'effectue en deux étapes au travers de la méthode `exécuter` (ligne 1). La première sauvegarde les couleurs de formes sélectionnées dans l'ensemble `anciennesValeurs`. La seconde est réalisée dans la méthode `modifierCouleur` (ligne 14) qui assigne les composantes d'une couleur à celles d'une forme donnée par le biais de la méthode `#modifier`. La méthode `annuler` (ligne 7) réassigne les anciennes couleurs aux formes correspondantes.




```
1  exécuter() {
2    Couleur c = (Couleur) nouvelleValeur
3    pour chaque Forme forme de sel.sélection faire
4      anciennesValeurs.ajouter(forme.couleurFond)
5      modifierCouleur(forme, c)
6  }
7  annuler() {
8    Entier i = 1
9    pour chaque Forme forme de sel.sélection faire
10     modifierCouleur(forme.couleurFond,
11               (Couleur)anciennesValeurs[i])
12     i = i+1
13 }
14 modifierCouleur(Forme f, Couleur c) {
15   #modifier(forme.couleurFond.r, r)
16   #modifier(forme.couleurFond.g, g)
17   #modifier(forme.couleurFond.b, b)
18 }
```

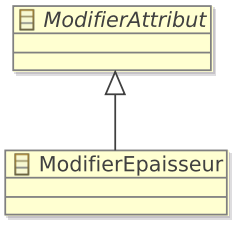
Modifier la couleur du contour de formes

La méthode `exécuter`, ligne 1, sauvegarde les anciennes couleurs de contour des formes puis appelle la méthode `modifierCouleur`. Cette dernière, ligne 14, modifie l'attribut `couleurContour` de la classe `Forme` en fonction des trois composantes de la nouvelle couleur. La méthode `annuler`, ligne 7, annule les modifications en réattribuant les anciennes couleurs au formes.




```
1  exécuter() {
2    Couleur c = (Couleur) nouvelleValeur
3    pour chaque Forme forme de sel.sélection faire
4      anciennesValeurs.ajouter(forme.couleurContour)
5    modifierCouleur(forme, c)
6  }
7  annuler() {
8    Entier i = 1
9    pour chaque Forme forme de sel.sélection faire
10     modifierCouleur(forme.couleurContour,
11                     (Couleur)anciennesValeurs[i])
12     i = i+1
13 }
14 modifierCouleur(Forme f, Couleur c) {
15   #modifier(forme.couleurContour.r, r)
16   #modifier(forme.couleurContour.g, g)
17   #modifier(forme.couleurContour.b, b)
18 }
```

Modifier l'épaisseur de formes



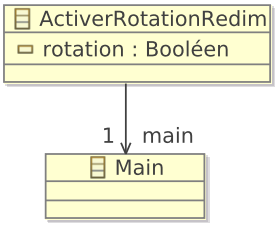

```
1  booléen peutExécuter() {
2    retourner super.peutExécuter et
3                nouvelleValeur est Double
4  }
5  exécuter() {
6    pour chaque Forme forme de sel.sélection faire
7      anciennesValeurs.ajouter(forme.epaisseur)
8    modifierEpaisseur()
9  }
10 annuler() {
11   Entier i = 1
12   pour chaque Forme forme de sel.sélection faire
13     #modifier(forme.épaisseur, anciennesValeurs[i])
14     i++
15 }
16 réexécuter() {
17   modifierEpaisseur()
18 }
19 modifierEpaisseur() {
20   Double ep = (Double)nouvelleValeur
21   pour chaque Forme forme de sel.sélection faire
22     #modifier(forme.épaisseur, ep)
23 }
```

Le changement de l'épaisseur de la bordure de formes suit le même processus que celui de l'action précédente. La méthode `exécuter` (ligne 5) sauvegarde les anciennes épaisseurs dans

l'ensemble `anciennesValeurs`, puis appelle la méthode `modifierEpaisseur` (ligne 19) qui affecte la nouvelle épaisseur aux formes. La méthode `annuler` (ligne 10) réaffecte les anciennes épaisseurs aux formes. La fonction `peutExécuter` (ligne 1) vérifie que le type de la nouvelle valeur correspond bien au type d'une épaisseur, en l'occurrence au type `Double`.

Action `ModifierRotationRedim`

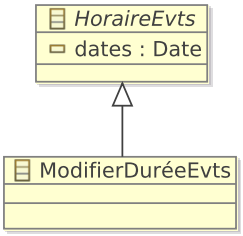
Le but de cette action est de définir si les instruments doivent pivoter ou redimensionner les formes sélectionnées. La classe `ActiverRotationRedim` possède un attribut `rotation` prenant la valeur « vrai » lorsque la rotation est considérée. Elle connaît également l'instrument `Main` qui gère la rotation et le redimensionnement. La méthode `exécuter` appelle la méthode `enRotation` de l'instrument `Main` pour définir si la rotation ou le redimensionnement est utilisé.




```
1  booléen peutExécuter() {
2    main!=null
3  }
4  exécuter() {
5    main.enRotation(rotation)
6  }
```

Modifier la durée d'évènements

L'action `ModifierDuréeEvts` hérite de l'action `HoraireEvts`. Par conséquent, `ModifierDuréeEvts` ne définit que la méthode `gérerHorairesEvts` dans le but de modifier la durée des évènements sélectionnés. Ainsi, pour chaque évènement sélectionné, une nouvelle date est spécifiée (ligne 5), tandis que l'ancienne est sauvegardée dans l'attribut `dates` (ligne 6).



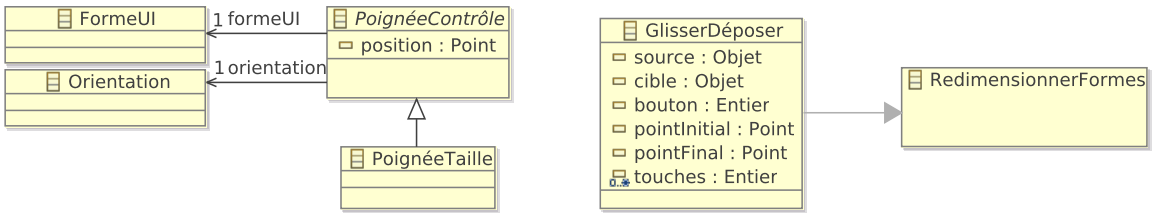

```
1  gérerHorairesEvts() {
2    pour i de 1 à |sélection| faire
3      Evènement evt = sélection[i]
4      Date durée    = evt.durée
5    #modifier(evt.durée, dates[i])
6      dates[i] = durée
7  }
```

C.4 Instruments

Cette section contient la définition des instruments triviaux ou non essentiels des études de cas des chapitres 9 et 10.

Poignées de redimensionnement

Comme le décrit le diagramme de classes de la figure C.5a, une poignée de redimensionnement (`PoignéeTaille`) est une poignée de contrôle intégrée dans la présentation concrète. La figure C.5b définit qu'une action `RedimensionnerFormes` est créée *via* un glisser-déposer. La condition de cette liaison, définie dans le pseudo-code ci-dessous, stipule que l'objet source de l'interaction doit correspondre à l'instrument (ligne 3).



(a) Diagramme de classes (b) Liaison entre l'action et l'interaction

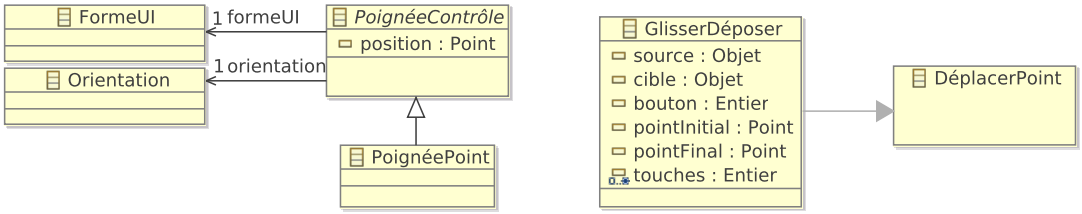
Fig. C.5: Description statique de la poignée de redimensionnement

Le feed-back intérimaire de cet instrument se rapporte à un changement de curseur lorsque l'action en cours est une action `RedimensionnerFormes` (lignes 4 et 7).

```
1 Liaisons PoignéeTaille pt {
2   GlisserDéposer gd -> RedimensionnerFormes {
3     condition : gd.source==pt
4     feedback  : remplacer curseur courant par curseur redimensionnement
5   }
6   défaut {
7     remplacer curseur courant par curseur par défaut
8   }
9 }
```

Poignées de déplacement de points

Les poignées de contrôle dédiées au déplacement des points des polygones fonctionnent de la même manière que les autres poignées. Le diagramme de classes de la figure C.6a définit qu'une poignée de déplacement de points est une poignée de contrôle intégrée dans la présentation concrète. La figure C.6b définit qu'une action `DéplacerPoint` est créée *via* un glisser-déposer. La condition de cette liaison, décrite dans le pseudo-code ci-dessous, stipule que l'objet source doit correspondre à l'instrument (ligne 3).



(a) Diagramme de classes (b) Liaison entre l'action et l'interaction

Fig. C.6: Description statique de la poignée de déplacement de points

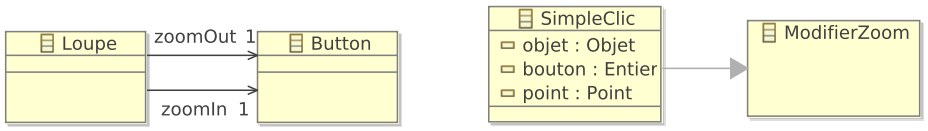
Le feed-back intérimaire de cet instrument est semblable à celui des poignées de redimensionnement. Il se rapporte à un changement de curseur lorsque l'action en cours est une action `DéplacerPoint` (ligne 4 et 7).

```
1 Liaisons PoignéeDéplacement pd {
```

```
2   GlisserDéposer gd -> DéplacerPoint {
3     condition : gd.source==pd
4     feedback  : remplacer curseur courant par curseur déplacement
5   }
6   défaut {
7     remplacer curseur courant par curseur par défaut
8   }
9 }
```

Loupe

La loupe est un instrument permettant de zoomer en avant ou en arrière sur les formes que présente l'éditeur. Comme le montre le diagramme de classes de la figure C.7a, cet instrument possède deux composants graphiques : un bouton pour zoomer en avant et un autre pour zoomer en arrière.



(a) Diagramme de classes (b) Liaison entre l'interaction et l'action

Fig. C.7: Description statique du zoom

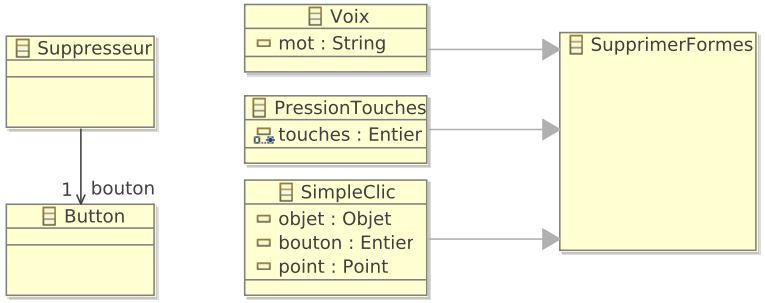
La figure C.7b définit qu'une interaction `SimpleClic` crée une action `ModifierZoom`. La condition de cette liaison, décrite dans la pseudo-code suivant, stipule que l'objet du clic doit être soit le bouton `zoomIn`, soit le `zoomOut`. Cet exemple d'instrument illustre le fait qu'un instrument peut posséder plusieurs composants graphiques avec lesquels l'utilisateur interagit.

```
1 Liaisons Loupe loupe {
2   SimpleClic sc -> ModifierZoom {
3     condition : sc.objet==loupe.zoomIn || sc.objet==loupe.zoomOut
4   }
5 }
```

Suppresseur

L'instrument `Suppresseur`, est destiné à la suppression de formes *via* l'utilisation de interactions présentées dans la figure C.8b. Les liaisons précisées dans cette figure définissent que les interactions `Voix`, `PressionTouches` et `SimpleClic` créent une action `SupprimerFormes`. Les conditions de ces liaisons sont décrites dans le pseudo-code ci-dessous. La condition de la première de ces liaisons stipule que le mot prononcé doit être le mot « supprimer » pour que les formes soient supprimées (ligne 3). La condition de la deuxième liaison définit que la pression de la touche « Supp. » crée une action `SupprimerFormes` (ligne 6). La condition de la dernière liaison (ligne 9) stipule que l'objet du simple clic doit être le bouton `bouton` (*cf.* figure C.8a).

```
1 Liaisons Suppresseur sup {
2   Voix v -> SupprimerFormes {
```



(a) Diagramme de classes

(b) Liaisons entre les interactions et l'action

Fig. C.8: Description statique de l'instrument `Suppresseur`

```
3       condition : v.mot=="supprimer"
4     }
5     PressionTouches pt -> SupprimerFormes {
6       condition : |pt.touches|==1 && pt.touches[1]=='Supp.'
7     }
8     SimpleClic sc -> SupprimerFormes {
9       condition : sc.objet==sup.bouton
10    }
11  }
```

Annexe D

Transformation de modèles pour le problème du blog

La transformation ATL pour le problème du blog fait intervenir plusieurs transformations. Le document XML doit tout d'abord être transformé en un modèle (changement d'espace technique) ; cette étape n'est pas précisée dans ce manuscrit étant donné que cet exemple utilise directement un modèle représentant un blog. La transformation d'un modèle de blog en un modèle SVG est ensuite réalisée à l'aide de la transformation ATL décrite dans la figure 2.5, page 25. Le code suivant écrit en KM3, langage dédié à la description de métamodèles [Jouault, 2006], décrit le métamodèle du blog.

```
1  package blog {
2    class Blog {
3      attribute nom : String;
4      reference billets[*] container : Billet;
5    }
6    class Billet {
7      attribute num     : Integer;
8      attribute titre   : String;
9      attribute contenu : String;
10     attribute date    : String;
11     attribute auteur  : String;
12   }
13 }
14 package PrimitiveTypes {
15   datatype Boolean;
16   datatype Integer;
17   datatype String;
18   datatype Double;
19 }
```

Le code KM3 suivant décrit une partie du métamodèle SVG.

```
1  package SVG {
2    abstract class Element {
3      reference owner[*] : SvgFile oppositeOf elements;
4      reference target[*] : Use oppositeOf use;
5      reference "attribute"[*] : Attribute oppositeOf attOwner;
6      reference position[0-1] container : Coordinates;
7      reference size[0-1] container : Dimension;
8      reference root[0-1] : Svg oppositeOf children;
9      attribute fill[0-1] : String;
10     attribute viewBox[0-1] : String;
```

```
11      reference group[0-1] : GroupingElement oppositeOf groupContent;
12      attribute identifier[0-1] : String;
13      reference drawsMarker[0-1] : Marker oppositeOf drawing;
14    }
15
16    abstract class StructuralElement extends Element { }
17    abstract class Transform extends Attribute { }
18    class Defs extends GroupingElement { }
19    class Symbol extends GroupingElement { }
20    abstract class Shape extends GraphicalElement { }
21    class Circle extends Shape { }
22    class Ellipse extends Shape { }
23    class Point extends Shape { }
24    class RelativeCoord extends Coordinates {}
25    class AbsoluteCoord extends Coordinates {}
26
27    class Image extends StructuralElement {
28      reference referee[*] : ReferencedFile oppositeOf referer;
29    }
30
31    class Svg extends StructuralElement {
32      reference owner_SVG[*] : SvgFile oppositeOf tag;
33      reference children[*] ordered container : Element oppositeOf root;
34      attribute namespace[0-1] : String;
35      attribute version[0-1] : String;
36      attribute baseProfile[0-1] : String;
37    }
38
39    abstract class GroupingElement extends StructuralElement {
40      reference groupContent[*] ordered container : Element oppositeOf group;
41    }
42
43    class G extends GroupingElement {
44      attribute name[0-1] : String;
45    }
46
47    class Use extends StructuralElement {
48      reference use[*] : Element oppositeOf target;
49    }
50
51    abstract class GraphicalElement extends Element {
52      attribute stroke[0-1] : String;
53    }
54
55    abstract class TextElement extends GraphicalElement {
56      attribute rotate[0-1] : Double;
57      attribute textLength[0-1] : String;
58      attribute fontSize[0-1] : String;
59    }
60
61    class Rect extends Shape {
62      attribute rx[0-1] : Double;
63      attribute ry[0-1] : Double;
64    }
65
66    class Line extends Shape {
67      reference between[2-2] : Point;
68      attribute markerEnd[0-1] : String;
69      attribute markerStart[0-1] : String;
70    }
71
72    class Polyline extends Shape {
73      reference waypoints[*] ordered container : Point;
74      attribute strokeDashArray[0-1] : String;
```

```
75      attribute markerEnd[0-1] : String;
76      attribute markerStart[0-1] : String;
77    }
78
79    class Polygon extends Shape {
80      reference waypoints[*] ordered : Point;
81      attribute markerEnd[0-1] : String;
82      attribute markerStart[0-1] : String;
83    }
84
85    class Path extends Shape {
86      attribute pathLength[0-1] : Double;
87      attribute d : String;
88      attribute markerEnd[0-1] : String;
89      attribute markerStart[0-1] : String;
90    }
91
92    class Marker extends Shape {
93      attribute markerUnits[0-1] : String;
94      attribute refX[0-1] : Double;
95      attribute refY[0-1] : Double;
96      attribute markerWidth[0-1] : Double;
97      attribute markerHeight[0-1] : Double;
98      attribute orient[0-1] : String;
99      reference drawing[*] container : Element oppositeOf drawsMarker;
100   }
101
102   class Text extends TextElement {
103     attribute lengthAdjust[0-1] : String;
104     attribute content : String;
105   }
106
107   class Tspan extends TextElement {
108     attribute content[0-1] : String;
109   }
110
111   class Tref extends TextElement {
112     reference xlinkHref : TextElement;
113   }
114
115   abstract class Attribute {
116     reference attOwner[*] : Element oppositeOf "attribute";
117   }
118
119   class Scale extends Transform {
120     attribute sx : Double;
121     attribute sy : Double;
122   }
123
124   class Translate extends Transform {
125     attribute tx : Double;
126     attribute ty : Double;
127   }
128
129   class Rotate extends Transform {
130     attribute angle : Double;
131     attribute cx : Double;
132     attribute cy : Double;
133   }
134
135   class Visibility extends Attribute {
136     attribute visible : Boolean;
137   }
138
```

```
139   class FontWeight extends Attribute {
140     attribute bold : Boolean;
141   }
142
143   class FontStyle extends Attribute {
144     attribute italic : Boolean;
145   }
146
147   class Dimension {
148     attribute width : Double;
149     attribute height : Double;
150   }
151
152   abstract class Coordinates {
153     attribute x : Double;
154     attribute y : Double;
155   }
156
157   abstract class ReferencedFile {
158     reference referer[*] : Image oppositeOf referee;
159     attribute name : String;
160   }
161
162   class SvgFile extends ReferencedFile {
163     reference tag : Svg oppositeOf owner_SVG;
164     reference elements[*] : Element oppositeOf owner;
165   }
166 }
```

Une fois la transformation d'un modèle de blog vers un modèle SVG effectuée, il est nécessaire de passer de l'espace technique MDA à l'espace technique XML pour obtenir le document SVG cible, en utilisant la transformation ATL suivante.

```
1  module SVG2XML;
2  create OUT : XML from IN : SVG;
3
4  helper context Real
5  def : notNull() : Boolean = self <> 0 and self <> 0.0;
6
7  helper context SVG!Scale
8  def : scale() : String =
9    if (self.sx = 1 or self.sx = 1.0)
10     then ''
11     else 'scale(' + self.sx.toString() +
12       if self.sy = self.sx
13       then ''
14       else ',' + self.sy.toString()
15       endif + ')'
16     endif;
17
18 helper context SVG!Translate
19 def : translate() : String =
20   if self.tx.notNull() or self.ty.notNull()
21     then 'translate(' + self.tx.toString() + ',' + self.ty.toString() + ')'
22     else ''
23   endif;
24
25 helper context SVG!Rotate
26 def : rotate() : String =
27   if self.angle.notNull()
28     then 'rotate(' + self.angle.toString() + ')'
29     else ''
30   endif;
```

```
31
32 rule Svg {
33   from
34     svg : SVG!Svg
35   to
36     root : XML!Root (
37       name <- 'svg',
38       children <- xmlns,
39       children <- version,
40       children <- thisModule.Attribute('width',
41         if not svg.size.oclIsUndefined()
42           then svg.size.width.toString()
43           else '100%'
44         endif),
45       children <- thisModule.Attribute('height',
46         if not svg.size.oclIsUndefined()
47           then svg.size.height.toString()
48           else '100%'
49         endif),
50       children <- svg.children
51     ),
52     xmlns : XML!Attribute (
53       name <- 'xmlns',
54       value <- svg.namespace ),
55     version : XML!Attribute (
56       name <- 'version',
57       value <- svg.version )
58   do {
59     if(not svg.viewBox.oclIsUndefined()) {
60       root.children <- thisModule.Attribute('viewBox', svg.viewBox);
61     }
62   }
63 }
64
65 rule G {
66   from
67     g : SVG!G
68   using {
69     transforms : Sequence(SVG!Transform) =
70       g.attribute->select(a|a.oclIsKindOf(SVG!Transform));
71     transformValue : String =
72       transforms->iterate(transf; str : String = ''|
73         str +
74         if transf.oclIsTypeOf(SVG!Scale)
75           then transf.scale()
76           else if transf.oclIsTypeOf(SVG!Translate)
77             then transf.translate()
78             else if transf.oclIsTypeOf(SVG!Rotate)
79               then transf.rotate()
80               else ''
81             endif
82           endif
83         endif +
84         if (transf <> transforms->last())
85           then ' '
86           else ''
87         endif);
88   }
89   to
90     elmt : XML!Element (
91       name <- 'g',
92       children <- g.groupContent )
93   do {
94     if (not g.fill.oclIsUndefined() and g.fill <> 'black') {
```

```
95        elmt.children <- thisModule.Attribute('fill', g.fill);
96      }
97      if (transforms->notEmpty() and transformValue <> '') {
98        elmt.children <- thisModule.Attribute('transform', transformValue);
99      }
100   }
101 }
102
103 rule Rect {
104   from
105     rect : SVG!Rect
106   to
107     elmt : XML!Element (
108       name <- 'rect',
109       children <- thisModule.Attribute('width',
110         if not rect.size.oclIsUndefined()
111           then rect.size.width.toString()
112           else '100%'
113         endif),
114       children <- thisModule.Attribute('height',
115         if not rect.size.oclIsUndefined()
116           then rect.size.height.toString()
117           else '100%'
118         endif)
119     )
120   do {
121     if (not rect.position.oclIsUndefined() and rect.position.x.notNull()) {
122       elmt.children <- thisModule.Attribute('x', rect.position.x.toString());
123     }
124     if (not rect.position.oclIsUndefined() and rect.position.y.notNull()) {
125       elmt.children <- thisModule.Attribute('y', rect.position.y.toString());
126     }
127     if (not rect.fill.oclIsUndefined() and rect.fill <> 'black') {
128       elmt.children <- thisModule.Attribute('fill', rect.fill);
129     }
130     if (not rect.stroke.oclIsUndefined() and rect.stroke <> 'none') {
131       elmt.children <- thisModule.Attribute('stroke', rect.stroke);
132     }
133   }
134 }
135
136 rule Circle {
137   from
138     circ : SVG!Circle
139   to
140     elmt : XML!Element (
141       name <- 'circle',
142       children <- thisModule.Attribute('r',
143         if not circ.size.oclIsUndefined()
144           then circ.size.width.toString()
145           else '0'
146         endif)
147     )
148   do {
149     if(not circ.position.oclIsUndefined() and circ.position.x.notNull()) {
150       elmt.children <- thisModule.Attribute('x', circ.position.x.toString());
151     }
152     if(not circ.position.oclIsUndefined() and circ.position.y.notNull()) {
153       elmt.children <- thisModule.Attribute('y', circ.position.y.toString());
154     }
155     if(not circ.fill.oclIsUndefined() and circ.fill <> 'black') {
156       elmt.children <- thisModule.Attribute('fill', circ.fill);
157     }
158     if(not circ.stroke.oclIsUndefined() and circ.stroke <> 'none') {
```

```
159         elmt.children <- thisModule.Attribute('stroke', circ.stroke);
160       }
161     }
162   }
163
164   rule Path {
165     from
166       path : SVG!Path
167     to
168       elmt : XML!Element (
169         name <- 'path',
170         children <- thisModule.Attribute('d', path.d)
171       )
172     do {
173       if(not path.fill.oclIsUndefined() and path.fill <> 'black') {
174         elmt.children <- thisModule.Attribute('fill', path.fill);
175       }
176       if(not path.stroke.oclIsUndefined() and path.stroke <> 'none') {
177         elmt.children <- thisModule.Attribute('stroke', path.stroke);
178       }
179     }
180   }
181
182   rule Text {
183     from
184       text : SVG!Text
185     to
186       elmt : XML!Element (
187         name <- 'text',
188         children <- txt ),
189       txt : XML!Text ( value <- text.content )
190     do {
191       if(not text.position.oclIsUndefined() and text.position.x.notNull()) {
192         elmt.children <- thisModule.Attribute('x', text.position.x.toString());
193       }
194       if(not text.position.oclIsUndefined() and text.position.y.notNull()) {
195         elmt.children <- thisModule.Attribute('y', text.position.y.toString());
196       }
197       if(not text.stroke.oclIsUndefined() and text.stroke <> 'none') {
198         elmt.children <- thisModule.Attribute('stroke', text.stroke);
199       }
200       if(not text.fontSize.oclIsUndefined() and text.fontSize <> 'medium') {
201         elmt.children <- thisModule.Attribute('font-size', text.fontSize);
202       }
203       if(not text.lengthAdjust.oclIsUndefined() and text.lengthAdjust <> 'start') {
204         elmt.children <- thisModule.Attribute('text-anchor', text.lengthAdjust);
205       }
206     }
207   }
208
209   rule Attribute(attrName : String, attrValue : String) {
210     to
211       attr : XML!Attribute (
212         name <- attrName,
213         value <- attrValue )
214     do { attr; }
215   }
```

Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données

Résumé

L'ingénierie du logiciel s'intéresse, entre autres, à trois aspects du développement des systèmes interactifs (SI) : la liaison entre les données sources et leurs présentations cibles ; la conception de la facette interactive ; l'exécution d'un même SI sur différentes plates-formes d'exécution. Les différentes évolutions du web et des données, la diversification des plates-formes d'exécution, ainsi que les techniques d'interactions modernes amènent à revoir la manière de traiter ces trois aspects. L'ingénierie des modèles (IDM) apporte une solution à l'exécution multi-plateforme en échelonnant la conception d'un SI sur différents niveaux d'abstraction. C'est sur ce principe que nos travaux se fondent. Nous avons tout d'abord défini un langage de correspondance, appelé Malan, dédié au lien entre les données sources d'un SI et leurs présentations cibles. Malan a pour avantages de : s'abstraire des plate-forme de données et d'IHM utilisées ; pouvoir réaliser des calculs complexes pour la disposition des éléments d'une présentation ; avoir été développé spécialement pour la liaison données-présentations, contrairement aux langages de transformations classiques. Nous avons ensuite conçu un modèle conceptuel d'interaction, appelé Malai, réunissant les caractéristiques majeures des principaux modèles d'interactions. Malai vise à : faciliter la conception du feed-back intérimaire des instruments ; réutiliser des interactions déjà définies ; considérer une action comme un objet à part entière ; décrire des interactions classiques et modernes ; générer du code pour une plate-forme d'exécution donnée.

Mots-clés : système interactif, correspondance de schémas, interaction, action, instrument, interface, IDM, Malan, Malai

A model for the conception of interactive systems manipulating data

Abstract

Software engineering focuses, amongst others, on three development aspects of interactive systems (IS): the link between source data and their target presentations; the conception of the interactive facet; the execution of the same IS on different execution platforms. The evolutions of the web and data, the diversification of execution platforms, and the modern interaction techniques requires a review of the processing of these three aspects. Model-driven engineering (MDE) proposes a solution to the multi-platform execution aspect by grading the conception of IS using different abstraction levels. Our work is based on this principle. Firstly, we defined a mapping language called Malan, dedicated to the link between source data and their target presentations. Malan has the advantages of being: independent of data and HCI platforms ; able to carry out complex calculations for the layout of a target presentation; developed in the data-to-presentation context, contrary to classical transformation languages. Secondly, we developed a conceptual interaction model called Malai, gathering together principles from the major interaction models. Malai aims to: facilitate the conception of the interim feedback of instruments; reuse interactions already defined ; consider an action as a first-class object; describe modern and classical interactions ; generate code for a given execution platform.

Keywords : interactive system, schema mapping, interaction, action, instrument, interface, MDE, Malan, Malai