Automatisation de la génération du modèle de traduction dans le cadre action-événement

par

Pierre-Yves Rozon

essai présenté au Centre de formation en technologies de l'information
en vue de l'obtention du grade de maître en génie logiciel
(maîtrise en génie logiciel incluant un cheminement de type cours en génie logiciel)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Longueuil, Québec, Canada, juin 2012

Sommaire

Le cadre action-événement est un outil de test d'interface graphique visant à remplacer les logiciels de capture et réexécution (*capture and playback*). Par l'emploi d'un modèle de traduction écrit par un programmeur, son adoption permet de minimiser la maintenance requise suite à un changement dans l'apparence de l'interface graphique du logiciel testé. Ce modèle centralise les informations relatives à l'interface graphique et indique comment l'utiliser pour exécuter chacune des actions du logiciel. Cet essai a pour but de créer un algorithme générant automatiquement le modèle de traduction. Ainsi, si l'apparence de l'interface graphique change, celui-ci n'aura qu'à être régénéré, éliminant ainsi complètement la maintenance nécessaire.

Le premier intrant du cadre action-événement est le modèle d'action, qui est une représentation de la spécification des exigences exprimée dans le langage informatique Spec#. Cet essai stipule qu'un autre format que le format textuel doit être employé afin de faciliter une analyse informatique. Celui-ci est un métamodèle orienté objet respectant la hiérarchisation des éléments d'un programme écrit selon ce paradigme : une classe contient des variables et des méthodes. Ces dernières contiennent des instructions. Ces instructions peuvent être de plusieurs types, notamment une expression symbolique, un appel de méthode, une instruction conditionnelle, une instruction répétitive ou une instruction de retour. Le deuxième intrant du cadre action-événement est un modèle de l'interface graphique décrivant celle-ci. Cet essai définit le format précis qu'il doit respecter. Celui-ci est un métamodèle orienté objet permettant d'identifier à quel type appartient chacun des composants de l'interface (bouton, zone de texte, liste déroulante, grille, etc.) et d'exprimer les liens hiérarchiques entre les composants (certains sont contenus à l'intérieur d'autres composants).

La génération automatisée du modèle de traduction se fait à partir d'une analyse des deux modèles en entrée ainsi que du code source du programme. Celui-ci doit d'abord être

transformé dans un format manipulable par un outil informatique que l'on appelle le modèle du programme. Le métamodèle retenu pour ce modèle est le même que celui du modèle d'action. L'algorithme peut difficilement travailler de façon directe sur les modèles en entrée. Afin de faciliter le travail, des structures de données intermédiaires doivent être utilisées. Ces structures sont des tables permettant d'associer des éléments communs entre deux modèles (table de correspondance action/méthode, table de correspondance classe/fenêtre et table de correspondance symbole/composant graphique) ou encore des structures faisant ressortir les éléments essentiels du modèle du programme (table des séquences d'instructions, graphe d'appels de méthodes, table des chemins d'appel d'action et table paramètres/expressions). L'algorithme principal analyse ces structures de données intermédiaires afin de générer le modèle de traduction. Il parcourt les chemins pertinents trouvés dans le graphe d'appels de méthodes et simule les actions qu'aurait entreprises l'utilisateur en s'aidant, notamment, d'une table indiquant le composant graphique source de chacun des paramètres.

En s'appuyant sur une liste de critères, l'algorithme a été implanté et testé à partir de trois études de cas. Malgré certaines limites, l'algorithme correspond très bien aux critères. La première limite demande que certaines contraintes soient imposées sur le code source du logiciel testé (avoir une méthode possédant le même nom que chacune des actions, par exemple). La deuxième limite est que l'algorithme ne gère pas les codes source comprenant des méthodes directement ou indirectement récursives ainsi que les appels de méthodes virtuelles. En ce qui concerne la troisième limite, l'algorithme omet de générer les instructions qui ouvrent les fenêtres appropriées lorsque l'action ne peut être directement déclenchée à partir de la fenêtre principale. Finalement, le code généré est, dans bien des cas, moins optimal que ce qu'aurait pu écrire un programmeur.

Remerciements

Je tiens tout d'abord à remercier mon directeur professionnel, monsieur Alexandre Moïse, ainsi que mon directeur académique, monsieur Évariste Valéry Bévo Wandji, pour m'avoir accompagné tout au long de la rédaction de cet essai. Leurs commentaires et suggestions ont été des plus précieux. Leur demande de rigueur et de précision ont contribué à renforcer le contenu de cet essai. Les suggestions de lectures connexes qu'ils m'ont faites ont été des plus enrichissantes.

Finalement, je remercie ma famille et mes amis pour m'avoir soutenu tout au long de mes études de maîtrise. Ma double vie de professionnel et d'étudiant n'aurait pu être possible sans leur compréhension de mon manque de disponibilité.

Table des matières

Liste des tableaux

Liste des figures

Glossaire

Application de bureau	Logiciel qui affiche son interface graphique dans un environnement graphique avec fenêtres
Chemin	Dans un graphe, suite de sommets deux à deux adjacents
Chemin élémentaire	Chemin dans lequel chacun des sommets ne se trouve qu'une seule fois
Constante littérale	Constante représentée par sa valeur
Constante symbolique	Constante représentée par un symbole plutôt que sa valeur réelle
Fenêtre modale	Fenêtre qui, tant qu'elle est ouverte, bloque l'accès aux autres fenêtres
Gestionnaire d'événement	Fonction ou méthode qui est exécutée lorsqu'un événement survient
Graphe	Ensemble de points dont certaines paires sont directement reliées par un (ou plusieurs) lien(s)
Méthode virtuelle	Méthode définie dans une classe et qui est destinée à être redéfinie dans les classes dérivées
Paramètre effectif	Valeur fournie pour un paramètre lors de l'appel d'une méthode
Paramètre formel	Variable représentant le paramètre à l'intérieur du corps d'une méthode
Pile	Structure de données dans laquelle les éléments sont retirés dans l'ordre inverse où ils ont été insérés
Programmation par contrat	Paradigme de programmation dans lequel le déroulement des traitements est régi par des règles
Puits	Dans un graphe orienté, sommet qui ne possède pas d'arc sortant
Scénario de test	Instructions détaillées qui décrivent les données de base, les procédures et les résultats prévus d'un test en particulier
Test	Opération destinée à valider le bon fonctionnement d'un logiciel

Introduction

Les interfaces graphiques occupent une place très importante dans la façon dont les utilisateurs interagissent avec les systèmes informatiques. Puisqu'il s'agit d'un moyen d'interaction entre l'utilisateur et le logiciel, il est important que l'interface ne contienne pas de faute[1], sans quoi l'utilisateur ne sera pas en mesure d'utiliser correctement le logiciel. Pour cette raison, il est important de tester l'interface graphique rigoureusement ([7]).

Le domaine du test logiciel a fait beaucoup de progrès au cours des dernières années, notamment avec l'arrivée de l'approche de développement piloté par les tests et de l'automatisation des tests unitaires. Cela a apporté de nombreux bénéfices, tels qu'une augmentation de la productivité des testeurs, une réduction des coûts de maintenance et une augmentation de l'efficacité des tests ([9]). Cependant, tester une interface graphique représente un défi différent des tests unitaires, et nécessite ses propres méthodes et outils.

La technique de test d'interface graphique la plus répandue est l'utilisation d'outils de capture et réexécution (*capture and playback*) ([7]). Ces outils enregistrent, lors de l'exécution d'une application fonctionnelle, toutes les actions de l'utilisateur et également le comportement de l'interface. Il est alors possible de rejouer la séquence, suite à la modification de l'application, et de vérifier automatiquement que le comportement observé de la part de l'interface graphique est le même. Cette technique, bien qu'elle permette l'automatisation de l'exécution des tests d'interface graphique, possède trois inconvénients majeurs. Premièrement, la génération des scénarios de test n'est pas automatisée. Deuxièmement, un changement purement visuel dans l'interface graphique entraîne bien souvent l'échec de tests qui devraient réussir et les scénarios de test doivent être réécrits pour refléter les changements dans

[1] Une faute dans l'interface graphique est tout affichage erroné ou comportement inattendu de la part de celle-ci. Par exemple, une donnée qui ne s'affiche pas dans une zone de texte, un bouton qui reste invisible alors qu'il devrait apparaître ou une grille n'ayant pas le bon nombre de lignes sont des fautes.

l'apparence de l'interface graphique, ce qui entraîne des coûts de maintenance énormes. Finalement, cette technique ne s'intègre pas dans une approche de développement piloté par les tests, car le programme doit être terminé avant de pouvoir écrire les scénarios de test.

Plusieurs travaux de recherche ont été menés afin de trouver des techniques de test d'interface graphique pour remplacer la capture et réexécution. Parmi ceux-ci, Duc Hoai Nguyen, Paul Strooper et Jörn Guy Süß ont proposé le cadre action-événement (*action-event framework*) dont le but est la génération automatisée de scénarios de test d'interface graphique d'un logiciel et l'exécution automatisée de ceux-ci ([10] et [11]). En plus de conserver l'avantage de l'automatisation de l'exécution des tests que possèdent les outils de capture et réexécution, ce cadre solutionne leur premier problème, qui est la génération manuelle des scénarios de test. Il solutionne partiellement le deuxième problème (changements dans l'apparence de l'interface graphique) par l'emploi d'un modèle de traduction, écrit par une personne, qui centralise les informations relatives à l'apparence de l'interface graphique, ce qui minimise la maintenance requise suite à un changement. Par contre, il n'adresse pas le troisième problème (ne s'intègre pas dans une approche de développement piloté par les tests).

Le but de cet essai est de modifier le cadre action-événement afin de solutionner complètement le deuxième problème (changements dans l'apparence de l'interface graphique) par la génération automatisée du modèle de traduction. Ainsi, si l'apparence de l'interface graphique change, celui-ci n'aura qu'à être régénéré. La recherche conduite dans cet essai a été menée selon le principe de la « recherche par conception » ([6]). Cette méthodologie, utilisée dans les sciences de conception telles que l'ingénierie et l'informatique, vise la création d'artéfacts innovateurs qui seront par la suite évalués selon des critères de valeur ou d'utilité. L'apprentissage se fait par la construction de l'artéfact. Cette méthodologie se distingue de celle, plus traditionnelle, des sciences naturelles, telles que la physique et la biologie, qui vise à comprendre et expliquer la réalité en l'observant, pour en dégager une théorie.

Le premier chapitre de cet essai présente le cadre action-événement. Les chapitres deux et trois décrivent le modèle d'action et le modèle de l'interface graphique, deux intrants nécessaires au cadre action-événement. Le quatrième chapitre présente l'algorithme, développé dans le cadre de cet essai, qui génère automatiquement le modèle de traduction. Le cinquième chapitre présente la validation de l'algorithme.

Chapitre 1

Le cadre action-événement

Le cadre action-événement (*action-event framework*) a pour but la génération automatisée de scénarios de test d'interface graphique d'un logiciel et l'exécution automatisée de ceux-ci. Il se veut une technique de remplacement pour les outils de capture et réexécution. Le présent chapitre résume le cadre action-événement. Celui-ci est décrit en détails dans [10] et [11].

1.1 Activités du cadre action-événement

Les différentes activités du cadre action-événement sont présentées à la figure 1.1 et sont détaillées dans les sous-sections suivantes.

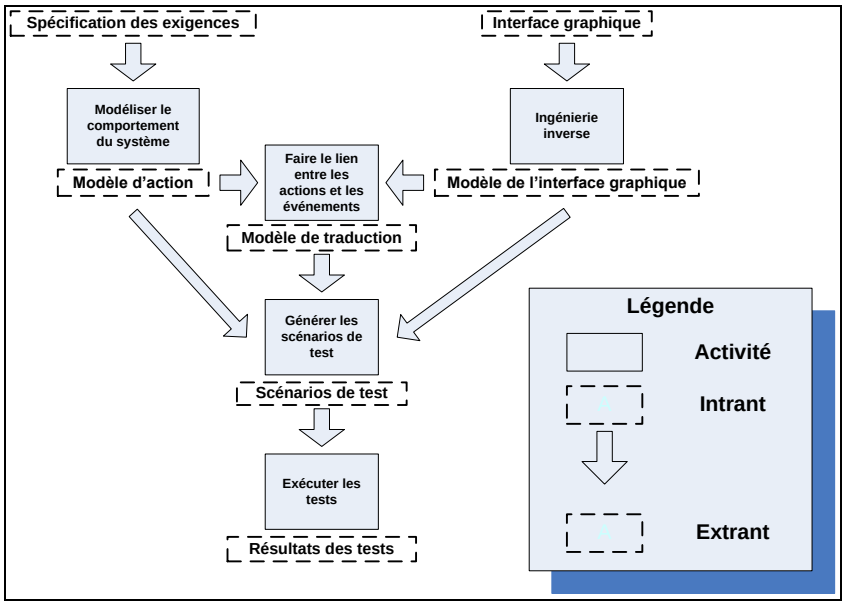


Figure 1.1 Cadre action-événement

Traduction libre
Inspiré de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 155

1.1.1 Modéliser le comportement du système

Cette activité consiste à traduire la spécification des exigences en un modèle d'action. Cette traduction se fait à la main, par une personne. La spécification des exigences est exprimée en langage naturel. Le modèle d'action en sortie est exprimé dans le langage de modélisation Spec#. Ce langage est une extension de C#, auquel ont été ajoutées des instructions permettant d'exprimer des éléments de spécification. Il fait partie de l'outil Spec Explorer, dont le but est « d'encoder le comportement attendu d'un système dans un format exécutable par une machine » ([8]). Spec Explorer se sert du modèle d'action d'un programme afin de générer des scénarios de test de la logique métier pour celui-ci. Le langage Spec# est décrit en détails dans [5]. La figure 1.3 montre un exemple de modèle d'action, écrit en Spec#, d'une application de gestion de tâches conçue à partir de la spécification des exigences de la figure 1.2. Le modèle d'action sera couvert plus en détails au chapitre 2.

L'application gère une liste de tâches.

Une tâche est définie par son nom et possède un état d'avancement, qui peut être :

- Non débutée (*New*)
- En cours (*Working*)
- Complétée (*Finished*)

L'application peut effectuer trois opérations :

- Ajouter une tâche
  - Crée une nouvelle tâche avec les informations spécifiées par l'utilisateur et l'ajoute dans la liste de tâches.
- Modifier une tâche
  - Modifie une tâche avec les informations spécifiées par l'utilisateur.
- Supprimer une tâche
  - Supprime une tâche de la liste de tâches.

Figure 1.2 Exemple de spécification des exigences

Inspiré de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 159

```
    // type declaration
    enum Progress {New, Finished, Working}
    class Task {
        string name;
        Progress progress;
    }

    // declare a ToDo list as a sequence of tasks
    type TodoList = Seq<Task>;
    // declare a ToDo list variable and initialize it
    TodoList todolist = Seq{};

    // create a new task. By default, the task name is empty.
    [Action]int newtask()
    {
        Task t=new Task("", Progress.New);
        todolist=todolist.Add(t);
        return todolist.Size;
    }
    // edit the ith task
    [Action]Task edittask(int i, string name, Progress progress)
    {
        todolist[i].name=name;
        todolist[i].progress=progress;
        return todolist[i];
    }
    // delete the ith task
    [Action]int deletetask(int i)
    {
        todolist=todolist.Take(i)+todolist.Drop(i+1);
        return todolist.Size;
}
```

Figure 1.3 Exemple de modèle d'action

Source : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 156

1.1.2 Ingénierie inverse de l'interface graphique

Cette activité consiste à construire un modèle abstrait de l'interface graphique à partir de l'interface elle-même, en procédant par ingénierie inverse. Cela peut se faire par une analyse statique ou dynamique de l'interface. Avec l'analyse statique, le code source de l'interface est analysé afin de construire le modèle. Il n'y a donc pas d'intervention humaine. Avec l'analyse dynamique, l'application est exécutée par une personne et un logiciel enregistre les composants contenus dans l'interface afin de construire le modèle. Le modèle de l'interface graphique en sortie est une liste de composants avec leurs événements et attributs. Ce modèle sera couvert plus en détails au chapitre 3.

1.1.3 Faire le lien entre les actions et les événements

Cette activité consiste à construire un modèle de traduction faisant la correspondance entre les actions définies dans le modèle d'action et les composants de l'interface définis dans le modèle de l'interface graphique. Il indique de quelle façon un utilisateur interagirait avec l'interface graphique pour effectuer chacune des actions. Cette opération se fait à la main, par une personne. Le modèle de traduction en sortie est exprimé dans le langage AEFMAP, développé dans le cadre des recherches conduites dans [10] et [11]. Un exemple de modèle de traduction pour l'application de gestion de tâches est présenté à la figure 1.4. Ce modèle sera couvert plus en détails au chapitre 4.

```
// create a new task either by clicking on the menu item or the toolbar
int newtask()  {
    Select {
        Execute(GUI.menuTODOADD.click);
        Execute(GUI.toolbarADD.click);
    }
    return GUI.tree.Size;
}

//edit a task by selecting the task, updating task information, then clicking on the Allow button.
Task edittask(int i, string name, Progress progress){
    Serialize{
        Execute (GUI.tree.select(i));
        Permute{
            ExecuteOp(GUI.textboxNAME.type, name);
            ExecuteOp(GUI.comboboxPROGRESS.select, progress);
        }
        Execute(GUI.buttonALLOW.click);
    }
    return new Task( GUI.tree.Node(i).Text,
                     GUI.comboboxPROGRESS.SelectedItem);
}

// delete a task by selecting the task, then clicking the menu item or the toolbar.
int deletetask(int i) {
    Serialize{
        ExecuteOp(GUI.tree.select(i));
        Select{
            Execute(GUI.menuTODODELETE.click);
            Execute(GUI.toolbarDELETE.click);
        }
    }
    return GUI.tree.Size;
}
```

Figure 1.4 Exemple de modèle de traduction

Source : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 157

1.1.4 Générer les scénarios de test

Cette activité consiste à générer des scénarios de test d'interface graphique à partir du modèle d'action, du modèle de l'interface graphique et du modèle de traduction (qui fait le pont entre les deux autres modèles). Parallèlement au modèle d'action, des scénarios de test d'action sont fournis. Les scénarios de test d'action sont générés automatiquement par l'outil Spec Explorer à partir du modèle d'action ([11]). À partir de cela, le générateur de scénarios de test d'interface graphique peut automatiquement traduire ces scénarios de test d'action en scénarios de test d'interface graphique grâce au modèle de traduction. Ce qui est généré en sortie est un script de test exécutable par le logiciel Quick Test Pro. Les auteurs du cadre action-événement ont choisi ce logiciel pour sa librairie complète de fonctions permettant d'accéder à l'interface graphique d'un logiciel et également pour son immense popularité chez les testeurs ([11]).

1.1.5 Exécuter les tests

Cette activité consiste à exécuter le script de test qui a été généré à l'étape précédente et à vérifier les résultats. Cette partie est entièrement automatisée par le logiciel Quick Test Pro.

1.2 Résumé du cadre action-événement et positionnement de l'essai

Le tableau 1.1 résume les activités du cadre action-événement. Il fait ressortir que seulement deux activités du cadre action-événement ne sont pas automatisées. Afin de faire un pas de plus vers une automatisation complète, cet essai automatisera la troisième activité, qui est de faire le lien entre les actions et les événements. La figure 1.5 illustre, à l'aide d'un rectangle de fond noir, où se situe le travail de cet essai dans le cadre action-événement. Les intrants de cette activité seront présentés aux chapitres 2 (modèle d'action) et 3 (modèle de l'interface graphique). L'algorithme qui génère le modèle de traduction sera présenté au chapitre 4 et validé au chapitre 5. Cet essai ne couvrira que les programmes écrits dans un langage orienté

objet et dont l'interface graphique est de type « application de bureau » (*desktop application*). De plus, il ne traitera pas de la façon de faire l'ingénierie inverse du code source afin d'obtenir le modèle d'action et le modèle de l'interface graphique.

Tableau 1.1 Résumé des activités du cadre action-événement

Activité	Entrée(s)	Sortie	Activité automatisée ou manuelle ?
Modéliser le comportement du système	Spécification des exigences (langage naturel)	Modèle d'action (Spec#)	Manuelle
Ingénierie inverse de l'interface graphique	Code source (analyse statique) ou application en cours d'exécution (analyse dynamique)	Modèle de l'interface graphique (liste de composants avec événements et attributs)	Automatisée (analyse statique) ou semi-automatisée (analyse dynamique)
Faire le lien entre les actions et les événements	Modèle d'action Modèle de l'interface graphique	Modèle de traduction (AEFMAP)	Manuelle
Générer les scénarios de test	Modèle d'action Scénarios de test d'action Modèle de l'interface graphique Modèle de traduction	Scénarios de test d'interface graphique (script Quick Test Pro)	Automatisée
Exécuter les tests	Scénarios de test d'interface graphique	Résultats des tests (affichage à l'écran ou sauvegarde dans un fichier)	Automatisée

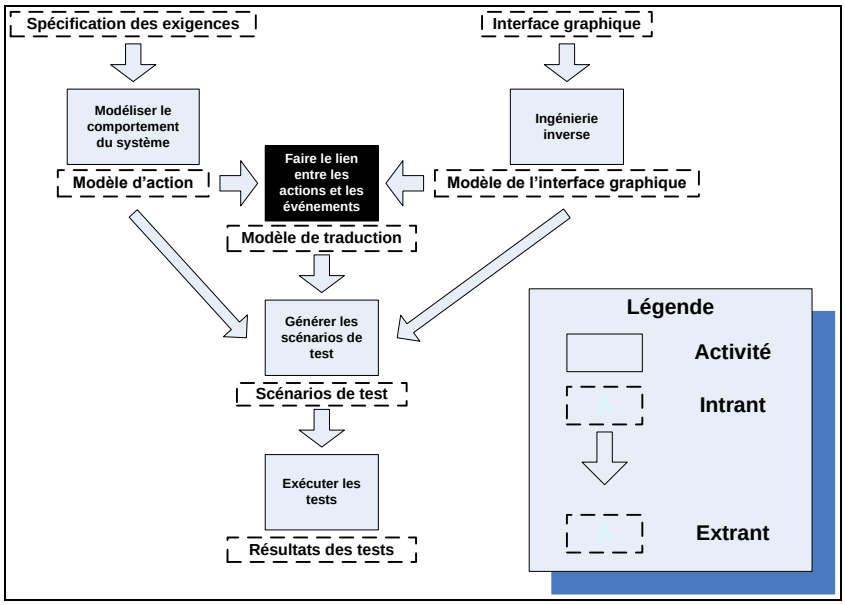


Figure 1.5 Positionnement de cet essai dans le cadre action-événement

Modifié de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 157

Chapitre 2

Modèle d'action

Ce chapitre couvre le modèle d'action, l'un des intrants de l'activité du cadre action-événement couverte par cet essai. Il définit le format, développé dans le cadre de cet essai, que celui-ci doit respecter.

2.1 Formats du modèle d'action

Il y a deux formats pour exprimer le modèle d'action. Le premier est un format servant à l'écriture, par une personne, du modèle d'action. Le second format, développé dans le cadre de cet essai, est une transformation du premier pour le rendre manipulable par un outil informatique. Cette sous-section décrit les deux formats.

2.1.1 Format initial du modèle d'action

Dans le cadre action-événement, une personne doit traduire la spécification des exigences (exprimée en langage naturel) en un modèle d'action. Ce modèle s'écrit dans le langage informatique Spec#. Le format initial du modèle d'action est donc du texte respectant la syntaxe de ce langage. La figure 1.3 du chapitre 1 a présenté un exemple de modèle d'action écrit en Spec#. Les détails de ce langage se trouvent dans [5]. Cette sous-section décrit seulement les éléments importants.

Spec# est un langage permettant de modéliser, de façon abstraite, le comportement attendu d'un système dans le but de générer des scénarios de test de la logique métier de façon automatisée. La façon de l'utiliser dans le cadre action-événement est semblable à celle de certaines méthodes formelles de vérification basées sur les modèles, par exemple Spin et

Promela ([4]). Dans cette méthode, afin d'éviter la complexité des implémentations, on construit un modèle simplifié du système qui préserve ses caractéristiques essentielles, tout en évitant les sources connues de complexité :

> « *When the software itself cannot be verified exhaustively, we can build a simplified model of the underlying design that preserves its essential characteristics but that avoids known sources of complexity. The design model can often be verified, while the full-scale implementation cannot.* »[2] ([4], p. x)

Par exemple, pour le cas de la figure 1.3, *todolist* est une séquence. Le modèle fait abstraction de comment elle sera implantée (un tableau, une liste chaînée, etc). Ces détails ne sont pas importants et complexifieraient le modèle. Ce qui importe, c'est que *todolist* ait le comportement d'une séquence. L'important à retenir ici est que l'on doit construire un modèle respectant les caractéristiques du système sans s'empêtrer dans des détails d'implémentation. Le lecteur intéressé à en connaître davantage sur la modélisation abstraite de systèmes pourra se référer à [4].

D'un point de vue syntaxique, Spec# est une extension du langage C#. Il possède donc la même syntaxe et mots clés que C#. Il ajoute à C# des éléments offrant la possibilité de faire de la programmation par contrat (*design by contract*). Dans ce paradigme de programmation, le programmeur spécifie des règles qui doivent être respectées. Dans Spec#, ces règles peuvent prendre l'une des trois formes suivantes :

- Un invariant de classe. Ce sont des propriétés sur les attributs d'une classe qui doivent être vraies en tout temps. On spécifie un invariant de classe à l'aide du mot clé *invariant*.
- Une pré-condition de méthode. C'est une condition qui doit être vraie pour pouvoir appeler la méthode. On spécifie une pré-condition de méthode à l'aide du mot clé *requires*.

[2] Lorsque le logiciel ne peut être vérifié exhaustivement, on peut construire un modèle simplifié du système qui préserve ses caractéristiques essentielles, tout en évitant les sources connues de complexité. Le modèle simplifié peut souvent être vérifié, tandis que l'implémentation complète ne peut généralement pas l'être. (traduction libre)

- Une post-condition de méthode. C'est une condition qu'une méthode doit respecter à la fin de son exécution. On spécifie une post-condition de méthode à l'aide du mot clé *ensures*.

La figure 2.1 présente un exemple d'invariant de classe pour une classe *Rectangle*. On s'assure ici que la longueur et la largeur ne sont jamais négatives et que si l'un des deux côtés est nul, alors l'autre l'est également. La figure 2.2 montre un exemple de pré et post-conditions pour une méthode qui permute deux éléments d'un tableau.

```
invariant 0 <= Dx && 0 <= Dy;
invariant Dx == 0 || Dy == 0 ==> Dx == 0 && Dy == 0;
```

Figure 2.1 Exemple d'invariant de classe

Source : Leino, R. et Müller, P. (2009), p. 15

```
public void Swap(int[] a, int i, int j)
  requires 0 <= i && i < a.Length;
  requires 0 <= j && j < a.Length;
  modifies a[i], a[j];
  ensures a[i] == old(a[j]) && a[j] == old(a[i]);
{
  int tmp = a[i];  a[i] = a[j];  a[j] = tmp;
}
```

Figure 2.2 Exemple de pré et post-conditions de méthode

Source : Leino, R. et Müller, P. (2009), p. 14

Spec Explorer possède un vérificateur statique de programmes qui analyse du code Spec# et qui est capable de déterminer si ce code respecte les invariants de classe ainsi que les pré et post-conditions de méthode ([5]). Il est aussi en mesure de générer automatiquement des scénarios de test de la logique métier d'un logiciel à partir de son modèle d'action.

2.1.2 Format du modèle d'action transformé pour un traitement informatique

Tel que vu à la section 2.1.1, le modèle d'action est fourni par une personne sous forme de texte. L'algorithme de génération du modèle de traduction nécessitera de manipuler les éléments du modèle d'action. Il est donc impératif de transformer le modèle d'action textuel en un format manipulable par un outil informatique. Le format proposé dans cet essai est un métamodèle orienté objet présenté dans le métamodèle des figures 2.3 et 2.4. Afin de pouvoir réutiliser ce métamodèle à d'autres fins (voir la section 4.1), il a été conçu dans le but de pouvoir modéliser non seulement un modèle d'action écrit en Spec#, mais également n'importe quel programme écrit dans un langage orienté objet.

- La métaclasse *ModeleProgramme* représente le modèle du programme en entier. Elle contient la liste des types définis dans le programme. Elle possède un attribut *langage*, qui indique dans quel langage le programme modélisé a été écrit.
- La métaclasse *Langage* est une métaclasse abstraite représentant un langage de programmation en général. Chacune de ses sous-métaclasses représente un langage de programmation spécifique et contient les types relatifs à ce langage.
- La métaclasse *Type* est une métaclasse abstraite représentant n'importe quel type défini dans le programme. Un type peut être un type de base (*int*, *string*, etc.) (sous-métaclasse *TypeDeBase*), une énumération (sous-métaclasse *Enumeration*) ou une classe (sous-métaclasse *Classe*).
- La métaclasse *Classe*, sous-métaclasse de *Type*, représente une classe définie dans le programme. Elle possède une liste de symboles, une liste de méthodes et une liste des événements qu'elle peut lancer.
- La métaclasse *Symbole* est une métaclasse abstraite représentant n'importe quel symbole défini dans le programme. Un symbole peut être une constante symbolique (sous-métaclasse *ConstanteSymbolique*), une variable (locale ou d'instance) (sous-métaclasse *Variable*) ou un paramètre d'une méthode (sous-métaclasse *Parametre*).

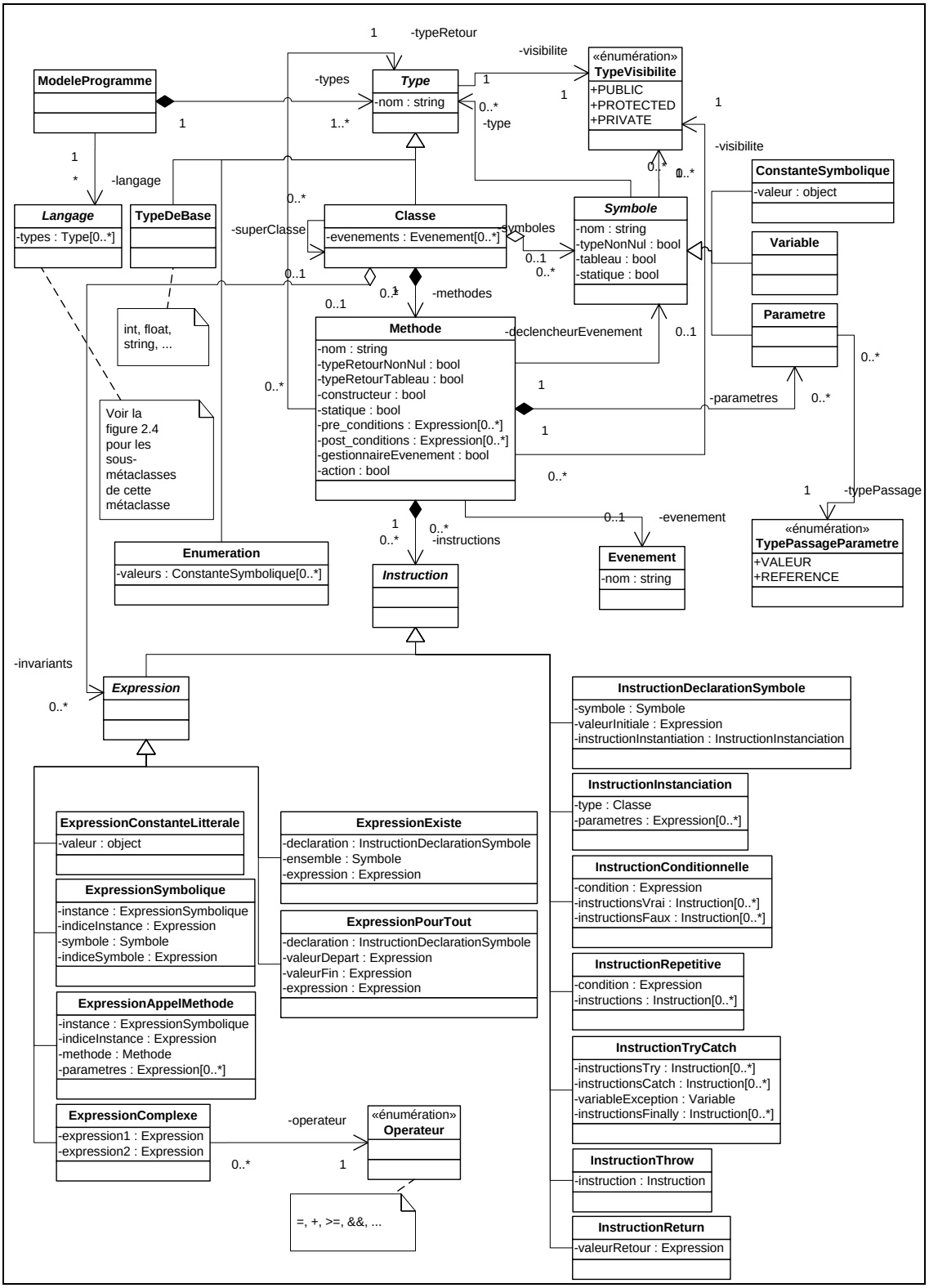


Figure 2.3 Métamodèle pour la modélisation d'actions dans le cadre action-événement

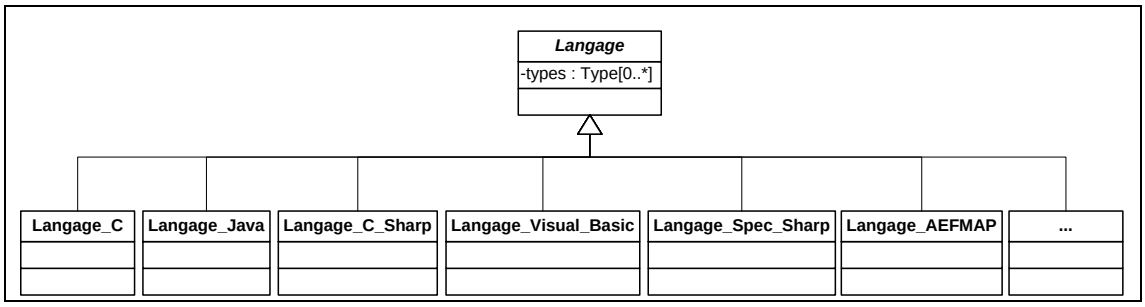


Figure 2.4 Métamodèle pour la modélisation d'actions dans le cadre action-événement (suite)

- La métaclasse *Methode* représente une méthode d'une classe. Elle possède un nom, un type de retour et contient une liste de ses paramètres, une liste de ses pré-conditions, une liste des ses post-conditions et une liste des instructions de son corps. Elle possède également une référence vers l'événement qu'elle gère si la méthode est un gestionnaire d'événement. Finalement, elle possède un attribut booléen indiquant si cette méthode est une action ou non (pertinent seulement lorsque le programme modélisé est un modèle d'action).
- La métaclasse *Instruction* est une métaclasse abstraite représentant n'importe quelle instruction. Une instruction peut être une instruction simple (par exemple une affectation, une addition, etc.), que l'on nomme une expression (sous-métaclasse *Expression*), une déclaration de symbole (sous-métaclasse *InstructionDeclarationSymbole*), une instruction d'instanciation (sous-métaclasse *InstructionInstanciation*), une instruction conditionnelle (sous-métaclasse *InstructionConditionnelle*), une instruction répétitive (sous-métaclasse *InstructionRepetitive*), une instruction « *try-catch* » (sous-métaclasse *InstructionTryCatch*), une instruction de lancement d'exception (sous-métaclasse *InstructionThrow*) ou une instruction de retour (sous-métaclasse *InstructionReturn*).
- La métaclasse *Expression*, sous-métaclasse de *Instruction*, est une métaclasse abstraite représentant n'importe quelle expression. Une expression peut être une constante littérale (sous-métaclasse *ExpressionConstanteLitterale*), un symbole (sous-métaclasse

ExpressionSymbolique), un appel de méthode (sous-métaclasse *ExpressionAppelMethode*), une expression « il existe » (sous-métaclasse *ExpressionExiste*), une expression « pour tout » (sous-métaclasse *ExpressionPourTout*) ou une expression composée de deux expressions séparées par un opérateur (sous-métaclasse *ExpressionComplexe*). Une expression peut être arbitrairement complexe en utilisant comme deuxième expression d'une *ExpressionComplexe* une autre *ExpressionComplexe*.

Une fois le modèle d'action chargé en mémoire selon cette structure, l'algorithme de génération du modèle de traduction pourra aisément le parcourir pour bien faire son travail.

Chapitre 3

Modèle de l'interface graphique

Ce chapitre couvre le modèle de l'interface graphique, l'un des intrants de l'activité du cadre action-événement couverte par cet essai. Il définit le format, développé dans le cadre de cet essai, que celui-ci doit respecter.

3.1 Modélisation d'interfaces graphiques

Plusieurs travaux de recherche sur la façon de modéliser une interface graphique ont été menés. La plupart des travaux récents privilégient l'utilisation de diagrammes UML puisque, comme le remarquent Paulo Pinheiro da Silva et Norman Paton, il est plus aisé d'utiliser des notations déjà existantes (il n'est ainsi pas nécessaire d'en apprendre une autre) et il est plus pratique d'avoir toute l'application modélisée avec la même notation (le reste du logiciel d'une application orientée objet sera également modélisé avec UML) :

> « *Therefore, it would be best not to have to invent new modelling constructs for the UI if existing ones can be used effectively. Further, it would be good to be able to use the same constructs for the UI as for the rest of the application. Indeed, a single notation could be useful for consolidating the complete design of an object-oriented user interface.* »[3] ([12], p. 1)

Dans [1], Kai Blankenhorn a fait une étude de l'applicabilité des différents diagrammes UML pour spécifier la disposition et l'agencement des composants d'une interface graphique. Il a tout d'abord établi que, étant donné la nature hiérarchisée des interfaces graphiques, les

[3] Il serait préférable de ne pas avoir à inventer de nouvelles notations pour la modélisation d'interfaces utilisateur si celles qui existent déjà peuvent être utilisées efficacement. En outre, il serait bon d'être en mesure d'utiliser les mêmes notations pour l'interface utilisateur que pour le reste de l'application. Une seule et unique notation pourrait ainsi être utile pour la spécification de la conception complète d'une application orientée objet. (traduction libre)

diagrammes structurels sont les plus aptes à les représenter ([1], p. 30). Les diagrammes structurels de UML sont le diagramme de classes, le diagramme de structure composite, le diagramme de composants, le diagramme de déploiement, le diagramme d'objets et le diagramme de paquetage. L'étude de Blankenhorn a par la suite démontré que, parmi ces diagrammes, le diagramme de classes et le diagramme de structure composite sont les mieux adaptés à la représentation d'interfaces graphiques ([1], p. 31). La figure 3.1 illustre un exemple de modèle d'interface graphique exprimé à l'aide d'un diagramme de classes. La figure 3.2 illustre la même interface à l'aide d'un diagramme de structure composite.

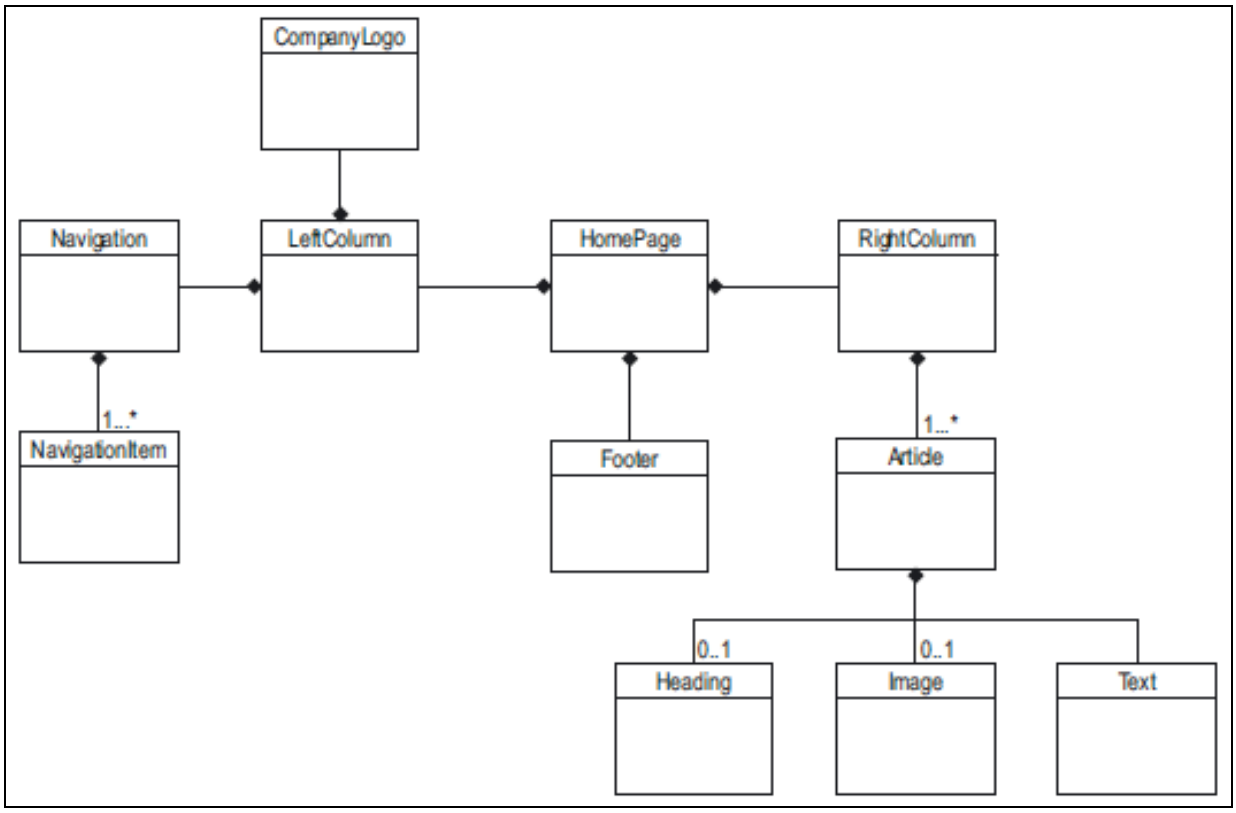


Figure 3.1 Exemple de modèle d'interface graphique exprimé à l'aide d'un diagramme de classes

Source : Blankenhorn, K. (2004), p. 32

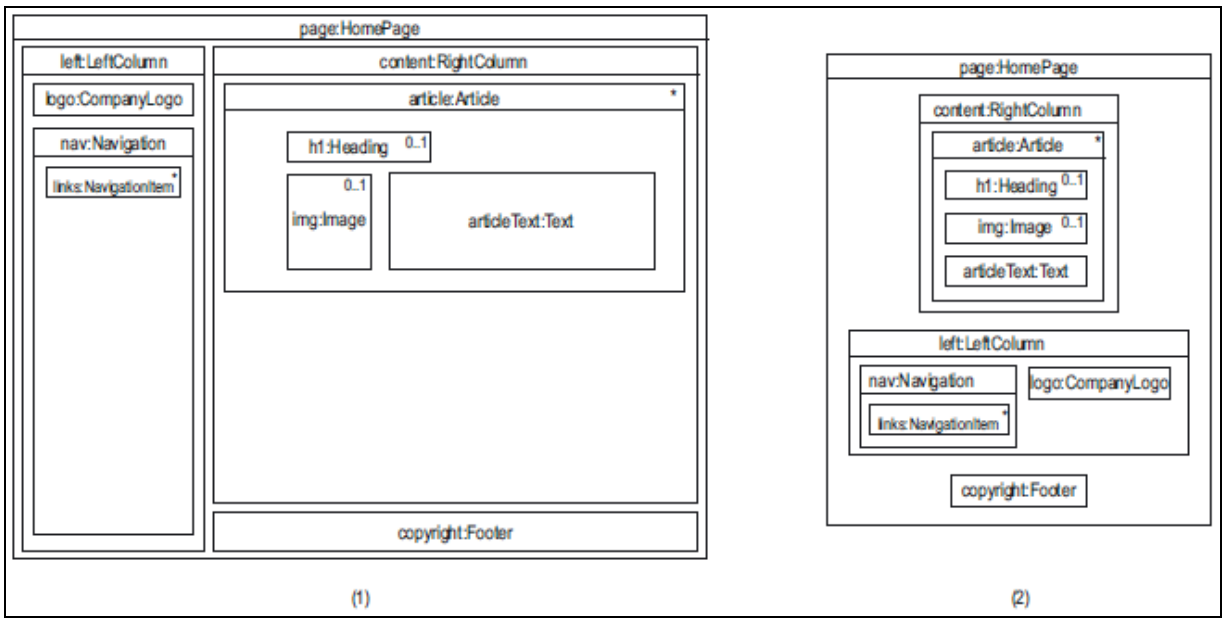


Figure 3.2 Exemple de modèle d'interface graphique exprimé à l'aide d'un diagramme de structure composite

Source : Blankenhorn, K. (2004), p. 32

Ces représentations illustrent les relations hiérarchiques entre les composants (conteneur vs contenu), mais n'énoncent pas le positionnement précis de ceux-ci. Pour pallier ce manque, l'auteur propose un métamodèle qui est une extension du métamodèle de UML ([1], p. 51) (figure 3.3). Il a utilisé les stéréotypes pour étendre le vocabulaire de UML et ainsi créer un nouveau langage de spécification d'interfaces graphiques. L'une de ses innovations importantes a été d'être en mesure de donner une sémantique à la position des objets dans un diagramme ([1], p. 50). Dans ce métamodèle, les composants graphiques doivent être regroupés en zones (*ScreenArea*), qui servent à exprimer la hiérarchie. De plus, il a identifié que les classes de composant graphique se divisent en deux catégories : les classes activables (*ActivatableUIFunctionality*) (qui provoquent une action, par exemple un bouton) et les classes statiques (*StaticUIFunctionality*) (qui affichent de l'information, par exemple une zone de texte). Finalement, ce métamodèle permet d'énoncer de quel type sont les composants graphiques contenus dans l'interface (zone de texte, image, etc.), tout en faisant abstraction du choix du langage d'implémentation.

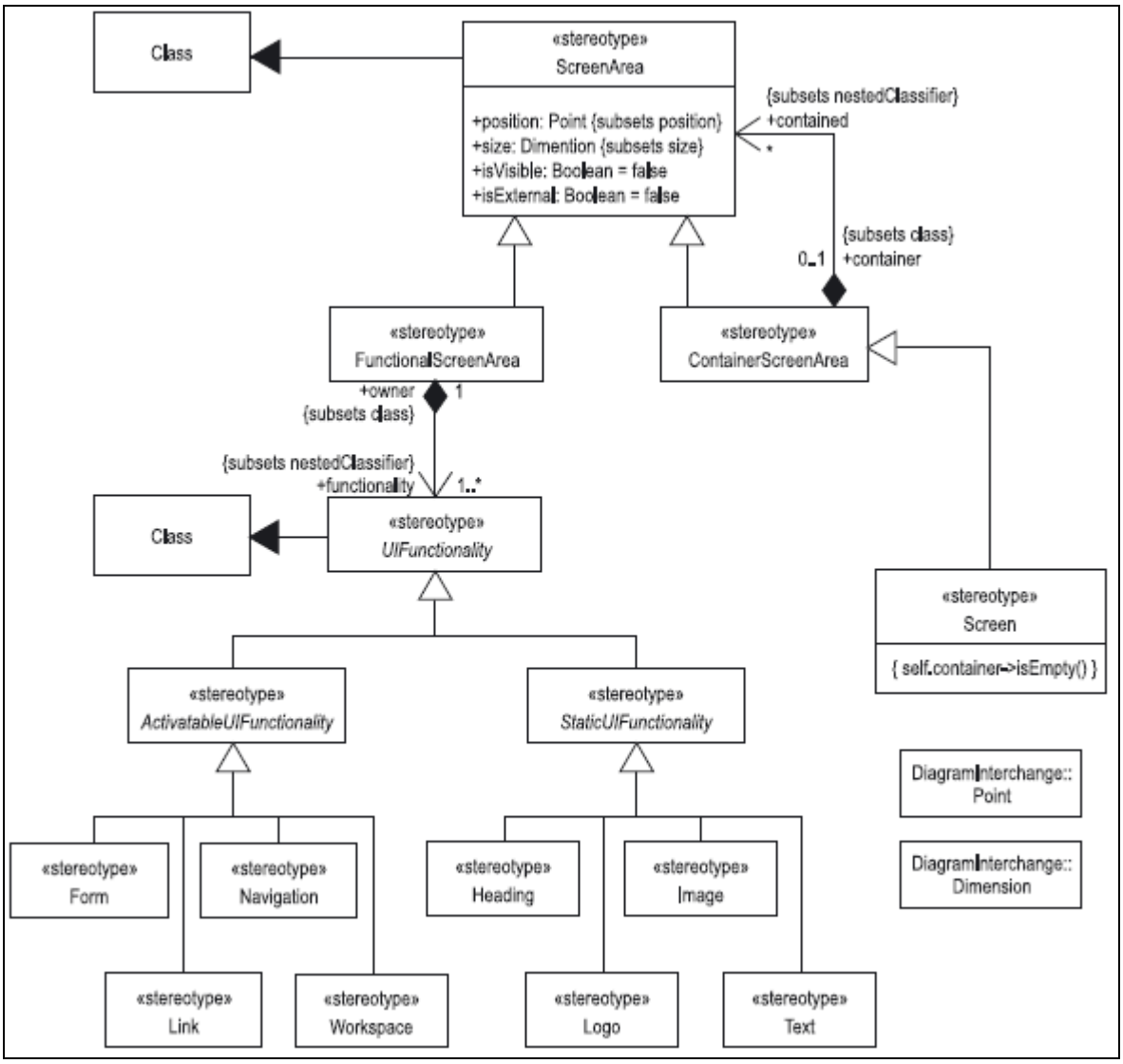


Figure 3.3 Métamodèle de Blankenhorn pour la modélisation d'interfaces graphiques
Source : Blankenhorn, K. (2004), p. 51

Pinheiro Da Silva et Paton ont proposé deux métamodèles semblables à celui de Blankenhorn. Le premier métamodèle permet une modélisation abstraite de très haut niveau ([12], p. 6) (figure 3.4). Il sert à spécifier des interfaces au tout début de la phase d'analyse, lorsqu'on n'a qu'une idée vague de son apparence. Il représente lui aussi les liens hiérarchiques (via les *AbstractContainer*), mais ne représente pas les positionnements ni le type concret des

composants graphiques (bouton, zone de texte, etc.). Il divise les classes de composant graphique en trois catégories plutôt que deux : les déclencheurs d'action (*ActionInvoker*), les classes d'affichage statique (*StaticDisplay*) et les classes de contrôle interactif (*InteractionControl*) (qui servent à la navigation, par exemple un menu). Les deux premières catégories représentent le même concept que celles du métamodèle de Blankenhorn.

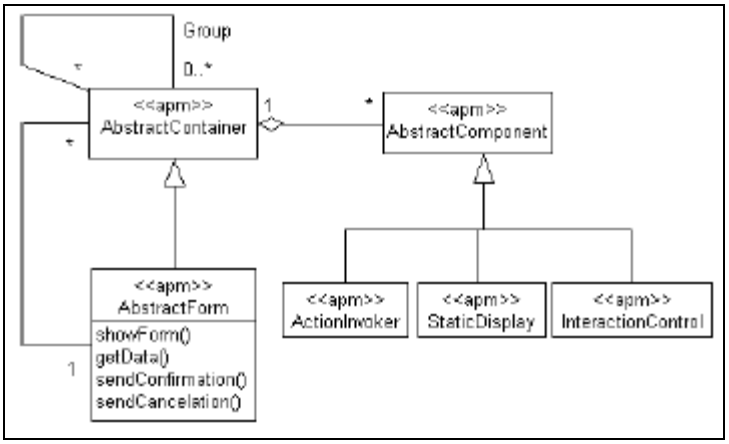


Figure 3.4 Métamodèle de Pinheiro da Silva et Paton pour la modélisation abstraite d'interfaces graphiques

Source : Pinheiro da Silva, P. et Paton, N. (2000), p. 6

Le deuxième métamodèle permet une modélisation concrète de très bas niveau ([12], p. 8) (figure 3.5). Il sert à spécifier une interface en vue de l'implémentation de celle-ci. Pour ce faire, il ajoute, au métamodèle abstrait, des sous-métaclasses aux catégories de composant graphique représentant non seulement le type de composant graphique (bouton, zone de texte, etc.), mais également le type de données cible dans le langage de programmation choisi. Il faut donc un métamodèle pour chacun des langages de programmation que l'on veut couvrir. La figure 3.5 est un exemple de métamodèle pour spécifier une interface qui sera implémentée en Java.

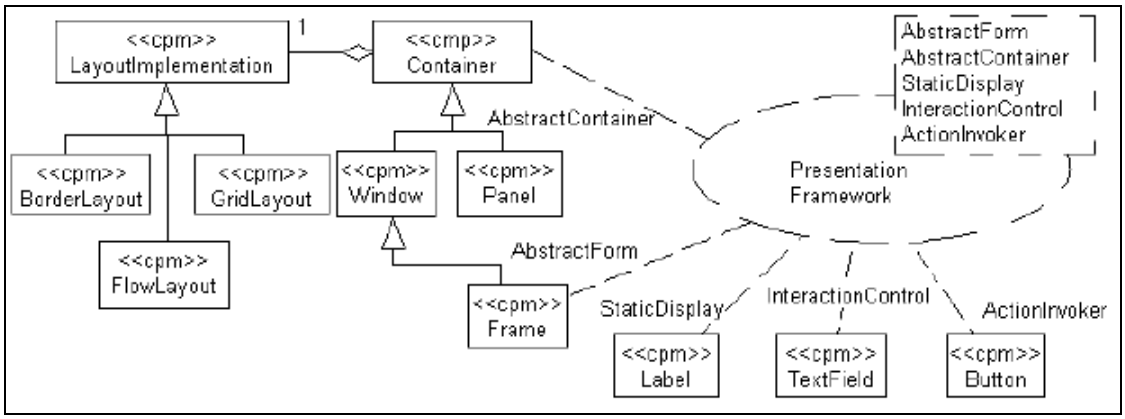


Figure 3.5 Métamodèle de Pinheiro da Silva et Paton pour la modélisation concrète d'interfaces graphiques (Java)

Source : Pinheiro da Silva, P. et Paton, N. (2000), p. 8

3.2 Métamodèle pour la modélisation d'interfaces graphiques dans le cadre action-événement

Le métamodèle pour la modélisation d'interfaces graphiques conçu dans le cadre de cet essai s'inspire des métamodèles présentés à la section 3.1. L'algorithme de génération du modèle de traduction aura besoin de connaître la hiérarchie des composants de l'interface. Ainsi, ce concept, qui était présent dans les autres métamodèles, a été repris (certains composants graphiques sont classés comme étant des conteneurs). L'abstraction du choix du langage d'implémentation des deux premiers métamodèles présentés est également existante, par l'emploi de métaclasses représentant le type de composant graphique (bouton, zone de texte, etc.) sans spécifier de type de données spécifique du langage de programmation cible. Le métamodèle de cet essai se distingue des autres par un plus grand nombre de types de composants graphiques couverts (33 contre seulement 8 pour le métamodèle de Blankenhorn) et par une catégorisation différente des composants graphiques (composants graphiques simples, de liste et conteneur), qui est plus adaptée aux besoins de cet essai. Le métamodèle complet est présenté à la figure 3.6 et est décrit dans le reste de cette sous-section. Ce métamodèle ne permet de modéliser que des interfaces graphiques de type « application de bureau » (*desktop application*).

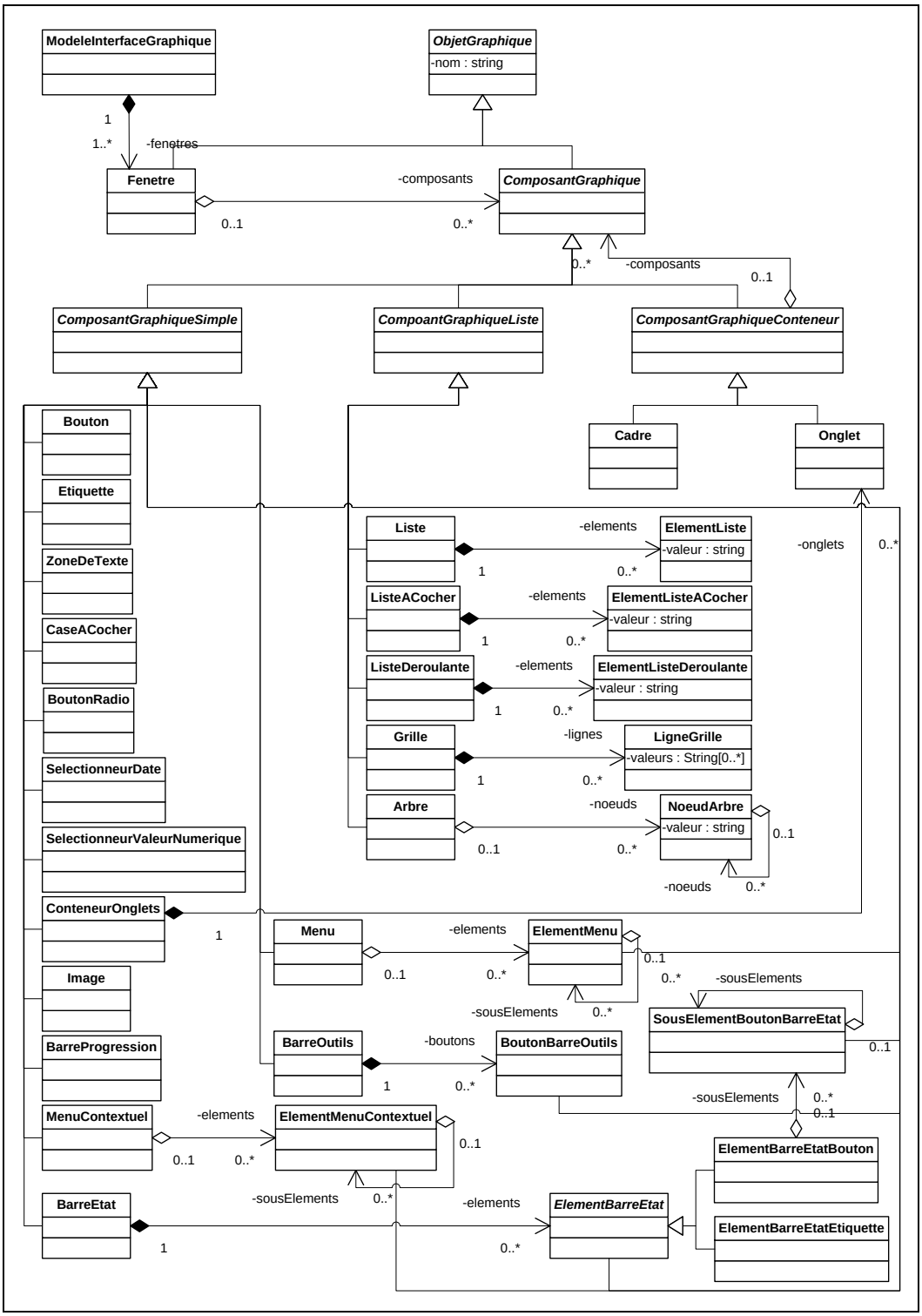


Figure 3.6 Métamodèle pour la modélisation d'interfaces graphiques dans le cadre action-événement

- La métaclasse *ModeleInterfaceGraphique* représente le modèle de l'interface graphique en entier. Elle contient la liste des fenêtres du logiciel modélisé.
- La métaclasse *ObjetGraphique* est une métaclasse abstraite représentant n'importe quel objet graphique d'une interface.
- La métaclasse *Fenetre* (sous-métaclasse de *ObjetGraphique*) représente une fenêtre d'une interface graphique. Elle contient la liste des composants graphiques situés dans cette fenêtre.
- La métaclasse *ComposantGraphique* (sous-métaclasse de *ObjetGraphique*) est une métaclasse abstraite représentant n'importe quel composant d'une interface. Un composant graphique est tout objet graphique pouvant aller dans une fenêtre.
- La métaclasse *ComposantGraphiqueConteneur* (sous-métaclasse de *ComposantGraphique*) est une métaclasse abstraite représentant n'importe quel composant graphique pouvant contenir d'autres composants graphiques. Un cadre et un onglet sont des exemples de conteneurs de composants graphiques.
- La métaclasse *ComposantGraphiqueListe* (sous-métaclasse de *ComposantGraphique*) est une métaclasse abstraite représentant n'importe quel composant graphique qui est une liste d'éléments. Une liste, une grille et un arbre sont des exemples de composants graphiques de liste.
- La métaclasse *ComposantGraphiqueSimple* (sous-métaclasse de *ComposantGraphique*) est une métaclasse abstraite représentant n'importe quel composant graphique simple, c'est-à-dire qui n'est pas composé d'autres composants (bref, un composant graphique qui est ni un conteneur, ni une liste). Un bouton, une zone de texte et une case à cocher sont des exemples de composants graphiques simples.

Les sous-métaclasses des métaclasses abstraites *ComposantGraphiqueSimple*, *ComposantGraphiqueConteneur* et *ComposantGraphiqueListe* sont des métaclasses concrètes représentant des composants graphiques réels tels un bouton ou une zone de texte. Afin que le

lecteur comprenne bien ce que chacune de ces métaclasses représente, la figure 3.7 présente un exemple d'une interface graphique contenant chacun des composants graphiques couverts par cet essai.

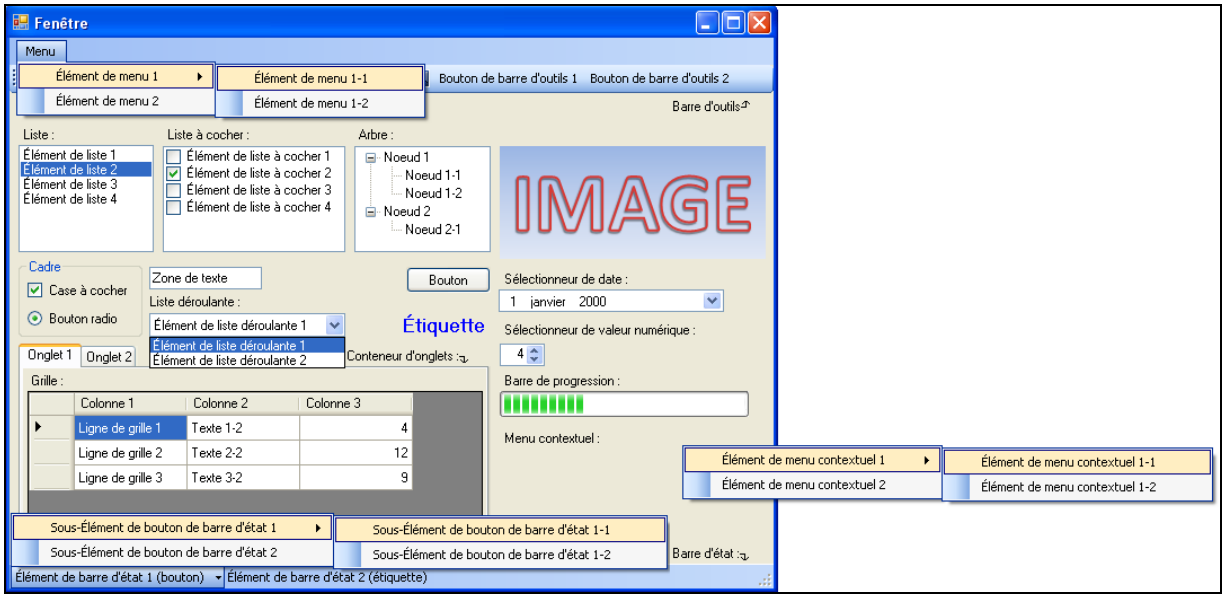


Figure 3.7 Exemple d'interface graphique contenant chacun des composants graphiques couverts par cet essai

Une fois le modèle de l'interface graphique chargé en mémoire selon cette structure, l'algorithme de génération du modèle de traduction pourra aisément le parcourir pour bien faire son travail.

Chapitre 4

Algorithme de génération du modèle de traduction

Ce chapitre décrit l'algorithme, développé au cours de cet essai, qui automatise la génération du modèle de traduction dans le cadre action-événement.

4.1 Intrant supplémentaire au cadre action-événement

Dans le cadre action-événement, le modèle de traduction est écrit par une personne, basé sur sa connaissance du code source du programme. Puisque le travail de cet essai consiste à automatiser cette partie, le module qui génère le modèle de traduction doit également prendre en entrée le code source du programme. Or, comme mentionné au chapitre 2 traitant du modèle d'action, le code source est du texte, et l'algorithme de génération du modèle de traduction devra manipuler les éléments du code source. Il est donc impératif de transformer le code source textuel en un format manipulable par un outil informatique, que l'on appellera le modèle du programme. Le métamodèle retenu est le même que celui proposé pour le modèle d'action, puisque celui-ci a été conçu dans le but de pouvoir modéliser n'importe quel programme écrit dans un langage orienté objet. Comme pour le modèle d'action, cet essai ne traitera pas de la façon de faire l'ingénierie inverse du code source pour construire le modèle. La figure 4.1 présente le cadre action-événement modifié avec le nouvel intrant.

4.2 Algorithme de génération du modèle de traduction

L'algorithme de génération du modèle de traduction est présenté à la figure 4.3. Puisque celui-ci parcourt abondamment le modèle du programme, la figure 4.2 effectue tout d'abord un rappel des éléments les plus importants de son métamodèle.

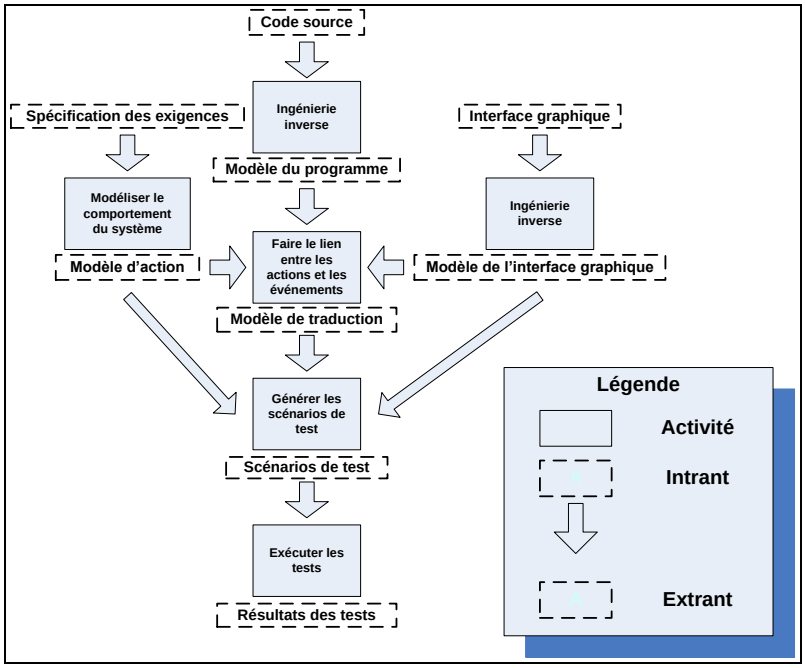


Figure 4.1 Cadre action-événement modifié

Modifié de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 157

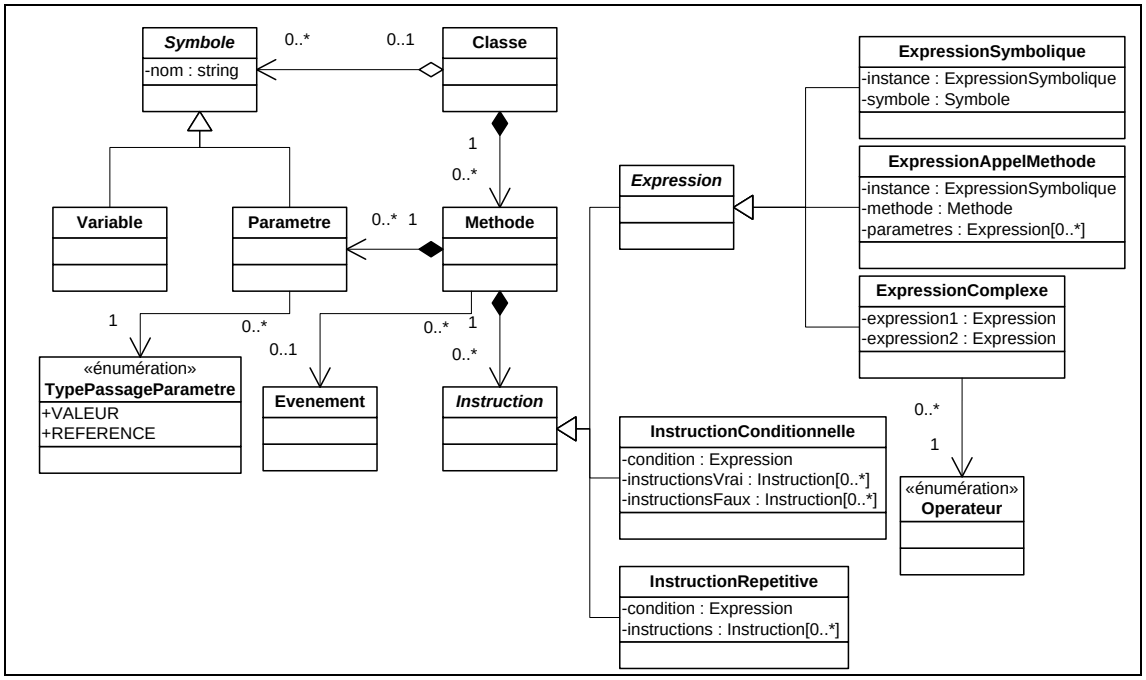


Figure 4.2 Métamodèle abrégé pour la modélisation du programme dans le cadre action-événement (rappel)

```
| Algorithme 1 - Génération du modèle de traduction                                                                    |
-------------------------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Modèle d'action                +
+           - Modèle de l'interface graphique +
+           - Modèle du programme             +
+                                             +
+ Sortie : - Modèle de traduction (texte)    +
++++++++++++++++++++++++++++++++++++++++++++

Générer la table des séquences d'instructions (algorithme 1.1.1)
Générer le graphe d'appels de méthodes (algorithme 1.1.2)
Générer la table de correspondance action/méthode (algorithme 1.1.3)
Générer la table des chemins d'appel d'action (algorithme 1.1.4)
Générer la table de correspondance classe/fenêtre (algorithme 1.1.5)
Générer la table de correspondance symbole/composant graphique (algorithme 1.1.6)

Pour chaque classe d'action (métaclasse Classe) du modèle d'action
    Pour chaque action (métaclasse Methode) de la classe d'action
        Générer la fonction de traduction de l'action (algorithme 1.2)
    Fin pour chaque
Fin pour chaque
```

Figure 4.3 Algorithme de génération du modèle de traduction (algorithme 1)

Les six premières étapes consistent à pré-calculer les structures de données dont l'algorithme aura besoin pour effectuer son travail. La septième étape consiste à générer une fonction de traduction pour chacune des actions du modèle d'action. Les sous-algorithmes sont décrits dans les sous-sections suivantes.

4.2.1 Algorithme pour générer la table des séquences d'instructions (algorithme 1.1.1)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de connaître toutes les séquences d'instructions possibles de toutes les méthodes du programme. Pour cette raison, une table contenant toutes ces séquences est pré-calculée afin de faciliter le travail de l'algorithme. Cette table possède deux membres : le membre de gauche est un pointeur sur une méthode du modèle du programme et le membre de droite est une liste des séquences d'instructions de la méthode. Une séquence d'instructions d'une méthode est une suite d'instructions consécutives correspondant à une exécution possible du code de la méthode. La figure 4.4 présente l'algorithme 1.1.1, qui génère cette table.

L'algorithme 1.1.1 initialise la table en ajoutant dans celle-ci une entrée pour chaque méthode (métaclasse *Methode*) trouvée dans le modèle du programme avec, comme liste de séquences,

une seule séquence vide. Il délègue la tâche de remplir cette liste à l'algorithme 1.1.1.1, qui est présenté à la figure 4.5.

```
| Algorithme 1.1.1 - Génération de la table des séquences d'instructions |

+++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Modèle du programme                 +
+                                                +
+ Sortie : - Table des séquences d'instructions  +
+++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque classe (métaclasse Classe) du modèle du programme
    Pour chaque méthode (métaclasse Methode) de la classe
        Ajouter une entrée dans la table des séquences d'instructions avec, dans la partie gauche,
        la méthode et, dans la partie droite, une liste de séquences contenant une séquence vide

        Créer toutes les séquences d'instructions de la méthode (algorithme 1.1.1.1)
    Fin pour chaque
Fin pour chaque
```

Figure 4.4 Algorithme pour générer la table des séquences d'instructions (algorithme 1.1.1)

```
| Algorithme 1.1.1.1 - Génération des séquences d'instructions |

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Liste d'instructions                           +
+          - Liste de séquences d'instructions             +
+                                                          +
+ Sortie : - Liste de séquences d'instructions modifiée     +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque instruction (métaclasse Instruction) de la liste d'instructions
    Si l'instruction est une instruction conditionnelle (métaclasse InstructionConditionnelle)
        Ajouter la condition de l'instruction conditionnelle dans toutes les séquences de la liste
        de séquences

        Cloner la liste de séquences d'instructions

        Ajouter, dans la liste clonée, les instructions contenues dans la partie "vrai" de
        l'instruction conditionnelle (appel récursif à cet algorithme)

        Ajouter les instructions contenues dans la partie "faux" de l'instruction conditionnelle
        dans toutes les séquences de la liste de séquences (appel récursif à cet algorithme)

        Ajouter le contenu de la liste clonée dans la liste principale
    Sinon si l'instruction est une instruction répétitive (métaclasse InstructionRepetitive)
        Ajouter la condition de l'instruction répétitive dans toutes les séquences de la liste de
        séquences

        Cloner la liste de séquences d'instructions

        Ajouter, dans la liste clonée, les instructions contenues dans l'instruction répétitive
        (appel récursif à cet algorithme)

        Ajouter le contenu de la liste clonée dans la liste principale
    Sinon si l'instruction est une instruction "try-catch" (métaclasse InstructionTryCatch)
        Ajouter les instructions de la partie "try" dans toutes les séquences de la liste de
        séquences (appel récursif à cet algorithme)

        Ajouter les instructions de la partie "finally" dans toutes les séquences de la liste de
        séquences (appel récursif à cet algorithme)
    Sinon
        Ajouter l'instruction dans toutes les séquences de la liste de séquences
    Fin si
Fin pour chaque
```

Figure 4.5 Algorithme pour générer les séquences d'instructions (algorithme 1.1.1.1)

Au départ, la méthode ne possède qu'une seule séquence vide, car son corps n'a pas encore été exploré. Une boucle est effectuée sur les instructions (métaclasse *Instruction*) de la méthode afin de construire ces séquences.

Si l'instruction rencontrée est une instruction ordinaire[4], elle est tout simplement ajoutée dans toutes les séquences découvertes jusqu'à présent, car une instruction ordinaire ne change pas le flot d'exécution. Les autres types d'instruction sont des instructions qui altèrent le flot d'exécution et doivent donc être traités de façon spéciale.

Si l'instruction rencontrée est une instruction conditionnelle (métaclasse *InstructionConditionnelle*), la condition est tout d'abord ajoutée dans toutes les séquences découvertes jusqu'à présent, car celle-ci sera nécessairement exécutée. Les instructions contenues dans la partie « vrai » ne seront pas nécessairement exécutées et constituent donc un embranchement que les séquences découvertes peuvent emprunter. Pour cette raison, les séquences découvertes sont clonées, pour ne pas les altérer, et les instructions de la partie « vrai » sont ajoutées dans les séquences clonées. Les instructions contenues dans la partie « faux » seront exécutées si celles de la partie « vrai » ne le sont pas. Pour cette raison, elles peuvent directement être ajoutées dans les séquences principales, car les instructions de la partie « vrai » ont été mises dans une liste de séquences alternatives. Une fois ce travail fait, les séquences clonées peuvent être fusionnées avec les séquences principales pour former l'ensemble des séquences possibles.

Si l'instruction rencontrée est une instruction répétitive (métaclasse *InstructionRepetitive*), la condition est tout d'abord ajoutée dans toutes les séquences découvertes jusqu'à présent, car celle-ci sera nécessairement exécutée. Les instructions contenues dans le corps de la boucle ne

[4] Une instruction ordinaire, dans le contexte de cet algorithme, est une instruction qui est autre chose qu'une instruction conditionnelle (métaclasse *InstructionConditionnelle*), une instruction répétitive (métaclasse *InstructionRepetitive*) ou une instruction « *try-catch* » (métaclasse *InstructionTryCatch*).

seront pas nécessairement exécutées (la condition peut être fausse dès le départ) et constituent donc un embranchement que les séquences découvertes peuvent emprunter. Pour cette raison, les séquences découvertes sont clonées, pour ne pas les altérer, et les instructions du corps de la boucle sont ajoutées dans les séquences clonées. Les instructions du corps de la boucle ne doivent être ajoutées qu'une seule fois dans les séquences, même si elles peuvent être exécutées plusieurs fois, car l'algorithme de génération du modèle de traduction ne requiert pas qu'elles s'y trouvent plusieurs fois. Une fois ce travail fait, les séquences clonées peuvent être fusionnées avec les séquences principales pour former l'ensemble des séquences possibles.

Si l'instruction rencontrée est une instruction « *try-catch* » (métaclasse *InstructionTryCatch*), les instructions de la partie « *try* » et de la partie « *finally* » sont ajoutées dans toutes les séquences découvertes jusqu'à présent. Les instructions de la partie « *catch* » ne sont pas considérées, car l'algorithme de génération du modèle de traduction a besoin qu'on lui fournisse toutes les séquences d'une exécution normale du programme, donc sans lancement d'exception.

À chaque fois qu'il est fait mention « d'ajouter les instructions d'une certaine partie d'une instruction dans les séquences d'instructions », cela implique que l'algorithme 1.1.1.1 doit être appelé récursivement pour traiter cela, car ces instructions peuvent être, par exemple, des instructions conditionnelles (métaclasse *InstructionConditionnelle*) et donc à leur tour contenir d'autres instructions qui devront elles aussi subir le même traitement.

4.2.2 Algorithme pour générer le graphe d'appels de méthodes (algorithme 1.1.2)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de savoir par quelle(s) autre(s) méthode(s) une méthode peut être appelée. Pour cette raison, un graphe d'appels de méthodes est pré-calculé afin de faciliter le travail de l'algorithme. Ce graphe est un graphe orienté acyclique $G = (V, E)$ dans lequel un sommet $v \in V$ est une méthode du modèle du programme et où un arc $(v_1, v_2) \in E$ indique que la méthode v_1 peut

appeler la méthode v_2. Un tel lien n'indique pas que la méthode v_1 appelle nécessairement la méthode v_2, mais bien qu'elle le fait dans certaines situations (l'appel de la méthode peut se trouver à l'intérieur d'une instruction conditionnelle par exemple). Ainsi, le graphe représente tous les liens possibles entre les méthodes. Le graphe doit être acyclique, car l'algorithme de génération du modèle de traduction développé dans cet essai ne gère pas les cycles dans les graphes. Une des limites de cet essai est donc que le programme testé ne peut pas avoir de méthodes directement ou indirectement récursives. La figure 4.6 présente l'algorithme 1.1.2, qui construit ce graphe.

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.1.2 - Génération du graphe d'appels de méthodes                                        |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Modèle du programme         +
+                                        +
+ Sortie : - Graphe d'appels de méthodes +
++++++++++++++++++++++++++++++++++++++++++

Ajouter les sommets dans le graphe :
    - Ajouter les méthodes du langage du modèle du programme dans le graphe
    - Ajouter les méthodes du modèle du programme dans le graphe

Ajouter les arcs dans le graphe :
    Pour chaque classe (métaclasse Classe) du modèle du programme
        Pour chaque méthode (métaclasse Methode) de la classe
            Pour chaque instruction (métaclasse Instruction) de la méthode
                Si l'instruction contient un appel de méthode (métaclasse ExpressionAppelMethode)
                    Ajouter un arc orienté de la méthode appelante vers la méthode appelée dans le
                    graphe d'appels de méthodes
                Fin si
            Fin pour chaque
        Fin pour chaque
    Fin pour chaque
```

Figure 4.6 Algorithme pour générer le graphe d'appels de méthodes (algorithme 1.1.2)

La première étape consiste à ajouter les sommets dans le graphe. Les sommets sont toutes les méthodes contenues dans le modèle du programme et également toutes les méthodes prédéfinies du langage avec lequel le programme a été écrit. La deuxième étape consiste à ajouter les arcs dans le graphe. Pour ce faire, une boucle est effectuée sur chaque instruction (métaclasse *Instruction*) de chaque méthode (métaclasse *Methode*) de chaque classe (métaclasse *Classe*) du modèle du programme. Si l'instruction est un appel de méthode (métaclasse *ExpressionAppelMethode*), alors un arc orienté de la méthode appelante vers la méthode appelée est ajouté dans le graphe. Afin de couvrir toutes les instructions, celles-ci sont parcourues en profondeur, c'est-à-dire que les instructions contenues dans les instructions

sont elles aussi examinées. Par exemple, une instruction conditionnelle (métaclasse *InstructionConditionnelle*) contient plusieurs instructions (une condition, plusieurs instructions dans la partie « vrai » et plusieurs instructions dans la partie « faux »). Chacune de ces sous-instructions peut elle aussi contenir plusieurs sous-instructions. Elles doivent toutes être explorées afin de ne manquer aucun appel de méthode. La figure 2.3 du chapitre 2 montre toutes les instructions qui peuvent contenir d'autres instructions.

Puisque les appels de méthodes sont déterminés à partir du modèle du programme, qui est lui-même construit à partir d'une analyse statique du code source du programme, l'algorithme 1.1.2 ne peut pas déterminer exactement quelle méthode sera effectivement appelée lors d'un appel de méthode virtuelle. Pour rappel, un appel de méthode est un appel de méthode virtuelle lorsque celui-ci s'effectue sur une référence d'une classe A qui réfère à une instance d'une classe B (la classe B étant une sous-classe directe ou indirecte de la classe A) et que c'est la méthode définie dans la classe B qui doit être appelée. La bonne méthode à appeler est impossible à déterminer par une analyse statique du code source, c'est pourquoi les compilateurs ne déterminent pas cette information à la compilation, mais plutôt à l'exécution du programme. C'est ce qu'on appelle la liaison dynamique ([3], p. 469). L'impossibilité d'avoir des méthodes virtuelles constitue donc une des limites de cet essai. Cependant, des chercheurs ont développé des algorithmes permettant de construire des graphes d'appels de méthodes contenant les appels de méthodes virtuelles les plus probables ([2] et [13]). Cette idée n'a pas été explorée dans le cadre de cet essai, mais constitue une piste de solution intéressante pour une prochaine version de l'algorithme.

4.2.3 Algorithme pour générer la table de correspondance action/méthode (algorithme 1.1.3)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de faire le lien entre une action du modèle d'action et la méthode correspondante dans le modèle du programme. Pour cette raison, une table faisant le lien entre les actions et les méthodes est pré-calculée afin de faciliter le travail de l'algorithme. Cette table possède deux membres : le

membre de gauche est un pointeur sur une action et le membre de droite est un pointeur sur la méthode. Chaque entrée de la table fait ainsi le lien entre l'action et la méthode. La figure 4.7 présente l'algorithme 1.1.3, qui génère cette table.

```
| Algorithme 1.1.3 - Génération de la table de correspondance action/méthode |

++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Modèle d'action                      +
+           - Modèle du programme                  +
+                                                  +
+ Sortie : Table de correspondance action/méthode +
++++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque classe (métaclasse Classe) d'action du modèle d'action
    Pour chaque action (métaclasse Methode) de la classe d'action
        Pour chaque classe (métaclasse Classe) du modèle du programme
            Si la classe du modèle du programme a le même nom que la classe d'action
                Pour chaque méthode (métaclasse Methode) de la classe du modèle du programme
                    Si la méthode est équivalente à l'action (voir explications)
                        Ajouter une entrée dans la table de correspondance pour cette action et
                        cette méthode
                    Fin si
                Fin pour chaque
            Fin si
        Fin pour chaque
    Fin pour chaque
Fin pour chaque
```

Figure 4.7 Algorithme pour générer la table de correspondance action/méthode (algorithme 1.1.3)

Afin de trouver toutes les actions, une boucle sur chaque classe (métaclasse *Classe*) du modèle d'action est effectuée. Pour chaque action (métaclasse *Methode*), l'algorithme 1.1.3 tente de trouver une méthode équivalente dans le modèle du programme en bouclant sur chaque classe (métaclasse *Classe*) de celui-ci pour trouver toutes les méthodes candidates (métaclasse *Methode*), jusqu'à ce qu'il trouve la méthode équivalente. Une méthode est équivalente à une action si elle se trouve dans une classe de même nom que la classe de l'action et si elle possède une signature identique (même nom, même nombre de paramètres et mêmes types pour les paramètres). Dans sa forme originale, le cadre action-événement ne requiert pas que le code source d'un programme possède une méthode équivalente à chaque action du modèle d'action. Il s'agit d'une contrainte imposée par l'algorithme développé dans cet essai afin de pouvoir automatiser la génération du modèle de traduction.

4.2.4 Algorithme pour générer la table des chemins d'appel d'action (algorithme 1.1.4)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de savoir comment une action peut être déclenchée. Pour cette raison, une table indiquant toutes les façons dont les actions peuvent être déclenchées est pré-calculée afin de faciliter le travail de l'algorithme. Cette table possède deux membres : le membre de gauche est un pointeur sur une action et le membre de droite est une liste des chemins d'appel possibles pour cette action. Un chemin d'appel d'action est une suite de méthodes qui doivent être appelées pour atteindre la méthode qui correspond à l'action. La figure 4.8 présente l'algorithme 1.1.4, qui génère cette table.

```
-----------------------------------------------------------------------------------------------------
| Algorithme 1.1.4 - Génération de la table des chemins d'appel d'action                            |
-----------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Graphe d'appels de méthodes             +
+           - Table de correspondance action/méthode +
+                                                    +
+ Sortie : - Table des chemins d'appel d'action      +
++++++++++++++++++++++++++++++++++++++++++++++++++++++

Calculer le graphe d'appels de méthodes transposé

Pour chaque action (métaclasse Methode) de la table de correspondance action/méthode
    Obtenir la méthode (métaclasse Methode) correspondant à l'action dans la table de
    correspondance action/méthode

    Ajouter, dans la table des chemins d'appel d'action, tous les chemins du graphe transposé
    débutant au sommet correspondant à la méthode et se terminant par un puits (algorithme 2)
Fin pour chaque

Inverser les chemins de la table des chemins d'appel d'action

Éliminer, de la table des chemins d'appel d'action, tous les chemins dont la première méthode n'est
pas un gestionnaire d'événement
```

Figure 4.8 Algorithme pour générer la table des chemins d'appels d'action (algorithme 1.1.4)

Pour chaque action (métaclasse *Methode*) du modèle d'action, la méthode (métaclasse *Methode*) correspondante du modèle du programme est déterminée grâce à la table de correspondance action/méthode construite par l'algorithme 1.1.3. Pour trouver tous les chemins d'appel de cette action, l'algorithme 1.1.4 devra explorer les chemins du graphe d'appels de méthodes construit par l'algorithme 1.1.2. Pour rappel, un chemin de longueur n d'un graphe est une suite de n sommets deux à deux adjacents débutant à v_0 et se terminant à

v_n. Un chemin élémentaire est un chemin dans lequel chacun des sommets ne se trouve qu'une seule fois. Un chemin d'appel d'une action est un chemin élémentaire du graphe d'appels de méthodes dans lequel la dernière méthode est celle correspondant à cette action. Trouver les chemins élémentaires d'un graphe se fait par un parcours en profondeur ou en largeur de celui-ci, en accumulant les sommets visités et en créant un nouveau parcours pour chaque embranchement emprunté. Cependant, dans le cas présent, trouver ces chemins pose problème, car ces algorithmes supposent que le sommet de départ est connu alors que, ici, c'est le sommet d'arrivée qui l'est. Pour cette raison, l'algorithme 1.1.4 doit utiliser la transposée du graphe d'appels de méthodes. Pour rappel, un graphe transposé G^T d'un graphe orienté G est un graphe orienté ayant les mêmes sommets et mêmes arcs que G, mais où le sens des arcs a été inversé. De cette façon, le parcours démarrera au sommet d'arrivée et construira les chemins à l'envers. À cause de cela, l'algorithme 1.1.4 inverse les chemins trouvés à postériori. De plus, il élimine des chemins trouvés ceux dont la première méthode n'est pas un gestionnaire d'événement, car le modèle de traduction qui devra être généré doit refléter comment un utilisateur interagit avec l'interface graphique pour déclencher cette action. Or, l'interaction d'un utilisateur avec l'interface graphique se fait par le déclenchement d'événements (un clic sur un bouton par exemple) qui appelle une série de méthodes pour effectuer son travail. Puisque l'on cherche des chemins débutant par un gestionnaire d'événement (donc un chemin se terminant par un gestionnaire d'événement dans le graphe transposé) et qu'un gestionnaire d'événement ne se fait pas appeler par d'autres méthodes (c'est l'utilisateur qui le déclenche), l'algorithme 1.1.4 doit trouver, dans le graphe transposé, des chemins dont le dernier sommet est un puits. Pour rappel, dans un graphe orienté, un puits est un sommet qui ne possède pas d'arc sortant. La figure 4.9 présente l'algorithme 2, utilisé par l'algorithme 1.1.4, qui retourne tous les chemins d'un graphe orienté acyclique débutant à un certain sommet et se terminant par un puits.

L'exploration du graphe se fait un sommet à la fois, en commençant par le sommet de départ. Le sommet actuellement observé est ajouté dans le chemin en cours. Si ce sommet est un puits, alors le chemin en cours est ajouté à la liste des chemins trouvés. Sinon, cet algorithme est appelé récursivement avec, comme sommet de départ, chacun des sommets sur lesquels le

sommet courant pointe. La récursivité se continuera ainsi de sommet en sommet, jusqu'à l'atteinte d'un puits.

```
-------------------------------------------------------------------------------------------------------------
| Algorithme 2 - Génération de tous les chemins d'un graphe orienté acyclique débutant à un                 |
|                certain sommet et se terminant par un puits                                                |
-------------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++
+ Entrée : - Graphe            +
           - Sommet de départ  +
+                              +
+ Sortie : - Liste des chemins +
++++++++++++++++++++++++++++++++

Initialiser la liste des chemins trouvés à vide
Initialiser le chemin en cours avec le sommet de départ
creerChemins(Graphe, Sommet de départ, Chemin en cours, Liste des chemins)
    (voir la définition de cette fonction utilitaire ci-bas)

-------------------------------------------------------------------------------------------------------------
| Fonction creerChemins                                                                                     |
-------------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Graphe                    +
           - Sommet courant            +
           - Chemin en cours           +
           - Liste des chemins         +
+                                      +
+ Sortie : - Liste des chemins modifiée +
++++++++++++++++++++++++++++++++++++++++

Si le sommet courant possède au moins un arc sortant
    Pour chaque arc sortant du sommet courant
        creerChemins(Graphe,
                     Sommet pointé par l'arc,
                     (Chemin en cours) U {Sommet pointé par l'arc},
                     Liste des chemins
                    )
    Fin pour chaque
Sinon (le sommet est un puits)
    Ajouter le chemin en cours dans la liste des chemins trouvés
Fin Si
```

Figure 4.9 Algorithme pour générer tous les chemins d'un graphe orienté acyclique débutant à un certain sommet et se terminant par un puits (algorithme 2)

4.2.5 Algorithme pour générer la table de correspondance classe/fenêtre (algorithme 1.1.5)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de faire le lien entre une classe du modèle du programme et une fenêtre du modèle de l'interface graphique. Pour cette raison, une table faisant le lien entre les classes et les fenêtres est pré-calculée afin de faciliter le travail de l'algorithme. Cette table possède deux membres : le membre de gauche est un pointeur sur une classe et le membre de droite est un pointeur sur la

fenêtre. Chaque entrée de la table fait ainsi le lien entre la classe et la fenêtre. La figure 4.10 présente l'algorithme 1.1.5, qui génère cette table.

```
| Algorithme 1.1.5 - Génération de la table de correspondance classe/fenêtre |

++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Modèle du programme                   +
+         - Modèle de l'interface graphique        +
+                                                  +
+ Sortie : - Table de correspondance classe/fenêtre +
++++++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque classe (métaclasse Classe) du modèle du programme
    Si la classe est une fenêtre (voir explications)
        Pour chaque fenêtre (métaclasse Fenetre) du modèle de l'interface graphique
            Si le nom de la fenêtre est le même que le nom de la classe
                Ajouter une entrée dans la table de correspondance pour cette classe et cette fenêtre
            Fin si
        Fin pour chaque
    Fin si
Fin pour chaque
```

Figure 4.10 Algorithme pour générer la table de correspondance classe/fenêtre (algorithme 1.1.5)

Une boucle est effectuée sur chaque classe (métaclasse *Classe*) du modèle du programme. Pour chacune d'elles, l'algorithme 1.1.5 détermine s'il s'agit d'une fenêtre. La façon de déterminer si une classe est une fenêtre est de regarder si la classe hérite d'une classe en particulier. Cette superclasse requise est spécifique au langage de programmation utilisé pour écrire le programme. Par exemple, si le programme a été écrit avec le langage Java, cette superclasse doit être *Window* ou *JFrame*; s'il a été écrit avec un langage de l'environnement .NET, cette superclasse doit être *Form*. Si la classe observée est une fenêtre, une boucle est alors effectuée sur chacune des fenêtres (métaclasse *Fenetre*) du modèle de l'interface graphique afin de trouver la fenêtre correspondante. La fenêtre correspond à la classe si elles possèdent le même nom.

4.2.6 Algorithme pour générer la table de correspondance symbole/composant graphique (algorithme 1.1.6)

Pour faire son travail, l'algorithme de génération du modèle de traduction aura besoin de faire le lien entre un symbole du modèle du programme et un composant graphique du modèle de

l'interface graphique. Pour cette raison, une table faisant le lien entre les symboles et les composants graphiques est pré-calculée afin de faciliter le travail de l'algorithme. Cette table possède deux membres : le membre de gauche est un pointeur sur un symbole et le membre de droite est un pointeur sur le composant graphique. Chaque entrée de la table fait ainsi le lien entre le symbole et le composant graphique. La figure 4.11 présente l'algorithme 1.1.6, qui génère cette table.

```
-----------------------------------------------------------------------------------------------------
| Algorithme 1.1.6 - Génération de la table de correspondance symbole/composant graphique          |
-----------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Table de correspondance classe/fenêtre               +
+                                                                 +
+ Sortie : - Table de correspondance symbole/composant graphique +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque classe (métaclasse Classe) de la table de correspondance classe/fenêtre
    Obtenir la fenêtre (métaclasse Fenetre) correspondant à la classe dans la table de
    correspondance classe/fenêtre

    Pour chaque symbole (métaclasse Symbole) de la classe
        Pour chaque composant graphique (métaclasse ComposantGraphique) de la fenêtre
            Si le nom du symbole est le même que le nom du composant graphique
                Ajouter une entrée dans la table de correspondance pour ce symbole et ce composant
                graphique
            Fin si
        Fin pour chaque
    Fin pour chaque
Fin pour chaque
```

Figure 4.11 Algorithme pour générer la table de correspondance symbole/composant graphique (algorithme 1.1.6)

Une boucle est effectuée sur chaque classe (métaclasse *Classe*) (membre de gauche) de la table de correspondance classe/fenêtre construite par l'algorithme 1.1.5. Les classes observées seront donc nécessairement des fenêtres. Les symboles contenus à l'intérieur de cette classe sont donc potentiellement des composants graphiques. L'algorithme 1.1.6 a pour but d'identifier ces composants. Une boucle est effectuée sur chaque symbole (métaclasse *Symbole*) de la classe. Si le modèle de l'interface graphique contient, pour la fenêtre correspondant à la classe, un composant graphique (métaclasse *ComposantGraphique*) de même nom que le symbole, alors il s'agit du composant graphique correspondant au symbole, et une entrée pour ce couple est ajoutée à la table.

4.2.7 Algorithme de génération de la fonction de traduction d'une action (algorithme 1.2)

Une fois les structures de données pré-calculées, l'algorithme de génération du modèle de traduction doit générer une fonction de traduction pour chacune des actions du modèle d'action. Cette fonction en sortie est du code écrit dans le langage AEFMAP. Tous les détails de ce langage peuvent être trouvés dans [10]. La grammaire de celui-ci est présentée à la figure 4.12.

```
<mapping-model>      ::=   <mapping-function>|<mapping-function> < mapping-model >
<mapping-function>   ::=   <function-signature> "{" <function-body> "}"
<function-signature> ::=   <return-type> <function-name> "(" ( <param-list> | "" ) ")"
<param-list>         ::=   <param-type> <param-name> |
                               <param-type> <param-name> "," <param-list>
<function-body>      ::=   <event-map> <return-statement>
<event-map>          ::=   <event-execution> | <seq-generator> "{" <event-executions> "}"
<seq-generator>      ::=   "Serialize" | "Select" | "Permute"
<event-executions>   ::=   <event-execution> | <event-execution> <event-map> |
                               <event-map> <event-execution>
<event-execution>    ::=   <exe-keyword> "(" <event-name> "," <event-input> ")" ";"
<exe-keyword>        ::=   "Execute"| "ExecuteOp"
```

Figure 4.12 Grammaire du langage AEFMAP

Source : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 156

Le corps d'une fonction de traduction consiste à exécuter des événements sur l'interface graphique et à retourner une valeur, s'il y a lieu. L'exécution d'un événement de l'interface graphique s'effectue à l'aide de la fonction *Execute* ou *ExecuteOp*. Ces deux fonctions prennent comme premier paramètre le nom de l'événement à générer sous la forme *GUI.<Composant_Graphique>.<Événement>* et comme deuxième paramètre la valeur d'entrée pour cet événement, s'il y a lieu. La différence entre les fonctions *Execute* et *ExecuteOp* est que la fonction *ExecuteOp* indique que l'exécution de l'événement est optionnelle. Les fonctions d'exécution des événements doivent se trouver à l'intérieur de blocs de code permettant de contrôler l'ordonnancement de ces événements. Il existe trois opérateurs d'ordonnancement : *Serialize*, *Permute* et *Select*. L'opérateur *Serialize* indique que

tous les événements qu'il contient doivent être exécutés dans l'ordre où ils ont été écrits. L'opérateur *Permute* indique que tous les événements qu'il contient doivent être exécutés et qu'ils peuvent l'être dans n'importe quel ordre. L'opérateur *Select* indique qu'un seul des événements qu'il contient doit être exécuté. Il sert à représenter la situation où l'utilisateur a plusieurs façons différentes d'effectuer une action. Les opérateurs d'ordonnancement peuvent être imbriqués.

La figure 4.13 présente l'algorithme 1.2, qui génère la fonction de traduction d'une action.

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.2 - Génération de la fonction de traduction d'une action                               |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Action                                                +
+           - Table des chemins d'appel d'action                    +
+           - Table des séquences d'instructions                    +
+           - Table de correspondance classe/fenêtre                +
+           - Table de correspondance symbole/composant graphique   +
+                                                                   +
+ Sortie : - Fonction de traduction de l'action (texte)            +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Écrire l'en-tête de la fonction de traduction :
    - Écrire le type de retour de l'action, suivi d'une espace
    - Écrire le nom de l'action, suivi d'une parenthèse ouvrante
    - Écrire les paramètres de l'action :
          Pour chaque paramètre (métaclasse Parametre) de l'action
              Écrire le type du paramètre, suivi d'une espace

              Écrire le nom du paramètre

              Si ce n'est pas le dernier paramètre
                  Écrire une virgule, suivie d'une espace
              Fin si
          Fin pour chaque
    - Écrire une parenthèse fermante, suivie d'une accolade ouvrante et d'un saut de ligne

Obtenir la liste des chemins d'appel de l'action dans la table des chemins d'appel d'action

Si la liste des chemins d'appel contient plus d'un chemin
    Écrire "Select", suivi d'une accolade ouvrante et d'un saut de ligne
Fin si

Pour chaque chemin d'appel de l'action de la liste des chemins d'appel d'action
    Écrire le code de traduction du chemin (algorithme 1.2.1)
Fin pour chaque

Si la liste des chemins d'appel contient plus d'un chemin
    Écrire une accolade fermante, suivie d'un saut de ligne (fermeture du bloc "Select")
Fin si

Écrire une accolade fermante, suivie d'un saut de ligne (fermeture de la fonction)
```

Figure 4.13 Algorithme pour générer la fonction de traduction d'une action (algorithme 1.2)

L'algorithme 1.2 génère tout d'abord l'en-tête de la fonction de traduction, soit le type de retour de l'action, le nom de l'action ainsi que les paramètres de l'action. Par la suite, il génère du code de traduction pour chacun des chemins d'appel de l'action provenant de la table des chemins d'appel d'action construite par l'algorithme 1.1.4. Chaque chemin étant une façon différente de déclencher l'action, le code généré doit être contenu dans un bloc « *Select* » s'il y a plus d'un chemin. La génération du code de traduction de chacun des chemins est déléguée à l'algorithme 1.2.1, qui est présenté à la figure 4.14.

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.2.1 - Génération du code de traduction d'un chemin d'appel d'une action              |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Chemin d'appel                                    +
+           - Table des séquences d'instructions              +
+           - Table de correspondance classe/fenêtre          +
+           - Table de correspondance symbole/composant graphique +
+                                                             +
+ Sortie : - Code de traduction du chemin (texte)            +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Calculer la table paramètres/expressions (algorithme 1.2.1.1)

Initialiser la pile des fenêtres ouvertes à vide

Écrire "Serialize", suivi d'une accolade ouvrante et d'un saut de ligne

Pour chaque méthode (métaclasse Methode) du chemin d'appel
    Écrire le code de traduction de cette méthode (algorithme 1.2.1.2)
Fin pour chaque

Écrire une accolade fermante, suivie d'un saut de ligne (fermeture du bloc "Serialize")
```

Figure 4.14 Algorithme pour générer le code de traduction d'un chemin d'appel d'une action (algorithme 1.2.1)

L'algorithme 1.2.1 obtient tout d'abord la table paramètres/expressions relative à ce chemin d'appel de la part de l'algorithme 1.2.1.1. Cette table, construite par une analyse du modèle du programme, indique, pour chaque paramètre de l'action, toutes les valeurs possibles qu'il peut prendre. Ces valeurs peuvent être réelles ou symboliques dans le cas où la valeur réelle d'un symbole ne peut être déterminée. Afin d'éclaircir le contenu de cette table, la partie droite de la figure 4.15 présente un exemple de table paramètres/expressions pour le code source contenu dans la partie gauche de cette figure.

Le paramètre *p1* est un exemple des différentes catégories d'expression qu'un paramètre peut prendre comme valeur : une constante (15), une variable (*p*) (car la valeur de *p* n'est pas

connue dans ce code) ou une propriété d'un composant graphique (*ZoneDeTexte.Text*). Les expressions contenues dans la table paramètres/expressions doivent être le plus près possible de leur valeur réelle. Par exemple, le paramètre *p2* peut prendre comme valeur la variable *y*. Or, comme les valeurs possibles de la variable *y* sont connues, ce sont celles-ci qui doivent être stockées dans la table et non le symbole de la variable. Le paramètre *p3* est un autre bon exemple de cela, car sa valeur est celle de la variable *z* et la valeur de celle-ci est le résultat de la fonction *f2*. Encore une fois, ce n'est pas le symbole de cette fonction qui doit être stocké dans la table, mais plutôt toutes les valeurs de retour possibles de celle-ci. L'algorithme 1.2.1.1, qui construit la table paramètres/expressions, sera présenté à la section 4.2.11.

```
void f1(int p) {
    int i;
    int x;
    int y;
    string z;

    i = 11;

    if (p < 5) {
        x = 15;
    } else if (p < 10) {
        x = p;
    } else {
        x = (int) ZoneDeTexte.Text;
    }

    if (x < 7) {
        y = i;
    } else {
        y = 88;
    }

    z = f2(x);

    action1(x, y, z);
}
```

```
int f2(int x) {
    string s;

    if (x < 14) {
        s = ListeDeroulante.SelectedItem;
    } else {
        s = Liste.SelectedItem;
    }

    return s;
}

void action1(int p1, int p2, string p3) {
}
```

Table paramètres/expressions :

Paramètres	Expressions
p1	15 p ZoneDeTexte.Text
p2	11 88
p3	ListeDeroulante.SelectedItem Liste.SelectedItem

Figure 4.15 Exemple de table paramètres/expressions

Pour revenir à l'algorithme 1.2.1, une fois qu'il a obtenu la table paramètres/expressions, il initialise la pile des fenêtres ouvertes à vide. Cette pile garde une trace de toutes les fenêtres qui ont été ouvertes et sert à connaître la fenêtre actuellement active (celle au sommet de la pile) et, lorsque celle-ci se fermera, celle qui deviendra active (la deuxième à partir du sommet de la pile). Par la suite, l'algorithme parcourt chaque méthode (métaclasse *Methode*) du chemin d'appel et génère du code de traduction spécifique à chacune des méthodes. Puisque les instructions générées devront être exécutées dans l'ordre, elles sont contenues dans un bloc

« *Serialize* ». La génération du code de traduction de chacune des méthodes est déléguée à l'algorithme 1.2.1.2, qui est présenté à la figure 4.16.

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.2.1.2 - Génération du code de traduction d'une méthode d'un chemin d'appel d'une      |
|                      action                                                                       |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Méthode                                               +
+          - Méthode suivante dans le chemin                       +
+          - Table des séquences d'instructions                    +
+          - Table de correspondance classe/fenêtre                +
+          - Table de correspondance symbole/composant graphique   +
+          - Table paramètres/expressions                          +
+          - Pile des fenêtres ouvertes                            +
+                                                                  +
+ Sortie : - Code de traduction de la méthode (texte)              +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Obtenir la fenêtre (métaclasse Fenetre) à laquelle la méthode appartient (s'il y en a une)
(voir explications)

Si la pile des fenêtres ouvertes est vide
    Empiler la fenêtre à laquelle la méthode appartient dans la pile des fenêtres ouvertes
Fin si

Effectuer le traitement des paramètres (algorithme 1.2.1.2.1)
Effectuer le traitement des gestionnaires d'événement (algorithme 1.2.1.2.2)
Tenir compte de l'ouverture des fenêtres (algorithme 1.2.1.2.3)
```

Figure 4.16 Algorithme pour générer le code de traduction d'une méthode d'un chemin d'appels d'une action (algorithme 1.2.1.2)

La première étape de l'algorithme 1.2.1.2 est d'initialiser la pile des fenêtres ouvertes si celle-ci est vide. La fenêtre initiale est celle à laquelle la première méthode du chemin d'appel appartient. Pour déterminer cette fenêtre, il faut regarder la classe (métaclasse *Classe*) à laquelle la méthode appartient puis, à l'aide de la table de correspondance classe/fenêtre construite par l'algorithme 1.1.5, on obtient la fenêtre (métaclasse *Fenetre*). Il est à noter que la première méthode du chemin d'appel appartient nécessairement à une fenêtre, car, tel qu'indiqué à la section 4.2.4, celle-ci est un gestionnaire d'événement et les méthodes qui sont des gestionnaires d'événement se trouvent toujours dans la classe de la fenêtre contenant le composant graphique qui déclenche l'événement.

Les deuxième et troisième étapes de l'algorithme 1.2.1.2 sont d'interagir avec l'interface graphique en fonction des paramètres de l'action et de déclencher les événements de l'interface graphique afin d'exécuter l'action. Les algorithmes 1.2.1.2.1 (section 4.2.8) et

1.2.1.2.2 (section 4.2.9) s'acquittent respectivement de cette tâche. De plus, l'algorithme 1.2.1.2 doit tenir compte des fenêtres qui sont ouvertes par le code afin de garder la pile des fenêtres ouvertes à jour. Cette tâche revient à l'algorithme 1.2.1.2.3 (section 4.2.10).

4.2.8 Algorithme pour le traitement des paramètres d'une action (algorithme 1.2.1.2.1)

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.2.1.2.1 - Algorithme pour le traitement des paramètres d'une action                    |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Table de correspondance classe/fenêtre                  +
           - Table de correspondance symbole/composant graphique     +
           - Table paramètres/expressions                            +
           - Fenêtre présentement active                             +
+                                                                    +
+ Sortie : - Une partie du code de traduction de la méthode (texte) +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Déterminer les paramètres/expressions à traiter :
    Pour chaque paramètre (métaclasse Parametre) de la table paramètres/expressions
        Pour chaque expression (métaclasse Expression) que le paramètre peut prendre comme valeur
            Si l'expression (métaclasse ExpressionSymbolique) est une propriété d'un composant
            graphique (voir explications)
                Si la fenêtre (métaclasse Fenetre) de ce composant graphique est la fenêtre
                présentement active
                    Marquer ce paramètre/expression comme étant à être traité
                Fin si
            Fin si
        Fin pour chaque
    Fin pour chaque

Si le nombre de paramètres à traiter est supérieur à 1
    Écrire "Permute", suivi d'une accolade ouvrante et d'un saut de ligne
Fin si

Pour chaque paramètre (métaclasse Parametre) à traiter
    Si le nombre d'expressions à traiter pour ce paramètre est supérieur à 1
        Écrire "Select", suivi d'une accolade ouvrante et d'un saut de ligne
    Fin si

    Pour chaque expression (métaclasse Expression) à traiter de ce paramètre
        Si l'expression est la propriété "Text" d'une zone de texte (métaclasse ZoneDeTexte)
            Écrire "Execute(GUI.<nom_zone_de_texte>.type, <nom_du_paramètre>);", suivi d'un saut
            de ligne
        Sinon
            Tous les autres cas possibles doivent se trouver ici...
        Fin si
    Fin pour chaque

    Si le nombre d'expressions traitées pour ce paramètre est supérieur à 1
        Écrire une accolade fermante, suivie d'un saut de ligne (fermeture du bloc "Select")
    Fin si
Fin pour chaque

Si le nombre de paramètres traités est supérieur à 1
    Écrire une accolade fermante, suivie d'un saut de ligne (fermeture du bloc "Permute")
Fin si

Supprimer, de la table paramètres/expressions, les paramètres/expressions qui ont été traités
```

Figure 4.17 Algorithme pour le traitement des paramètres d'une action (algorithme 1.2.1.2.1)

Dans le cadre action-événement, chaque paramètre d'une action correspond à une valeur fournie par l'utilisateur à l'application. Le modèle de traduction indique comment l'utilisateur utilise l'interface graphique pour communiquer cette information à l'application. Par exemple, le modèle de traduction présenté à la figure 1.4 du chapitre 1 indique que le paramètre *i* de l'action *edittask* représente l'élément que l'utilisateur souhaite sélectionner et qu'il le fait en choisissant un élément dans l'arbre *tree*; le paramètre *name* représente le nouveau nom de la tâche et l'utilisateur communique cette information en tapant le nom dans la zone de texte *textboxNAME*. L'algorithme de génération du modèle de traduction doit déterminer automatiquement ces moyens de communication. C'est ce que fait l'algorithme 1.2.1.2.1, qui est présenté à la figure 4.17.

L'algorithme 1.2.1.2.1 utilise la table paramètres/expressions pour déterminer quel(s) composant(s) graphique(s) permettent à l'utilisateur de communiquer la valeur des paramètres à l'application. Il commence tout d'abord par déterminer les paramètres/expressions qu'il sera capable de traiter au cours de cette itération. Ceux-ci sont ceux dont l'expression est une propriété d'un composant graphique et dont celui-ci se trouve dans la fenêtre présentement active. La façon de déterminer si une expression est une propriété d'un composant graphique est de regarder s'il existe une entrée pour l'instance de cette expression symbolique (métaclasse *ExpressionSymbolique*) dans la table de correspondance symbole/composant graphique construite par l'algorithme 1.1.6. Les paramètres/expressions non retenus seront traités lors d'une prochaine itération, lorsque la fenêtre dans laquelle ils se trouvent sera active. Si plus d'un paramètre peut être traité au cours de cette itération, le code généré doit être dans un bloc « *Permute* », car, dans une interface graphique, l'utilisateur a le choix de l'ordre dans lequel il entre les informations. Par exemple, dans la figure 1.4 du chapitre 1, l'écriture dans la zone de texte *textboxNAME* et la sélection d'un élément dans la liste déroulante *comboboxPROGRESS* se situent dans un bloc « *Permute* », car l'utilisateur peut d'abord écrire dans la zone de texte, puis choisir un élément dans la liste déroulante, ou faire l'inverse. Si un paramètre peut prendre plusieurs valeurs différentes, autrement dit s'il y a plus d'une expression dans la table paramètres/expressions pour ce paramètre, le code généré pour ce paramètre doit être dans un bloc « *Select* ». Cette situation signifie qu'il existe plusieurs

façons différentes de communiquer la valeur du paramètre et qu'une seule d'entre elles doit être choisie lors d'une exécution du logiciel. Par exemple, dans l'application de gestion de tâches, l'utilisateur pourrait spécifier l'état d'avancement à l'aide d'une liste déroulante contenant une liste de valeurs prédéfinies ou, si la valeur souhaitée n'en fait pas partie, taper une valeur supplémentaire dans une zone de texte. L'emploi d'un bloc « *Select* » aura pour effet de générer plusieurs scénarios de test différents par le cadre action-événement afin de refléter les différentes possibilités. Une fois les blocs de contrôle créés, le code affectant la valeur des paramètres à une propriété d'un composant graphique est généré. Ce code est la fonction *Execute* et ses paramètres dépendent de la propriété et du composant. Par exemple, pour la propriété *Text* d'une zone de texte (métaclasse *ZoneDeTexte*), ce code est « *Execute(GUI.<nom_zone_de_texte>.type, <nom_du_paramètre>);* », où « *type* » signifie taper (écrire) du texte. Le code requis pour les autres types de composant se trouve dans [10]. Une fois que tous les paramètres/expressions pouvant être traités au cours de cette itération l'ont été, ceux-ci sont supprimés de la table paramètres/expressions pour ne pas qu'ils soient traités à nouveau lors d'une prochaine itération.

4.2.9 Algorithme pour le traitement des gestionnaires d'événement (algorithme 1.2.1.2.2)

En plus d'indiquer le moyen de communication des paramètres à l'application, le modèle de traduction doit déclencher les événements de l'interface graphique qui permettent d'exécuter l'action. C'est ce que fait l'algorithme 1.2.1.2.2, qui est présenté à la figure 4.18.

Les événements à déclencher sont ceux pour lesquels il existe une méthode (métaclasse *Methode*) qui est gestionnaire de cet événement dans le chemin d'appel de l'action. L'algorithme 1.2.1.2.2 est appelé pour chaque méthode du chemin d'appel. Si la méthode n'est pas un gestionnaire d'événement, l'algorithme ne fait rien. Sinon, il obtient le composant graphique (métaclasse *ComposantGraphique*) qui déclenche l'événement en cherchant le symbole déclencheur dans la table de correspondance symbole/composant graphique construite par l'algorithme 1.1.6. Une fois le composant trouvé, il écrit le code permettant de

déclencher cet événement. Celui-ci est de la forme « *Execute(GUI.<nom_du_composant>.<nom_événement>);* ». La figure 4.18 montre un exemple de code pour l'événement « *click* » d'un bouton (métaclasse *Bouton*). Un traitement spécial doit cependant être fait pour les composants se trouvant à l'intérieur d'autres composants : les composants supérieurs doivent préalablement être activés. Par exemple, pour l'interface graphique de la figure 3.7 du chapitre 3, si on veut cliquer sur l'élément de menu 1-1, il faut d'abord cliquer sur le menu principal, puis cliquer sur l'élément de menu 1.

```
-------------------------------------------------------------------------------------------------------
| Algorithme 1.2.1.2.2 - Algorithme pour le traitement des gestionnaires d'événement                  |
-------------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Méthode                                                 +
           - Table de correspondance symbole/composant graphique    +
+                                                                    +
+ Sortie : - Une partie du code de traduction de la méthode (texte) +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Si la méthode (métaclasse Methode) est un gestionnaire d'événement
    Obtenir le composant graphique (métaclasse ComposantGraphique), dans la table de correspondance
    symbole/composant graphique, qui déclenche cet événement (voir explications)

    Si le composant graphique est un bouton (métaclasse Bouton)
        Si l'événement est un clic
            Écrire "Execute(GUI.<nom_du_bouton>.click);", suivi d'un saut de ligne
        Sinon
            Tous les autres cas possibles doivent se trouver ici...
        Fin si
    Sinon
        Tous les autres cas possibles doivent se trouver ici...
    Fin si
Fin si
```

Figure 4.18 Algorithme pour le traitement des gestionnaires d'événement (algorithme 1.2.1.2.2)

4.2.10 Algorithme pour la prise en charge des fenêtres ouvertes (algorithme 1.2.1.2.3)

Tout au long du parcours des méthodes, l'algorithme de génération du modèle de traduction doit garder la pile des fenêtres ouvertes à jour. C'est ce que fait l'algorithme 1.2.1.2.3, qui est présenté à la figure 4.19.

```
| Algorithme 1.2.1.2.3 - Algorithme pour la prise en charge des fenêtres ouvertes |

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Méthode                                                  +
+          - Méthode suivante dans le chemin                         +
+          - Fenêtre de la méthode                                   +
+          - Table des séquences d'instructions                      +
+          - Table de correspondance classe/fenêtre                  +
+          - Table de correspondance symbole/composant graphique     +
+          - Table paramètres/expressions                            +
+          - Pile des fenêtres ouvertes                              +
+                                                                    +
+ Sortie : - Pile des fenêtres ouvertes modifiée                     +
+          - Une partie du code de traduction de la méthode (texte) +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Pour chaque séquence d'instructions de la méthode
    Si la méthode est la dernière méthode du chemin ou l'appel à la méthode suivante dans le
    chemin se trouve dans la séquence d'instructions actuelle
        Pour chaque instruction (métaclasse Instruction) de la séquence d'instructions
            Si l'instruction est un appel de méthode (métaclasse ExpressionAppelMethode)
                Si l'appel de méthode est une ouverture de fenêtre (voir explications)
                    Empiler la nouvelle fenêtre ouverte dans la pile des fenêtres ouvertes
                Sinon si l'appel de méthode est une fermeture de fenêtre (voir explications)
                    Dépiler la fenêtre
                Sinon si l'appel de méthode n'est pas l'appel à la méthode suivante dans le chemin
                    Tenir compte de l'ouverture de fenêtres pour la méthode appelée
                    (appel récursif à cet algorithme)
                Sinon (l'appel de méthode est l'appel à la méthode suivante dans le chemin)
                    Mettre fin à cet algorithme
                Fin si

                Effectuer le traitement des paramètres (algorithme 1.2.1.2.1)

                Si la fenêtre présentement active est modale
                    Si la fenêtre présentement active contient un bouton (métaclasse Bouton) de nom
                    "btnOK" (voir explications)
                        Fermer la fenêtre à l'aide du bouton OK :
                            - Écrire "Execute(GUI.btnOK.click);", suivi d'un saut de ligne
                    Sinon
                        Fermer la fenêtre à l'aide de son bouton de fermeture :
                            - Écrire "Execute(GUI.close);", suivi d'un saut de ligne
                    Fin si

                    Dépiler la fenêtre présentement active de la pile des fenêtres ouvertes
                Fin si
            Fin si
        Fin pour chaque
    Fin si
Fin pour chaque
```

Figure 4.19 Algorithme pour la prise en charge des fenêtres ouvertes (algorithme 1.2.1.2.3)

L'algorithme 1.2.1.2.3 est appelé pour chaque méthode (métaclasse *Methode*) du chemin d'appel. Il parcourt chaque séquence d'instructions de la méthode afin d'en trouver une faisant partie du chemin d'appel en cours (l'appel à la méthode suivante dans le chemin doit se trouver dans cette séquence d'instructions). Afin de simplifier grandement l'algorithme, une seule des séquences candidates sera explorée. Lors de la conception de cet algorithme, il a donc été supposé que chacune d'entre elles aurait donné le même résultat, c'est-à-dire que les mêmes fenêtres auraient été ouvertes ou fermées. L'algorithme parcourt chaque instruction

(métaclasse *Instruction*) de la séquence d'instructions retenue et analyse les appels de méthode (métaclasse *ExpressionAppelMethode*) trouvés. Si l'appel de méthode est une ouverture de fenêtre, alors la nouvelle fenêtre doit être empilée dans la pile des fenêtres ouvertes. Un appel de méthode est une ouverture de fenêtre si l'instance sur laquelle elle est appelée est une fenêtre et que la méthode appelée possède un nom particulier, spécifique au langage de programmation utilisé pour écrire le programme. Par exemple, si le programme a été écrit avec le langage Java, cette méthode est *setVisible*; s'il a été écrit avec un langage de l'environnement .NET, cette méthode est *Show* ou *ShowDialog*. Si l'appel de méthode est une fermeture de fenêtre, alors la fenêtre doit être dépilée. Si l'appel de méthode est l'appel à la méthode suivante dans le chemin, alors l'algorithme 1.2.1.2.3 doit prendre fin, car la suite sera prise en charge la prochaine fois que l'algorithme 1.2.1.2.3 sera appelé, c'est-à-dire lors de la prochaine itération de l'algorithme 1.2.1. Si l'appel de méthode n'est pas l'un de ces trois cas de figure, alors il s'agit d'un appel de méthode ordinaire et cette méthode doit à son tour être explorée, car elle peut elle aussi ouvrir ou fermer des fenêtres. Pour cette raison, l'algorithme 1.2.1.2.3 doit être appelé récursivement pour cette méthode. Une fois que le traitement approprié a été effectué selon le type d'appel de la méthode, l'algorithme 1.2.1.2.1, qui a été présenté à la section 4.2.8, doit encore une fois être appelé afin d'effectuer le traitement des paramètres de l'action. Ce travail doit être refait, car, si une nouvelle fenêtre a été ouverte, il est possible que de nouveaux paramètres puissent être traités. Pour terminer l'algorithme, un traitement spécial doit être fait pour les fenêtres modales. Dans une interface graphique, une fenêtre modale est une fenêtre qui, tant qu'elle est ouverte, bloque l'accès aux autres fenêtres. Elles servent à demander à l'utilisateur de l'information qu'il est impératif de connaître avant de poursuivre. Une fois les informations entrées, l'utilisateur doit fermer la fenêtre. C'est ce que la dernière partie de l'algorithme 1.2.1.2.3 fait. Lors de la conception de cet algorithme, il a été décidé d'imposer une convention sur la façon de fermer cette fenêtre. Le choix qui a été fait est de supposer qu'un bouton (métaclasse *Bouton*) de nom *btnOK* se trouve dans la fenêtre. Celui-ci doit faire le traitement approprié, par exemple sauvegarder les valeurs entrées dans des variables, et fermer la fenêtre. Si aucun traitement particulier ne doit être fait, alors ce bouton peut être omis. La fenêtre est alors tout simplement fermée à l'aide de son bouton de fermeture.

4.2.11 Algorithme pour générer la table paramètres/expressions (algorithme 1.2.1.1)

Cette section décrit la dernière partie de l'algorithme de génération du modèle de traduction qui n'a pas encore été présentée, soit la génération de la table paramètres/expressions. Cette table a été décrite à la section 4.2.7 et un exemple a été présenté à la figure 4.15. Ce sous-algorithme est présenté aux figures 4.20, 4.21 et 4.22.

```
-----------------------------------------------------------------------------------------------------
| Algorithme 1.2.1.1 - Génération de la table paramètres/expressions relative à un chemin           |
-----------------------------------------------------------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Chemin d'appel                    +
+         - Table des séquences d'instructions +
+                                              +
+ Sortie : - Table paramètres/expressions      +
++++++++++++++++++++++++++++++++++++++++++++++++

Initialiser la table paramètres/expressions :
    Pour chaque paramètre (métaclasse Parametre) de la dernière méthode (métaclasse Methode) du
    chemin
        Ajouter une entrée dans la table paramètres/expressions avec, dans la partie gauche, le
        paramètre et, dans la partie droite, une liste d'expressions vide
    Fin pour chaque

Pour chaque méthode (métaclasse Methode) du chemin, à partir de l'avant-dernière, en sens inverse
    Pour chaque séquence d'instructions de la méthode
        Si l'appel à la méthode suivante (métaclasse ExpressionAppelMethode) dans le chemin est
        dans la séquence d'instructions actuelle
            Pour chaque instruction (métaclasse Instruction) de la séquence d'instructions, en sens
            inverse
                Si l'appel à la méthode suivante (métaclasse ExpressionAppelMethode) dans le chemin
                n'a pas encore été trouvé
                    Effectuer le traitement approprié à cette situation (algorithme 1.2.1.1.1)
                Sinon
                    Effectuer le traitement approprié à cette situation (algorithme 1.2.1.1.2)
                Fin si
            Fin pour chaque
        Fin si
    Fin pour chaque
Fin pour chaque

Nettoyer la table paramètres/expressions :
    - Éliminer les doublons
    - Éliminer les paramètres/expressions pour lesquelles l'expression (métaclasse Expression) ne
      provient pas d'un composant graphique (métaclasse ComposantGraphique)
```

Figure 4.20 Algorithme pour générer la table paramètres/expressions (algorithme 1.2.1.1)

```
| Algorithme 1.2.1.1.1 - Partie de l'algorithme de génération de la table paramètres/expressions  |
|                        qui gère le cas où l'appel à la méthode suivante dans le chemin n'a pas  |
|                        encore été trouvé                                                        |
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Instruction                          +
+          - Méthode suivante dans le chemin      +
+          - Table paramètres/expressions         +
+                                                 +
+ Sortie : - Table paramètres/expressions modifiée +
+++++++++++++++++++++++++++++++++++++++++++++++++++

Si l'instruction (métaclasse Instruction) contient l'appel à la méthode suivante (métaclasse
ExpressionAppelMethode) dans le chemin
    Pour chaque paramètre (métaclasse Parametre) de la méthode (métaclasse Methode) suivante dans
    le chemin
        Si le paramètre formel est dans la partie gauche de la table
            Ajouter le paramètre effectif dans la liste des expressions que ce paramètre peut
            prendre comme valeur (partie droite de la table)
        Sinon si le paramètre formel est dans une des listes de la partie droite de la table
            Remplacer le paramètre formel par le paramètre effectif dans cette liste (partie
            droite de la table)
        Fin si
    Fin pour chaque
Fin si
```

Figure 4.21 Algorithme pour générer la table paramètres/expressions (suite) (algorithme 1.2.1.1.1)

```
| Algorithme 1.2.1.1.2 - Partie de l'algorithme de génération de la table paramètres/expressions  |
|                        qui gère le cas où l'appel à la méthode suivante dans le chemin a déjà   |
|                        été trouvé                                                               |
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++
+ Entrée : - Instruction                          +
+          - Table paramètres/expressions         +
+                                                 +
+ Sortie : - Table paramètres/expressions modifiée +
+++++++++++++++++++++++++++++++++++++++++++++++++++

Déterminer si l'instruction est une affectation ou un appel de méthode :
    Si l'instruction est une expression complexe (métaclasse ExpressionComplexe) et que son
    opérateur est l'affectation
        L'instruction est une affectation
    Sinon si l'instruction est une déclaration de symbole (métaclasse InstructionDeclarationSymbole)
    et qu'elle contient une initialisation
        L'instruction est une affectation
    Sinon si l'instruction est une expression d'appel de méthode (métaclasse ExpressionAppelMethode)
        L'instruction est un appel de méthode
    Fin si

Si l'instruction est une affectation
    Remplacer le membre de gauche de l'affectation dans la partie droite de la table par tous les
    sous-membres de la partie droite de l'affectation
Sinon si l'instruction est un appel de méthode
    Pour chaque paramètre (métaclasse Parametre) de la méthode appelée
        Remplacer les paramètres effectifs passés par référence par la valeur qui leur est affectée
        dans la méthode
        (même principe que l'algorithme 1.2.1.1, mais où le parcours des séquences se fait de la
        première à la dernière instruction)
    Fin pour chaque
Fin si
```

Figure 4.22 Algorithme pour générer la table paramètres/expressions (suite) (algorithme 1.2.1.1.2)

L'algorithme 1.2.1.1 commence par initialiser la table paramètres/expressions en ajoutant une entrée pour chacun des paramètres (métaclasse *Parametre*) de la dernière méthode (métaclasse *Methode*) du chemin. Celle-ci est la méthode pour laquelle on veut connaître la source des paramètres. Cette information se détermine en parcourant le code en sens inverse, car, tel que mentionné à la section 4.2.7, les expressions trouvées doivent être le plus près possible de leur valeur originale. Par exemple, à la figure 4.15, le paramètre *p1* prend la variable *x* comme valeur. Or, un peu plus tôt dans le code, la valeur 15 est affectée à la variable *x*. Au moment de découvrir cette instruction, la variable *x* devra donc être remplacée par 15 dans les valeurs possibles au paramètre *p1*.

Puisque le code doit être parcouru en sens inverse, la boucle sur chacune des méthodes (métaclasse *Methode*) du chemin se fait en sens inverse. Celle-ci débute à l'avant-dernière méthode du chemin, car ce qui est recherché sont les valeurs données aux paramètres lors de l'appel à la dernière méthode du chemin. Dans le même ordre d'idée, les instructions (métaclasse *Instruction*) de chaque séquence d'instructions des méthodes sont parcourues en sens inverse. Dans ce parcours, l'algorithme ne fait rien tant que l'appel à la méthode suivante (métaclasse *ExpressionAppelMethode*) dans le chemin n'a pas été rencontré, car les affectations qui suivent cet appel n'influenceront pas ce qui lui est passé en paramètre. Lorsque l'appel à cette méthode est trouvé, le traitement réel débute. Ce traitement consiste à ajouter les valeurs passées en paramètre à la méthode (les paramètres effectifs) comme valeurs potentielles aux paramètres formels (un paramètre formel est le symbole représentant le paramètre à l'intérieur du corps d'une méthode). Dans le cas où le paramètre formel se trouve dans la partie gauche de la table, le paramètre effectif est ajouté dans la partie droite de la table (liste des valeurs possibles de ce paramètre). Cette situation correspond au cas où l'appel de méthode trouvé est celui de la dernière méthode du chemin et ce sont ses paramètres qui sont dans la partie gauche de la table. Dans le cas où le paramètre formel se trouve dans une des listes de la partie droite de la table, celui-ci est remplacé par le paramètre effectif. Cette situation correspond au cas où le paramètre effectif passé à l'appel d'une méthode est le paramètre formel d'une autre méthode. Il s'agit du seul cas où un paramètre formel peut se retrouver dans la partie droite de la table.

Une fois que l'appel à la méthode suivante dans le chemin a été trouvé, la suite du parcours inverse des instructions sert à remplacer les valeurs contenues dans la table lorsque celles-ci changent. Il existe deux types d'instruction qui peuvent modifier une variable : une affectation (une affectation ordinaire (métaclasse *ExpressionComplexe* avec un opérateur d'affectation) ou une déclaration de symbole (métaclasse *InstructionDeclarationSymbole*) suivie d'une initialisation) ou un appel de méthode (métaclasse *ExpressionAppelMethode*) avec passage de paramètre(s) par référence. La première étape du sous-algorithme 1.2.1.1.2 est de déterminer si l'instruction actuellement observée est l'une de ces situations. S'il s'agit d'une affectation, le symbole auquel une valeur est affectée est remplacé, dans la partie droite de la table, par la valeur qui lui est affectée. S'il s'agit d'un appel de méthode, les instructions des séquences d'instructions de celle-ci sont explorées, de la première à la dernière instruction, afin de trouver des affectations aux paramètres passés par référence et d'insérer les valeurs affectées dans la partie droite de la table.

La dernière étape de l'algorithme 1.2.1.1 est de nettoyer la table paramètres/expressions qu'il a construite. Il se peut en effet que des doublons se trouvent dans la table, car le parcours de chacune des séquences d'instructions des méthodes implique qu'une même instruction peut être visitée plus d'une fois et donc que les valeurs affectées aient été insérées plus d'une fois dans la table. De plus, les expressions qui ne proviennent pas d'un composant graphique (métaclasse *ComposantGraphique*) sont éliminées, car l'algorithme de génération du modèle de traduction ne s'intéresse qu'à celles qui proviennent d'un composant graphique.

Chapitre 5

Validation de l'algorithme

Ce chapitre décrit le travail qui a été fait afin de valider l'algorithme présenté au chapitre 4.

5.1 Approche de validation

Tableau 5.1 Grille d'évaluation de l'équivalence de deux modèles de traduction

	Critère	Critère rencontré ?
1	Présence d'une fonction de traduction pour chacune des actions	
2	Présence de l'en-tête de la fonction de traduction (type de retour, nom, types et noms des paramètres)	
3	Présence des mêmes instructions	
4	Présence d'un bloc « *Select* » lorsqu'il existe un choix à faire entre plusieurs groupes d'instructions	
5	Présence d'un bloc « *Serialize* » contenant les instructions devant être exécutées séquentiellement, s'il y a lieu	
6	Présence d'un bloc « *Permute* » lorsque l'ordre de plusieurs instructions peut être permuté	
7	Présence de code affectant la valeur de chacun des paramètres à la propriété d'un composant graphique	
8	Présence de code déclenchant l'événement débutant l'action	
9	Présence de code fermant les fenêtres modales lorsque leur utilisation est terminée, s'il y a lieu	

L'algorithme de génération du modèle de traduction développé dans le cadre de cet essai a été testé sur trois études de cas. Celles-ci sont présentées en détails aux annexes 1, 2 et 3. Elles ont été inspirées des exemples présentés dans les articles décrivant le cadre action-événement ([10] et [11]). Ces exemples n'ont pu être repris tels quels, puisque leur code source n'est pas disponible. De plus, ils ont été complexifiés afin de tester l'algorithme plus en profondeur.

Pour chacune des études de cas, un modèle de traduction a été écrit manuellement. La validation consiste donc à vérifier que le modèle de traduction obtenu en sortie de l'algorithme est équivalent à celui écrit manuellement. Cependant, puisqu'un modèle de traduction est du code écrit dans un langage informatique, il est difficile d'évaluer l'équivalence de deux modèles. En effet, deux programmes peuvent être syntaxiquement très différents tout en étant sémantiquement égaux. Afin de déterminer l'équivalence de deux modèles de traduction, une grille d'évaluation a été mise sur pied (tableau 5.1). Les critères de cette grille sont la présence et l'agencement des principaux éléments d'un modèle de traduction, sans égard à la forme avec laquelle il a été écrit.

5.2 Couverture des tests

Les trois études de cas n'ont évidemment pas pu couvrir toutes les facettes d'un algorithme aussi complexe, malgré qu'une attention particulière ait été apportée afin d'en couvrir le plus possible. Le tableau 5.2 résume les aspects de l'algorithme qui ont été couverts par les tests et ceux qui ne l'ont pas été.

5.3 Évaluation des résultats

L'analyse des résultats des trois études de cas fait ressortir que l'algorithme correspond très bien aux critères du tableau 5.1. Cependant, un problème a été identifié. En effet, lorsque l'action ne peut être déclenchée à partir de la fenêtre principale (comme c'est le cas dans les exemples 2 et 3), l'algorithme omet les instructions qui ouvrent la fenêtre permettant de lancer l'action.

L'étude de cas 3 a montré que l'emploi de boîtes de dialogue nécessitant une action de la part de l'utilisateur, telle que répondre à une question de type « oui / non », cause problème. Il ne s'agit cependant pas d'une faille de l'algorithme, car il n'est carrément pas possible de gérer cette situation avec le langage AEFMAP, puisque celui-ci n'a pas d'instruction conditionnelle.

Tableau 5.2 Couverture de l'algorithme par les études de cas

Algorithme	Couverture	Commentaire
1	Complète	
1.1.1	Complète	
1.1.1.1	Complète	
1.1.2	Complète	
1.1.3	Complète	
1.1.4	Complète	
1.1.5	Complète	
1.1.6	Complète	
1.2	Complète	
1.2.1	Complète	
1.2.1.1	Complète	
1.2.1.1.1	Partielle	Cet algorithme n'est pas complètement couvert, car les études de cas ne contiennent pas la situation où le paramètre formel d'une méthode est passé comme paramètre effectif de l'appel d'une autre méthode.
1.2.1.1.2	Partielle	Cet algorithme n'est pas complètement couvert, car les études de cas ne contiennent pas d'affectation d'une expression complexe à un symbole.
1.2.1.2	Complète	
1.2.1.2.1	Partielle	Cet algorithme n'est pas complètement couvert, car les études de cas n'utilisent pas tous les types de composant graphique possibles, ni toutes les propriétés possibles.
1.2.1.2.2	Partielle	Cet algorithme n'est pas complètement couvert, car les études de cas n'utilisent pas tous les types de composant graphique possibles, ni tous les événements possibles.
1.2.1.2.3	Complète	
2	Complète	

Le dernier constat est que le code généré, bien que bon, pourrait être optimisé. Par exemple, pour l'action *ajouterTache* de l'étude de cas 1, deux blocs « *Serialize* » sont contenus dans un bloc « *Select* » (figure A1.10). Seul le début de ces blocs varie. La manière optimale d'écrire cela est celle préconisée par le modèle de traduction attendu (figure A1.8), où seules les

différences ont été mises dans un bloc « *Select* ». Le contenu similaire vient par la suite, sans redondance.

5.4 Limites de l'approche de validation

Cette approche de validation comporte deux limites. Premièrement, la grille d'évaluation (tableau 5.1) n'a pas été validée. Pour s'assurer qu'elle est bonne, plusieurs tests, où l'on compare des modèles rédigés par plusieurs programmeurs, devraient être conduits. Deuxièmement, les modèles de traduction attendus et les évaluations ont été réalisés par la personne qui a conçu l'algorithme. Il est possible que, inconsciemment, la connaissance de l'algorithme influence la rédaction de la solution attendue. Pour assurer une meilleure objectivité de la validation, celle-ci devrait être faite par une autre personne.

Conclusion

Cet essai avait pour but de développer un algorithme générant automatiquement le modèle de traduction d'une application testée avec le cadre action-événement, afin d'éliminer la maintenance que ce dernier requiert suite à des changements dans l'apparence de l'interface graphique. Malgré certaines limites, il a accompli cet objectif avec succès.

Contributions

Cet essai a défini de façon précise le métamodèle des deux intrants du cadre action-événement, en plus de définir un troisième intrant nécessaire à l'algorithme. Les premier et troisième métamodèles, ceux du modèle d'action et du modèle du programme, permettent de représenter du code informatique écrit dans un langage orienté objet. Le deuxième métamodèle, celui du modèle de l'interface graphique, permet de lister les composants et les types des composants d'une interface graphique, ainsi que d'exprimer les liens hiérarchiques qu'ils ont entre eux.

Il a été déterminé que l'algorithme développé pouvait difficilement traiter les modèles en entrée. Des structures de données intermédiaires, ainsi que les algorithmes permettant de les construire, ont été conçus. Ces structures sont des tables permettant d'associer des éléments communs entre deux modèles (table de correspondance action/méthode, table de correspondance classe/fenêtre et table de correspondance symbole/composant graphique) ou encore des structures faisant ressortir les éléments essentiels du modèle du programme (table des séquences d'instructions, graphe d'appels de méthodes, table des chemins d'appel d'action et table paramètres/expressions).

L'algorithme principal analyse ces structures de données intermédiaires afin de générer le modèle de traduction. Il parcourt les chemins pertinents trouvés dans le graphe d'appels de

méthodes et simule les actions qu'aurait entreprises l'utilisateur en s'aidant, notamment, d'une table indiquant le composant graphique source de chacun des paramètres.

Limites

La conception de l'algorithme fait en sorte que deux contraintes doivent être imposées sur le code source du logiciel testé (avoir une méthode possédant le même nom que chacune des actions et la présence obligatoire d'un bouton de nom *btnOK* dans les fenêtres modales). De plus, l'algorithme ne gère pas les codes source comprenant des méthodes directement ou indirectement récursives, ainsi que les appels de méthodes virtuelles.

Les tests effectués sur trois études de cas ont fait ressortir que l'algorithme omet de générer les instructions qui ouvrent les fenêtres appropriées lorsque l'action ne peut être directement déclenchée à partir de la fenêtre principale. L'autre constat est que le code généré est, dans bien des cas, moins optimal que ce qu'aurait écrit une personne.

Avenues futures

Les futurs travaux sur ce sujet devraient porter sur le solutionnement des limites de l'algorithme. Une piste de solution concernant les appels de méthodes virtuelles a été citée dans l'essai. Le graphe d'appels de méthodes pourrait possiblement être exploité afin de déterminer comment ouvrir les fenêtres secondaires à partir de la fenêtre principale et ainsi régler le problème des instructions d'ouverture de ces fenêtres qui ne sont pas générées. Finalement, il serait intéressant d'analyser si les techniques d'optimisation de code utilisées par les compilateurs sont applicables à cet algorithme.

Cet algorithme devrait aussi faire l'objet de tests beaucoup plus poussés dans le futur, c'est-à-dire sur un très grand nombre de logiciels, mais également sur des programmes de plus grande taille. Cela n'a pas été fait dans le cadre de cet essai, car, pour faire cela, il faudrait développer un module permettant de traduire le code source textuel des modèles d'action et du

programme en modèles orienté objet (ces modèles ont été construits manuellement lors des tests). L'automatisation de cette tâche pourrait, à elle seule, constituer le sujet d'un autre essai. De plus, la grille d'évaluation doit être validée.

Liste des références

[1] Blankenhorn, K., *A UML Profile for GUI Layout*, University of Applied Sciences Furtwangen, Furtwangen, 2004, 114 p.

[2] Grove, D., DeFouw, G., Dean, J. et Chambers, C., *Call Graph Construction in Object-Oriented Languages*, OOPSLA '97 Conference Proceedings, 1997.

[3] Grune, D., Bal, H., Jacobs, C. et Langendoen, K., *Compilateurs*, 1ère éd., Dunod, Paris, 2002, 774 p.

[4] Holzmann, G., *The Spin Model Checker*, 1ère éd., Addison-Wesley, Boston, 2004, 596 p.

[5] Leino, R. et Müller, P., *Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs*, http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=specsharp&DownloadId=84056, 17 septembre 2009.

[6] March, S. et Smith, G., *Design and natural science research on information technology*, Decision Support Systems 15, 1995, p. 251-266.

[7] Memon, A. M. et Soffa, M., *Regression Testing of GUIs*, ESEC/FSE'03, 2003, p. 118-127.

[8] Microsoft Research, *Model-based Testing with SpecExplorer*, http://research.microsoft.com/en-us/projects/specexplorer/, 10 février 2011.

[9] Naik, K. et Tripathy, P, *Software Testing and Quality Assurance*, 1ère éd., Wiley, Hoboken, 2008, 616 p.

[10] Nguyen, D. H., Strooper, P. et Suess, J. G., *Automated Functionality Testing through GUIs*, Conferences in Research and Practice in Information Technology (CRPIT), 2010, p. 153-162.

[11] Nguyen, D. H., Strooper, P. et Suess, J. G., *Model-Based Testing of Multiple GUI Variants Using the GUI Test Generator*, AST' 10, mai 2010, p. 24-30.

[12] Pinheiro da Silva, P. et Paton, N., *User Interface Modelling with UML*, 10th European-Japanese Conference on Information Modelling and Knowledge Representation, 2000, 15 p.

[13] Tip, F. et Palsberg, J., *Scalable Propagation-Based Call Graph Construction Algorithms*, OOPSLA '00 Conference Proceedings, 2000, p. 281-293.

Annexe 1

Étude de cas 1

Cette annexe présente en détails la première étude de cas ayant servi à tester l'algorithme de génération du modèle de traduction. Les particularités de cet exemple sont la possibilité de déclencher la plupart des actions de deux façons différentes, l'utilisation de méthodes avec des paramètres par référence et la présence d'une fenêtre modale.

A1.1 Présentation de l'étude de cas

Cette étude de cas consiste en une application permettant de gérer une liste de tâches. La spécification des exigences de cette application est présentée à la figure A1.1. Son modèle d'action est présenté à la figure A1.2. Son interface graphique est présentée aux figures A1.3 et A1.4. Son code source complet, écrit en Visual Basic .NET, se trouve à la section A1.5. Le diagramme de classes de la figure A1.5 et les diagrammes de séquence des figures A1.6 et A1.7 résument les aspects importants du code source.

L'application gère une liste de tâches.

Une tâche est définie par son nom et possède un état d'avancement, qui peut être :

- Non débutée
- En cours
- Complétée

L'application peut effectuer trois opérations :

- Ajouter une tâche
  - Crée une nouvelle tâche avec les informations spécifiées par l'utilisateur et l'ajoute dans la liste de tâches.
- Modifier une tâche
  - Modifie une tâche avec les informations spécifiées par l'utilisateur.
- Supprimer une tâche
  - Supprime une tâche de la liste de tâches.

Figure A1.1 Spécification des exigences

Inspiré de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 159

```
enum Etats {NON_DEBUTEE, EN_COURS, COMPLETEE}

class Tache {
    int noTache;
    string! nom;
    Etats etat;
}

class GestionnaireTaches {
    Seq<Tache> taches = Seq{};

    [Action] void ajouterTache(string! nom, Etats etat) {
        Tache nouvelleTache = new Tache(0, nom, etat);
        taches.Add(nouvelleTache);
    }

    [Action] void modifierTache(Tache tache, string! nouveauNom, Etats nouvelEtat) {
        tache.nom = nouveauNom;
        tache.etat = nouvelEtat;
    }

    [Action] void supprimerTache(Tache tache) {
        taches.Remove(tache);
    }
}
```

Figure A1.2 Modèle d'action

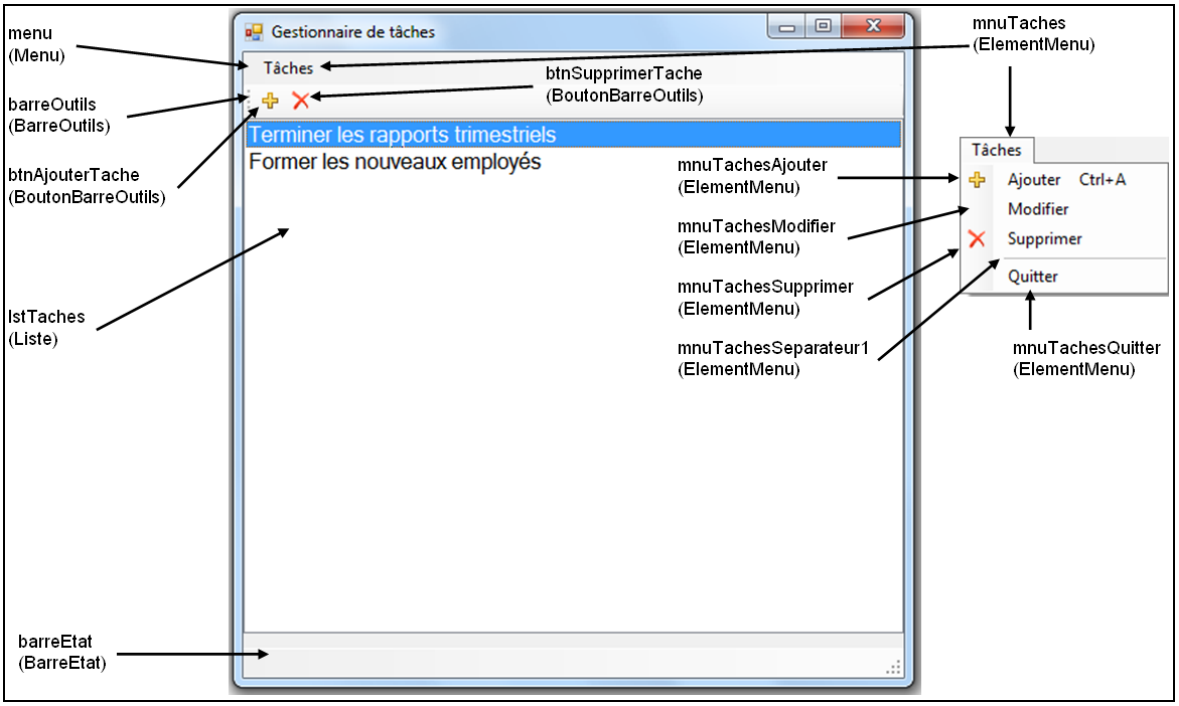


Figure A1.3 Fenêtre principale

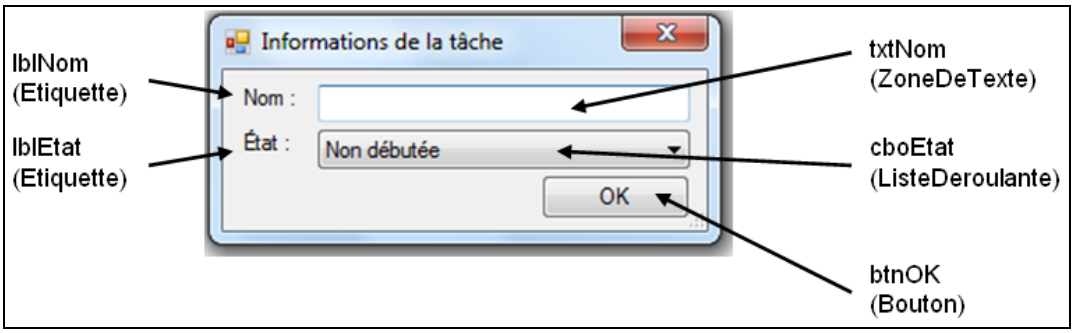


Figure A1.4 Fenêtre d'entrée de données

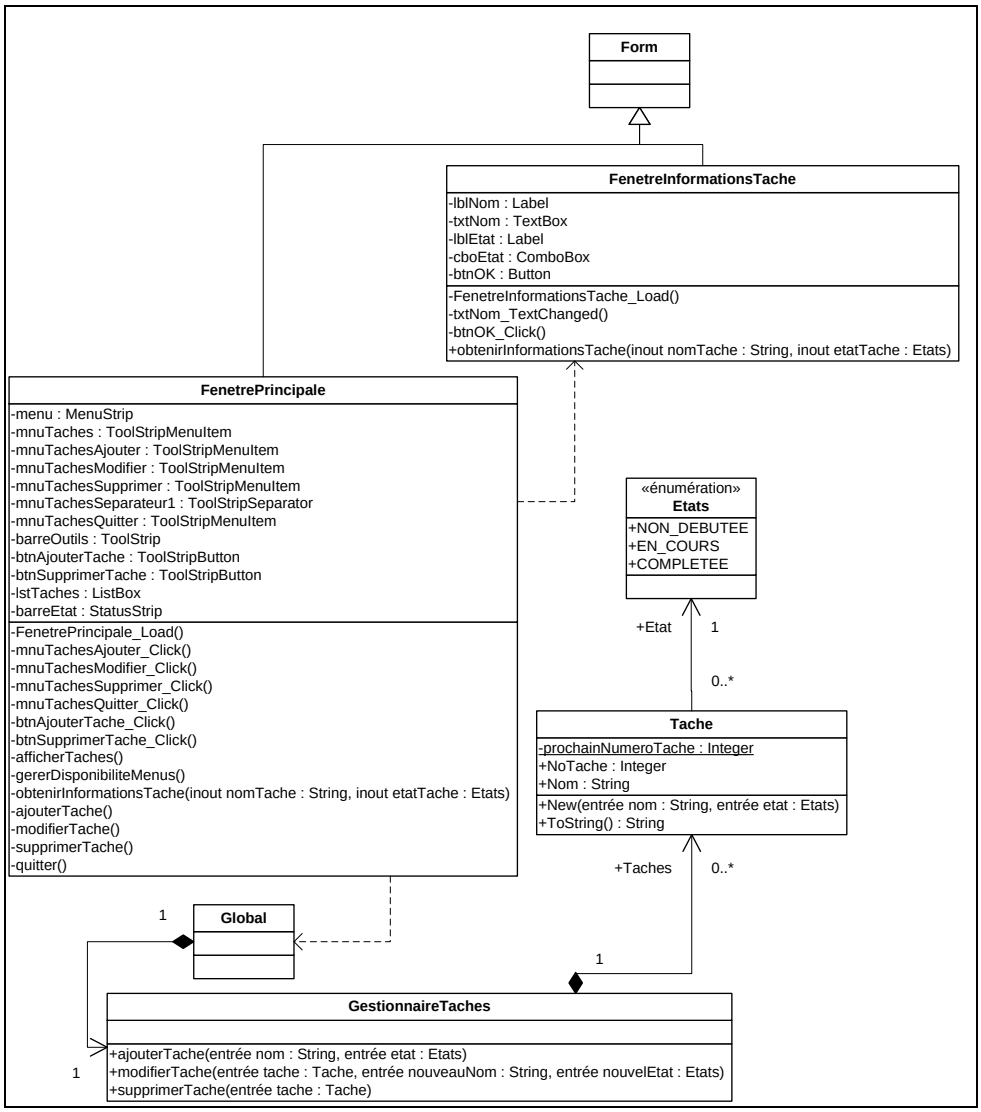


Figure A1.5 Diagramme de classes

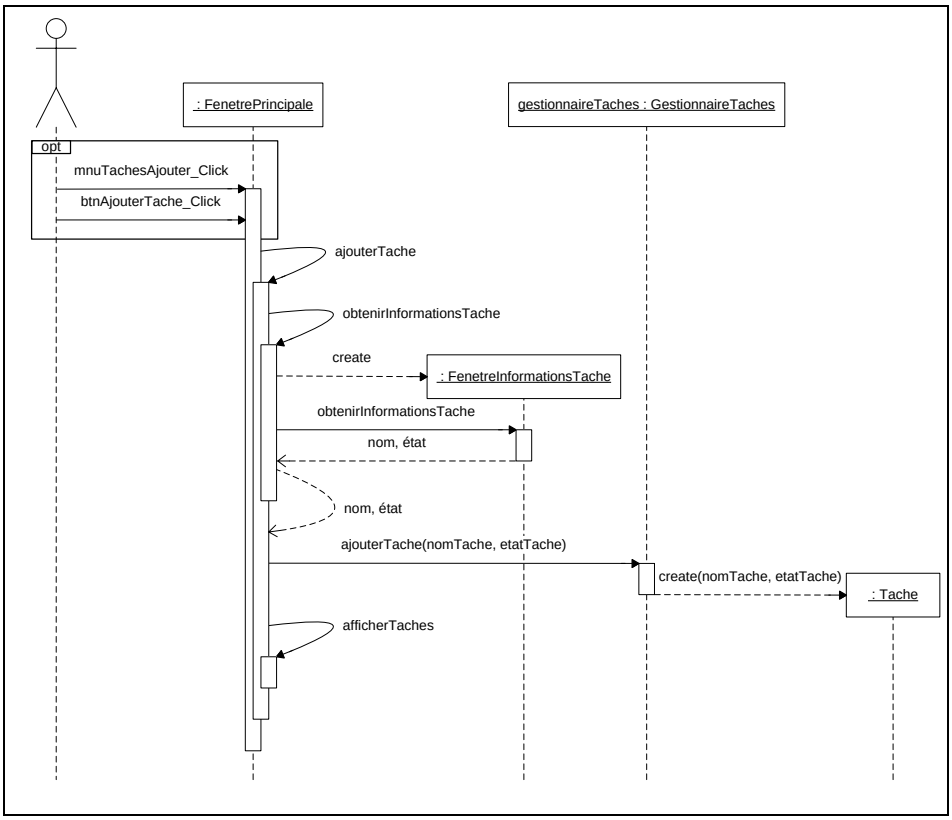


Figure A1.6 Diagramme de séquence de l'opération « ajouter une tâche »

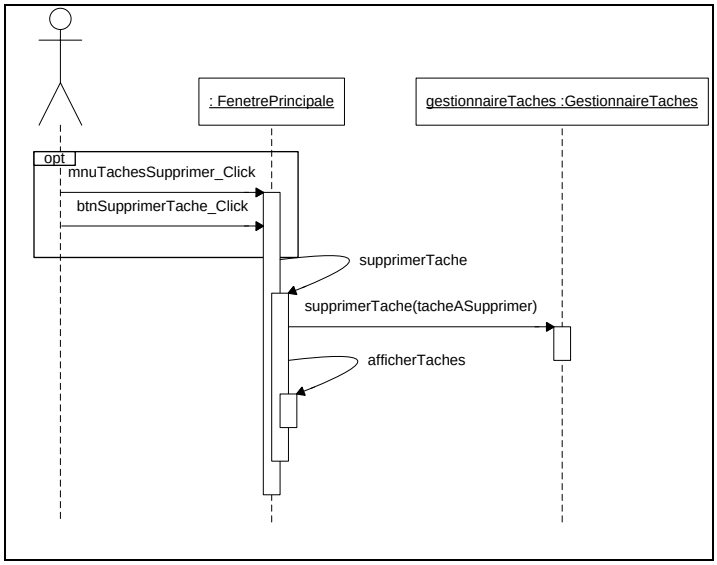


Figure A1.7 Diagramme de séquence de l'opération « supprimer une tâche »

L'utilisateur peut lancer une opération à partir du menu « Tâches » de la fenêtre principale (figure A1.3). Certaines de ces opérations peuvent également être lancées à partir de la barre d'outils. La fenêtre d'entrée de données (figure A1.4) est une fenêtre modale servant à demander à l'utilisateur les informations lors de l'ajout ou de la modification d'une tâche.

Le code source de l'application est divisé en six classes. La classe *Etats* est une énumération des états possibles d'une tâche. La classe *Tache* représente une tâche et contient ses informations. La classe *GestionnaireTaches* représente un conteneur de tâches. La classe *Global* contient les instances des variables globales. Les classes *FenetrePrincipale* et *FenetreInformationsTache* sont les fenêtres. Elles héritent de la classe *Form*, car cette application a été écrite en Visual Basic .NET.

Le diagramme de séquence de la figure A1.6 montre les interactions entre les objets lors de l'opération d'ajout d'une tâche. La modification d'une tâche suit le même principe. L'utilisateur lance d'abord l'opération en cliquant sur le menu ou sur la barre d'outils. Cela appelle une méthode *ajouterTache* dans la classe de la fenêtre principale. Celle-ci appelle une méthode *obtenirInformationsTache* qui crée la fenêtre d'entrée de données et appelle sur celle-ci une méthode *obtenirInformationsTache* qui affiche la fenêtre de façon modale et, une fois que l'utilisateur a cliqué sur le bouton OK, retourne les informations entrées via des paramètres par référence. Finalement, la méthode *ajouterTache* de la fenêtre principale appelle la méthode *ajouterTache* du gestionnaire de tâches avec les informations obtenues afin d'ajouter la tâche.

Le diagramme de séquence de la figure A1.7 montre les interactions entre les objets lors de l'opération de suppression d'une tâche. L'utilisateur sélectionne d'abord une tâche dans la liste de tâches. Il lance ensuite l'opération en cliquant sur le menu ou sur la barre d'outils. Cela appelle la méthode *supprimerTache* dans la classe de la fenêtre principale. Celle-ci appelle la méthode *supprimerTache* du gestionnaire de tâches afin de supprimer la tâche.

A1.2 Modèle de traduction attendu en sortie

Un exemple de modèle de traduction pour cette application, écrit par une personne, est présenté à la figure A1.8. Le modèle de traduction obtenu en sortie de l'algorithme doit être équivalent à celui-ci. Un tel modèle doit d'abord lancer les opérations en tenant compte des deux possibilités, lorsque c'est le cas. Il doit ensuite entrer les informations dans la fenêtre modale et cliquer sur le bouton OK.

```
void ajouterTache(string nom, Etats etat) {
    Serialize {
        Select {
            Serialize {
                Execute(GUI.mnuTaches.click);
                Execute(GUI.mnuTachesAjouter.click);
            }
            Execute(GUI.btnAjouterTache.click);
        }
        Permute {
            Execute(GUI.txtNom.type, nom);
            Execute(GUI.cboEtat.select, etat);
        }
        Execute(GUI.btnOK.click);
    }
}

void modifierTache(Tache tache, string nouveauNom, Etats nouvelEtat) {
    Serialize {
        Execute(GUI.lstTaches.select, tache);
        Execute(GUI.mnuTaches.click);
        Execute(GUI.mnuTachesModifier.click);
        Permute {
            Execute(GUI.txtNom.type, nouveauNom);
            Execute(GUI.cboEtat.select, nouvelEtat);
        }
        Execute(GUI.btnOK.click);
    }
}

void supprimerTache(Tache tache) {
    Serialize {
        Execute(GUI.lstTaches.select, tache);
        Select {
            Serialize {
                Execute(GUI.mnuTaches.click);
                Execute(GUI.mnuTachesSupprimer.click);
            }
            Serialize {
                Execute(GUI.btnSupprimerTache.click);
            }
        }
    }
}
```

Figure A1.8 Modèle de traduction attendu en sortie

A1.3 Fonctionnement de l'algorithme

Cette section montre, étape par étape, le fonctionnement de l'algorithme de génération du modèle de traduction sur cette étude de cas.

A1.3.1 Pré-calcul des structures de données

Dans cet exemple, la table des séquences d'instructions est simple, car le code source ne contient pas d'instruction conditionnelle ni d'instruction répétitive. Donc, chaque méthode a comme liste de séquences d'instructions une seule séquence contenant toutes les instructions de la méthode (algorithme 1.1.1).

L'analyse du code source de la section A1.5 donne le graphe d'appels de méthodes de la figure A1.9 (algorithme 1.1.2).

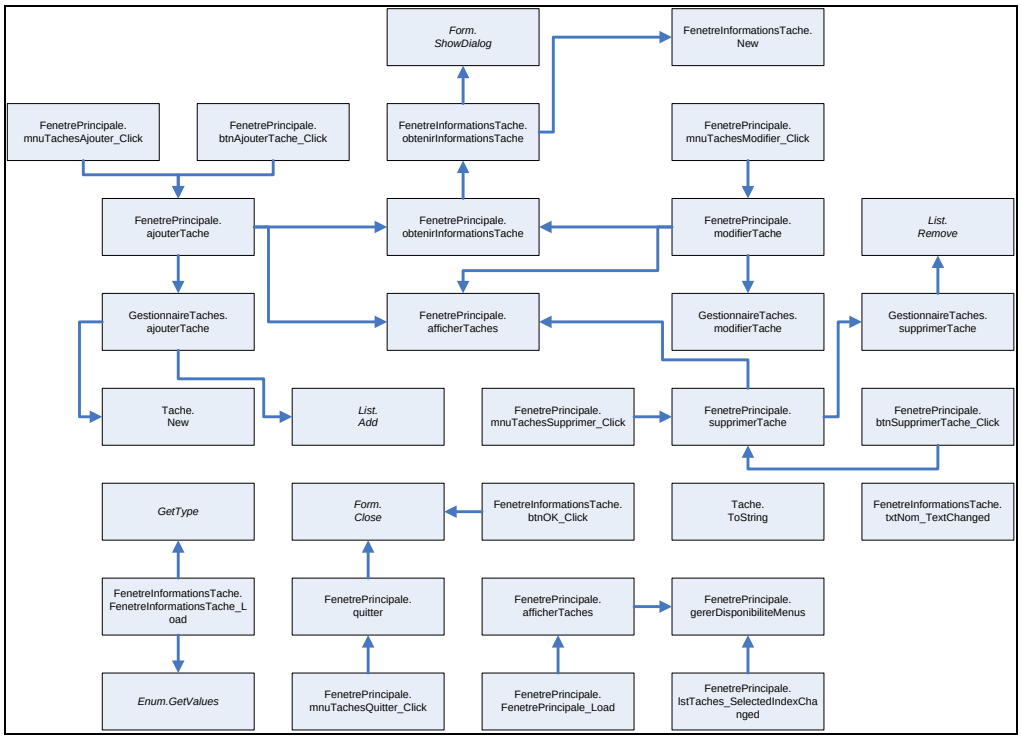


Figure A1.9 Graphe d'appels de méthodes

L'analyse du code source donne également la table de correspondance action/méthode du tableau A1.1 (algorithme 1.1.3), la table de correspondance classe/fenêtre du tableau A1.2 (algorithme 1.1.5) et la table de correspondance symbole/composant graphique du tableau A1.3 (algorithme 1.1.6). Celles-ci sont construites en associant des éléments du modèle d'action et du modèle de l'interface graphique à des éléments du modèle du programme, par correspondance de noms. Il est à noter que même si, par exemple, le programme contient deux méthodes qui se nomment *ajouterTache* (une dans la classe *FenetrePrincipale* et l'autre dans la classe *GestionnaireTaches*), c'est celle de la classe *GestionnaireTache* qui est associée à l'action *ajouterTache*, car ces deux éléments sont situés dans une classe de même nom et possèdent la même signature. Les classes *FenetrePrincipale* et *FenetreInformationsTache* sont identifiées comme fenêtres, car elles héritent de la classe *Form*.

Tableau A1.1 Table de correspondance action/méthode

Action	Méthode
ajouterTache	GestionnaireTaches.ajouterTache
modifierTache	GestionnaireTaches.modifierTache
supprimerTache	GestionnaireTaches.supprimerTache

Tableau A1.2 Table de correspondance classe/fenêtre

Classe	Fenêtre
FenetrePrincipale	FenetrePrincipale
FenetreInformationsTache	FenetreInformationsTache

L'analyse des chemins du graphe d'appels de méthodes de la figure A1.9 donne la table des chemins d'appel d'action du tableau A1.4 (algorithme 1.1.4). Pour rappel, les chemins retenus sont ceux débutant par un gestionnaire d'événement et se terminant par une méthode correspondant à une action de la table de correspondance action/méthode.

Tableau A1.3 Table de correspondance symbole/composant graphique

Symbole	Composant Graphique
FenetrePrincipale.menu (MenuStrip)	FenetrePrincipale.menu (Menu)
FenetrePrincipale.mnuTaches (ToolStripMenuItem)	FenetrePrincipale.mnuTaches (ElementMenu)
FenetrePrincipale.mnuTachesAjouter (ToolStripMenuItem)	FenetrePrincipale.mnuTachesAjouter (ElementMenu)
FenetrePrincipale.mnuTachesModifier (ToolStripMenuItem)	FenetrePrincipale.mnuTachesModifier (ElementMenu)
FenetrePrincipale.mnuTachesSupprimer (ToolStripMenuItem)	FenetrePrincipale.mnuTachesSupprimer (ElementMenu)
FenetrePrincipale.mnuTachesSeparateur1 (ToolStripSeparator)	FenetrePrincipale.mnuTachesSeparateur1 (ElementMenu)
FenetrePrincipale.mnuTachesQuitter (ToolStripMenuItem)	FenetrePrincipale.mnuTachesQuitter (ElementMenu)
FenetrePrincipale.barreOutils (ToolStrip)	FenetrePrincipale.barreOutils (BarreOutils)
FenetrePrincipale.btnAjouterTache (ToolStripButton)	FenetrePrincipale.btnAjouterTache (BoutonBarreOutils)
FenetrePrincipale.btnSupprimerTache (ToolStripButton)	FenetrePrincipale.btnSupprimerTache (BoutonBarreOutils)
FenetrePrincipale.lstTaches (ListBox)	FenetrePrincipale.lstTaches (Liste)
FenetrePrincipale.barreEtat (StatusStrip)	FenetrePrincipale.barreEtat (BarreEtat)
FenetreInformationsTache.lblNom (Label)	FenetreInformationsTache.lblNom (Etiquette)
FenetreInformationsTache.txtNom (TextBox)	FenetreInformationsTache.txtNom (ZoneDeTexte)
FenetreInformationsTache.lblEtat (Label)	FenetreInformationsTache.lblEtat (Etiquette)
FenetreInformationsTache.cboEtat (ComboBox)	FenetreInformationsTache.cboEtat (ListeDeroulante)
FenetreInformationsTache.btnOK (Button)	FenetreInformationsTache.btnOK (Bouton)

Tableau A1.4 Table des chemins d'appel d'action

Action	Chemin d'appel
ajouterTache	FenetrePrincipale.mnuTachesAjouter_Click → FenetrePrincipale.ajouterTache → GestionnaireTaches.ajouterTache
	FenetrePrincipale.btnAjouterTache_Click → FenetrePrincipale.ajouterTache → GestionnaireTaches.ajouterTache
modifierTache	FenetrePrincipale.mnuTachesModifier_Click → FenetrePrincipale.modifierTache → GestionnaireTaches.modifierTache
supprimerTache	FenetrePrincipale.mnuTachesSupprimer_Click → FenetrePrincipale.supprimerTache → GestionnaireTaches.supprimerTache
	FenetrePrincipale.btnSupprimerTache_Click → FenetrePrincipale.supprimerTache → GestionnaireTaches.supprimerTache

A1.3.2 Génération du modèle de traduction

Cette section explore la génération du modèle de traduction de l'action « ajouter une tâche » (algorithme 1.2). Le traitement des autres actions est similaire.

L'en-tête de la fonction de traduction est d'abord écrit, suivi du début d'un bloc « *Select* », puisque l'action possède plus d'un chemin d'appel. L'étape suivante est de calculer, pour chaque chemin d'appel de cette action, la table paramètres/expressions (algorithme 1.2.1.1). Dans cet exemple, la table est identique pour les deux chemins. Au départ, la liste d'expressions de chacun des paramètres est initialisée à vide (tableau A1.5, version 1). Le code du chemin d'appel est parcouru, en sens inverse. Lorsque l'appel à la méthode *GestionnaireTache.ajouterTache* est trouvé dans l'avant-dernière méthode du chemin (section A1.5, ligne de code 177), la liste d'expressions de chacun des paramètres est remplie avec les paramètres effectifs de l'appel (tableau A1.5, version 2). Le parcours inverse du code continue par la suite. L'appel à la méthode *FenetrePrincipale.obtenirInformationsTache* est alors rencontré (section A1.5, ligne de code 175). Puisque cette méthode possède des paramètres par référence, ses séquences d'instructions sont explorées, de la première à la dernière instruction. Dans cette méthode, on rencontre l'appel à la méthode *FenetreInformationsTache.obtenirInformationsTache* (section A1.5, ligne de code 168). Puisque cette méthode possède elle aussi des paramètres par référence, ses séquences d'instructions sont à leur tour explorées, de la première à la dernière instruction. Dans celles-ci, une nouvelle valeur est affectée aux paramètres par référence (section A1.5, lignes de code 105 et 106). Ces variables sont donc remplacées dans la partie droite de la table paramètres/expressions par la valeur qui leur est affectée (tableau A1.5, version 3).

Une fois la table paramètres/expressions construite, chaque méthode du chemin d'appel est parcourue afin de générer le code de traduction (algorithmes 1.2.1 et 1.2.1.2). Le début du bloc « *Serialize* » est écrit. La première méthode du chemin est *FenetrePrincipale.mnuTachesAjouter_Click*. Puisque la méthode appartient à une classe (*FenetrePrincipale*) identifiée comme fenêtre dans la table de correspondance classe/fenêtre et

Tableau A1.5 Table paramètres/expressions pour l'action « ajouter une tâche »

Paramètres	Expressions (version 1)	Expressions (version 2)	Expressions (version 3)
nom	{}	{nomTache}	{FenetreInformationsTache. txtNom.Text}
etat	{}	{etatTache}	{FenetreInformationsTache. cboEtat.SelectedItem}

que la pile des fenêtres ouvertes est vide, cette fenêtre est empilée dans la pile des fenêtres ouvertes. Pour cette méthode, l'algorithme 1.2.1.2.2 est applicable, puisqu'il s'agit d'un gestionnaire d'événement. Le code déclenchant cet événement doit donc être écrit. Puisque ce menu est contenu dans un autre menu, celui-ci doit d'abord être activé. Le code généré est donc le clic sur le menu *mnuTaches*, puis le clic sur le menu *mnuTachesAjouter*. La deuxième méthode du chemin est *FenetrePrincipale.ajouterTache*. Celle-ci contient l'appel à la méthode *FenetrePrincipale.obtenirInformationsTache*, qui elle-même contient l'appel à la méthode *FenetreInformationsTache.obtenirInformationsTaches*. Puisque cette dernière contient l'ouverture d'une fenêtre, l'algorithme 1.2.1.2.3 est applicable. La fenêtre est empilée dans la pile des fenêtres ouvertes. L'algorithme 1.2.1.2.1 est appelé pour traiter les paramètres. Puisque, selon la table paramètres/expressions, la source des paramètres *nom* et *etat* est une expression dont le symbole est identifié comme composant graphique dans la table de correspondance symbole/composant graphique et que la fenêtre de ce composant est présentement active, ces paramètres doivent être traités. Le début d'un bloc « *Permute* » est d'abord écrit, puisque plus d'un paramètre peut être traité au cours de cette itération. Le code tapant le nom dans la zone de texte *txtNom* et sélectionnant le bon état dans la liste déroulante *cboEtat* est écrit. Ces deux paramètres/expressions sont retirés de la table puisqu'ils ont été traités. Puisque la fenêtre qui a été ouverte est modale, la fin de l'algorithme 1.2.1.2.3 doit la fermer. Un bouton de nom *btnOK* est présent, donc le code cliquant sur celui-ci est écrit. La fenêtre est dépilée de la pile des fenêtres ouvertes. Plus aucun traitement n'est possible par la suite pour ce chemin. Les étapes précédemment écrites recommencent pour le deuxième chemin d'appel de l'action. La seule différence est la façon de déclencher l'action (en utilisant

la barre d'outils plutôt que le menu). Le modèle de traduction obtenu en sortie est celui de la figure A1.10.

```
void ajouterTache(string nom, Etats etat) {
    Select {
        Serialize {
            Execute(GUI.mnuTaches.click);
            Execute(GUI.mnuTachesAjouter.click);
            Permute {
                Execute(GUI.txtNom.type, nom);
                Execute(GUI.cboEtat.select, etat);
            }
            Execute(GUI.btnOK.click);
        }
        Serialize {
            Execute(GUI.btnAjouterTache.click);
            Permute {
                Execute(GUI.txtNom.type, nom);
                Execute(GUI.cboEtat.select, etat);
            }
            Execute(GUI.btnOK.click);
        }
    }
}

void modifierTache(Tache tache, string nouveauNom, Etats nouvelEtat) {
    Serialize {
        Execute(GUI.lstTaches.select, tache);
        Execute(GUI.mnuTaches.click);
        Execute(GUI.mnuTachesModifier.click);
        Permute {
            Execute(GUI.txtNom.type, nouveauNom);
            Execute(GUI.cboEtat.select, nouvelEtat);
        }
        Execute(GUI.btnOK.click);
    }
}

void supprimerTache(Tache tache) {
    Select {
        Serialize {
            Execute(GUI.lstTaches.select, tache);
            Execute(GUI.mnuTaches.click);
            Execute(GUI.mnuTachesSupprimer.click);
        }
        Serialize {
            Execute(GUI.lstTaches.select, tache);
            Execute(GUI.btnSupprimerTache.click);
        }
    }
}
```

Figure A1.10 Modèle de traduction obtenu en sortie

A1.4 Évaluation

Afin de vérifier si l'algorithme de génération du modèle de traduction fonctionne bien sur cette étude de cas, les critères de comparaison de deux modèles de traduction ont été appliqués sur les modèles de traduction obtenu et attendu (tableau A1.6). L'analyse montre que les deux

modèles sont équivalents. Cependant, une observation rapide permet de constater que le modèle de traduction attendu a été écrit de manière à contenir moins d'instructions redondantes.

Tableau A1.6 Évaluation de l'équivalence entre les modèles de traduction obtenu et attendu

	Critère rencontré ?	Commentaire
1	Oui	
2	Oui	
3	Oui	
4	Oui	L'utilisateur a le choix d'utiliser le menu ou la barre d'outils pour déclencher les actions. Ce choix est contenu dans un bloc « *Select* » pour les deux modèles.
5	Oui	Le déclenchement de l'action par le menu ou la barre d'outils doit d'abord se faire. L'écriture dans les champs utilisateur doit être faite ensuite. Le bouton OK doit ensuite être cliqué. Ces étapes sont contenues, dans l'ordre, dans un bloc « *Serialize* » pour les deux modèles.
6	Oui	L'ordre de l'écriture dans les champs utilisateur peut être permuté. Cela se trouve dans un bloc « *Permute* » pour les deux modèles.
7	Oui	
8	Oui	
9	Oui	

A1.5 Code source

```
001    Public Enum Etats
002        NON_DEBUTEE
003        EN_COURS
004        COMPLETEE
005    End Enum
006
007    Public Class Tache
008        Private Shared prochainNumeroTache As Integer = 1
009
010        Private _noTache As Integer
011        Private _nom As String
012        Private _etat As Etats
```

```
013
014         Public ReadOnly Property NoTache As Integer
015             Get
016                 Return _noTache
017             End Get
018         End Property
019
020         Public Property Nom As String
021             Get
022                 Return _nom
023             End Get
024             Set(ByVal value As String)
025                 _nom = value
026             End Set
027         End Property
028
029         Public Property Etat As Etats
030             Get
031                 Return _etat
032             End Get
033             Set(ByVal value As Etats)
034                 _etat = value
035             End Set
036         End Property
037
038         Public Sub New(ByVal nom As String, ByVal etat As Etats)
039             Me._noTache = prochainNumeroTache
040             Me.Nom = nom
041             Me.Etat = etat
042
043             prochainNumeroTache += 1
044         End Sub
045
046         Public Overrides Function ToString() As String
047             Return Nom
048         End Function
049     End Class
050
051     Public Class GestionnaireTaches
052         Private _taches As New List(Of Tache)
053
054         Public ReadOnly Property Taches As List(Of Tache)
055             Get
056                 Return _taches
057             End Get
```

```
058         End Property
059
060         Public Sub ajouterTache(ByVal nom As String, ByVal etat As Etats)
061             Dim nouvelleTache As New Tache(nom, etat)
062
063             _taches.Add(nouvelleTache)
064         End Sub
065
066         Public Sub modifierTache(ByVal tache As Tache, ByVal nouveauNom As String, ByVal
    nouvelEtat As Etats)
067             tache.Nom = nouveauNom
068             tache.Etat = nouvelEtat
069         End Sub
070
071         Public Sub supprimerTache(ByVal tache As Tache)
072             _taches.Remove(tache)
073         End Sub
074     End Class
075
076     Public Class FenetreInformationsTache
077         ''''''''''''''
078         ' Événements '
079         ''''''''''''''
080
081         Private Sub FenetreInformationsTache_Load(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles MyBase.Load
082             cboEtat.DataSource = System.Enum.GetValues(GetType(Etats))
083         End Sub
084
085         Private Sub txtNom_TextChanged(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles txtNom.TextChanged
086             Dim boutonOKActif As Boolean = (txtNom.Text <> "")
087
088             btnOK.Enabled = boutonOKActif
089         End Sub
090
091         Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles btnOK.Click
092             Me.Close()
093         End Sub
094
095         ''''''''''''''''''''''
096         ' Fonction principale '
097         ''''''''''''''''''''''
098
```

```
099         Public Sub obtenirInformationsTache(ByRef nomTache As String, ByRef etatTache As
    Etats)
100             txtNom.Text = nomTache
101             cboEtat.SelectedItem = etatTache
102 
103             Me.ShowDialog()
104 
105             nomTache = txtNom.Text
106             etatTache = CType(cboEtat.SelectedItem, Etats)
107         End Sub
108     End Class
109 
110     Public Class FenetrePrincipale
111         ''''''''''''''
112         ' Événements '
113         ''''''''''''''
114 
115         Private Sub FenetrePrincipale_Load(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles MyBase.Load
116             afficherTaches()
117         End Sub
118 
119         Private Sub mnuTachesAjouter_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles mnuTachesAjouter.Click
120             ajouterTache()
121         End Sub
122 
123         Private Sub mnuTachesModifier_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles mnuTachesModifier.Click
124             modifierTache()
125         End Sub
126 
127         Private Sub mnuTachesSupprimer_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles mnuTachesSupprimer.Click
128             supprimerTache()
129         End Sub
130 
131         Private Sub mnuTachesQuitter_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles mnuTachesQuitter.Click
132             quitter()
133         End Sub
134 
135         Private Sub btnAjouterTache_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnAjouterTache.Click
136             ajouterTache()
```

```
137         End Sub
138
139         Private Sub btnSupprimerTache_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnSupprimerTache.Click
140             supprimerTache()
141         End Sub
142
143         Private Sub lstTaches_SelectedIndexChanged(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles lstTaches.SelectedIndexChanged
144             gererDisponibiliteMenus()
145         End Sub
146
147         ''''''''''''''''''''''''
148         ' Fonctions principales '
149         ''''''''''''''''''''''''
150
151         Private Sub afficherTaches()
152             lstTaches.DataSource = Nothing
153             lstTaches.DataSource = [Global].gestionnaireTaches.Taches
154             gererDisponibiliteMenus()
155         End Sub
156
157         Private Sub gererDisponibiliteMenus()
158             Dim menusDisponibles As Boolean = (lstTaches.SelectedIndex >= 0)
159
160             mnuTachesModifier.Enabled = menusDisponibles
161             mnuTachesSupprimer.Enabled = menusDisponibles
162             btnSupprimerTache.Enabled = menusDisponibles
163         End Sub
164
165         Private Sub obtenirInformationsTache(ByRef nomTache As String, ByRef etatTache As
    Etats)
166             Dim fenetreInformationsTache As New FenetreInformationsTache
167
168             fenetreInformationsTache.obtenirInformationsTache(nomTache, etatTache)
169         End Sub
170
171         Private Sub ajouterTache()
172             Dim nomTache As String = ""
173             Dim etatTache As Etats = Etats.NON_DEBUTEE
174
175             obtenirInformationsTache(nomTache, etatTache)
176
177             [Global].gestionnaireTaches.ajouterTache(nomTache, etatTache)
178
```

```
179             afficherTaches()
180         End Sub
181
182         Private Sub modifierTache()
183             Dim tacheAModifier As Tache = CType(lstTaches.SelectedItem, Tache)
184             Dim nomTache As String
185             Dim etatTache As Etats
186
187             nomTache = tacheAModifier.Nom
188             etatTache = tacheAModifier.Etat
189
190             obtenirInformationsTache(nomTache, etatTache)
191
192             [Global].gestionnaireTaches.modifierTache(tacheAModifier, nomTache, etatTache)
193
194             afficherTaches()
195         End Sub
196
197         Private Sub supprimerTache()
198             Dim tacheASupprimer As Tache = CType(lstTaches.SelectedItem, Tache)
199
200             [Global].gestionnaireTaches.supprimerTache(tacheASupprimer)
201
202             afficherTaches()
203         End Sub
204
205         Private Sub quitter()
206             Me.Close()
207         End Sub
208     End Class
209
210     Public Class [Global]
211         Public Shared gestionnaireTaches As New GestionnaireTaches
212     End Class
```

Annexe 2

Étude de cas 2

Cette annexe présente en détails la deuxième étude de cas ayant servi à tester l'algorithme de génération du modèle de traduction. Les particularités de cet exemple sont la présence de deux champs utilisateur différents pouvant servir de source à un même paramètre et la présence de plusieurs fenêtres.

A2.1 Présentation de l'étude de cas

L'application gère les commandes client et les bons de travail d'une entreprise manufacturière.

Commandes

Une commande client contient les informations suivantes :

- Numéro de la commande
- Nom du client
- Produit commandé
- Produit de remplacement (facultatif)
- Quantité commandée
- Prix total

Les produits de l'entreprise sont codés par une série de lettres suivie d'une série de chiffres. Les produits d'une même famille (série de lettres identiques) sont très similaires entre eux et n'ont qu'une différence technique mineure, peu perceptible par le client.

L'entreprise souhaite garder son inventaire de produits très bas. Pour cette raison, il arrive souvent que le produit commandé par un client ne soit pas en stock. Le commis peut alors décider de remplacer le produit commandé par un produit équivalent.

Bons de travail

Un bon de travail contient les informations suivantes :

- Numéro du bon de travail
- Produit à fabriquer
- Quantité à fabriquer
- Commande sur laquelle le bon de travail est basé (facultatif)

Un bon de travail peut être créé à partir d'une commande client. Si la commande indique un produit de remplacement, le bon de travail est créé avec ce produit. Sinon, il est créé avec le produit commandé.

Un bon de travail peut également être créé sans commande client, afin de pouvoir augmenter l'inventaire de produits.

Opérations

L'application peut effectuer cinq opérations :

- Lire une commande
- Lire un bon de travail
- Ajouter une nouvelle commande
- Ajouter un nouveau bon de travail
- Ajouter une nouvelle commande et le bon de travail correspondant simultanément

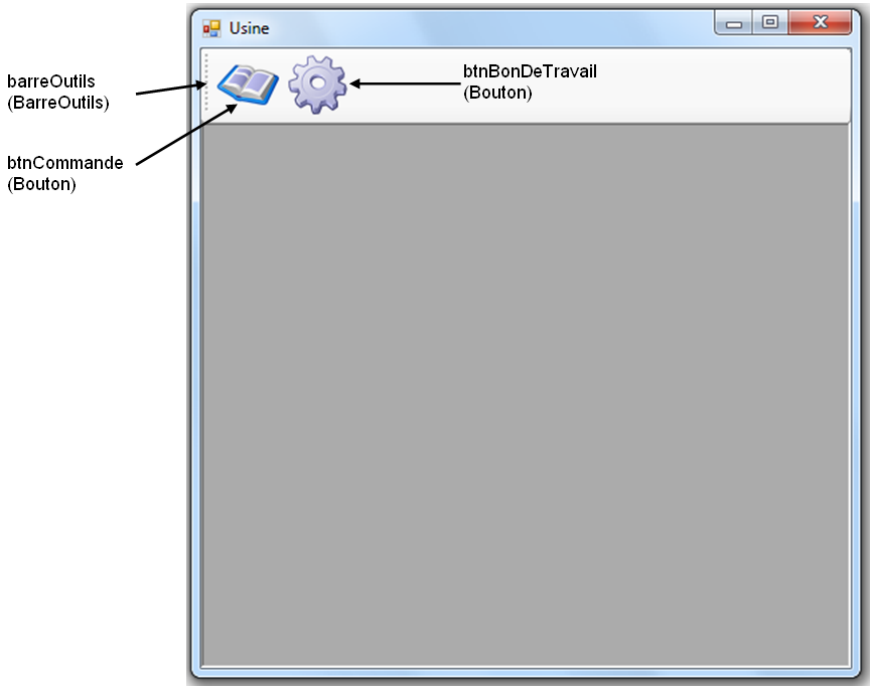
Figure A2.1 Spécification des exigences

Cette étude de cas consiste en une application permettant de gérer les commandes client et les bons de travail d'une entreprise manufacturière. La spécification des exigences de cette

application est présentée à la figure A2.1. Son modèle d'action est présenté à la figure A2.2. Son interface graphique est présentée aux figures A2.3, A2.4 et A2.5. Son code source complet, écrit en Visual Basic .NET, se trouve à la section A2.5. Le diagramme de classes de la figure A2.6 et les diagrammes de séquence des figures A2.7 et A2.8 résument les aspects importants du code source.

```
class Produit {
    int noProduit;
    string! nomProduit;
}

class Commande {
    int noCommande;
    string! client;
    Produit! produit;
    Produit produitDeRemplacement;
    int quantite;
    double prix;
}

class BonDeTravail {
    int noBonDeTravail;
    Produit! produit;
    int quantite;
    Commande commandeOrigine;
}

class GestionnaireUsine {
    Seq<Produit> carnetProduits = Seq{};
    Seq<Commande> carnetCommandes = Seq{};
    Seq<BonDeTravail> carnetBonsDeTravail = Seq{};

    [Action] void lireCommande(Commande! commande) {

    }

    [Action] void lireBonDeTravail(BonDeTravail! bonDeTravail) {

    }

    [Action] void ajouterCommande(string! client, Produit! produit, Produit produitDeRemplacement, int quantite, double prix)
        requires quantite >= 1;
        requires prix >= 0;
    {
        Commande nouvelleCommande = new Commande(0, client, produit, produitDeRemplacement, quantite, prix);
        carnetCommandes.Add(nouvelleCommande);
    }

    [Action] void ajouterBonDeTravail(Produit! produit, int quantite, Commande commandeOrigine)
        requires quantite >= 1;
    {
        BonDeTravail nouveauBonDeTravail = new BonDeTravail(0, produit, quantite, commandeOrigine);
        carnetBonsDeTravail.Add(nouveauBonDeTravail);
    }
}
```

Figure A2.2 Modèle d'action



A2.3 Fenêtre principale

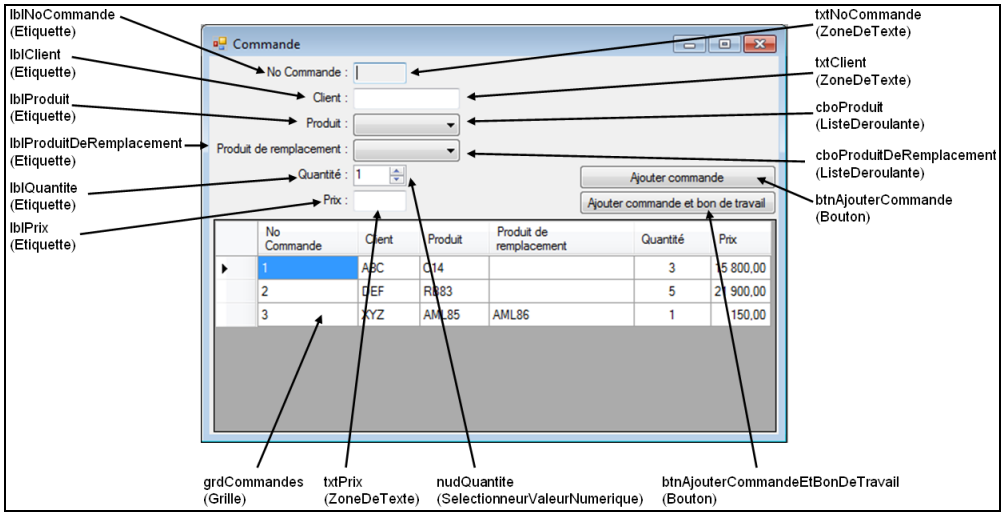


Figure A2.4 Fenêtre des commandes

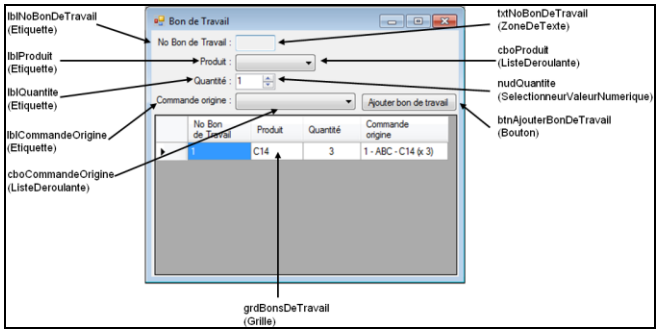


Figure A2.5 Fenêtre des bons de travail

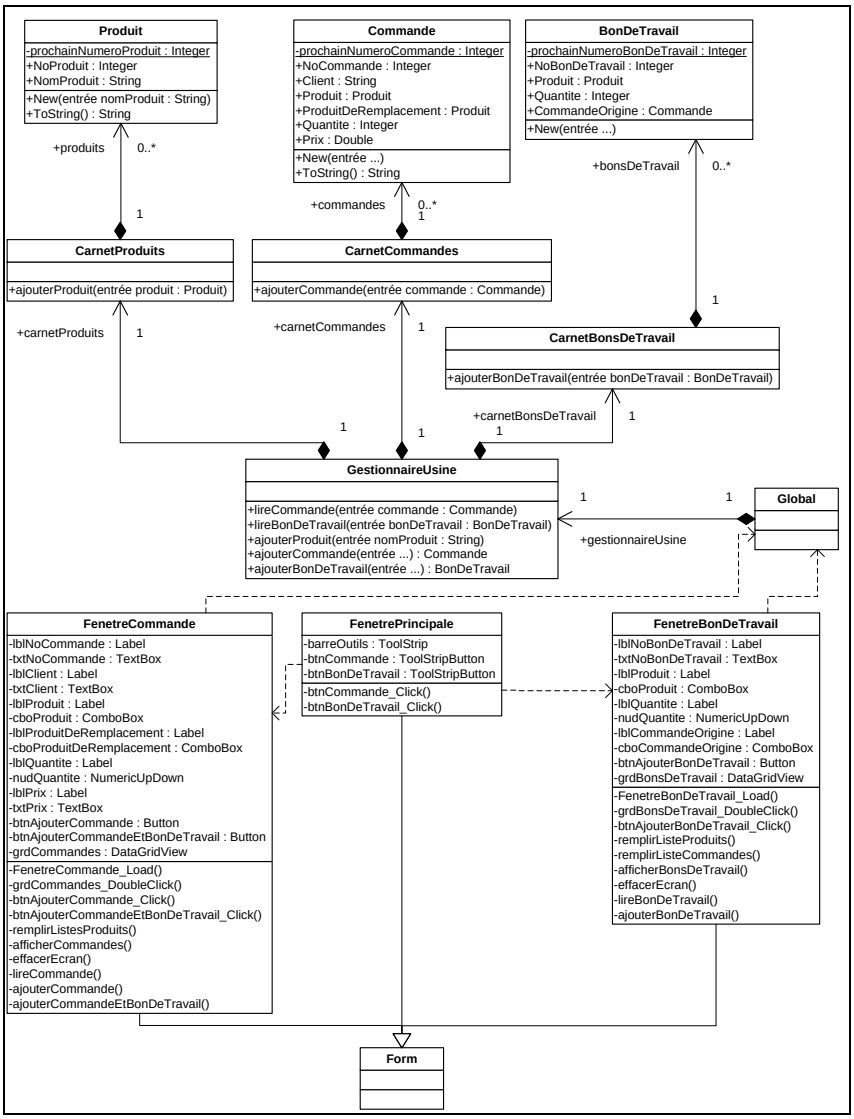


Figure A2.6 Diagramme de classes

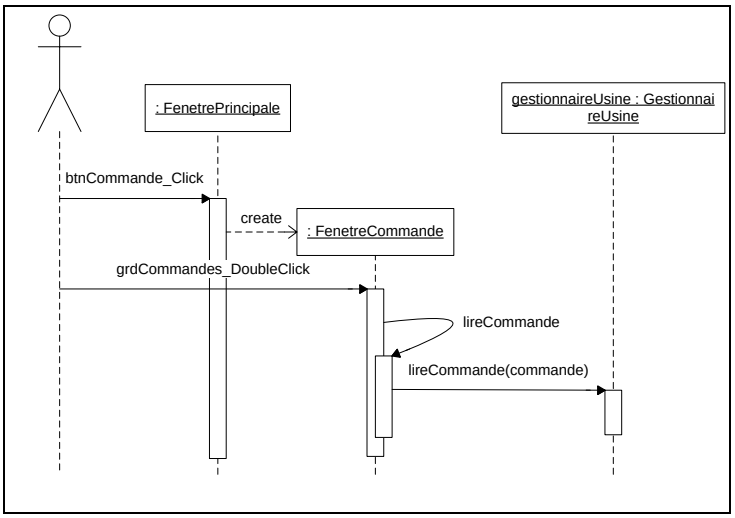


Figure A2.7 Diagramme de séquence de l'opération « lire une commande »

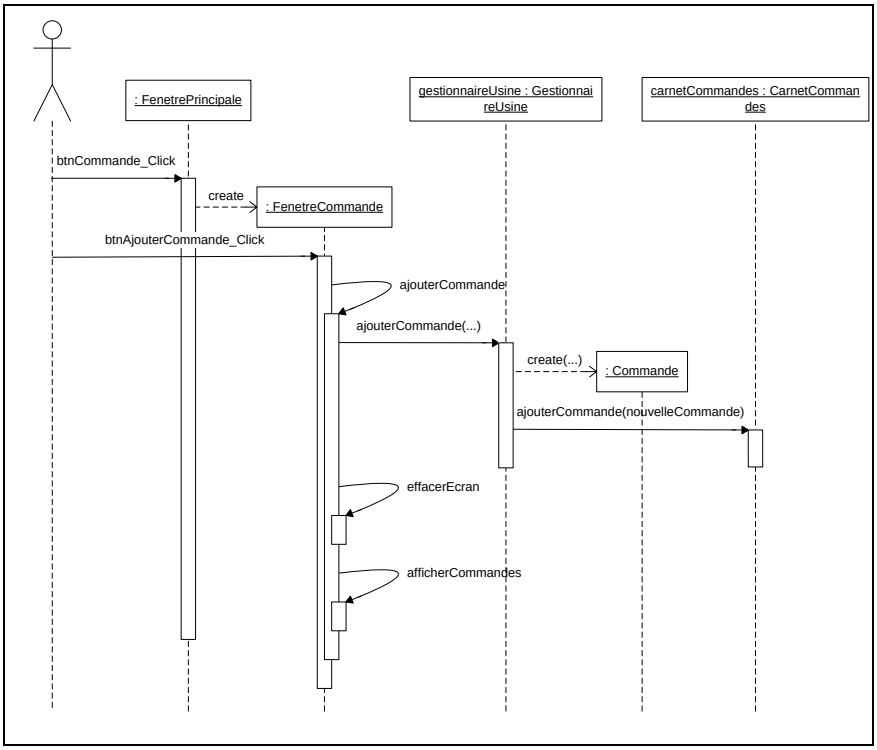


Figure A2.8 Diagramme de séquence de l'opération « ajouter une nouvelle commande »

L'utilisateur accède aux fenêtres des commandes et des bons de travail en cliquant sur le bouton approprié de la barre d'outils de la fenêtre principale. La liste des éléments déjà existants apparaît dans la grille du bas de la fenêtre. Afin de lire un élément, l'utilisateur doit double-cliquer sur celui-ci dans la grille et les informations apparaissent alors dans la section du haut. Pour ajouter un nouvel élément, l'utilisateur doit remplir les champs et cliquer sur le bouton ajouter. Il est possible de créer une commande et le bon de travail associé à cette commande simultanément.

Le code source de l'application est divisé en onze classes. Les classes *Produit*, *Commande* et *BonDeTravail* représentent respectivement un produit, une commande et un bon de travail. Elles contiennent leurs informations. Les classes *CarnetProduits*, *CarnetCommandes* et *CarnetBonsDeTravail* représentent respectivement la liste des produits, commandes et bons de travail. La classe *GestionnaireUsine* contient tous les carnets. La classe *Global* contient les instances des variables globales. Les classes *FenetrePrincipale*, *FenetreCommande* et *FenetreBonDeTravail* sont les fenêtres. Elles héritent de la classe *Form*, car cette application a été écrite en Visual Basic .NET.

Le diagramme de séquence de la figure A2.7 montre les interactions entre les objets lors de l'opération de lecture d'une commande. L'opération de lecture d'un bon de travail suit le même principe. L'utilisateur ouvre d'abord la fenêtre de commande en cliquant sur le bouton *Commande* dans la fenêtre principale. Il double-clique sur la commande qu'il désire consulter dans la grille du bas. Cela appelle une méthode *lireCommande* dans la classe de la fenêtre. Celle-ci appelle la méthode *lireCommande* du gestionnaire d'usine afin de lire la commande.

Le diagramme de séquence de la figure A2.8 montre les interactions entre les objets lors de l'opération d'ajout d'une commande. L'opération d'ajout d'un bon de travail suit le même principe. L'utilisateur ouvre d'abord la fenêtre de commande en cliquant sur le bouton *Commande* dans la fenêtre principale. Il remplit les champs et clique sur le bouton « Ajouter commande ». Cela appelle une méthode *ajouterCommande* dans la classe de la fenêtre.

Celle-ci appelle la méthode *ajouterCommande* du gestionnaire d'usine. Celle-ci appelle à son tour la méthode *ajouterCommande* du carnet de commandes afin d'ajouter la commande.

A2.2 Modèle de traduction attendu en sortie

```
void lireCommande(Commande commande) {
    Serialize {
        Execute(GUI.btnCommande.click);
        Execute(GUI.grdCommandes.select, commande);
        Execute(GUI.grdCommandes.doubleclick);
    }
}

void lireBonDeTravail(BonDeTravail bonDeTravail) {
    Serialize {
        Execute(GUI.btnBonDeTravail.click);
        Execute(GUI.grdBonsDeTravail.select, bonDeTravail);
        Execute(GUI.grdBonsDeTravail.doubleclick);
    }
}

void ajouterCommande(string client, Produit produit, Produit produitDeRemplacement, int quantite, double prix) {
    Serialize {
        Execute(GUI.btnCommande.click);
        Permute {
            Execute(GUI.txtClient.type, client);
            Execute(GUI.cboProduit.select, produit);
            Execute(GUI.cboProduitDeRemplacement.select, produitDeRemplacement);
            Execute(GUI.nudQuantite.type, quantite);
            Execute(GUI.txtPrix.type, prix);
        }
        Select {
            Execute(GUI.btnAjouterCommande.click);
            Execute(GUI.btnAjouterCommandeEtBonDeTravail.click);
        }
    }
}

void ajouterBonDeTravail(Produit produit, int quantite, Commande commandeOrigine) {
    Select {
        Serialize {
            Execute(GUI.btnBonDeTravail.click);
            Permute {
                Execute(GUI.cboProduit.select, produit);
                Execute(GUI.nudQuantite.type, quantite);
                Execute(GUI.cboCommandeOrigine.select, commandeOrigine);
            }
            Execute(GUI.btnAjouterBonDeTravail.click);
        }
        Serialize {
            Execute(GUI.btnCommande.click);
            Permute {
                Execute(GUI.nudQuantite.type, quantite);
                Select {
                    Execute(GUI.cboProduit.select, produit);
                    Execute(GUI.cboProduitDeRemplacement.select, produit);
                }
            }
            Execute(GUI.btnAjouterCommandeEtBonDeTravail.click);
        }
    }
}
```

Figure A2.9 Modèle de traduction attendu en sortie

Un exemple de modèle de traduction pour cette application, écrit par une personne, est présenté à la figure A2.9. Le modèle de traduction obtenu en sortie de l'algorithme doit être équivalent à celui-ci. Un tel modèle doit d'abord ouvrir la fenêtre appropriée à l'action à partir de la barre d'outils de la fenêtre principale. Dans le cas d'une opération de lecture, il doit sélectionner l'élément désiré dans la grille et double-cliquer sur celle-ci. Dans le cas d'une

opération d'ajout, il doit entrer les informations dans la fenêtre et cliquer sur le bouton qui effectue l'opération.

A2.3 Fonctionnement de l'algorithme

Cette section montre, étape par étape, le fonctionnement de l'algorithme de génération du modèle de traduction sur cette étude de cas.

A2.3.1 Pré-calcul des structures de données

L'analyse du code source de la section A2.5 donne la table des séquences d'instructions du tableau A2.1 et le graphe d'appels de méthodes de la figure A2.10

Tableau A2.1 Table des séquences d'instructions

Méthode		Instructions

FenetreCommande.
ajouterCommandeEtBonDeTravail
	# 1	`Dim nouvelleCommande As Commande`
		`Dim produit As Produit`
		`cboProduitDeRemplacement.SelectedIndex <> -1`
		`produit = cboProduitDeRemplacement.SelectedItem`
		`nouvelleCommande = _`
		`[Global].gestionnaireUsine.ajouterCommande(…)`
		`[Global].gestionnaireUsine.ajouterBonDeTravail(…)`
		`effacerEcran()`
		`afficherCommandes()`

	# 2	`Dim nouvelleCommande As Commande`
		`Dim produit As Produit`
		`cboProduitDeRemplacement.SelectedIndex <> -1`
		`produit = cboProduit.SelectedItem`
		`nouvelleCommande = _`
		`[Global].gestionnaireUsine.ajouterCommande(…)`
		`[Global].gestionnaireUsine.ajouterBonDeTravail(…)`
		`effacerEcran()`
		`afficherCommandes()`

Toutes les autres méthodes possèdent comme liste de séquences d'instructions une seule séquence contenant toutes leurs instructions (car elles ne contiennent pas d'instruction conditionnelle ni d'instruction répétitive).		

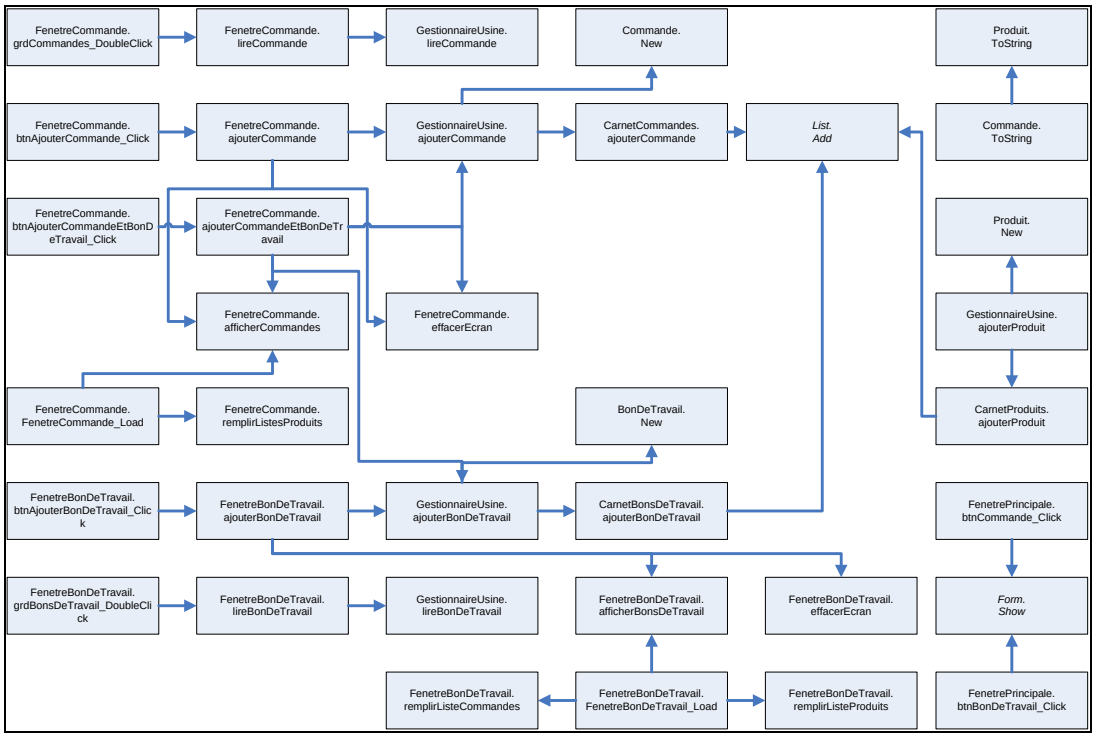


Figure A2.10 Graphe d'appels de méthodes

L'analyse du code source donne également la table de correspondance action/méthode du tableau A2.2 (algorithme 1.1.3), la table de correspondance classe/fenêtre du tableau A2.3 (algorithme 1.1.5) et la table de correspondance symbole/composant graphique du tableau A2.4 (algorithme 1.1.6). Celles-ci sont construites en associant des éléments du modèle d'action et du modèle de l'interface graphique à des éléments du modèle du programme, par correspondance de noms. Il est à noter que même si, par exemple, le programme contient trois méthodes qui se nomment *ajouterCommande* (une dans la classe *FenetreCommande*, une dans la classe *GestionnaireUsine* et l'autre dans la classe *CarnetProduits*), c'est celle de la classe *GestionnaireUsine* qui est associée à l'action *ajouterCommande*, car ces deux éléments sont situés dans une classe de même nom et possèdent la même signature. Les classes *FenetrePrincipale*, *FenetreCommande* et *FenetreBonDeTravail* sont identifiées comme fenêtres, car elles héritent de la classe *Form*.

Tableau A2.2 Table de correspondance action/méthode

Action	Méthode
lireCommande	GestionnaireUsine.lireCommande
lireBonDeTravail	GestionnaireUsine.lireBonDeTravail
ajouterCommande	GestionnaireUsine.ajouterCommande
ajouterBonDeTravail	GestionnaireUsine.ajouterBonDeTravail

Tableau A2.3 Table de correspondance classe/fenêtre

Classe	Fenêtre
FenetrePrincipale	FenetrePrincipale
FenetreCommande	FenetreCommande
FenetreBonDeTravail	FenetreBonDeTravail

Tableau A2.4 Table de correspondance symbole/composant graphique

Symbole	Composant Graphique
FenetrePrincipale.barreOutils (ToolStrip)	FenetrePrincipale.barreOutils (BarreOutils)
FenetrePrincipale.btnCommande (ToolStripButton)	FenetrePrincipale.btnCommande (BoutonBarreOutils)
FenetrePrincipale.btnBonDeTravail (ToolStripButton)	FenetrePrincipale.btnBonDeTravail (BoutonBarreOutils)
FenetreCommande.lblNoCommande (Label)	FenetreCommande.lblNoCommande (Etiquette)
FenetreCommande.txtNoCommande (TextBox)	FenetreCommande.txtNoCommande (ZoneDeTexte)
FenetreCommande.lblClient (Label)	FenetreCommande.lblClient (Etiquette)
FenetreCommande.txtClient (TextBox)	FenetreCommande.txtClient (ZoneDeTexte)
FenetreCommande.lblProduit (Label)	FenetreCommande.lblProduit (Etiquette)
FenetreCommande.cboProduit (ComboBox)	FenetreCommande.cboProduit (ListeDeroulante)
FenetreCommande.lblProduitDeRemplacement (Label)	FenetreCommande.lblProduitDeRemplacement (Etiquette)
FenetreCommande.cboProduitDeRemplacement (ComboBox)	FenetreCommande.cboProduitDeRemplacement (ListeDeroulante)
FenetreCommande.lblQuantite (Label)	FenetreCommande.lblQuantite (Etiquette)
FenetreCommande.nudQuantite (NumericUpDown)	FenetreCommande.nudQuantite (SelectionneurValeurNumerique)
FenetreCommande.lblPrix (Label)	FenetreCommande.lblPrix (Etiquette)
FenetreCommande.txtPrix (TextBox)	FenetreCommande.txtPrix (ZoneDeTexte)
FenetreCommande.btnAjouterCommande (Button)	FenetreCommande.btnAjouterCommande (Bouton)
FenetreCommande.btnAjouterCommandeEtBonDeTravail (Button)	FenetreCommande.btnAjouterCommandeEtBonDeTravail (Bouton)
FenetreCommande.grdCommandes (DataGridView)	FenetreCommande.grdCommandes (Grille)
FenetreBonDeTravail.lblNoBonDeTravail (Label)	FenetreBonDeTravail.lblNoBonDeTravail (Etiquette)
FenetreBonDeTravail.txtNoBonDeTravail (TextBox)	FenetreBonDeTravail.txtNoBonDeTravail (ZoneDeTexte)
FenetreBonDeTravail.lblProduit (Label)	FenetreBonDeTravail.lblProduit (Etiquette)
FenetreBonDeTravail.cboProduit (ComboBox)	FenetreBonDeTravail.cboProduit (ListeDeroulante)

Symbole	Composant Graphique
FenetreBonDeTravail.lblQuantite (Label)	FenetreBonDeTravail.lblQuantite (Etiquette)
FenetreBonDeTravail.nudQuantite (NumericUpDown)	FenetreBonDeTravail.nudQuantite (SelectionneurValeurNumerique)
FenetreBonDeTravail.lblCommandeOrigine (Label)	FenetreBonDeTravail.lblCommandeOrigine (Etiquette)
FenetreBonDeTravail.cboCommandeOrigine (ComboBox)	FenetreBonDeTravail.cboCommandeOrigine (ListeDeroulante)
FenetreBonDeTravail.btnAjouterBonDeTravail (Button)	FenetreBonDeTravail.btnAjouterBonDeTravail (Bouton)
FenetreBonDeTravail.grdBonsDeTravail (DataGridView)	FenetreBonDeTravail.grdBonsDeTravail (Grille)

L'analyse des chemins du graphe d'appels de méthodes de la figure A2.10 donne la table des chemins d'appel d'action du tableau A2.5 (algorithme 1.1.4). Pour rappel, les chemins retenus sont ceux débutant par un gestionnaire d'événement et se terminant par une méthode correspondant à une action de la table de correspondance action/méthode.

Tableau A2.5 Table des chemins d'appel d'action

Action	Chemin d'appel
lireCommande	FenetreCommande.grdCommandes_DoubleClick → FenetreCommande.lireCommande → GestionnaireUsine.lireCommande
lireBonDeTravail	FenetreBonDeTravail.grdBonsDeTravail_DoubleClick → FenetreBonDeTravail.lireBonDeTravail → GestionnaireUsine.lireBonDeTravail
ajouterCommande	FenetreCommande.btnAjouterCommande_Click → FenetreCommande.ajouterCommande → GestionnaireUsine.ajouterCommande
	FenetreCommande.btnAjouterCommandeEtBonDeTravail_Click → FenetreCommande.ajouterCommandeEtBonDeTravail → GestionnaireUsine.ajouterCommande
ajouterBonDeTravail	FenetreBonDeTravail.btnAjouterBonDeTravail_Click → FenetreBonDeTravail.ajouterBonDeTravail → GestionnaireUsine.ajouterBonDeTravail
	FenetreCommande.btnAjouterCommandeEtBonDeTravail_Click → FenetreCommande.ajouterCommandeEtBonDeTravail → GestionnaireUsine.ajouterBonDeTravail

A2.3.2 Génération du modèle de traduction

Cette section explore la génération du modèle de traduction de l'action « ajouter un bon de travail » (algorithme 1.2). Le traitement des deux actions de lecture est similaire à ce qui a été

présenté à l'étude de cas 1 et ne sera donc pas repris ici. Le traitement de l'action « ajouter une commande » est similaire à celui de l'ajout d'un bon de travail.

L'en-tête de la fonction de traduction est d'abord écrit, suivi du début d'un bloc « *Select* », puisque l'action possède plus d'un chemin d'appel. L'étape suivante est de calculer, pour chaque chemin d'appel de cette action, la table paramètres/expressions (algorithme 1.2.1.1).

Pour le premier chemin d'appel, la liste d'expressions de chacun des paramètres est initialisée à vide au départ (tableau A2.6, chemin 1, version 1). Le code du chemin d'appel est parcouru, en sens inverse. Lorsque l'appel à la méthode *GestionnaireUsine.ajouterBonDeTravail* est trouvé dans l'avant-dernière méthode du chemin (section A2.5, ligne de code 378), la liste d'expressions de chacun des paramètres est remplie avec les paramètres effectifs de l'appel (tableau A2.6, chemin 1, version 2).

Tableau A2.6 Table paramètres/expressions pour l'action « ajouter un bon de travail »

Chemin	Paramètres	Expressions (version 1)	Expressions (version 2)	Expressions (version 3)
1	produit	{}	{cboProduit.SelectedItem}	
	quantite	{}	{nudQuantite.Value}	
	commandeOrigine	{}	{cboCommandeOrigine. SelectedItem}	
2	produit	{}	{produit}	{cboProduit.SelectedItem, cboProduitDeRemplacement. SelectedItem}
	quantite	{}	{nudQuantite.Value}	{nudQuantite.Value}
	commandeOrigine	{}	{nouvelleCommande}	{nouvelleCommande}

Pour le deuxième chemin d'appel, la liste d'expressions de chacun des paramètres est initialisée à vide au départ (tableau A2.6, chemin 2, version 1). Le code du chemin d'appel est parcouru, en sens inverse. Lorsque l'appel à la méthode *GestionnaireUsine.ajouterBonDeTravail* est trouvé dans l'avant-dernière méthode du chemin

(section A2.5, ligne de code 312), la liste d'expressions de chacun des paramètres est remplie avec les paramètres effectifs de l'appel (tableau A2.6, chemin 2, version 2). Le parcours inverse du code continue par la suite. Des affectations à la variable *produit* sont rencontrées (provenant de deux séquences d'instructions différentes, voir tableau A2.1) (section A2.5, lignes de code 306 et 308). Cette variable est donc remplacée dans la partie droite de la table paramètres/expressions par les valeurs qui lui sont affectées (tableau A2.6, chemin 2, version 3).

Une fois la table paramètres/expressions construite, chaque méthode de chacun des chemins d'appel est parcourue afin de générer le code de traduction (algorithmes 1.2.1 et 1.2.1.2). Ce paragraphe ne décrit que le traitement du deuxième chemin, car celui-ci possède des caractéristiques différentes de ce qui a été étudié dans l'étude de cas 1. Le début du bloc « *Serialize* » est écrit. La première méthode du chemin est *FenetreCommande.btnAjouterCommandeEtBonDeTravail_Click*. Puisque la méthode appartient à une classe (*FenetreCommande*) identifiée comme fenêtre dans la table de correspondance classe/fenêtre et que la pile des fenêtres ouvertes est vide, cette fenêtre est empilée dans la pile des fenêtres ouvertes. L'algorithme 1.2.1.2.1 est appelé pour traiter les paramètres. Puisque, selon la table paramètres/expressions, la source des paramètres *produit* et *quantite* est une expression dont le symbole est identifié comme composant graphique dans la table de correspondance symbole/composant graphique et que la fenêtre de ce composant est présentement active, ces paramètres doivent être traités. Le début d'un bloc « *Permute* » est d'abord écrit, puisque plus d'un paramètre peut être traité au cours de cette itération. Le code sélectionnant le produit et tapant la quantité est écrit. Puisque, selon la table paramètres/expressions, le paramètre *produit* peut avoir comme source deux composants graphiques différents, le code sélectionnant le produit dans chacune des listes déroulantes doit se trouver dans un bloc « *Select* ». Les paramètres/expressions de *produit* et *quantite* sont retirés de la table puisqu'ils ont été traités. Le paramètre/expression du paramètre *commandeOrigine* ne sera jamais utilisé, puisqu'il ne provient pas d'un composant graphique. Une fois cela fait, l'algorithme détermine que l'algorithme 1.2.1.2.2 est applicable pour la première méthode du chemin, puisqu'il s'agit d'un gestionnaire d'événement. Le code

déclenchant cet événement doit donc être écrit. Plus aucun traitement n'est possible par la suite pour ce chemin. Le modèle de traduction obtenu en sortie est celui de la figure A2.11.

```
void lireCommande(Commande commande) {
    Serialize {
        Execute(GUI.grdCommandes.select, commande);
        Execute(GUI.grdCommandes.doubleclick);
    }
}

void lireBonDeTravail(BonDeTravail bonDeTravail) {
    Serialize {
        Execute(GUI.grdBonsDeTravail.select, bonDeTravail);
        Execute(GUI.grdBonsDeTravail.doubleclick);
    }
}

void ajouterCommande(string client, Produit produit, Produit produitDeRemplacement, int quantite, double prix) {
    Select {
        Serialize {
            Permute {
                Execute(GUI.txtClient.type, client);
                Execute(GUI.cboProduit.select, produit);
                Execute(GUI.cboProduitDeRemplacement.select, produitDeRemplacement);
                Execute(GUI.nudQuantite.type, quantite);
                Execute(GUI.txtPrix.type, prix);
            }
            Execute(GUI.btnAjouterCommande.click);
        }
        Serialize {
            Permute {
                Execute(GUI.txtClient.type, client);
                Execute(GUI.cboProduit.select, produit);
                Execute(GUI.cboProduitDeRemplacement.select, produitDeRemplacement);
                Execute(GUI.nudQuantite.type, quantite);
                Execute(GUI.txtPrix.type, prix);
            }
            Execute(GUI.btnAjouterCommandeEtBonDeTravail.click);
        }
    }
}

void ajouterBonDeTravail(Produit produit, int quantite, Commande commandeOrigine) {
    Select {
        Serialize {
            Permute {
                Execute(GUI.cboProduit.select, produit);
                Execute(GUI.nudQuantite.type, quantite);
                Execute(GUI.cboCommandeOrigine.select, commandeOrigine);
            }
            Execute(GUI.btnAjouterBonDeTravail.click);
        }
        Serialize {
            Permute {
                Execute(GUI.nudQuantite.type, quantite);
                Select {
                    Execute(GUI.cboProduit.select, produit);
                    Execute(GUI.cboProduitDeRemplacement.select, produit);
                }
            }
            Execute(GUI.btnAjouterCommandeEtBonDeTravail.click);
        }
    }
}
```

Figure A2.11 Modèle de traduction obtenu en sortie

A2.4 Évaluation

Afin de vérifier si l'algorithme de génération du modèle de traduction fonctionne bien sur cette étude de cas, les critères de comparaison de deux modèles de traduction ont été appliqués sur les modèles de traduction obtenu et attendu (tableau A2.7). L'analyse montre que les deux modèles ne sont pas équivalents, car un des critères de comparaison n'est pas rencontré

(présence des mêmes instructions). Ces instructions sont celles permettant d'ouvrir la fenêtre de commande ou la fenêtre de bon de travail à partir de la fenêtre principale.

Tableau A2.7 Évaluation de l'équivalence entre les modèles de traduction obtenu et attendu

	Critère rencontré ?	Commentaire
1	Oui	
2	Oui	
3	Non	Le modèle de traduction obtenu ne contient pas les instructions permettant d'ouvrir les fenêtres appropriées à partir de la barre d'outils de la fenêtre principale.
4	Oui	L'utilisateur a le choix de faire une seule opération à la fois (ajouter une commande ou un bon de travail) ou de faire ces deux opérations en même temps. Ces choix sont contenus dans un bloc « *Select* » pour les deux modèles.
5	Oui	L'écriture dans les champs utilisateur doit d'abord se faire. Le bouton effectuant l'action doit ensuite être cliqué. Ces étapes sont contenues, dans l'ordre, dans un bloc « *Serialize* » pour les deux modèles.
6	Oui	L'ordre de l'écriture dans les champs utilisateur peut être permuté. Cela se trouve dans un bloc « *Permute* » pour les deux modèles.
7	Oui	
8	Oui	
9	N/A	

A2.5 Code source

```
001    Public Class Produit
002        Private Shared prochainNumeroProduit As Integer = 1
003
004        Private _noProduit As Integer
005        Private _nomProduit As String
006
007        Public ReadOnly Property NoProduit As Integer
008            Get
009                Return _noProduit
010            End Get
011        End Property
012
```

```
013         Public ReadOnly Property NomProduit As String
014             Get
015                 Return _nomProduit
016             End Get
017         End Property
018 
019         Public Sub New(ByVal nomProduit As String)
020             Me._noProduit = prochainNumeroProduit
021             Me._nomProduit = nomProduit
022 
023             prochainNumeroProduit += 1
024         End Sub
025 
026         Public Overrides Function ToString() As String
027             Return NomProduit
028         End Function
029     End Class
030 
031     Public Class Commande
032         Private Shared prochainNumeroCommande As Integer = 1
033 
034         Private _noCommande As Integer
035         Private _client As String
036         Private _produit As Produit
037         Private _produitDeRemplacement As Produit
038         Private _quantite As Integer
039         Private _prix As Double
040 
041         Public ReadOnly Property NoCommande As Integer
042             Get
043                 Return _noCommande
044             End Get
045         End Property
046 
047         Public ReadOnly Property Client As String
048             Get
049                 Return _client
050             End Get
051         End Property
052 
053         Public ReadOnly Property Produit As Produit
054             Get
055                 Return _produit
056             End Get
057         End Property
```

```
058
059         Public ReadOnly Property ProduitDeRemplacement As Produit
060             Get
061                 Return _produitDeRemplacement
062             End Get
063         End Property
064
065         Public ReadOnly Property Quantite As Integer
066             Get
067                 Return _quantite
068             End Get
069         End Property
070
071         Public ReadOnly Property Prix As Double
072             Get
073                 Return _prix
074             End Get
075         End Property
076
077         Public Sub New(ByVal client As String, ByVal produit As Produit, ByVal
    produitDeRemplacement As Produit, ByVal quantite As Integer, ByVal prix As Double)
078             Me._noCommande = prochainNumeroCommande
079             Me._client = client
080             Me._produit = produit
081             Me._produitDeRemplacement = produitDeRemplacement
082             Me._quantite = quantite
083             Me._prix = prix
084
085             prochainNumeroCommande += 1
086         End Sub
087
088         Public Overrides Function ToString() As String
089             Return CStr(NoCommande) & " - " & Client & " - " & Produit.ToString() & " (x "
    & CStr(Quantite) & ")"
090         End Function
091     End Class
092
093     Public Class BonDeTravail
094         Private Shared _prochainNumeroBonDeTravail As Integer = 1
095
096         Private _noBonDeTravail As Integer
097         Private _produit As Produit
098         Private _quantite As Integer
099         Private _commandeOrigine As Commande
100
```

```
101         Public ReadOnly Property NoBonDeTravail As Integer
102             Get
103                 Return _noBonDeTravail
104             End Get
105         End Property
106 
107         Public ReadOnly Property Produit As Produit
108             Get
109                 Return _produit
110             End Get
111         End Property
112 
113         Public ReadOnly Property Quantite As Integer
114             Get
115                 Return _quantite
116             End Get
117         End Property
118 
119         Public ReadOnly Property CommandeOrigine As Commande
120             Get
121                 Return _commandeOrigine
122             End Get
123         End Property
124     End Class
125 
126         Public Sub New(ByVal produit As Produit, ByVal quantite As Integer, ByVal
    commandeOrigine As Commande)
127             Me._noBonDeTravail = _prochainNumeroBonDeTravail
128             Me._produit = produit
129             Me._quantite = quantite
130             Me._commandeOrigine = commandeOrigine
131 
132             _prochainNumeroBonDeTravail += 1
133         End Sub
134 
135     Public Class CarnetProduits
136         Private _produits As New List(Of Produit)
137 
138         Public ReadOnly Property Produits As List(Of Produit)
139             Get
140                 Return New List(Of Produit)(_produits)
141             End Get
142         End Property
143 
144         Public Sub ajouterProduit(ByVal produit As Produit)
```

```
145             _produits.Add(produit)
146         End Sub
147     End Class
148 
149     Public Class CarnetCommandes
150         Private _commandes As New List(Of Commande)
151 
152         Public ReadOnly Property Commandes As List(Of Commande)
153             Get
154                 Return New List(Of Commande)(_commandes)
155             End Get
156         End Property
157 
158         Public Sub ajouterCommande(ByVal commande As Commande)
159             _commandes.Add(commande)
160         End Sub
161     End Class
162 
163     Public Class CarnetBonsDeTravail
164         Private _bonsDeTravail As New List(Of BonDeTravail)
165 
166         Public ReadOnly Property BonsDeTravail As List(Of BonDeTravail)
167             Get
168                 Return New List(Of BonDeTravail)(_bonsDeTravail)
169             End Get
170         End Property
171 
172         Public Sub ajouterBonDeTravail(ByVal bonDeTravail As BonDeTravail)
173             _bonsDeTravail.Add(bonDeTravail)
174         End Sub
175     End Class
176 
177     Public Class GestionnaireUsine
178         Private _carnetProduits As New CarnetProduits()
179         Private _carnetCommandes As New CarnetCommandes()
180         Private _carnetBonsDeTravail As New CarnetBonsDeTravail()
181 
182         Public ReadOnly Property Produits As List(Of Produit)
183             Get
184                 Return _carnetProduits.Produits
185             End Get
186         End Property
187 
188         Public ReadOnly Property Commandes As List(Of Commande)
189             Get
```

```
190             Return _carnetCommandes.Commandes
191         End Get
192     End Property
193
194     Public ReadOnly Property BonsDeTravail As List(Of BonDeTravail)
195         Get
196             Return _carnetBonsDeTravail.BonsDeTravail
197         End Get
198     End Property
199
200     Public Sub lireCommande(ByVal commande As Commande)
201         ' Méthode bidon pour faire fonctionner l'algorithme de génération du modèle de
    traduction
202     End Sub
203
204     Public Sub lireBonDeTravail(ByVal bonDeTravail As BonDeTravail)
205         ' Méthode bidon pour faire fonctionner l'algorithme de génération du modèle de
    traduction
206     End Sub
207
208     Public Sub ajouterProduit(ByVal nomProduit As String)
209         Dim nouveauProduit As New Produit(nomProduit)
210
211         _carnetProduits.ajouterProduit(nouveauProduit)
212     End Sub
213
214     Public Function ajouterCommande(ByVal client As String, ByVal produit As Produit,
    ByVal produitDeRemplacement As Produit, ByVal quantite As Integer, ByVal prix As
    Double) As Commande
215         Dim nouvelleCommande As New Commande(client, produit, produitDeRemplacement,
    quantite, prix)
216
217         _carnetCommandes.ajouterCommande(nouvelleCommande)
218
219         Return nouvelleCommande
220     End Function
221
222     Public Function ajouterBonDeTravail(ByVal produit As Produit, ByVal quantite As
    Integer, ByVal commandeOrigine As Commande) As BonDeTravail
223         Dim nouveauBonDeTravail As New BonDeTravail(produit, quantite, commandeOrigine)
224
225         _carnetBonsDeTravail.ajouterBonDeTravail(nouveauBonDeTravail)
226
227         Return nouveauBonDeTravail
228     End Function
```

```
End Class

Public Class FenetreCommande
    ''''''''''''''
    ' Événements '
    ''''''''''''''

    Private Sub FenetreCommande_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        remplirListesProduits()
        afficherCommandes()
    End Sub

    Private Sub grdCommandes_DoubleClick(ByVal sender As Object, ByVal e As System.EventArgs) Handles grdCommandes.DoubleClick
        lireCommande()
    End Sub

    Private Sub btnAjouterCommande_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnAjouterCommande.Click
        ajouterCommande()
    End Sub

    Private Sub btnAjouterCommandeEtBonDeTravail_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnAjouterCommandeEtBonDeTravail.Click
        ajouterCommandeEtBonDeTravail()
    End Sub

    ''''''''''''''''''''''''
    ' Fonctions principales '
    ''''''''''''''''''''''''

    Private Sub remplirListesProduits()
        cboProduit.DataSource = [Global].gestionnaireUsine.Produits
        cboProduit.SelectedIndex = -1

        cboProduitDeRemplacement.DataSource = [Global].gestionnaireUsine.Produits
        cboProduitDeRemplacement.SelectedIndex = -1
    End Sub

    Private Sub afficherCommandes()
        grdCommandes.DataSource = [Global].gestionnaireUsine.Commandes
    End Sub

    Private Sub effacerEcran()
```

```
270             txtNoCommande.Text = ""
271             txtClient.Text = ""
272             cboProduit.SelectedIndex = -1
273             cboProduitDeRemplacement.SelectedIndex = -1
274             nudQuantite.Value = 1
275             txtPrix.Text = ""
276         End Sub
277 
278         Private Sub lireCommande()
279             Dim commande As Commande
280 
281             commande = CType(grdCommandes.CurrentRow.DataBoundItem, Commande)
282 
283             txtNoCommande.Text = commande.NoCommande
284             txtClient.Text = commande.Client
285             cboProduit.SelectedItem = commande.Produit
286             cboProduitDeRemplacement.SelectedItem = commande.ProduitDeRemplacement
287             nudQuantite.Value = commande.Quantite
288             txtPrix.Text = commande.Prix
289 
290             ' Appel de méthode bidon pour faire fonctionner l'algorithme de génération du
    modèle de traduction
291             [Global].gestionnaireUsine.lireCommande(commande)
292         End Sub
293 
294         Private Sub ajouterCommande()
295             [Global].gestionnaireUsine.ajouterCommande(txtClient.Text,
    CType(cboProduit.SelectedItem, Produit), CType(cboProduitDeRemplacement.SelectedItem,
    Produit), CInt(nudQuantite.Value), CDbl(txtPrix.Text))
296 
297             effacerEcran()
298             afficherCommandes()
299         End Sub
300 
301         Private Sub ajouterCommandeEtBonDeTravail()
302             Dim nouvelleCommande As Commande
303             Dim produit As Produit
304 
305             If cboProduitDeRemplacement.SelectedIndex <> -1 Then
306                 produit = cboProduitDeRemplacement.SelectedItem
307             Else
308                 produit = cboProduit.SelectedItem
309             End If
310 
311             nouvelleCommande = [Global].gestionnaireUsine.ajouterCommande(txtClient.Text,
```

```
CType(cboProduit.SelectedItem, Produit), CType(cboProduitDeRemplacement.SelectedItem,
Produit), CInt(nudQuantite.Value), CDbl(txtPrix.Text))
312             [Global].gestionnaireUsine.ajouterBonDeTravail(produit,
CInt(nudQuantite.Value), nouvelleCommande)
313
314             effacerEcran()
315             afficherCommandes()
316         End Sub
317     End Class
318
319     Public Class FenetreBonDeTravail
320         ''''''''''''''
321         ' Événements '
322         ''''''''''''''
323
324         Private Sub FenetreBonDeTravail_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
325             remplirListeProduits()
326             remplirListeCommandes()
327             afficherBonsDeTravail()
328         End Sub
329
330         Private Sub grdBonsDeTravail_DoubleClick(ByVal sender As Object, ByVal e As
System.EventArgs) Handles grdBonsDeTravail.DoubleClick
331             lireBonDeTravail()
332         End Sub
333
334         Private Sub btnAjouterBonDeTravail_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAjouterBonDeTravail.Click
335             ajouterBonDeTravail()
336         End Sub
337
338         ''''''''''''''''''''''''
339         ' Fonctions principales '
340         ''''''''''''''''''''''''
341
342         Private Sub remplirListeProduits()
343             cboProduit.DataSource = [Global].gestionnaireUsine.Produits
344             cboProduit.SelectedIndex = -1
345         End Sub
346
347         Private Sub remplirListeCommandes()
348             cboCommandeOrigine.DataSource = [Global].gestionnaireUsine.Commandes
349             cboCommandeOrigine.SelectedIndex = -1
350         End Sub
```

```
351
352         Private Sub afficherBonsDeTravail()
353             grdBonsDeTravail.DataSource = [Global].gestionnaireUsine.BonsDeTravail
354         End Sub
355
356         Private Sub effacerEcran()
357             txtNoBonDeTravail.Text = ""
358             cboProduit.SelectedIndex = -1
359             nudQuantite.Value = 1
360             cboCommandeOrigine.SelectedIndex = -1
361         End Sub
362
363         Private Sub lireBonDeTravail()
364             Dim bonDeTravail As BonDeTravail
365
366             bonDeTravail = CType(grdBonsDeTravail.CurrentRow.DataBoundItem, BonDeTravail)
367
368             txtNoBonDeTravail.Text = bonDeTravail.NoBonDeTravail
369             cboProduit.SelectedItem = bonDeTravail.Produit
370             nudQuantite.Value = bonDeTravail.Quantite
371             cboCommandeOrigine.SelectedItem = bonDeTravail.CommandeOrigine
372
373             ' Appel de méthode bidon pour faire fonctionner l'algorithme de génération du
    modèle de traduction
374             [Global].gestionnaireUsine.lireBonDeTravail(bonDeTravail)
375         End Sub
376
377         Private Sub ajouterBonDeTravail()
378             [Global].gestionnaireUsine.ajouterBonDeTravail(CType(cboProduit.SelectedItem,
    Produit), CInt(nudQuantite.Value), CType(cboCommandeOrigine.SelectedItem, Commande))
379
380             effacerEcran()
381             afficherBonsDeTravail()
382         End Sub
383     End Class
384
385     Public Class FenetrePrincipale
386         Private Sub btnCommande_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnCommande.Click
387             Dim fenetreCommande As New FenetreCommande()
388             fenetreCommande.MdiParent = Me
389             fenetreCommande.Show()
390         End Sub
391
392         Private Sub btnBonDeTravail_Click(ByVal sender As System.Object, ByVal e As
```

```
    System.EventArgs) Handles btnBonDeTravail.Click
393             Dim fenetreBonDeTravail As New FenetreBonDeTravail()
394             fenetreBonDeTravail.MdiParent = Me
395             fenetreBonDeTravail.Show()
396         End Sub
397     End Class
398
399     Public Class [Global]
400         Public Shared gestionnaireUsine As New GestionnaireUsine()
401     End Class
```

Annexe 3

Étude de cas 3

Cette annexe présente en détails la troisième étude de cas ayant servi à tester l'algorithme de génération du modèle de traduction. Les particularités de cet exemple sont la présence de plusieurs fenêtres et l'utilisation de boîtes de dialogue.

A3.1 Présentation de l'étude de cas

Cette étude de cas consiste en un logiciel de courriels. La spécification des exigences de cette application est présentée à la figure A3.1. Son modèle d'action est présenté à la figure A3.2. Son interface graphique est présentée aux figures A3.3 et A3.4. Son code source complet, écrit en Visual Basic .NET, se trouve à la section A3.5. Le diagramme de classes de la figure A3.5 et les diagrammes de séquence des figures A3.6, A3.7 et A3.8 résument les aspects importants du code source.

L'application est un logiciel de courriels.

Les courriels peuvent être emmagasinés dans quatre répertoires :

- Boîte de réception
- Éléments envoyés
- Éléments supprimés
- Courrier indésirable

L'application peut effectuer cinq opérations :

- Lire un courriel
  - Le courriel s'affiche à l'écran.
- Écrire un courriel
  - Le destinataire est obligatoire.
  - Si l'utilisateur n'entre pas d'objet pour le courriel, l'application doit lui demander s'il veut envoyer le courriel quand même.
  - Le courriel écrit se retrouve dans le répertoire « Éléments envoyés ».
- Répondre à un courriel
  - Le courriel écrit se retrouve dans le répertoire « Éléments envoyés ».
- Transférer un courriel
  - Le destinataire est obligatoire.
  - Le courriel écrit se retrouve dans le répertoire « Éléments envoyés ».
- Supprimer un courriel
  - Le courriel supprimé se retrouve dans le répertoire « Éléments supprimés ».

Figure A3.1 Spécification des exigences

Inspiré de : Nguyen, D. H., Strooper, P. et Suess, J. G. (2010), p. 28

```
class Courriel {
    string! de;
    string! a;
    string cc;
    string objet;
    string message;
    datetime dateEnvoi;
}

class GestionnaireCourriels {
    string! proprietaireBoiteCourriel;

    Seq<Courriel> boiteReception = Seq{};
    Seq<Courriel> elementsEnvoyes = Seq{};
    Seq<Courriel> elementsSupprimes = Seq{};
    Seq<Courriel> courrierIndesirable = Seq{};

    [Action] void lireCourriel(Courriel! courriel) {

    }

    [Action] void envoyerCourriel(string! a, string cc, string objet, string message) {
        Courriel nouveauCourriel = new Courriel(proprietaireBoiteCourriel, a, cc, objet, message, Date.Now);
        elementsEnvoyes.Add(nouveauCourriel);
    }

    [Action] void supprimerCourriel(Courriel! courriel)
        boiteReception.Remove(courriel);
        elementsEnvoyes.Remove(courriel);
        courrierIndesirable.Remove(courriel);

        elementsSupprimes.Add(courriel);
    }
}
```

Figure A3.2 Modèle d'action

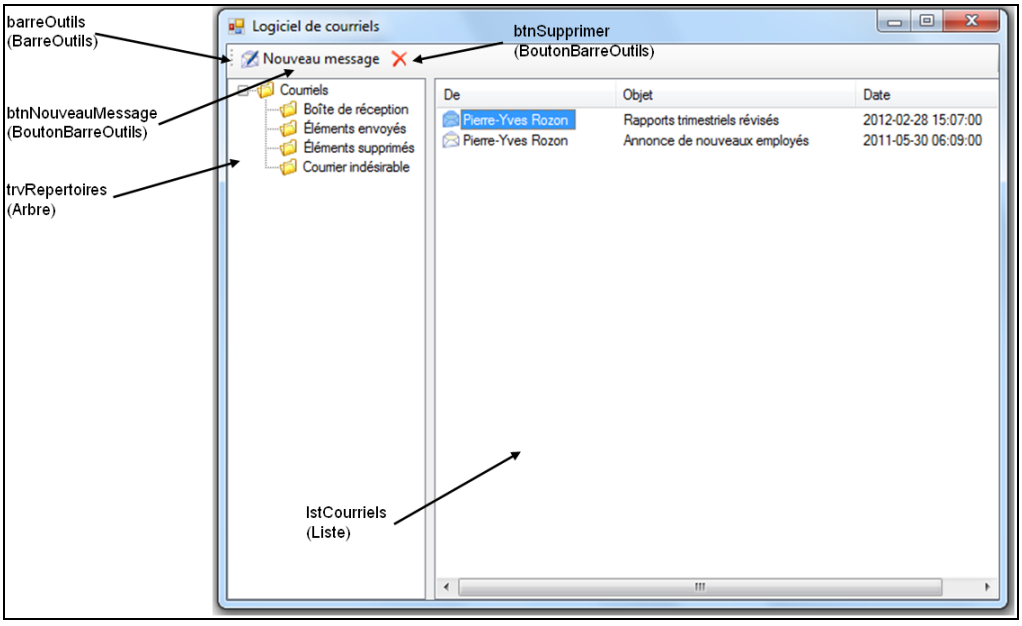


Figure A3.3 Fenêtre principale

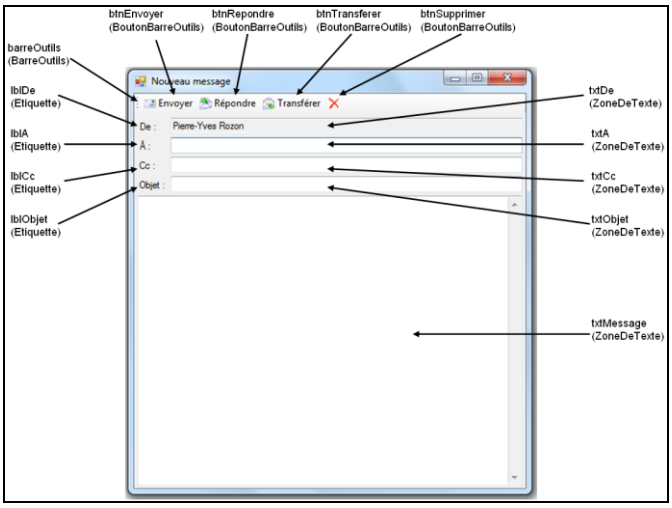


Figure A3.4 Fenêtre de lecture et de rédaction de courriel

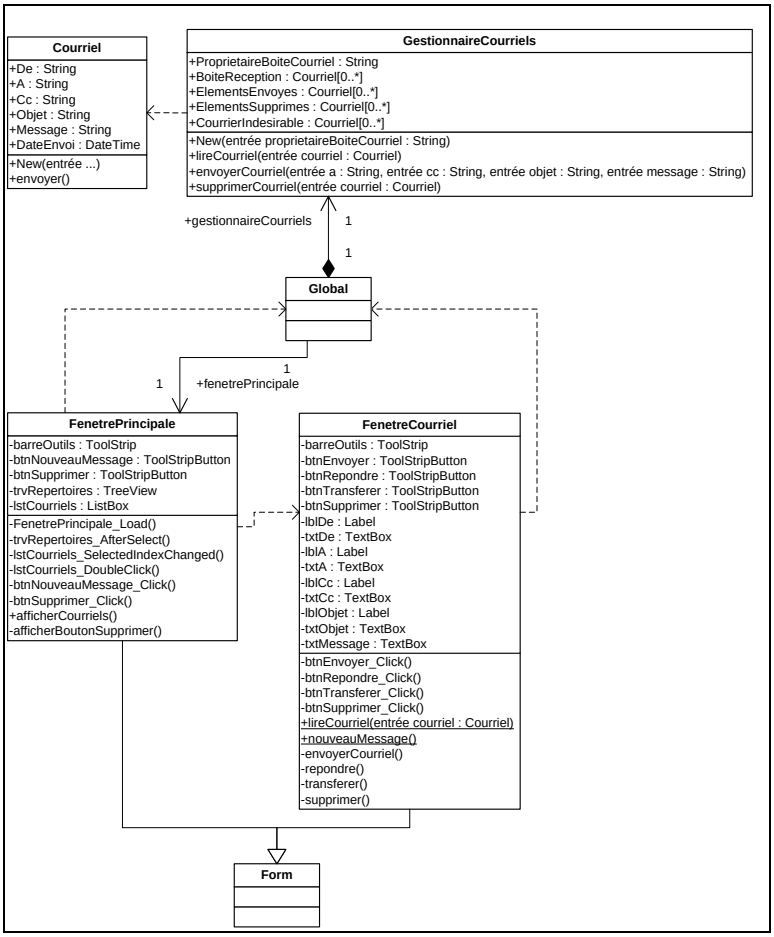


Figure A3.5 Diagramme de classes

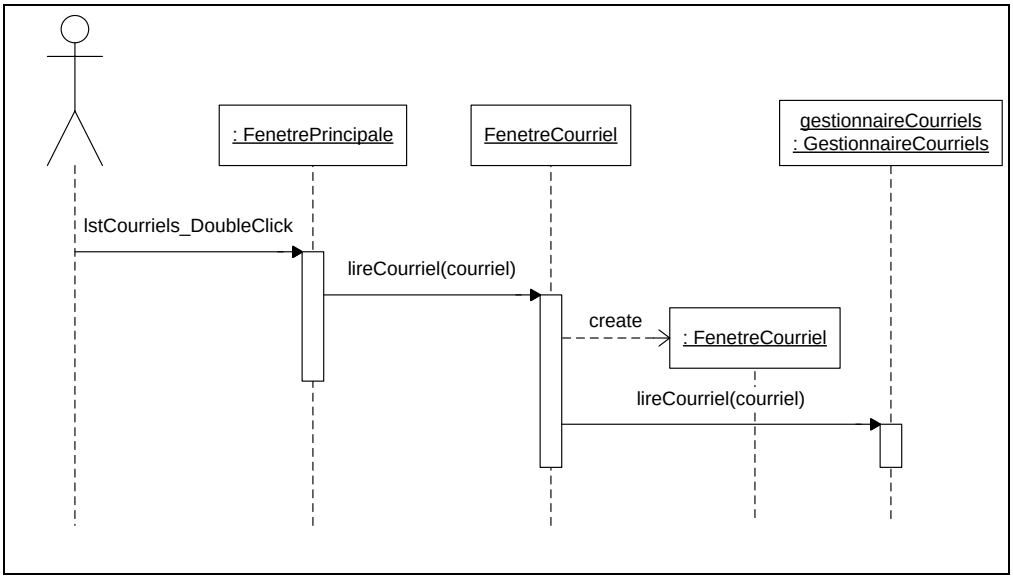


Figure A3.6 Diagramme de séquence de l'opération « lire un courriel »

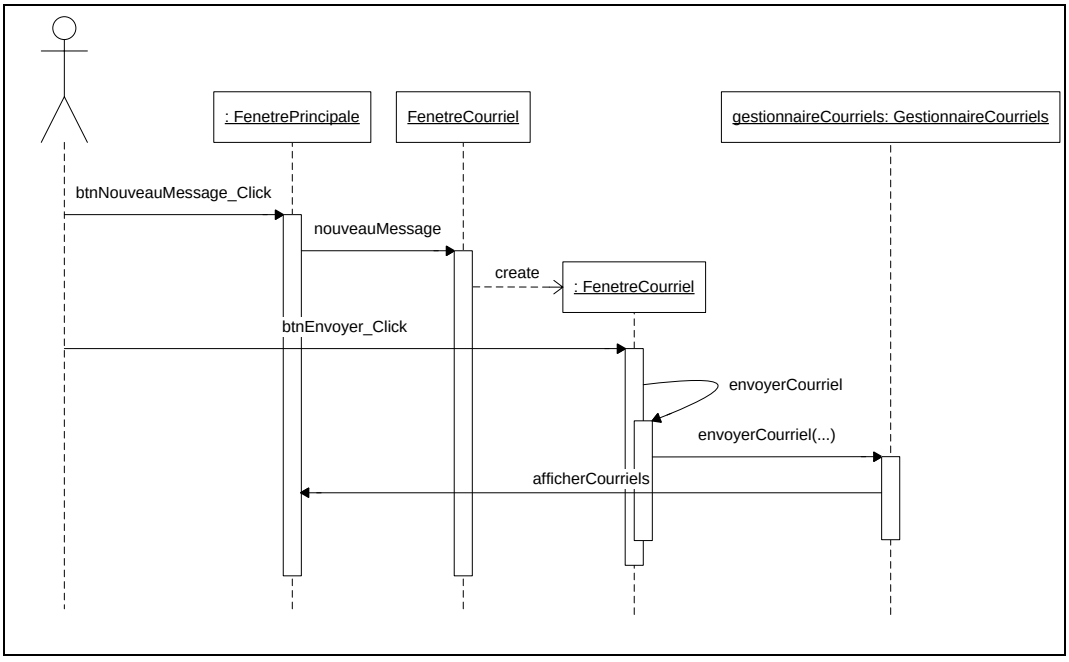


Figure A3.7 Diagramme de séquence de l'opération « écrire un courriel »

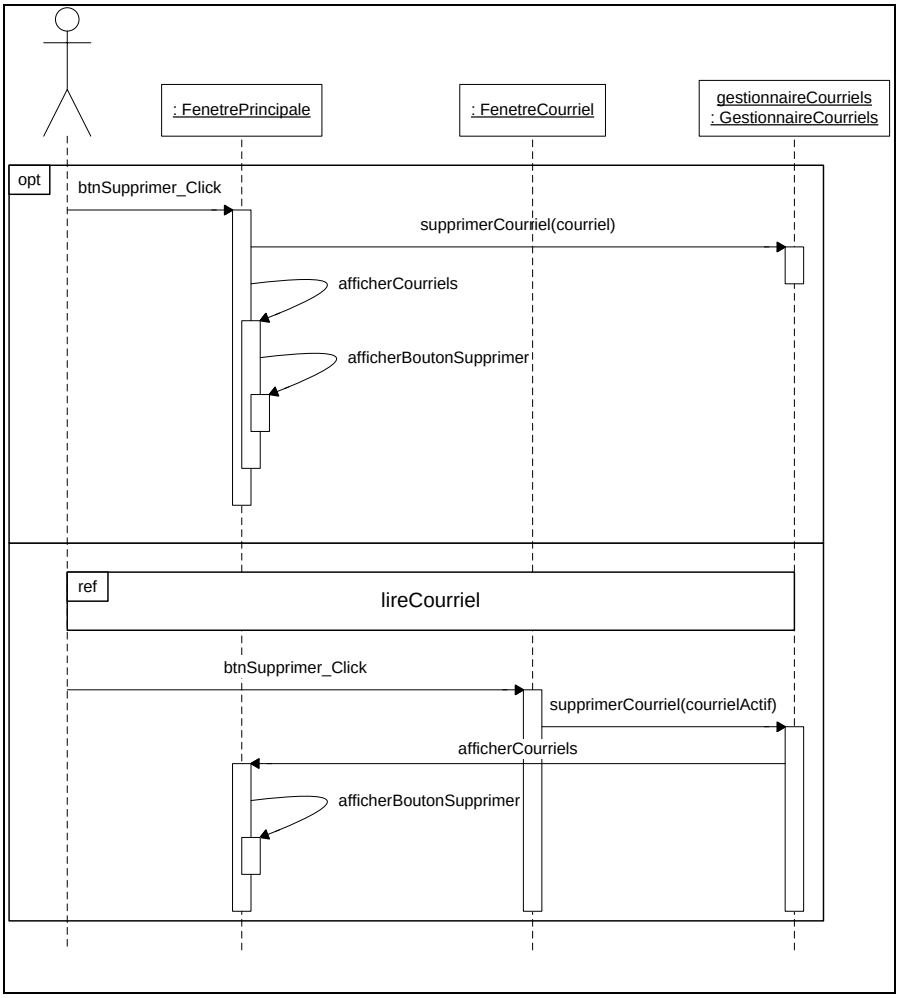


Figure A3.8 Diagramme de séquence de l'opération « supprimer un courriel »

L'utilisateur peut naviguer dans les répertoires à l'aide de l'arbre situé dans la section gauche de la fenêtre principale. Il peut lire un courriel en double-cliquant sur celui-ci dans la liste des courriels de la section droite de la fenêtre principale. Il peut écrire un nouveau courriel en cliquant sur le bouton « Nouveau message » puis en cliquant sur le bouton « Envoyer » de la fenêtre de rédaction de courriel. La suppression d'un courriel se fait en cliquant sur le bouton « supprimer » de la fenêtre principale ou de la fenêtre de lecture de courriel.

Le code source de l'application est divisé en cinq classes. La classe *Courriel* représente un courriel et contient ses informations. La classe *GestionnaireCourriels* contient les quatre

répertoires et leurs courriels. La classe *Global* contient les instances des variables globales. Les classes *FenetrePrincipale* et *FenetreCourriel* sont les fenêtres. Elles héritent de la classe *Form*, car cette application a été écrite en Visual Basic .NET.

Le diagramme de séquence de la figure A3.6 montre les interactions entre les objets lors de l'opération de lecture d'un courriel. L'utilisateur lance d'abord l'opération en double-cliquant sur le courriel dans la liste de courriels. Cela appelle une méthode *lireCourriel* dans la classe de la fenêtre de lecture de courriel qui crée la fenêtre et appelle à son tour une méthode *lireCourriel* dans le gestionnaire de courriels afin de lire le courriel.

Le diagramme de séquence de la figure A3.7 montre les interactions entre les objets lors de l'opération d'écriture d'un courriel. L'utilisateur clique d'abord sur le bouton « Nouveau message » dans la barre d'outils de la fenêtre principale. Il remplit les champs et clique sur le bouton « Envoyer ». Cela appelle une méthode *envoyerCourriel* dans la classe de la fenêtre. Celle-ci appelle la méthode *envoyerCourriel* du gestionnaire de courriels afin d'envoyer le courriel.

Le diagramme de séquence de la figure A3.8 montre les interactions entre les objets lors de l'opération de suppression d'un courriel. Cette opération peut s'effectuer de deux façons différentes. La première façon est de sélectionner un courriel dans la liste de courriels et de cliquer sur le bouton supprimer de la fenêtre principale. La deuxième façon est de lire un courriel et de cliquer sur le bouton supprimer de la fenêtre de lecture de courriel. Dans les deux cas, cela appelle la méthode *supprimerCourriel* du gestionnaire de courriels.

A3.2 Modèle de traduction attendu en sortie

Un exemple de modèle de traduction pour cette application, écrit par une personne, est présenté à la figure A3.9. Le modèle de traduction obtenu en sortie de l'algorithme doit être équivalent à celui-ci. Dans le cas de l'opération de lecture, un tel modèle doit sélectionner le courriel dans la liste et double-cliquer sur celle-ci. Dans le cas de l'opération d'envoi, il doit

cliquer sur le bouton « Nouveau message », entrer les informations dans la fenêtre et cliquer sur le bouton « Envoyer ». Dans le cas de l'opération de suppression, il doit sélectionner le bon courriel (en le sélectionnant dans la liste de la fenêtre principale ou en le lisant) et cliquer sur le bouton supprimer.

```
void lireCourriel(Courriel courriel) {
    Serialize {
        Execute(GUI.lstCourriels.select, courriel);
        Execute(GUI.lstCourriels.doubleclick);
    }
}

void envoyerCourriel(string a, string cc, string objet, string message) {
    Serialize {
        Execute(GUI.btnNouveauMessage.click);
        Permute {
            Execute(GUI.txtA.type, a);
            Execute(GUI.txtCc.type, cc);
            Execute(GUI.txtObjet.type, objet);
            Execute(GUI.txtMessage.type, message);
        }
        Execute(GUI.btnEnvoyer.click);
    }
}

void supprimerCourriel(Courriel courriel) {
    Serialize {
        Execute(GUI.lstCourriels.select, courriel);
        Select {
            Serialize {
                Execute(GUI.btnSupprimer.click);
            }
            Serialize {
                Execute(GUI.lstCourriels.doubleclick);
                Execute(GUI.btnSupprimer.click);
            }
        }
    }
}
```

Figure A3.9 Modèle de traduction attendu en sortie

A3.3 Fonctionnement de l'algorithme

Cette section montre, étape par étape, le fonctionnement de l'algorithme de génération du modèle de traduction sur cette étude de cas.

A3.3.1 Pré-calcul des structures de données

L'analyse du code source de la section A3.5 donne le graphe d'appels de méthodes de la figure A3.10 (algorithme 1.1.2). La table des séquences d'instructions obtenue (algorithme

1.1.1) n'est pas reproduite ici par souci de concision et également parce qu'elle n'est pas pertinente pour la partie de l'exemple qui sera étudiée.

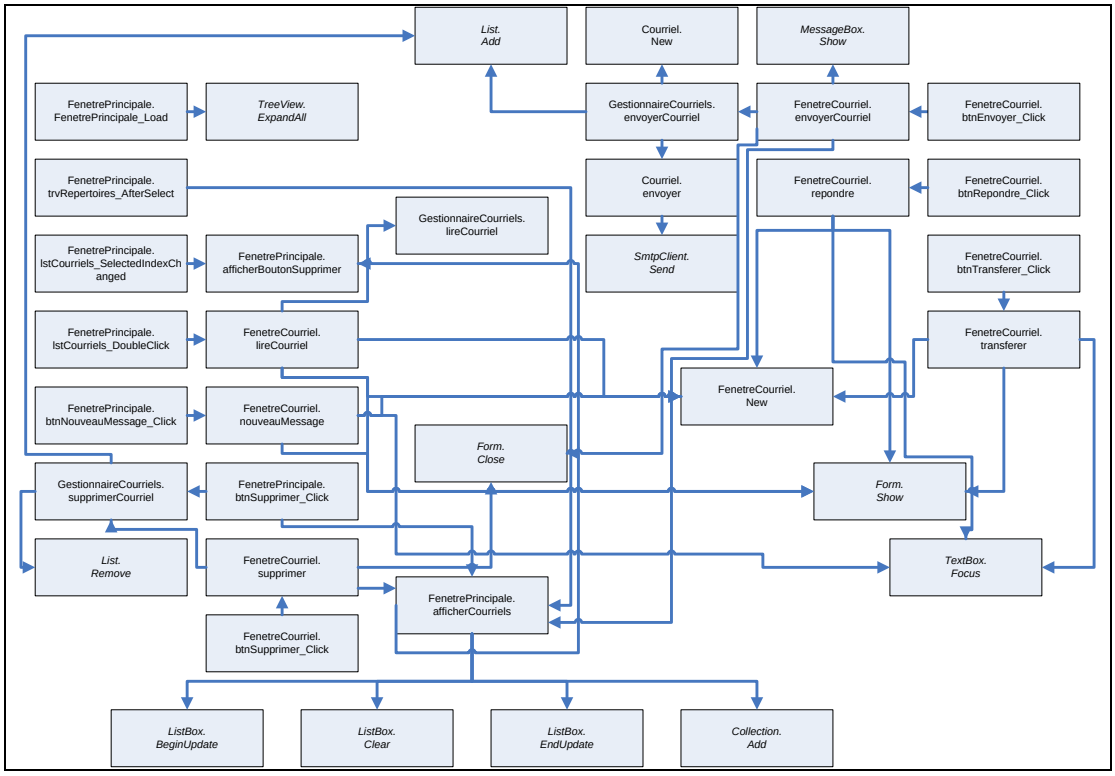


Figure A3.10 Graphe d'appels de méthodes

L'analyse du code source donne également la table de correspondance action/méthode du tableau A3.1 (algorithme 1.1.3), la table de correspondance classe/fenêtre du tableau A3.2 (algorithme 1.1.5) et la table de correspondance symbole/composant graphique du tableau A3.3 (algorithme 1.1.6). Celles-ci sont construites en associant des éléments du modèle d'action et du modèle de l'interface graphique à des éléments du modèle du programme, par correspondance de noms. Il est à noter que même si, par exemple, le programme contient deux méthodes qui se nomment *envoyerCourriel* (une dans la classe *FenetreCourriel* et l'autre dans la classe *GestionnaireCourriels*), c'est celle de la classe *GestionnaireCourriels* qui est associée à l'action *envoyerCourriel*, car ces deux éléments sont situés dans une classe de

même nom et possèdent la même signature. Les classes *FenetrePrincipale* et *FenetreCourriel* sont identifiées comme fenêtres, car elles héritent de la classe *Form*.

Tableau A3.1 Table de correspondance action/méthode

Action	Méthode
lireCourriel	GestionnaireCourriels.lireCourriel
envoyerCourriel	GestionnaireCourriels.envoyerCourriel
supprimerCourriel	GestionnaireCourriels.supprimerCourriel

Tableau A3.2 Table de correspondance classe/fenêtre

Classe	Fenêtre
FenetrePrincipale	FenetrePrincipale
FenetreCourriel	FenetreCommande

Tableau A3.3 Table de correspondance symbole/composant graphique

Symbole	Composant Graphique
FenetrePrincipale.barreOutils (ToolStrip)	FenetrePrincipale.barreOutils (BarreOutils)
FenetrePrincipale.btnNouveauMessage (ToolStripButton)	FenetrePrincipale.btnNouveauMessage (BoutonBarreOutils)
FenetrePrincipale.btnSupprimer (ToolStripButton)	FenetrePrincipale.btnSupprimer (BoutonBarreOutils)
FenetrePrincipale.trvRepertoires (TreeView)	FenetrePrincipale.trvRepertoires (Arbre)
FenetrePrincipale.lstCourriels (ListBox)	FenetrePrincipale.lstCourriels (Liste)
FenetreCourriel.barreOutils (ToolStrip)	FenetreCourriel.barreOutils (BarreOutils)
FenetreCourriel.btnEnvoyer (ToolStripButton)	FenetreCourriel.btnEnvoyer (BoutonBarreOutils)
FenetreCourriel.btnRepondre (ToolStripButton)	FenetreCourriel.btnRepondre (BoutonBarreOutils)
FenetreCourriel.btnTransferer (ToolStripButton)	FenetreCourriel.btnTransferer (BouronBarreOutils)
FenetreCourriel.btnSupprimer (ToolStripButton)	FenetreCourriel.btnSupprimer (BoutonBarreOutils)
FenetreCourriel.lblDe (Label)	FenetreCourriel.lblDe (Etiquette)
FenetreCourriel.txtDe (TextBox)	FenetreCourriel.txtDe (ZoneDeTexte)
FenetreCourriel.lblA (Label)	FenetreCourriel.lblA (Etiquette)
FenetreCourriel.txtA (TextBox)	FenetreCourriel.txtA (ZoneDeTexte)
FenetreCourriel.lblCc (Label)	FenetreCourriel.lblCc (Etiquette)
FenetreCourriel.txtCc (TextBox)	FenetreCourriel.txtCc (ZoneDeTexte)
FenetreCourriel.lblObjet (Label)	FenetreCourriel.lblObjet (Etiquette)
FenetreCourriel.txtObjet (TextBox)	FenetreCourriel.txtObjet (ZoneDeTexte)
FenetreCourriel.txtMessage (TextBox)	FenetreCourriel.txtMessage (ZoneDeTexte)

L'analyse des chemins du graphe d'appels de méthodes de la figure A3.10 donne la table des chemins d'appel d'action du tableau A3.4 (algorithme 1.1.4). Pour rappel, les chemins retenus sont ceux débutant par un gestionnaire d'événement et se terminant par une méthode correspondant à une action de la table de correspondance action/méthode.

Tableau A3.4 Table des chemins d'appel d'action

Action	Chemin d'appel
lireCourriel	FenetrePrincipale.lstCourriels_DoubleClick → FenetreCourriel.lireCourriel → GestionnaireCourriels.lireCourriel
envoyerCourriel	FenetreCourriel.btnEnvoyer_Click → FenetreCourriel.envoyerCourriel → GestionnaireCourriels.envoyerCourriel
supprimerCourriel	FenetrePrincipale.btnSupprimer_Click → GestionnaireCourriels.supprimerCourriel
	FenetreCourriel.btnSupprimer_Click → FenetreCourriel.supprimer → GestionnaireCourriels.supprimerCourriel

A3.3.2 Génération du modèle de traduction

Cette section explore la génération du modèle de traduction de l'action « envoyer un courriel » (algorithme 1.2). Le traitement des deux autres actions est similaire à ce qui a déjà été traité dans les deux autres études de cas et ne sera donc pas expliqué ici.

L'en-tête de la fonction de traduction est d'abord écrit. L'étape suivante est de calculer la table paramètres/expressions pour le seul chemin d'appel de cette action (algorithme 1.2.1.1).

La liste d'expressions de chacun des paramètres est initialisée à vide au départ (tableau A3.5, version 1). Le code du chemin d'appel est parcouru, en sens inverse. Lorsque l'appel à la méthode *GestionnaireCourriels.envoyerCourriel* est trouvé dans l'avant-dernière méthode du chemin (section A3.5, ligne de code 224), la liste d'expressions de chacun des paramètres est remplie avec les paramètres effectifs de l'appel (tableau A3.5, version 2). Le parcours inverse du code continue par la suite. Des affectations aux variables contenues dans la table paramètres/expressions sont rencontrées (section A3.5, lignes de code 212 à 215). Ces

variables sont donc remplacées dans la partie droite de la table paramètres/expressions par les valeurs qui leur sont affectées (tableau A3.5, version 3).

Tableau A3.5 Table paramètres/expressions pour l'action « envoyer un courriel »

Paramètres	Expressions (version 1)	Expressions (version 2)	Expressions (version 3)
a	{}	{a}	{txtA.Text}
cc	{}	{cc}	{txtCc.Text}
objet	{}	{objet}	{txtObjet.Text}
message	{}	{message}	{txtMessage.Text}

Une fois la table paramètres/expressions construite, chaque méthode du chemin d'appel est parcourue afin de générer le code de traduction (algorithmes 1.2.1 et 1.2.1.2). Le début du bloc « *Serialize* » est écrit. La première méthode du chemin est *FenetreCourriel.btnEnvoyer_Click*. Puisque la méthode appartient à une classe (*FenetreCourriel*) identifiée comme fenêtre dans la table de correspondance classe/fenêtre et que la pile des fenêtres ouvertes est vide, cette fenêtre est empilée dans la pile des fenêtres ouvertes. L'algorithme 1.2.1.2.1 est appelé pour traiter les paramètres. Puisque, selon la table paramètres/expressions, la source des paramètres est une expression dont le symbole est identifié comme composant graphique dans la table de correspondance symbole/composant graphique et que la fenêtre de ce composant est présentement active, ces paramètres doivent être traités. Le début d'un bloc « *Permute* » est d'abord écrit, puisque plus d'un paramètre peut être traité au cours de cette itération. Le code tapant les informations du courriel est écrit. Ces paramètres/expressions sont retirés de la table puisqu'ils ont été traités. Une fois cela fait, l'algorithme détermine que l'algorithme 1.2.1.2.2 est applicable pour la première méthode du chemin, puisqu'il s'agit d'un gestionnaire d'événement. Le code déclenchant cet événement doit donc être écrit. Plus aucun traitement n'est possible par la suite pour ce chemin. Le modèle de traduction obtenu en sortie est celui de la figure A3.11.

```
void lireCourriel(Courriel courriel) {
    Serialize {
        Execute(GUI.lstCourriels.select, courriel);
        Execute(GUI.lstCourriels.doubleclick);
    }
}

void envoyerCourriel(string a, string cc, string objet, string message) {
    Serialize {
        Permute {
            Execute(GUI.txtA.type, a);
            Execute(GUI.txtCc.type, cc);
            Execute(GUI.txtObjet.type, objet);
            Execute(GUI.txtMessage.type, message);
        }
        Execute(GUI.btnEnvoyer.click);
    }
}

void supprimerCourriel(Courriel courriel) {
    Select {
        Serialize {
            Execute(GUI.btnSupprimer.click);
        }
        Serialize {
            Execute(GUI.lstCourriels.select, courriel);
            Execute(GUI.btnSupprimer.click);
        }
    }
}
```

Figure A3.11 Modèle de traduction obtenu en sortie

A3.4 Évaluation

Afin de vérifier si l'algorithme de génération du modèle de traduction fonctionne bien sur cette étude de cas, les critères de comparaison de deux modèles de traduction ont été appliqués sur les modèles de traduction obtenu et attendu (tableau A3.6). L'analyse montre que les deux modèles ne sont pas équivalents, car un des critères de comparaison n'est pas rencontré (présence des mêmes instructions). Ces instructions sont celles permettant d'ouvrir la fenêtre de lecture et de rédaction de courriel à partir de la fenêtre principale (par exemple, il faut cliquer sur le bouton « Nouveau message » avant d'envoyer un courriel). Il s'agit du même problème qui avait été décelé à l'étude de cas 2. Cependant, cette analyse ne fait pas ressortir un autre problème de ce modèle de traduction, puisque ce même problème se retrouve dans le modèle de traduction attendu. En effet, à la ligne de code 220 de la section A3.5, l'application demande à l'utilisateur de confirmer s'il veut continuer lorsqu'il n'entre pas d'objet à son courriel. Le modèle de traduction doit donc répondre « oui » à cette question lorsque l'objet est une chaîne vide dans le scénario de test. Or, il est impossible d'exprimer cela avec le

langage AEFMAP, puisque ce langage ne contient pas d'instruction conditionnelle ni d'instruction permettant de cliquer sur un bouton d'une boîte de dialogue.

Tableau A3.6 Évaluation de l'équivalence entre les modèles de traduction obtenu et attendu

	Critère rencontré ?	Commentaire
1	Oui	
2	Oui	
3	Non	Le modèle de traduction obtenu ne contient pas les instructions permettant d'ouvrir les fenêtres appropriées à partir de la fenêtre principale (par exemple, le bouton « Nouveau message » avant d'envoyer un courriel).
4	Oui	L'utilisateur peut supprimer un courriel de deux façons différentes. Ces deux choix sont contenus dans un bloc « *Select* » pour les deux modèles.
5	Oui	Pour les opérations de lecture et de suppression d'un courriel, la sélection de ce dernier doit d'abord se faire avant les clics. Pour l'opération d'envoi d'un courriel, l'écriture dans les champs utilisateur doit se faire avant de cliquer sur le bouton. Ces étapes sont contenues, dans l'ordre, dans un bloc « *Serialize* » pour les deux modèles.
6	Oui	L'ordre de l'écriture dans les champs utilisateur peut être permuté. Cela se trouve dans un bloc « *Permute* » pour les deux modèles.
7	Oui	
8	Oui	
9	N/A	

A3.5 Code source

```
001    Public Class Courriel
002        Private _de As String
003        Private _a As String
004        Private _cc As String
005        Private _objet As String
006        Private _message As String
007        Private _dateEnvoi As DateTime
008
009        Public ReadOnly Property De As String
```

```
010             Get
011                 Return _de
012             End Get
013         End Property
014
015         Public ReadOnly Property A As String
016             Get
017                 Return _a
018             End Get
019         End Property
020
021         Public ReadOnly Property Cc As String
022             Get
023                 Return _cc
024             End Get
025         End Property
026
027         Public ReadOnly Property Objet As String
028             Get
029                 Return _objet
030             End Get
031         End Property
032
033         Public ReadOnly Property Message As String
034             Get
035                 Return _message
036             End Get
037         End Property
038
039         Public ReadOnly Property DateEnvoi As DateTime
040             Get
041                 Return _dateEnvoi
042             End Get
043         End Property
044
045         Public Sub New(ByVal de As String, ByVal a As String, ByVal cc As String, ByVal
        objet As String, ByVal message As String, ByVal dateEnvoi As DateTime)
046             Me._de = de
047             Me._a = a
048             Me._cc = cc
049             Me._objet = objet
050             Me._message = message
051             Me._dateEnvoi = dateEnvoi
052         End Sub
053
```

```
054         Public Sub envoyer()
055             Dim courriel As New System.Net.Mail.MailMessage
056             Dim smtpClient As New System.Net.Mail.SmtpClient
057 
058             courriel.Sender = New System.Net.Mail.MailAddress(De)
059             courriel.To.Add(A)
060             courriel.CC.Add(Cc)
061             courriel.Subject = Objet
062             courriel.Body = Message
063 
064             smtpClient.Send(courriel)
065         End Sub
066     End Class
067 
068     Public Class GestionnaireCourriels
069         Private _proprietaireBoiteCourriel As String
070 
071         Private _boiteReception As New List(Of Courriel)
072         Private _elementsEnvoyes As New List(Of Courriel)
073         Private _elementsSupprimes As New List(Of Courriel)
074         Private _courrierIndesirable As New List(Of Courriel)
075 
076         Public ReadOnly Property ProprietaireBoiteCourriel As String
077             Get
078                 Return _proprietaireBoiteCourriel
079             End Get
080         End Property
081 
082         Public ReadOnly Property BoiteReception As List(Of Courriel)
083             Get
084                 Return _boiteReception
085             End Get
086         End Property
087 
088         Public ReadOnly Property ElementsEnvoyes As List(Of Courriel)
089             Get
090                 Return _elementsEnvoyes
091             End Get
092         End Property
093 
094         Public ReadOnly Property ElementsSupprimes As List(Of Courriel)
095             Get
096                 Return _elementsSupprimes
097             End Get
098         End Property
```

```
099
100         Public ReadOnly Property CourrierIndesirable As List(Of Courriel)
101             Get
102                 Return _courrierIndesirable
103             End Get
104         End Property
105
106         Public Sub New(ByVal proprietaireBoiteCourriel As String)
107             Me._proprietaireBoiteCourriel = proprietaireBoiteCourriel
108         End Sub
109
110         Public Sub lireCourriel(ByVal courriel As Courriel)
111             ' Méthode bidon pour faire fonctionner l'algorithme de génération du modèle de
    traduction
112         End Sub
113
114         Public Sub envoyerCourriel(ByVal a As String, ByVal cc As String, ByVal objet As
    String, ByVal message As String)
115             Dim courriel As New Courriel(_proprietaireBoiteCourriel, a, cc, objet, message,
    Date.Now)
116
117             courriel.envoyer()
118             _elementsEnvoyes.Add(courriel)
119         End Sub
120
121         Public Sub supprimerCourriel(ByVal courriel As Courriel)
122             BoiteReception.Remove(courriel)
123             ElementsEnvoyes.Remove(courriel)
124             CourrierIndesirable.Remove(courriel)
125
126             ElementsSupprimes.Add(courriel)
127         End Sub
128     End Class
129
130     Public Class FenetreCourriel
131         Private courrielActif As Courriel
132
133         ''''''''''''''
134         ' Événements '
135         ''''''''''''''
136
137         Private Sub btnEnvoyer_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnEnvoyer.Click
138             envoyerCourriel()
139         End Sub
```

```
140
141         Private Sub btnRepondre_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnRepondre.Click
142             repondre()
143         End Sub
144
145         Private Sub btnTransferer_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnTransferer.Click
146             transferer()
147         End Sub
148
149         Private Sub btnSupprimer_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnSupprimer.Click
150             supprimer()
151         End Sub
152
153         ''''''''''''''''''''''''
154         ' Fonctions principales '
155         ''''''''''''''''''''''''
156
157         Public Shared Sub lireCourriel(ByVal courriel As Courriel)
158             Dim fenetreCourriel As New FenetreCourriel
159
160             fenetreCourriel.courrielActif = courriel
161
162             fenetreCourriel.Text = courriel.Objet
163
164             fenetreCourriel.btnEnvoyer.Visible = False
165             fenetreCourriel.btnRepondre.Visible = True
166             fenetreCourriel.btnTransferer.Visible = True
167             fenetreCourriel.btnSupprimer.Visible = True
168
169             fenetreCourriel.txtDe.ReadOnly = True
170             fenetreCourriel.txtA.ReadOnly = True
171             fenetreCourriel.txtCc.ReadOnly = True
172             fenetreCourriel.txtObjet.ReadOnly = True
173             fenetreCourriel.txtMessage.ReadOnly = True
174
175             fenetreCourriel.txtDe.Text = courriel.De
176             fenetreCourriel.txtA.Text = courriel.A
177             fenetreCourriel.txtCc.Text = courriel.Cc
178             fenetreCourriel.txtObjet.Text = courriel.Objet
179             fenetreCourriel.txtMessage.Text = courriel.Message
180
181             fenetreCourriel.Show()
```

```
182
183             ' Appel de méthode bidon pour faire fonctionner l'algorithme de génération du
    modèle de traduction
184             [Global].gestionnaireCourriels.lireCourriel(courriel)
185         End Sub
186
187         Public Shared Sub nouveauMessage()
188             Dim fenetreCourriel As New FenetreCourriel
189
190             fenetreCourriel.Text = "Nouveau message"
191
192             fenetreCourriel.btnEnvoyer.Visible = True
193             fenetreCourriel.btnRepondre.Visible = False
194             fenetreCourriel.btnTransferer.Visible = False
195             fenetreCourriel.btnSupprimer.Visible = False
196
197             fenetreCourriel.txtDe.ReadOnly = True
198
199             fenetreCourriel.txtDe.Text =
    [Global].gestionnaireCourriels.ProprietaireBoiteCourriel
200
201             fenetreCourriel.Show()
202             fenetreCourriel.txtA.Focus()
203         End Sub
204
205         Private Sub envoyerCourriel()
206             Dim a As String
207             Dim cc As String
208             Dim objet As String
209             Dim message As String
210             Dim continuer As DialogResult = DialogResult.Yes
211
212             a = txtA.Text
213             cc = txtCc.Text
214             objet = txtObjet.Text
215             message = txtMessage.Text
216
217             Try
218                 If a <> "" Then
219                     If objet.Trim = "" Then
220                         continuer = MessageBox.Show("Votre message ne contient pas d'objet.
    Désirez-vous continuer ?", "Envoi de courriel", MessageBoxButtons.YesNo,
    MessageBoxIcon.Question)
221                     End If
222
```

```
223                 If continuer = DialogResult.Yes Then
224                     [Global].gestionnaireCourriels.envoyerCourriel(a, cc, objet,
    message)
225 
226                     Me.Close()
227                     [Global].fenetrePrincipale.afficherCourriels()
228                 End If
229             Else
230                 MessageBox.Show("Vous devez entrer un destinataire.")
231             End If
232         Catch ex As Exception
233             MessageBox.Show(ex.Message)
234         End Try
235     End Sub
236 
237     Private Sub repondre()
238         Dim fenetreCourriel As New FenetreCourriel
239 
240         fenetreCourriel.Text = "Re: " & courrielActif.Objet
241 
242         fenetreCourriel.btnEnvoyer.Visible = True
243         fenetreCourriel.btnRepondre.Visible = False
244         fenetreCourriel.btnTransferer.Visible = False
245         fenetreCourriel.btnSupprimer.Visible = False
246 
247         fenetreCourriel.txtDe.ReadOnly = True
248 
249         fenetreCourriel.txtDe.Text =
    [Global].gestionnaireCourriels.ProprietaireBoiteCourriel
250         fenetreCourriel.txtA.Text = courrielActif.De
251         fenetreCourriel.txtCc.Text = courrielActif.Cc
252         fenetreCourriel.txtObjet.Text = "Re: " & courrielActif.Objet
253         fenetreCourriel.txtMessage.Text = System.Environment.NewLine & _
254             System.Environment.NewLine & _
255             "_____" & _
256             System.Environment.NewLine & _
257             courrielActif.Objet & _
258             System.Environment.NewLine & _
259             System.Environment.NewLine & _
260             courrielActif.Message
261 
262         fenetreCourriel.Show()
263         fenetreCourriel.txtMessage.Focus()
264         fenetreCourriel.txtMessage.SelectionLength = 0
265     End Sub
```

```
266
267         Private Sub transferer()
268             Dim fenetreCourriel As New FenetreCourriel
269
270             fenetreCourriel.Text = "Tr: " & courrielActif.Objet
271
272             fenetreCourriel.btnEnvoyer.Visible = True
273             fenetreCourriel.btnRepondre.Visible = False
274             fenetreCourriel.btnTransferer.Visible = False
275             fenetreCourriel.btnSupprimer.Visible = False
276
277             fenetreCourriel.txtDe.ReadOnly = True
278
279             fenetreCourriel.txtDe.Text =
    [Global].gestionnaireCourriels.ProprietaireBoiteCourriel
280             fenetreCourriel.txtObjet.Text = "Tr: " & courrielActif.Objet
281             fenetreCourriel.txtMessage.Text = System.Environment.NewLine & _
282                 System.Environment.NewLine & _
283                 "_____________________________________________" & _
284                 System.Environment.NewLine & _
285                 courrielActif.Objet & _
286                 System.Environment.NewLine & _
287                 System.Environment.NewLine & _
288                 courrielActif.Message
289
290             fenetreCourriel.Show()
291             fenetreCourriel.txtMessage.Focus()
292             fenetreCourriel.txtMessage.SelectionLength = 0
293         End Sub
294
295         Private Sub supprimer()
296             [Global].gestionnaireCourriels.supprimerCourriel(courrielActif)
297             Me.Close()
298
299             [Global].fenetrePrincipale.afficherCourriels()
300         End Sub
301     End Class
302
303     Public Class FenetrePrincipale
304         ''''''''''''''
305         ' Événements '
306         ''''''''''''''
307
308         Private Sub FenetrePrincipale_Load(ByVal sender As Object, ByVal e As
    System.EventArgs) Handles Me.Load
```

```
309             [Global].fenetrePrincipale = Me
310 
311             trvRepertoires.ExpandAll()
312 
313             lstCourriels.SmallImageList = ImageList1
314             lstCourriels.View = View.Details
315         End Sub
316 
317         Private Sub trvRepertoires_AfterSelect(ByVal sender As System.Object, ByVal e As
    System.Windows.Forms.TreeViewEventArgs) Handles trvRepertoires.AfterSelect
318             afficherCourriels()
319         End Sub
320 
321         Private Sub lstCourriels_SelectedIndexChanged(ByVal sender As System.Object, ByVal
    e As System.EventArgs) Handles lstCourriels.SelectedIndexChanged
322             afficherBoutonSupprimer()
323         End Sub
324 
325         Private Sub lstCourriels_DoubleClick(ByVal sender As Object, ByVal e As
    System.EventArgs) Handles lstCourriels.DoubleClick
326             Dim courriel As Courriel = CType(lstCourriels.SelectedItems(0).Tag, Courriel)
327 
328             FenetreCourriel.lireCourriel(courriel)
329         End Sub
330 
331         Private Sub btnNouveauMessage_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnNouveauMessage.Click
332             FenetreCourriel.nouveauMessage()
333         End Sub
334 
335         Private Sub btnSupprimer_Click(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnSupprimer.Click
336             Dim courriel As Courriel = CType(lstCourriels.SelectedItems(0).Tag, Courriel)
337 
338             [Global].gestionnaireCourriels.supprimerCourriel(courriel)
339 
340             afficherCourriels()
341         End Sub
342 
343         ''''''''''''''''''''''''
344         ' Fonctions principales '
345         ''''''''''''''''''''''''
346 
347         Public Sub afficherCourriels()
348             Dim listeCourriels As List(Of Courriel) = Nothing
```

```
349             Dim listItem As ListViewItem
350
351             Select Case trvRepertoires.SelectedNode.Index
352                 Case 0
353                     listeCourriels = [Global].gestionnaireCourriels.BoiteReception
354                 Case 1
355                     listeCourriels = [Global].gestionnaireCourriels.ElementsEnvoyes
356                 Case 2
357                     listeCourriels = [Global].gestionnaireCourriels.ElementsSupprimes
358                 Case 3
359                     listeCourriels = [Global].gestionnaireCourriels.CourrierIndesirable
360             End Select
361
362             lstCourriels.BeginUpdate()
363             lstCourriels.Clear()
364
365             lstCourriels.Columns.Add("De", 150, HorizontalAlignment.Left)
366             lstCourriels.Columns.Add("Objet", 200, HorizontalAlignment.Left)
367             lstCourriels.Columns.Add("Date", 150, HorizontalAlignment.Left)
368
369             For Each courriel As Courriel In listeCourriels
370                 listItem = New ListViewItem(courriel.De)
371                 listItem.SubItems.Add(courriel.Objet)
372                 listItem.SubItems.Add(courriel.DateEnvoi)
373                 listItem.ImageIndex = 1
374                 listItem.Tag = courriel
375                 lstCourriels.Items.Add(listItem)
376             Next
377
378             lstCourriels.EndUpdate()
379
380             afficherBoutonSupprimer()
381         End Sub
382
383         Private Sub afficherBoutonSupprimer()
384             If lstCourriels.SelectedItems.Count > 0 Then
385                 btnSupprimer.Enabled = True
386             Else
387                 btnSupprimer.Enabled = False
388             End If
389         End Sub
390     End Class
391
392     Public Class [Global]
393         Public Shared gestionnaireCourriels As New GestionnaireCourriels("Pierre-Yves
```

```
      Rozon")
394         Public Shared fenetrePrincipale As FenetrePrincipale
395     End Class
```