Julien Miltat – Julien Rosener – Thomas Petazzoni

Appel de procédures natives depuis Java

Projet LO43

Semestre d'automne 2002-2003

Université de Technologie de Belfort Montbéliard

Table des matières

1 Présentation et objectif **3**

2 JNI : Java Native Interface **3**

3 Architecture du programme **4**
3.1 L'interface graphique 4
3.1.1 Interface.java 4
3.1.2 MainPanel.java 5
3.1.3 NouveauMondeDialog.java 5
3.1.4 EnvironnementPanel.java 5
3.1.5 Simulation.java 6
3.1.6 Quelques précisions 7
3.2 Le moteur de simulation en C++ 7
3.2.1 common.h 7
3.2.2 environnement.h 8
3.2.3 environnement.cc 8
3.2.4 fourmi.h 9
3.2.5 fourmi.cc 10
3.2.6 fourmiliere.h 10
3.2.7 fourmiliere.cc 11
3.2.8 glue.cc 11
3.2.9 graphics.h 12
3.2.10 listefourmis.cc 13

4 Schéma de l'architecture **14**

5 Difficultés rencontrées **14**

6 Améliorations possibles **15**

7 Conclusion **15**

Références **15**

1 Présentation et objectif

Le projet 9, *Appel de procédures natives depuis Java*, consiste à coupler le C/C++ avec Java, par l'intermédiaire de l'interface *JNI*. L'objet de l'étude était donc de découper le projet de simulation de fourmilière en deux parties :

- Interface graphique en Java
- Simulation des fourmis et de l'environnement en C++

Les deux parties sont ensuite reliées grâce à *JNI*. Comme base de travail, nous disposons de diverses classes C++ (fourmi, fourmiliere, liste_fourmis, environnement, etc..), d'un début de simulation et d'une interface graphique *X11*.

L'objectif du projet n'était donc pas d'améliorer la qualité de la simulation en elle-même mais d'apprendre à utiliser JNI afin de réaliser une autre interface graphique, en Java.

2 JNI : Java Native Interface

La *Java Native Interface* (JNI) est une fonctionnalité intéressante de la plateforme Java. Les applications utilisant JNI peuvent incorporer du code natif écrit en C ou en C++ avec du code écrit en language Java. Ceci permet de bénéficier des avantages du Java, sans avoir à abandonner du code écrit précédemment dans d'autres langages. Par ailleurs, pour des raisons de performance, il peut être utile de conserver une partie d'un logiciel dans un langage natif, plus rapide à éxécuter.

La *JNI* est une interface *bi-directionnelle* : elle permet à Java d'accéder à du code natif, et à du code natif d'accéder à Java.

3 Architecture du programme

Notre programme se décompose en deux parties principales :
– L'interface graphique en language Java
– La simuation des fourmis et de l'environnement en C++

Ces deux parties interagissent via la *JNI*, en utilisant les deux sens d'interactions possibles : de Java vers le code natif, et du code natif vers le Java.

Ainsi, Java appelle des fonctions du code natif pour créer un environnement, ajouter une fourmilière, connaître le nombre de fourmis par fourmilière, la quantité de nourriture par fourmilière, etc...

Par ailleurs, Java met à disposition du code natif des méthodes permettant de dessiner dans l'interface graphique : dessiner de la nourriture, dessiner des pheromones, dessiner un obstacle, etc...

Au départ nous n'utilisions pas le deuxième sens d'interaction (code natif qui appelle du Java) : apres chaque tour de simulation le Java allait lire l'environnement et les tableaux de pheromones dans chaque fourmilière, et redessinait l'ensemble de l'écran. Ceci étant réalisé toutes les 100ms, le logiciel devait trop lourd dès que la taille du monde augmentait ou que le nombre de fourmis devenait trop grand.

Nous avons donc décidé d'utiliser le deuxième sens d'interaction afin de ne redessiner à chaque fois que ce qui avait change. Plus exactement, nous dessinons dans une *BufferedImage*, qui est recopiée à l'écran à chaque fois que nécessaire.

3.1 L'interface graphique

L'interface graphique en Java est présente dans le sous répertoire `interface`.

3.1.1 Interface.java

Ce fichier contient la classe *Interface* héritant de *JFrame*, qui crée la fenêtre principale du projet, avec les menus. La méthode *actionPerformed* est appelée dès qu'un menu est lancé.

La sélection du menu Fichier/Nouveau entrenera l'affichage d'une boite de dialogue (*NouveauMondeDialog.java*). Si l'on confirme la création d'un nouveau monde, alors la méthode *createWorld* est appellée. Cette méthode créé un nouvel *EnvironnementPanel*, pour représenter le monde, ainsi qu'un *MainPanel*, encapsulant le *EnvironnementPanel* et une zone de texte pour l'affichage de message. La méthode *changeText* permet d'ailleurs de changer le texte de cette zone de texte.

La création d'un nouveau monde entrainera le redimensionnement automatique de la fenêtre.

3.1.2 MainPanel.java

Ce composant graphique encapsule simplement l'objet *EnvironnementPanel* passé en paramètre dans le constructeur, avec une zone de texte, le tout à l'aide d'un *GridBagLayout*.

La méthode *changeText*, appelée depuis la classe *Interface* permet de changer le texte de la zone de texte.

3.1.3 NouveauMondeDialog.java

NouveauMondeDialog est une classe héritant de *JDialog*, permettant à l'utilisateur de créer un nouveau monde en entrant divers paramètres (taille, nombre de fourmilière, quantité de nourriture, nombre d'obstacle, taux d'évaporation des phéromones, etc...).

Lorsque l'utilisateur clique sur *Ok*, la méthode *createWorld* de la classe *Interface* est appellée.

3.1.4 EnvironnementPanel.java

La classe *EnvironnementPanel* est la plus complexe : elle hérite de *JPanel*, et permet de représenter sous forme graphique le monde dans lequel évoluent les fourmis.

Le constructeur de la classe prend de nombreux paramètres : taille du monde, quantité de nourriture, nombre d'obstacles, taux d'évaporation des phéromones et une référence vers l'objet *Interface*. A partir de là, on créé un *Panel* d'une taille bien précise, on créé un objet *BufferedImage* dans lequel on va dessiner. Ensuite, on créé une *HashTable* qui contiendra pour chaque numéro de fourmilière l'objet *Color* associé. Puis on initialise la *BufferedImage* (dessin du fond et du contour). On créé l'objet *Simulation* (voir plus bas), puis toutes les fourmilières. Enfin, on créé un *Timer* qui appelera la méthode *actionPerformed* toutes les 100ms.

La méthode *stop* arrête la simulation et détruit l'objet *Simulation*.

Les méthodes *simuStop* et *simuStart* permettent respectivement d'arrêter et de redémarrer la simulation.

La méthode *actionPerformed* est appellée toutes les 100ms, lance un tour de simulation, puis mets à jour le texte dans la zone de texte (c'est ici qu'on

utilise la référence vers l'objet *Interface*). Enfin, le *Panel* est repaint, afin de laisser apparaître les modifications.

La méthode *addFourmiliere* ajoute une nouvelle fourmilière au monde, et choisit une couleur pour cette fourmilière. Si c'est une des quatres premières fourmilières, la couleur est prédéfinie, sinon on essaie de trouver aléatoirement une couleur pas trop proche du blanc, pas trop proche du noir, et pas trop proche de la couleur d'une fourmilière existante. On utilise une *HastTable* pour stocker les divers objets *Color*.

Les méthodes *paintFourmi, clearFourmi, paintObst, paintFood, clearFood, paintPhero, clearPhero, paintFourmiliere* et *clearFourmiliere* permettent de dessiner des éléments dans la *BufferedImage*. Quand cela est nécessaire, on dispose du numéro de la fourmilière concernée, afin de pouvoir retrouver la couleur correspondante grâce à la *HashTable*.

La méthode *paint* est surchargée, afin de redessiner l'image *BufferedImage*. De même, la méthode *update* est surchargée, afin d'appeler la méthode *paint*.

3.1.5 Simulation.java

La classe *Simulation* est la classe permettant l'interaction entre le Java et le C++, et ce, dans les deux sens possibles.

Tout d'abord, un certain nombre de méthode *natives* sont déclarées. Ce sont des méthodes qui ne sont pas codées en Java, mais codées en C++, que l'on pourra appeler depuis le Java comme si c'était des méthodes de la classe *Simulation*. Ces méthodes permettent d'intéragir avec le moteur de simulation en C++.

La section `static` de la classe permet de charger la librairie correspondante, contenant le code natif associé aux méthodes natives. La méthode native *initIDs* est appellée, son rôle sera exposé plus tard.

Le constructeur de la classe utilise la méthode native `newenvironnement` pour créer un nouvel environnement au niveau du moteur de simulation en C++. On conserve une référence sur l'objet *EnvironnementPanel* dans lequel le monde est représenté graphiquement.

La méthode *destroy* détruit l'environnement C++ associé (et donc les objects fourmilieres, fourmis, etc...).

En effet, les méthodes *paintFourmi, clearFourmi, paintObst, paintFood, clearFood, paintPhero, clearPhero, paintFourmiliere* et *clearFourmiliere*, ac-

cedées depuis le code C++, sont des wrappers pour les méthodes correspondantes dans la classe *EnvironnementPanel*.

3.1.6 Quelques précisions

Le programme `javah`, lancé sur le byte code compilé de la classe *Simulation*, va générer un fichier `.h` contenant les prototypes des fonctions à implémenter en C/C++ au niveau de la bibliothèque dynamique pour que celà fonctionne.

Extrait :

```
/*
 * Class:     Simulation
 * Method:    initIDs
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Simulation_initIDs
  (JNIEnv *, jclass);
```

Voici le prototype d'une méthode native Java qui ne prend rien en argument et ne retourne rien. La méthode C/C++ correspondante prend en argument un pointeur vers l'environnement JNI et une *jclass* car la méthode en question est statique (au niveau de Java). Pour les méthodes non statiques, le deuxième argument est toujours un *jobject* :

```
/*
 * Class:     Simulation
 * Method:    getFourmilierePosY
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_Simulation_getFourmilierePosY
  (JNIEnv *, jobject, jint);
```

On a ici une méthode Java qui prend un entier en argument, et renvoie un autre entier.

3.2 Le moteur de simulation en C++

Le moteur de simulation en C++ est présent dans le sous-répertoire `simu`.

3.2.1 common.h

Le fichier d'include `common.h` contient simplement deux macros utiles pour le debugging.

3.2.2 environnement.h

Ce fichier contient simplement la déclaration de la classe *environnement*. Quelques méthodes `inline` y sont implémentées, mais la majeure partie des méthodes est implémentée au niveau de `environnement.cc`

3.2.3 environnement.cc

Le fichier `environnement.cc` implémente la classe `environnement`, déclarée dans `environnement.h`.

Plus particulièrement on y trouvera :

Le constructeur . Il se charge d'allouer la grille correspondant au monde, de l'initialiser, de générer des obstacles et de la nourriture, et ce en fonction des divers paramètres passés en arguments. On notera également que le constructeur prend en paramètre une référence sur la classe *mygraphics*.

Le destructeur . Il se charge de libérer la mémoire utilisée par la grille représentant le monde.

addFourmiliere . Cette méthode permet d'ajouter une fourmilière à l'environnement. Pour celà, on essaie d'abord de trouver une position adéquate pour la fourmilière : pas trop proche d'un ostacle et pas trop proche d'une autre fourmilière. Une fois qu'on a trouvé une zone adéquate, on la marque, on réalloue le tableau des fourmilières, et on y ajoute la nouvelle fourmilière. Enfin, on appelle *showFourmiliere* pour afficher la nouvelle fourmilière.

isDirPossible . Cette méthode prend en argument une position et une direction, et indique si cette direction est possible, c'est à dire si il n'y a pas d'obstacle génant.

simu est la méthode appellée à chaque tour de simulation. Elle se charge simplement d'appeler la méthode *simu* de chaque fourmilière du monde.

isFoodNear prend en argument une position, et retourne un entier positif correspondant à une direction si il y a de la nourriture à proximité. Si ce n'est pas le cas, cette méthode retourne -1.

getFood permet de prélever de la nourriture à la position donnée, dans une limite donnée.

addFood permet d'ajouter de la nourriture à une position donnée.

repaintCase repaint complètement une case donnée, en vérifiant si il s'agit d'un obstacle, de nourriture ou d'une fourmi. Et enfin, elle réaffiche les phéromones, toujours pour une case donnée. Cette méthode est appelée

par les méthodes d'effacements *unshowFourmi, unshowFood, etc...*, afin de redessiner ce qu'il y a "derrière".

Les méthodes de dessin : *showFourmi, unshowFourmi, showFood, unshowFood, showObst, unshowObst, showPhero, unshowPhero, showFourmilière* et *unshowFourmiliere* utilise les méthodes de la classe *mygraphics* (elle-même liée à Java) afin de reporter sur l'interface graphique les modifications du monde. Ces méthodes ne sont pas de simples wrappers, car ils se chargent de redessiner, grâce à la méthode *repaintCase*, ce qui se trouver "derrière" un élément lorsque celui-ci disparait.

3.2.4 fourmi.h

Ce fichier contient les définitions des classes suivantes :

fourmi Classe de base permettant de représenter en mémoire une fourmi. On y trouve des informations telle que l'age, la position courante, et la position précédente de la fourmi, un pointeur vers la fourmilière, etc... Quelques méthodes `inline` sont implémentées à ce niveau, mais étant données que certaines méthodes (et notamment `simu`) sont virtuelles pures, cette classe est *abstraite* et donc non instantiable.

Guerriere Classe héritant de fourmi, et représentant plus particulièrement une fourmi de type *Guerrière*. Les différentes méthodes sont implémentées dans le fichier `fourmi.cc`, et notamment la méthode `simu`, appellée à chaque tour de simulation pour faire évoluer la fourmi.

Reine Classe héritant de fourmi, et représentant plus particulièrement une *Reine*. De même que pour *Guerrière*, les diverses méthodes (et notamment `simu`) sont implémentées dans `fourmi.cc`.

Ouvriere Classe héritant de fourmi, et représentant plus particulièrement une *Ouvrière*. De même que pour *Guerrière*, les diverses méthodes (et notamment `simu`) sont implémentées dans `fourmi.cc`.

FourmiElt Cette classe encapsule simplement un pointeur vers un objet de type *fourmi*, et des pointeurs *suivant* et *précédent*. Cette classe est utilisée par la classe *ListeFourmis* pour réaliser des listes chainées de fourmis.

ListeFourmis Cette classe permet de réaliser une liste chainée de fourmis. Elle propose donc les méthodes classiques permettant de manipuler une liste chainée : *ajouter, supprimer, existe, get, getFirst, getNext, cherche*, etc.. La plupart des méthodes sont aussi implémentées dans `listesfourmis.cc`.

3.2.5 fourmi.cc

Ce fichier implémente toutes les méthodes des classes déclarées dans le fichier `fourmi.h`, à savoir *fourmi, Reine, Ouvrière* et *Guerrière*

Pour *fourmi* On dispose d'un constructeur, d'un constructeur par recopie et d'une méthode permettant de connaitre le numéro de la fourmi. L'opérateur de flux `<<` est aussi implémenté.

Pour *Reine* On dispose d'un constructeur et d'un constructeur par recopie qui appellent simplement celui de la classe mère. L'opérateur de flux `<<` est aussi implémenté, ainsi qu'une méthode `affiche` donnant diverses informations concernant la *Reine*. La méthode `simu` de *Reine* ne fait rien.

Pour *Ouvrière* Le constructeur appelle celui de la classe mère, puis initialise quelques champs propres à la classe *Ouvrière*. Le constructeur par recopie est implémenté en conséquence. L'opérateur de flux `<<` ainsi que la méthode `affiche` sont implémentées de manière basique. La méthode `simu`, elle, est relativement complexe : elle gère le mouvement de l'ouvrière. Le mouvement de l'ouvrière est aléatoire, jusqu'a ce qu'on trouve a manger. Dès que l'on a trouvé à manger, on rentre le plus directement possible à la fourmilière en déposant des phéromones. Si une ouvrière détecte des phéromones, elle est capable de les suivres, de manière à aller chercher de la nourriture qui avait été découverte précédemment. Quelques subtilités permettant d'éviter les obstacles lors du retour à la fourmilière, ou le bouclage en fin de ligne de phéromone sont aussi présents, afin d'améliorer la simulation.

Pour *Guerrière* Le constructeur appelle celui de la classe mère, puis initialise quelques champs propres à la classe *Guerrière*. Le constructeur par recopie est implémenté en conséquence. L'opérateur de flux `<<` ainsi que la méthode `affiche` sont implémentées de manière basique. La méthode `simu` modélise simplement et va-et-vient continu des fourmis de type *Guerrière*.

3.2.6 fourmiliere.h

Ce fichier contient la définition de la classe *fourmiliere*, représentant une fourmilière dans le monde. On y trouvera en particulier des informations comme la reserve de nourriture, le taux d'évaporation des phéromones, la taille du monde, la position de la fourmilière, le nombre de fourmi, un tableau bi-dimensionnel donnant en chaque point du monde la quantité de phéromone, un pointeur vers l'environnement ainsi qu'une liste de fourmis.

Quelques méthodes `inline` sont implémentées ici, et notamment *hasPhero*, *getPhero*, *evapPhero* et *addPhero*. Les méthodes *evapPhero* et *addPhero* en plus d'évaporer ou d'ajouter des phéromones s'occupent de l'aspect graphique, consistant à dessiner ou effacer des phéromones lorsque celà et nécessaire (et uniquement lorsque celà est nécessaire. Les autres méthodes `inline` sont simplement des accesseurs permettant d'accéder aux informations privées de la classe.

Les autres méthodes sont implémentées au niveau du fichier `fourmiliere.cc`.

3.2.7 fourmiliere.cc

Ce fichier implémente les diverses méthodes de la classe *fourmiliere*.

Le constructeur initialise divers champs de la classe en fonction des paramètres donnés, puis alloue un tableau bi-dimensionnel permettant de représenter en chaque point du monde la quantité de phéromone présente. Puis, il créé un certain nombre de fourmis au sein de la fourmilière.

Le destructeur se charge simplement de libérer le tableau de phéromone.

Les méthodes `ajouter` et `supprimer` sont simplement des wrappers vers les méthodes correspondantes dans la classe *ListeFourmis*.

On dispose également de deux opérateurs de flux `<<` et `>>`, ainsi que d'une méthode `affiche` donnant des informations sur la fourmilière.

La méthode `isPheroNear` permet de déterminer à partir d'une position donnée dans quelle direction doit évoluer la fourmi pour suivre les phéromones. Si il n'y a pas de phéromones à suivre, on retourne -1.

3.2.8 glue.cc

Ce fichier est la glue permettant l'interaction entre Java et le code natif, c'est à dire les appels depuis Java des méthodes natives de la classe *Simulation*.

Toutes les fonctions implémentées sont celles dont le prototype se trouve dans le fichier `Simulation.h`, généré par `javah` à partir du byte code de la classe *Simulation*. Ces fonctions sont simplement des wrappers vers des méthodes de *environnement*, permettant à Java d'agir sur le monde.

Un pointeur statique vers l'environnement courant est maintenu, ainsi qu'un pointeur vers la classe *mygraphics*. Ce premier pointeur est initialisé au moment de la construction de l'environnement (appel de la méthode native `newenvironnement`), tandis que le second pointeur est initialisé par la fonction `Java_Simulation_initIDs`, appelée statiquement par la classe `Simulation`. Ceci signifie que cette fonction est appelée dès que la classe *Simulation* existe, elle n'a pas besoin d'être instantiée.

C'est dans ce fichier que se situe le coeur de l'interaction Java vers code natif.

3.2.9 graphics.h

Ce fichier contient la déclaration et l'implémentation (courte) de la classe *mygraphics*. Cette classe est utilisée par le code C++ afin d'agir sur l'interface graphique en Java, et plus exactement sur le *EnvironnementPanel* représentant le monde.

On maintient en champ privé des champs `jmethodID` pour chaque méthode de la classe *Simulation* qui sera appelée depuis cette classe. Le constructeur de la classe (appelée par `Java_Simulation_initIDs` dans le fichier `glue.cc`) initialise ces champs. Pour celà, le constructeur utilise la méthode *GetMethodID* de la classe *JNIEnv* dont un pointeur nous est fourni en paramètre. Ensuite pour chaque méthode, on va rechercher l'identifiant en précisant la classe, le nom de la méthode, et sa signature.

Ainsi

```
paintFourmiliereMID = env->GetMethodID(cls, "paintFourmiliere",
"(III)V");
```

Permet de rechercher dans la classe *cls*, la méthode s'appelant *paintFourmiliere*, prenant trois `int` et retournant `void` (ceci est représenté par la chaine `(III)V`.

On pourrait effectuer cette recherche à chaque appel de méthode, mais étant donné que ces recherches sont assez longues, on ne les éxécute qu'une seule fois, et on utilisera ensuite les `jmethodID` pour appeler directement les méthodes.

Ce constructeur étant appelé statiquement par Java (avant l'existence de toute instance de la classe *Simulation*), on ne dispose pas d'une référence sur l'objet Java *Simulation*, simplement d'une référence sur la classe (`jclass` et non `jobject`). Or pour appeler les méthodes, on a besoin d'une référence sur l'objet.

On dispose donc de deux méthodes `objregister` et `objunregister`, appelées respectivement par `newenvironnement` et `delenvironnement` de `glue.cc` qui permettant de préciser l'objet *Simulation* sur lequel on doit agir. Seulement cette référence étant détruite après l'appel de la fonction, on doit utiliser la méthode `NewGlobalRef` de la classe *JNIEnv* pour copier cette référence dans un champ privé de notre classe.

Maintenant que l'on dispose de tous les identifiants de méthodes (`jmethodID`) et d'une référence vers l'objet *Simulation* (`jobject`), on peut implémenter

les différents wrappers.

Par exemple :

```
void paintFood(int x, int y)
  {
    env->CallVoidMethod(obj, paintFoodMID, x, y);
  }
```

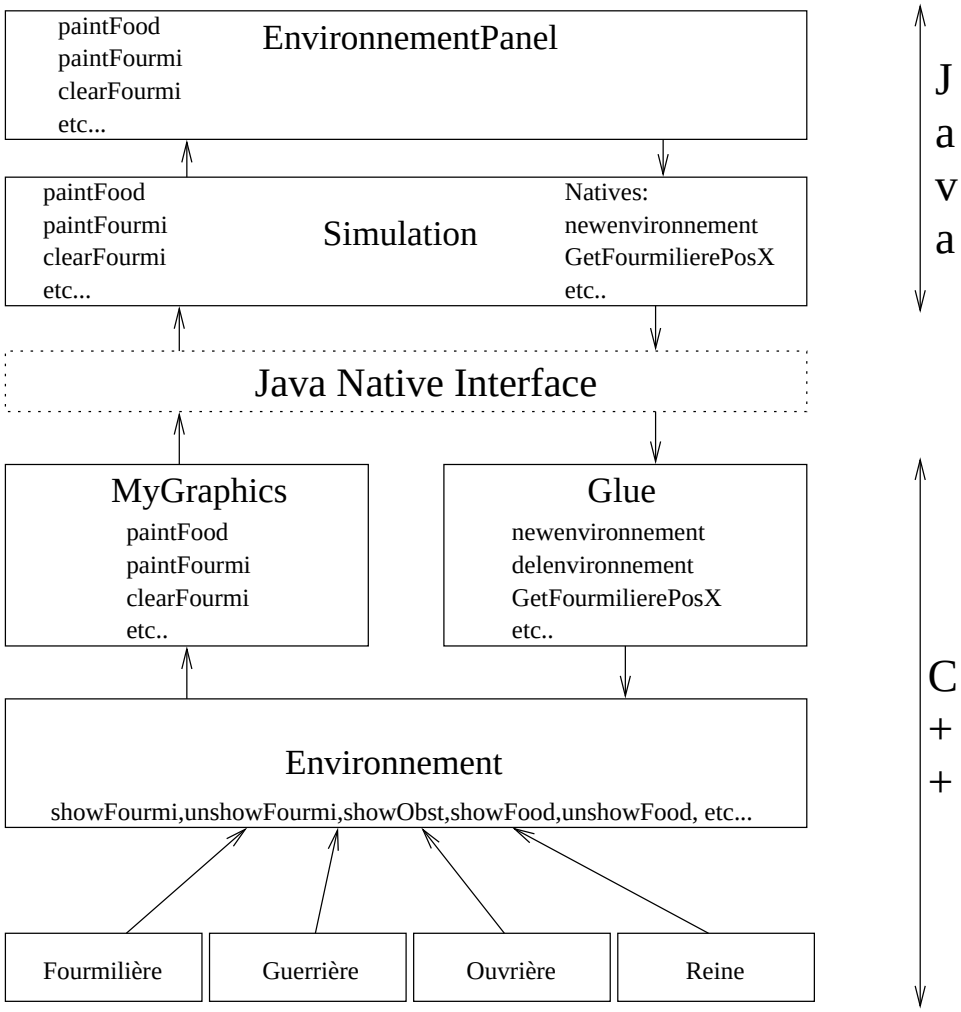
permettra d'appeler la méthode désignée par l'identifiant `paintFoodMID`, retournant `void`, avec les paramètres `x` et `y`.

Toutes les méthodes de graphisme sont implémentées ici en tant que wrappers vers les méthodes de la classe *Simulation* en Java. C'est ici le coeur de l'interaction code natif vers Java.

3.2.10 listefourmis.cc

Ce fichier implémente le constructeur de la classe *FourmiElt*, et les diverses méthodes de la classe *ListeFourmis*. Nous ne donnerons pas un descriptif détaillé de ces méthodes, car elles correspondent à une implémentation classique de liste chainée.

4 Schéma de l'architecture



5 Difficultés rencontrées

Il a fallu tout d'abord prendre connaissance avec le fonctionnement de la *JNI* qui est relativement difficile à comprendre au premier abord. Pour celà, nous avons réalisé un petit programme de test mettant en oeuvre cette *JNI* entre du code Java et C++. Une large première partie du projet a donc consisté à comprendre le fonctionnement de la *Java Native Interface*.

L'étape de structuration du programme n'était pas simple, car nous disposions de plusieurs solutions différentes, et le choix fut difficile. Finalement, nous avons choisi la solution qui permettait d'avoir l'interface graphique la plus indépendante du moteur de simulation : les méthodes Java venaient lire

directement les informations dans le moteur de simulation, ce qui permettait de redessiner à chaque fois l'ensemble de l'environnement.

Toutefois, comme nous l'avons précisé plus haut, cette solution s'est avérée inutilisable avec des environnements trop grands ou un nombre de fourmis trop importants, et nous avons donc du ajouter des méthodes *Java*, accessibles depuis le code natif qui ont permis de redessiner à chaque fois seulement les modifications par rapport à l'état précédent de l'environnement. Il a donc fallu retravailler une grande partie du code, alors que le projet était déjà bien avancé.

6 Améliorations possibles

Diverses améliorations sont possibles :

- Améliorer l'interaction entre l'interface graphique et le moteur de simulation, en permettant d'ajouter en temps réel des fourmilières, des fourmis de tous types, d'ajouter de la nourriture, de changer les paramètres d'évaporation des phéromones, etc...
- Améliorer la simulation pour prendre en compte la ponte et la mort de fourmi, ainsi que prévoir une simulation correcte pour les *Guerrière* et la *Reine*.

7 Conclusion

A partir de code réalisé durant les séances de TP, qui nous ont permis de nous familiariser avec le langage C++, ce projet nous a fait découvrir la *Java Native Interface* et nous a permis de pratiquer une nouvelle fois le langage *Java*.

Le travail simultané sur 2 langages orientés objets fut très enrichissant, et étudier la manière de faire interagir ces deux langages très intéressant.

Références

[1] The Java Native Interface, Programmer's Guide and Specification, *Sheng Liang* : `ftp://ftp.javasoft.com/docs/specs/jni.pdf`

[2] Java Native Interface : `http://java.sun.com/j2se/1.4.1/docs/guide/jni/index.html`

[3] Java Native Interface, http ://dominique.revuz.free.fr/Xpose2001/JNI/, `Dominique.Revuz@univ-mlv.fr`