Bibliothèque de gestion d'agents

1 Introduction

Cette bibliothèque de gestion d'agents permet, assez simplement, de créer des agents, de les ordonnancer (séquentiellement ou aléatoirement) et de permettre la communication entre agents (point à point ou broadcast).

Cette bibliothèque est composée de 3 classes principales :

- `Scheduler` ; la gestion de l'ordonnancement et du nommage des agents.
- `Agent` ; les agents ordonnancés doivent hériter de cette classe.
- `Message` ; les messages échangés par les agents doivent hériter de cette classe.

Ainsi, un objet de la classe `Scheduler` est chargé d'ordonnancer toutes les instances des classes qui héritent de la classe `Agent`.

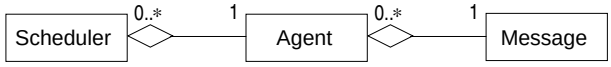

Figure 1: Diagramme de classes : `Scheduler`, `Agent` et `Message`

ATTENTION :

- Un `Scheduler` doit être instancié AVANT les agents qu'il doit ordonnancer.
- Il est toujours préférable d'instancier un agent dynamiquement – via un `new` –. En effet, cela permet, à n'importe quel moment, de supprimer un agent à l'aide d'un `delete`. Un agent peut ainsi "se suicider" (`delete this`) ou "commettre un meurtre" en supprimant un autre agent – connu par son pointeur – via un `delete`. Ainsi, la création – via un `new` – ou la suppression – via un `delete` – d'un agent peut se faire en cours de cycle.
- Un `Agent` doit posséder une méthode `void live(double dt)`. C'est cette méthode qui est appelée cycliquement par l'ordonnanceur. Si l'ordonnanceur est en mode "Temps Réel", le paramètre `dt`, exprimé en secondes, correspond au temps écoulé depuis la dernière activation. Sinon, le paramètre `dt` est égal à `0.0`.

Exemple :

```
#include <iostream>
#include "MAS.h"
#include "A.h"    // Description de la class A

using namespace std;

int main(void)
{
 Scheduler sched;
 sched.setRandomMode(true);       // Passage en ordonnancement aleatoire
 sched.setRealTimeMode(true);     // Passage en mode "temps reel"

 // Creation de 5 Agents A : "A.1", "A.2", "A.3", "A.4", "A.5"
 for(size_t i=0; i<5 ;i++) { Agent* a = new A;
                             cout << "Creation de " << a->getName() << endl;
 }

 while (1)         // Execution d'un cycle de simulation :
 {                 // Appel, 1 et 1 seule fois dans le cycle, de la
  sched.cycle();   // methode void live(double dt) de chacun des Agents
 }

 return 0;    // A la fin, lorsque l'ordonnanceur est detruit, tous les agents
}             // connus par l'ordonnanceur sont automatiquement detruits.
```

2 La classe `Agent`

Un agent d'une nouvelle classe (classe `A` dans l'exemple précédent) doit hériter directement ou indirectement de la classe `Agent`. L'interface publique de cette classe `Agent` est la suivante :

```
class Agent {

 DEFCLASS(Agent)      // Macro liee au nommage des agents

  friend ostream& operator<<(ostream& os, const Agent& anAgent);

 public:

   // Allocateurs/Desallocateurs

           Agent(void);
           Agent(const Agent& anAgent);
           Agent& operator=(const Agent& anAgent);
  virtual ~Agent(void);

  virtual  void live(double dt) {} // dt en seconde: temps depuis la derniere activation

           void suspend(void);
           void restart(void);
           bool isSuspended(void) const;

   // Comparaisons

  friend   bool operator==(const Agent& anAgent1, const Agent& anAgent2);
  friend   bool operator!=(const Agent& anAgent1, const Agent& anAgent2);

   // Inspecteurs

           string getName(void) const;  // N'a pas de sens dans le constructeur
           unsigned long getSuffix(void) const; // Idem : n'a pas de sens ...
           string getClass(void) const; // DEFCLASS: virtual getClassName ...
           bool   isA(string aClass) const;

   // Gestion des messages

           size_t   getNbMessages(void) const;
           Message* getNextMessage(void); // Le suivant
           void     clearMessageBox(void);
           void     setSensitivity(string aClass,bool yesNo); // Sensibilite (cf broadcast)
   virtual size_t   sendMessageTo(Message& aM,Agent *dest) const; // retourne 1(ok),0(ko)
   virtual void     broadcastMessage(Message& aM) const;

 protected:

   // Methodes a appeler par une classe derivee

                               // Methode qui doit etre appelee dans le constructeur
           void newAgent(void); // d'une classe derivee => Arbre d'heritage

   //  display a appeler dans une classe derivee       // display est une methode
   virtual void display(ostream& os) const;            // appelee dans operator<<

   //  isEqualTo a appeler dans une classe derivee     // isEqualTo est une methode
   virtual bool isEqualTo(const Agent& anAgent) const; // appelee dans operator==
};
```

Cette classe `Agent` propose les méthodes classiques que tous les objets C++ doivent avoir (constructeur, constructeur par recopie, opérateur d'affectation, destructeur, opérateurs de comparaison, opérateur d'injection dans un flot de sortie pour l'affichage). De plus, elle propose les méthodes suivantes:

◇ De gestion du comportement des agents

- `void live(double dt)` ; via une redéfinition de cette méthode dans une classe dérivée de la classe `Agent`, il est possible de spécifier le comportement associé aux agents de cette classe dérivée.
- `void suspend(void)` ; permet de suspendre le comportement d'un agent. Sa méthode `live` ne sera plus appelée automatiquement lors de l'exécution de la méthode `cycle` de l'ordonnanceur.
- `void restart(void)` ; permet de "relancer" le comportement d'un agent.
- `bool isSuspended(void)` ; permet de savoir si un agent est suspendu.

◇ De gestion des noms d'agents et des classes

- `string getName(void)` ; permet d'obtenir le nom d'un agent (cf nommage des agents). La méthode `getName` n'a pas de sens dans le constructeur. En effet, dans le constructeur d'une classe, le nom n'est pas définitif puisqu'il est, a priori possible, de dériver cette classe.
- `unsigned long getSuffix(void)` ; permet d'obtenir le suffixe d'un agent (cf nommage des agents). Cette méthode n'a pas de sens dans le constructeur.
- `string getClass(void)` ; permet d'obtenir le nom de la classe d'un agent (cf nommage des agents).
- `bool isA(string aClass)` ; permet, en prenant en compte l'arbre d'héritage, de savoir si un agent est un objet d'une classe particulière (voir la section 3.3).

◇ De gestion de la boîte aux lettres des agents et de l'envoi de messages (voir la classe `Message` décrite dans la section 5). Signalons que les messages ont une priorité et que la priorité 0 (priorité par défaut) est la priorité la plus faible.

- `size_t getNbMessages(void)` ; permet de connaître le nombre de messages en attente dans la boîte aux lettres de l'agent.
- `Message* getNextMessage(void)` ; permet d'obtenir un pointeur sur le message suivant (attention, la destruction du message obtenu (via un pointeur) est à la charge de l'agent... penser à faire un delete...!).
- `void clearMessageBox(void)` ; vide la boîte de réception.
- `void setSensitivity(string aClass,bool yesNo)` ; permet de rendre sensible l'agent à un certain type de message envoyé en broadcast.
- `size_t sendMessageTo(Message& aM,Agent *dest)` ; permet d'envoyer un message à un agent particulier.
  Retourne `1` si ok, `0` si l'envoi n'est pas possible (i.e. l'agent `dest` n'existe pas).
- `void broadcastMessage(Message& aM)` ; permet d'envoyer un message en broadcast (les agents sensibles au(x) type(s) du message envoyé recevront alors le message).

La classe `Agent` comporte également les méthodes `protected` :

- `virtual void display(ostream& os)` ; cette méthode appelée par l'`operator<<` permet d'afficher un agent.
- `virtual bool isEqualTo(const Agent& anAgent)` ; cette méthode appelée par la fonction amie `operator==` permet de tester l'égalité de deux agents.

Ainsi on a :

```
ostream& operator<<(ostream& os,const Agent& anAgent)
{
 anAgent.display(os);
 return os;
}
```

et

```
bool operator==(const Agent& anAgent1, const Agent& anAgent2)
{
 return anAgent1.isEqualTo(anAgent2);
}
```

L'intérêt principal de ces deux méthodes `display` et `isEqualTo` est que l'on peut facilement les redéfinir dans les classes dérivées de la classe `Agent`.

Ainsi, si l'on reprend l'exemple de la classe `A` qui hérite de la classe `Agent`, on peut écrire :

```
void A::display(ostream& os) const
{
 Agent::display(os);              // Affichage de la partie Agent de A

 ...      // Affichage des attributs propres a un objet de la classe A
}
```

et

```
bool A::isEqualTo(const A& anA) const
{
 ...    // Test de l'egalite des attributs propres a un objet de la classe A

 if (!(Agent::isEqualTo(anA))) return false; // Test de la partie Agent de A
 return true;
}
```

Maintenant, si l'on considère une classe `B`, qui hérite de la classe `A`, ces 2 méthodes peuvent être redéfinies ainsi :

```
void B::display(ostream& os) const
{
 A::display(os);             // Affichage de la partie A de l'object B

 ...      // Affichage des attributs propres a un objet de la classe B
}
```

et

```
bool B::isEqualTo(const B& aB) const
{
 ...    // Test de l'egalite des attributs propres a un objet de la classe B

 if (!(A::isEqualTo(aB))) return false;  // Test de la partie A de l'objet B
 return true;
}
```

AUTRE METHODE DE LA CLASSE `Agent` : `void newAgent(void)`

Cette méthode ne doit pas être redéfinie. Elle doit juste être impérativement appelée dans les constructeurs des classes dérivant (directement ou non) de la classe `Agent`.
Cette méthode `newAgent` de la classe `Agent` sert en effet au nommage des agents. L'absence d'appel à cette fonction dans les constructeurs peut entraîner des incohérences au niveau du nom des agents.

ATTENTION : cette méthode `newAgent` est fortement liée à la macro `DEFCLASS` dont un appel doit se trouver dans la déclaration de la classe. Ainsi, dans la déclaration de la classe `Agent` (voir page 2), nous pouvons voir l'appel suivant : `DEFCLASS(Agent)`. De la même façon, dans la déclaration d'une classe `A`, il doit y avoir l'appel suivant : `DEFCLASS(A)`... Et la même chose pour une classe `B`, etc...

A titre d'illustration d'utilisation de `newAgent`, si l'on reprend l'exemple de la classe `A` qui hérite de la classe `Agent`, les 2 constructeurs de cette classe `A` doivent appeler la méthode `newAgent` :

```
A::A(void) : Agent()
{
 newAgent();
}

//--
A::A(const A& anA) : Agent(anA)
{
 newAgent();
 ...         // Copie a effectuer pour la classe A
}
```

Maintenant, si l'on considère une classe `B`, qui hérite de la classe `A`, les 2 constructeurs de cette classe `B` doivent également appeler la méthode `newAgent` :

```
B::B(void) : A()
{
 newAgent();
}

//--
B::B(const B& aB) : A(aB)
{
 newAgent();
 ...         // Copie a effectuer pour la classe B
}
```

2.1 Exemple : une classe A héritant de la classe `Agent`

Soient le fichier `A.h` suivant :

```
#ifndef _A_H_            /////////////// A.h ///////////////
#define _A_H_

#include <iostream>
#include "MAS.h"

using namespace std;

class A : public Agent
{
  DEFCLASS(A)

  friend ostream& operator<<(ostream& os, const A& anA);

 public :

   // Allocateurs/Desallocateurs

            A(void);
            A(const A& anA);
            A& operator=(const A& anA);
   virtual ~A(void);
```

```
   virtual  void live(double dt);

   // Comparaisons

   friend  bool operator==(const A& anA1, const A& anA2);
   friend  bool operator!=(const A& anA1, const A& anA2);

   // Inspecteurs/modificateurs

 protected :

   // Methodes a appeler par une classe derivee

   // display: a appeler dans une classe derivee      // display est une
   virtual void display(ostream& os) const;           // methode appelee
                                                      // dans operator<<

   // isEqualTo: a appeler dans une classe derivee (dans operator==)
   virtual bool isEqualTo(const A& anA) const;

 private :

   // Methodes privees d'allocation/desallocation

   void _copy(const A& anA);
   void _destroy(void);
};

#endif // _A_H_
```

et le fichier `A.cpp` suivant :

```
#include "A.h"           ////////////// A.cpp ////////////

//--
A::A(void) : Agent()
{
 newAgent();
}

//--
A::A(const A& anA) : Agent(anA)
{
 newAgent();
 _copy(anA);
}

//--
A& A::operator=(const A& anA)
{
 if (this != &anA)
 {
  Agent::operator=(anA);
  _destroy();
  _copy(anA);
 }
 return *this;
}
```

```
//--
A::~A(void)
{
 _destroy();
}

//--
void A::live(double dt)
{
 // "Comportement" d'un Agent de la classe A
 cout << "A::My name is " << getName() << endl;
}

//--
bool operator==(const A& anA1, const A& anA2)
{
 return anA1.isEqualTo(anA2);
}

//--
bool operator!=(const A& anA1, const A& anA2)
{
 return !(anA1==anA2);
}

//--
ostream& operator<<(ostream& os, const A& anA)
{
 anA.display(os);
 return os;
}

//--
void A::display(ostream& os) const
{
 Agent::display(os);
 // Affichage des attributs de la classe A (Ici, pas d'attribut dans A!)
}

//--
bool A::isEqualTo(const A& anA) const
{
 // Test des attributs de la classe A (Ici, pas d'attribut dans A!)
 if (!(Agent::isEqualTo(anA))) return false;
 return true;
}

//--
void A::_copy(const A& anA)
{
 // Affectation des attributs de la classe A (Ici, pas d'attribut dans A!)
}

//--
void A::_destroy(void)
{
 // Destruction des attributs de la classe A (Ici, pas d'attribut dans A!)
}
```

2.2 Exemple : une classe B héritant de la classe A

Soient le fichier B.h suivant :

```
#ifndef _B_H_            /////////////// B.h ///////////////
#define _B_H_

#include <iostream>

#include "MAS.h"
#include "A.h"

using namespace std;

class B : public A
{
  DEFCLASS(B)

  friend ostream& operator<<(ostream& os, const B& aB);

 public :

   // Allocateurs/Desallocateurs

            B(void);
            B(const B& aB);
            B& operator=(const B& aB);
   virtual ~B(void);

   virtual  void live(double dt);

   // Comparaisons

   friend  bool operator==(const B& aB1, const B& aB2);
   friend  bool operator!=(const B& aB1, const B& aB2);

   // Inspecteurs/modificateurs

 protected :

   // Methodes a appeler par une classe derivee

   // display: a appeler dans une classe derivee      // display est une
   virtual void display(ostream& os) const;           // methode appelee
                                                      // dans operator<<

   // isEqualTo: a appeler dans une classe derivee (dans operator==)
   virtual bool isEqualTo(const B& aB) const;

 private :

   // Methodes privees d'allocation/desallocation

   void _copy(const B& aB);
   void _destroy(void);
};

#endif // _B_H_
```

et le fichier `B.cpp` suivant :

```
#include "B.h"           /////////////// B.cpp //////////////
//--
B::B(void) : A()
{
 newAgent();
}

//--
B::B(const B& aB) : A(aB)
{
 newAgent();
 _copy(aB);
}

//--
B& B::operator=(const B& aB)
{
 if (this != &aB)
 {
  A::operator=(aB);
  _destroy();
  _copy(aB);
 }
 return *this;
}

//--
B::~B(void)
{
 _destroy();
}

//--
void B::live(double dt)
{
 // Comportement" d'un Agent de la classe B
 cout << "B::My name is " << getName() << endl;
}

//--
bool operator==(const B& aB1, const B& aB2)
{
 return aB1.isEqualTo(aB2);
}

//--
bool operator!=(const B& aB1, const B& aB2)
{
 return !(aB1==aB2);
}

//--
ostream& operator<<(ostream& os, const B& aB)
{
 aB.display(os);
 return os;
}
```

```
//--
void B::display(ostream& os) const
{
 A::display(os);
 // Affichage des attributs de la classe B (Ici, pas d'attribut dans B!)
}

//--
bool B::isEqualTo(const B& aB) const
{
 // Test des attributs de la classe B (Ici, pas d'attribut dans B!)
 if (!(A::isEqualTo(aB))) return false;
 return true;
}

//--
void B::_copy(const B& aB)
{
 // Affectation des attributs de la classe B (Ici, pas d'attribut dans B!)
}

//--
void B::_destroy(void)
{
 // Destruction des attributs de la classe B (Ici, pas d'attribut dans B!)
}
```

3 Nommage des agents et gestion des instances

Nous avons vu dans la section précédente consacrée à la classe `Agent` que les agents avaient un nom.
Encore une fois, cette fonctionnalité est possible grâce :

- à l'appel à la macro `DEFCLASS` dans chaque déclaration de classe héritant (directement ou non) de la classe `Agent` ;
- à l'appel à la fonction `newAgent` dans chaque constructeur de classe héritant (directement ou non) de la classe `Agent`.

Si l'on respecte bien les appels à `DEFCLASS` et à `newAgent`, le nom d'un agent est une chaînes de caractères composée du nom de la classe de l'agent suivi d'un numéro d'instance. Ainsi, la première instance d'une classe `A` aura comme nom "A.1", la deuxième "A.2", etc...
Le suffixe d'un agent (voir la méthode `getSuffix`) correspond à son numéro d'instance.

3.1 Rappels sur la classe `Agent`

Gestion des noms d'agents et des classes dans la classe `Agent` :

- `string getName(void)` ; permet d'obtenir le nom d'un agent. Rappel: la méthode `getName` n'a pas de sens dans le constructeur. En effet, dans le constructeur d'une classe, le nom n'est pas définitif puisqu'il est, a priori possible, de dériver cette classe.
- `unsigned long getSuffix(void)` ; permet d'obtenir le suffixe d'un agent. Cette méthode n'a pas de sens dans le constructeur.

- `string getClass(void)` ; permet d'obtenir le nom de la classe d'un agent.

- `bool isA(string aClass)` ; permet, en prenant en compte l'arbre d'héritage, de savoir si un agent est un objet d'une classe particulière (dans l'exemple précédent avec les classes `A` et `B`, un appel à la méthode `isA` sur une instance de la classe `B` retourne `true` si le paramètre passé à `isA` est `"Agent"`, `"A"` ou `"B"`.

3.2 Fonctionnalités de gestion des noms et des instances

La librairie de gestion d'agents fournit également quelques services concernant la gestion des instances.
Ainsi, les fonctions sont disponibles :

- `void getAllAgents(string aClass, vector<Agent*>& anAgentVector)` ; cette fonction permet d'obtenir dans un vecteur STL la liste des agents appartenant à une classe donnée (l'arbre d'héritage étant pris en compte : dans l'exemple précédent avec les classes `A` et `B`, un appel à `getAllAgents` avec comme premier paramètre `"A"` donne toutes les instances de `A` mais aussi celles de `B`).

- `Agent* getAgent(string aName)` ; cette fonction permet, connaissant un nom d'agent, d'obtenir un pointeur sur cet agent.

- `bool exist(const Agent* anAgent)` ; cette fonction permet, connaissant un pointeur sur un agent, de savoir si cet agent existe encore...

Remarque : `getAllAgents`, `getAgent` et `exist` sont des fonctions...
...pas des méthodes de classe !

3.3 Un point sur la méthode `isA` de la classe `Agent`

La méthode `bool isA(string aClass)` de la classe `Agent` permet, en prenant en compte l'arbre d'héritage, de savoir si un agent est un objet d'une classe particulière.
Concrètement, connaissant un pointeur sur un `Agent`, l'idée est ici de savoir si l'agent "pointé" est un objet d'une classe dérivée particulière.

Exemple d'utilisation

- Soit `Agt` une classe dérivée de la classe `Agent`.

- Soient `A` et `B` deux classes dérivées de la classe `Agt`.

Le morceau de programme suivant permet :

- de récupérer des pointeurs sur tous les agents de la classe de base `Agt` (c'est–à–dire, des agents de la classe `Agt`, des agents de la classe `A` et des agents de la classe `B`) ;

- d'appliquer un traitement particulier en fonction des classes effectives des agents "pointés".

```
...
vector<Agent*> v;
...
getAllAgents("Agt",v);
...
for(size_t i=0;i<v.size();i++)
{
 Agent* agent=v[i];

 if (agent->isA("A")) {
                        A* a=(A*)agent:
                        a->doSomethingsForA();   // Traitement particulier pour A
 }
 else
 if (agent->isA("B")) {
                        B* b=(B*)agent:
                        b->doSomethingsForB();   // Traitement particulier pour B
 }
 else
 {
  Agt* agt=(Agt*)agent:
  agt->doSomethingsForAgt();                     // Traitement particulier pour Agt
 }
}
...
```

4 La classe `Scheduler`

L'instanciation d'un objet de la classe `Scheduler` est indispensable au bon fonctionnement d'une application multi–agents.

```
class Scheduler
{
 public:

  // Allocateurs/Desallocateurs

                     Scheduler(void);
  // Non implemente: Scheduler(const Scheduler& aScheduler);
  // Non implemente: Scheduler& operator=(const Scheduler& aScheduler);
            virtual ~Scheduler(void);

                void cycle(void);

                void setRandomMode(bool randomMode);
                bool getRandomMode(void) const;

                void setRealTimeMode(bool realTimeMode);
                bool getRealTimeMode(void) const;
};
```

L'utilisation classique d'un ordonnanceur a été présentée lors du premier exemple (voir page 1).

Ainsi, nous pouvons résumer les différentes grandes étapes de la création d'une application multi-agents :

```
...
int main(void)
{
 Scheduler sched;     // Instanciation d'un ordonnanceur

 sched.setRandomMode(true);     // Passage eventuel en ordonnancement aleatoire
 sched.setRealTimeMode(true);   // Passage eventuel en mode "temps reel"
 ... // Instanciation dynamique des agents presents au debut de la simulation
 while (1)
 {                  // Execution d'un cycle de simulation :
  sched.cycle();    // Appel, 1 et 1 seule fois dans le cycle, de la
                    // methode void live(double dt) de chacun des Agents
  vector<Agent*> v;
  getAllAgents("Agent",v);
  if (v.size()==0) break;    // Ou bien une condition plus simple... ou pas de condition!
 }
 return 0;   // A la fin, lorsque l'ordonnanceur est detruit, tous les agents
}            // connus par l'ordonnanceur sont automatiquement detruits.
```

Un ordonnanceur possède donc les méthodes suivantes :

- `setRandomMode(bool randomMode)` ; permet de fixer le mode d'activation (séquentiel ou aléatoire) des agents. Si `randomMode` est `true`, activation du mode aléatoire. Sinon, activation du mode séquentiel.
- `bool getRandomMode(void)` ; permet de récupérer le mode d'activation (séquentiel ou aléatoire) lié à l'ordonnanceur.
- `setRealTimeMode(bool realTimeMode)` ; permet de fixer le mode "temps réel" ou non. Si `realTimeMode` est `true`, activation du mode "temps réel". Sinon activation du mode non "temps réel".
- `bool getRealTimeMode(void)` ; permet de récupérer le mode ("temps réel" ou non) lié à l'ordonnanceur.
- `void cycle(void)` ; permet de faire "vivre" une et une seule fois un agent... c'est à dire, exécuter un cycle de simulation. Le mode est séquentiel ou aléatoire, "temps réel" ou non, en fonction des réglages faits avec `setRandomMode` et `setRealTimeMode`.

Remarque importante : s'il y a plusieurs `Scheduler` instanciés, les agents sont tous ordonnancés par le premier `Scheduler` ayant été instancié (l'"ordonnanceur courant") !

5 La classe `Message`

Nous avons vu que les agents pouvaient s'échanger des messages. Tous les messages échangés entre les agents doivent hériter (directement ou non) de la classe `Message`.
Les messages ont une priorité. La priorité 0 (priorité par défaut) est la priorité la plus faible. Plus la priorité associée à un message est élevée, plus le message est prioritaire.

Comme pour la classe `Agent`, il existe des contraintes sur l'écriture du code `C++`. Ainsi :

- Dans la déclaration d'une classe correspondant à un nouveau type de message, il doit y avoir un appel à la macro `DEFCLASS`.
- Dans les constructeurs des classes héritant (directement ou non) de la classe `Message`, il doit y avoir un appel à la méthode `newMessage` de cette même classe `Message`.

Ces deux appels sont indispensables au bon fonctionnement des échanges de messages. Un message doit donc hériter directement ou indirectement de la classe `Message`. L'interface publique de cette classe `Message` est la suivante :

```
class Message {

 DEFCLASS(Message)    // Pour le bon fonctionnement ...

  friend ostream& operator<<(ostream& os, const Message& aMessage);

 public:

   // Allocateurs/Desallocateurs

           Message(void);
           Message(const Message& aMessage);
           Message& operator=(const Message& aMessage);
  virtual ~Message(void);

           void   setPriority(size_t priority);
           size_t getPriority(void) const;

   // Comparaisons

  friend   bool operator==(const Message& aMessage1, const Message& aMessage2);
  friend   bool operator!=(const Message& aMessage1, const Message& aMessage2);

   // Inspecteurs

           string getClass(void)   const; // DEFCLASS: virtual getClassName ...
           bool   isA(string aClass) const;
           Agent* getEmitter(void) const; // Avant l'envoi d'un message,
                                          // l'emetteur est NULL ...!

 protected:

   // Methodes a appeler par une classe derivee

                               // Methode qui doit etre appelee dans le cons-
         void newMessage(void); // tructeur d'une classe derivee
                               // => Arbre d'heritage

   //  display a appeler dans une classe derivee        // display est une
   virtual void display(ostream& os) const;             // methode appelee
                                                        // dans operator<<

   //  isEqualTo a appeler dans une classe derivee      // isEqualTo est une
   virtual bool isEqualTo(const Message& aMessage) const; // methode appelee
                                                          // dans operator==

};
```

Cette classe `Message` propose les méthodes classiques que tous les objets C++ doivent avoir (constructeur, constructeur par recopie, opérateur d'affectation, destructeur, opérateurs de comparaison, opérateur d'injection dans un flot de sortie pour l'affichage).

La classe `Message` propose également les méthodes suivantes:

- `void setPriority(size_t priority)` ; permet de fixer la priorité associée à un message. La priorité 0 (priorité par défaut) est la priorité la plus faible. Plus la priorité associée à un message est élevée, plus le message est prioritaire.
- `size_t getPriority(void)` ; permet d'obtenir la priorité associée à un message.
- `string getClass(void)` ; permet d'obtenir le nom de la classe d'un message.
- `bool isA(string aClass)` ; permet, en prenant en compte l'arbre d'héritage, de savoir si un message est un objet d'une classe particulière.
- `Agent* getEmitter(void)` ; permet de connaître l'agent ayant envoyé le message. Attention : l'émetteur d'un message n'est disponible qu'une fois le message envoyé..!

La classe `Message` possède également des méthodes très similaires à celles disponibles dans la classe `Agent`. Ainsi, la classe `Message` comporte également les méthodes `protected` :

- `virtual void display(ostream& os)` ; cette méthode appelée par l'`operator<<` permet d'afficher un message.
- `virtual bool isEqualTo(const Message& aMessage)` ; cette méthode appelée par la fonction amie `operator==` permet de tester l'égalité de deux messages.

Comme nous l'avons déjà indiqué, la classe `Message` possède également une méthode `newMessage` fortement liée à la macro `DEFCLASS` dont un appel doit se trouver dans la déclaration de la classe.
Cette méthode `void newMessage(void)` ne doit pas être redéfinie. Elle doit juste être impérativement appelée dans les constructeurs des classes dérivant (directement ou non) de la classe `Message`.
Cette méthode `newMessage` sert à créer l'arbre d'héritage entre messages. L'absence d'appel à cette fonction dans les constructeurs peut entraîner des incohérences au niveau des envois de messages.

5.1 Envoi/Réception de message par un agent

Un agent a à sa disposition un certain nombre de méthodes (voir page 3) lui permettant de gérer sa boîtes aux lettres, d'envoyer un message à un agent particulier, de se rendre sensibles à certains types de messages émis en broadcast et, enfin, d'envoyer en broadcast un message aux agents sensibles au(x) type(s) du message envoyé (en prenant en compte l'arbre d'héritage des messages).

5.2 Un point sur la méthode `isA` de la classe `Message`

La méthode `isA` de la classe `Message` permet à un `Agent` de gérer simplement sa boîte à messages même si des messages de différents types y sont stockés.

Exemple d'utilisation

- Soient `MA` et `MB` deux classes dérivées de la classe `Message`.

Le morceau de programme suivant permet à un agent de type `UneClasseAgent` de lire sa boîte à messages en distingant les messages de type `MA` et ceux de type `MB`.

```
void UneClasseAgent::live(double dt)
{
 ...
 while (getNbMessages())
 {
  Message* m = getNextMessage();

  if (m->isA("MA") { MA* ma=(MA*)m;
                     // Utilisation, via le pointeur ma, du message de type MA
  }
  else
  if (m->isA("MB") { MB* mb=(MB*)m;
                     // Utilisation, via le pointeur mb, du message de type MB
  }

  delete m; // Important pour eviter les "fuites" memoire
 }
 ...
}
```

6 Générateur de code pour les classes `Agent` et `Message`

Afin de simplifier le travail du programmeur, un générateur de classes dérivant de `Agent` ou de `Message` a été réalisé.

Il se trouve dans le répertoire : `Generateurs`. et s'utilise de la façon suivante :

`$ ./genere Agent` pour créer une classe dérivant (directement ou non) de `Agent`

ou bien

`$ ./genere Message` pour créer une classe dérivant (directement ou non) de `Message`

Dans les deux cas, le nom de la classe est demandé ainsi que le nombre de classes dérivées... et les noms de ces classes. Le générateur crée alors un fichier `.h` et un fichier `.cpp`.

Le générateur fournit un squelette minimal permettant de créer une nouvelle classe... Les parties à modifier (arguments du constructeur, ...) sont repérées dans le code à l'aide de `###`.

... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... C'est bien pratique !
... Et en plus j'ai encore de la place !

7 A savoir: quelques fonctions utilitaires

La bibliothèque de gestion d'`Agent` fournit également quelques fonctions utilitaires concernant la gestion du temps et la gestion de nombres aléatoires :

voir le fichier `LibMoRis/include/UtilAgent.h`

- Gestion du temps :
  - `double getTimeMicroSeconds(void);`
    retourne le temps courant en microsecondes.
  - `double getTimeMilliSeconds(void);`
    retourne le temps courant en millisecondes.
  - `double getTimeSeconds(void);`
    retourne le temps courant en secondes.
- Gestion de nombres aléatoires :
  - `void initRandom(void);`
    initialise le générateur de nombres aléatoires.
  - `size_t randomMinMax(size_t min, size_t max);`
    permet d'obtenir un nombre entier positif compris entre `min` et `max`.
  - `double random01(void);`
    permet d'obtenir un nombre réel compris entre `0` et `1`.

8 Bugs connus

Sous Cygwin, le mode temps réel ne fonctionne probablement pas correctement à cause d'un bug connu de la fonction `gettimeofday`...
Pour essayer de remédier à ce problème j'ai ajouté une attente – `usleep(1);` – juste avant l'appel à `gettimeofday`.... voir le fichier `UtilAgent.cpp`.
Bien sûr, cet ajout se fait uniquement lorsque l'on compile `UtilAgent.cpp` sous Cygwin !

9 A faire un jour

- Utiliser des "smart pointers" pour gérer les messages... permet d'éviter de recopier un message avant qu'il ne soit mis dans la boîte aux lettres d'un agent.
- Revoir la méthode d'ordonnancement (`Scheduler::cycle(void)`) et évaluer les performances lorsque beaucoup d'agent doivent être gérés.

10 A savoir sur l'héritage multiple pour programmeurs avertis...

Afin de faire de l'héritage multiple :

- dans la classe `Agent`, il y a la méthode `void newAgent(Agent* This);`
- dans la classe `Message`, il y a la méthode `void newMessage(Message* This);`

Pour l'utilisation de l'héritage multiple, voir les exemples se trouvant ici :

`GestionAgents/Exemples/ExemplesPourProgrammeursAvertis/HeritageMultiple`

11 A savoir sur la classe `Scheduler`... pour programmeurs avertis... pour faire des SMA de SMA

En réalité, la classe `Scheduler` comporte en plus de ce qui a été annoncé :

- Un constructeur par recopie et l'opérateur d'affectation

  $\Longrightarrow$ la copie d'un ordonnanceur `o` dans un autre ordonnanceur implique la copie des agents ordonnancés par l'ordonnanceur `o`.

- Des méthodes permettant de gérer/modifier l'"ordonnanceur courant". Le terme "ordonnanceur courant" doit être compris ici comme l'ordonnanceur auquel est automatiquement rattaché un agent lorsque celui-ci est instancié.

Pour avoir accès à ces nouvelles fonctionnalités, la ligne numéro 4 du fichier

`GestionAgents/LibMoRis/include/Scheduler.h`

doit être dé-commentée !... Et la bibliothèque re-compilée.

`//#define SMAdeSMA` $\Longrightarrow$ `#define SMAdeSMA`

```
class Scheduler
{
 public:

   // Allocateurs/Desallocateurs

                     Scheduler(void);
                     Scheduler(const Scheduler& aScheduler); // Progr. avertis!
                     Scheduler& operator=(const Scheduler& aScheduler); // Idem
            virtual ~Scheduler(void);

                void cycle(void);

                void setRandomMode(bool randomMode);
                bool getRandomMode(void) const;

                void setRealTimeMode(bool realTimeMode);
                bool getRealTimeMode(void) const;

   // Gestion de plusieurs ordonnanceurs
   //-- Pour programmeurs avertis ... pour faire des SMA de SMA !
   //
   //-- Constructeur par recopie et affectation
   // Implemente !    Scheduler(const Scheduler& aScheduler);
   // Implemente !    Scheduler& operator=(const Scheduler& aScheduler);
   //
   //-- Changement et memorisation de l'ordonnanceur courant........
    static void       setCurrentSched(Scheduler& aScheduler);
    static void       setCurrentSched(Scheduler* aSchedulerPtr);
    static Scheduler* getCurrentSched(void);
   //
};
```

Pour l'utilisation de plusieurs ordonnanceurs, voir l'exemple se trouvant ici :

`GestionAgents/Exemples/ExemplesPourProgrammeursAvertis/SMAdeSMA`

12 A savoir sur la classe `Agent`... pour programmeurs avertis... pour changer la méthode automatiquement activée par l'ordonnanceur.....!

Jusqu'à présent, nous avons vu que, pour une classe `A` qui hérite de la classe `Agent`, l'ordonnanceur appelle automatiquement la méthode `void A::live(double dt);` de chacune des instances de cette classe `A`.

Ceci peut être modifié très simplement car, en réalité, la classe `Agent` comporte en plus de ce qui a été annoncé :

- `void setLiveMethod(liveMethodType newLiveMethod);`
- `liveMethodType getLiveMethod(void);`

Où le type `liveMethodType` correspond à la définition d'un nouveau type "Pointeur sur des méthodes activables par l'ordonnanceur":

```
typedef void (Agent::*liveMethodType)(double dt);
```

La méthode `void Agent::setLiveMethod(liveMethodType newLiveMethod);` permet, pour une instance donnée, de changer la méthode qui sera automatiquement appelée par l'ordonnanceur.
La méthode `liveMethodType Agent::getLiveMethod(void);` permet de récupérer un pointeur sur la méthode actuellement appelée automatiquement pour une instance donnée.

Exemple d'utilisation de la méthode `setLiveMethod`:

- Soit une classe `A` ayant une méthode `void A::behavior(double dt);`
- Soit `A *a = new A( avec d'éventuels paramètres );`
- L'appel `a->setLiveMethod((liveMethodType)&A::behavior);` aura pour conséquence que, pour l'instance `a`, l'ordonnanceur appelera automatiquement la méthode `A::behavior`

Remarques:

- Si le paramètre passé à `setLiveMethod` est `NULL`, c'est le comportement par défaut qui est remis. Ainsi, par la suite, la méthode `live` sera appelée automatiquement par l'ordonnanceur.
- Au cours de la vie d'une instance, ce n'est pas nécessairement toujours la même méthode qui sera appelée automatiquement pas l'ordonnanceur... puisque, à tout moment, avec un appel à `setLiveMethod`, il est possible de changer de méthode !

Pour l'utilisation sur un exemple complet de ces nouvelles méthodes de la classe `Agent`, voir l'exemple se trouvant ici :

`GestionAgents/Exemples/ExemplesPourProgrammeursAvertis/Exemple_setLiveMethod`

13 Fichier `MAS.h`

Ce fichier reprend les différents `.h` et fonctions utiles...

```
#ifndef _MAS_H_
#define _MAS_H_


//%%%%%%%%%%%%%%%%%%%%%%  MACRO DEFCLASS  %%%%%%%%%%%%%%%%%%%%%%%%%%

#define DEFCLASS(className) \
private: virtual string getClassName(void) const\
{ \
  return #className; \
} \
private : virtual void* virtualCopy(void) const \
{ \
 return (void*)new className(*this); \
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class Agent;
class Scheduler;
class Message;

#include "Agent.h"
#include "Message.h"
#include "Scheduler.h"

#include "UtilAgent.h"

extern void   getAllAgents(string aClass, vector<Agent*>& anAgentVector);
extern Agent* getAgent(string aName);
extern bool   exist(const Agent* anAgent);


/*

*******************************   Utilities  (UtilAgent.h) ******************

////////////////////////////// Gestion du temps ////////////////////////////////

extern double getTimeMicroSeconds(void);              // En microsecondes
extern double getTimeMilliSeconds(void);              // En millisecondes
extern double getTimeSeconds(void);                   // En secondes

////////////////////////////// Generation aleatoire ////////////////////////////

extern void   initRandom(void);                    // Initialisation generateur
extern size_t randomMinMax(size_t min, size_t max);// Generation dans [min,max]
extern double random01(void);                      // Generation dans [0.0,1.0]

////////////////////////////////////////////////////////////////////////////////
```

```
******************************  class Agent (Agent.h) **********************

typedef void (Agent::*liveMethodType)(double dt); // Pour get/setLiveMethod...

class Agent {

 DEFCLASS(Agent)

  friend ostream& operator<<(ostream& os, const Agent& anAgent);

 public:

   // Allocateurs/Desallocateurs

           Agent(void);
           Agent(const Agent& anAgent);
           Agent& operator=(const Agent& anAgent);
  virtual ~Agent(void);

  virtual  void live(double dt)               // dt en seconde : temps depuis
           {                                  // la derniere activation
            // Rien pour un Agent de base
            (void)dt;  // Pour eviter un warning
           }

           void suspend(void);
           void restart(void);
           bool isSuspended(void) const;

           void setLiveMethod(liveMethodType newLiveMethod); // Progr. avertis!
           liveMethodType getLiveMethod(void);               // Progr. avertis!

   // Comparaisons

  friend   bool operator==(const Agent& anAgent1, const Agent& anAgent2);
  friend   bool operator!=(const Agent& anAgent1, const Agent& anAgent2);

   // Inspecteurs

           string getName(void) const;  // N'a pas de sens dans le constructeur
           unsigned long getSuffix(void) const; // Idem : n'a pas de sens ...
           string getClass(void) const; // DEFCLASS: virtual getClassName ...
           bool   isA(string aClass) const;

   // Gestion des messages

           size_t   getNbMessages(string aClass="Message") const;
           Message* getNextMessage(string aClass="Message"); // Le suivant
           void     clearMessageBox(void);

           void     setSensitivity(string aClass,bool yesNo);
                                                              // retourne
   virtual size_t   sendMessageTo(Message& aM,Agent *dest) const;// 1(ok),0(ko)
   virtual void     broadcastMessage(Message& aM) const;

 protected:

   // Methodes a appeler par une classe derivee

                               // Methode qui doit etre appelee dans le constructeur
           void newAgent(void);  // d'une classe derivee => Arbre d'heritage
           void newAgent(Agent* This); // Idem mais pour l'heritage multiple

   //  display a appeler dans une classe derivee       // display est une methode
   virtual void display(ostream& os) const;            // appelee dans operator<<

   //  isEqualTo a appeler dans une classe derivee     // isEqualTo est une
   virtual bool isEqualTo(const Agent& anAgent) const; // methode appelee
};                                                     // dans operator==
```

```
*****************************  class Message (Message.h) ******************

class Message {

 DEFCLASS(Message)

  friend ostream& operator<<(ostream& os, const Message& aMessage);

 public:

   // Allocateurs/Desallocateurs

           Message(void);
           Message(const Message& aMessage);
           Message& operator=(const Message& aMessage);
  virtual ~Message(void);

           void   setPriority(size_t priority);
           size_t getPriority(void) const;

   // Comparaisons

  friend   bool operator==(const Message& aMessage1, const Message& aMessage2);
  friend   bool operator!=(const Message& aMessage1, const Message& aMessage2);

   // Inspecteurs

           string getClass(void)   const; // DEFCLASS: virtual getClassName ...
           bool   isA(string aClass) const;
           Agent* getEmitter(void) const; // Avant l'envoi d'un message,
                                          // l'emetteur est NULL ...!
 protected:

   // Methodes a appeler par une classe derivee

                               // Methode qui doit etre appelee dans le constructeur
         void newMessage(void); // d'une classe derivee => Arbre d'heritage
         void newMessage(Message* This); // Idem mais pour l'heritage multiple

   //  display a appeler dans une classe derivee        // display est une
   virtual void display(ostream& os) const;             // methode appelee
                                                        // dans operator<<

   //  isEqualTo a appeler dans une classe derivee      // isEqualTo est une
   virtual bool isEqualTo(const Message& aMessage) const; // methode appelee
                                                          // dans operator==
};
```

```
*****************************  class Scheduler (Scheduler.h) ***************

class Scheduler
{

 public:

 // Allocateurs/Desallocateurs

                     Scheduler(void);
                     Scheduler(const Scheduler& aScheduler); // Progr. avertis!
                     Scheduler& operator=(const Scheduler& aScheduler); // Idem
            virtual ~Scheduler(void);

                void cycle(void);

                void setRandomMode(bool randomMode);
                bool getRandomMode(void) const;

                void setRealTimeMode(bool realTimeMode);
                bool getRealTimeMode(void) const;

  //-- Gestion de plusieurs ordonnanceurs...
  //-- Pour programmeurs avertis ... pour faire des SMA de SMA !
  //
  //-- Constructeur par recopie et affectation
  // Implemente !    Scheduler(const Scheduler& aScheduler);
  // Implemente !    Scheduler& operator=(const Scheduler& aScheduler);
  //
  //-- Changement et memorisation de l'ordonnanceur courant........
   static void       setCurrentSched(Scheduler& aScheduler);
   static void       setCurrentSched(Scheduler* aSchedulerPtr);
   static Scheduler* getCurrentSched(void);
  //
};

*/

#endif // _MAS_H_
```

14 Installation

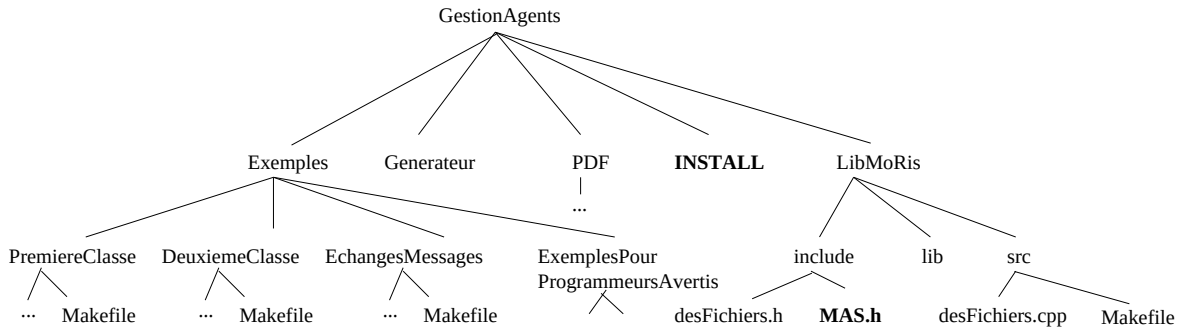

Figure 2: Description de l'arborescence de la bibliothèque

Aller dans le répertoire `LibMoRis/src` et faire `$ make`
Une bibliothèque `libAgents.a` est alors placée dans le répertoire `LibMoRis/lib`
Des exemples de programmes sont disponibles dans le répertoire `Exemples`.

... mais on peut faire plus simple !

En étant dans le répertoire `GestionAgents`, faire tout simplement `$ ./INSTALL`
Il faut ensuite aller dans les répertoires avec les divers exemples et faire `$ make`