Simulation TP5
Architecture Logicielle et Qualité
Système Multi-Agents

Arquillière Mathieu - Zangla Jérémy

26 mars 2020

Table des matières

1 Introduction **2**

2 Organisation **2**
2.1 Arborescence . . . 2
2.2 Fonctionnement . . . 2
2.3 Patrons de conception . . . 3
2.3.1 Observer . . . 3
2.3.2 Singleton . . . 4
2.3.3 Builder . . . 5

3 Implémentation **6**
3.1 Les agents . . . 6
3.1.1 Harvester . . . 6
3.1.2 Hunter . . . 7
3.2 World . . . 9
3.3 System . . . 10

A Manuel d'utilisation **12**

B Compilation **12**

C Documentation **13**

Table des figures

1 Résultat de la commande *tree* . . . 2
2 Diagramme de classe du patron *Observer* (source : Site de M.Bouhours) . . . 3
3 Interface *Observer* . . . 3
4 Classe *Observable* . . . 4
5 Diagramme de classe du patron *Singleton* (source : Site de M.Bouhours) . . . 4
6 Initialisation et méthode du singleton pour *World* . . . 5
7 Diagramme de classe du patron *Builder* (source : Site de M.Bouhours) . . . 5
8 Méthode de la classe *Base* qui génère des *Harvester* . . . 5
9 Méthode *move* et *update* de la classe *Harvester* . . . 6
10 Méthode *update* de la classe *Hunter* : interactions avec les *Harvester* . . . 7
11 Méthode *update* de la classe *Hunter* : gestion de sa vie . . . 8
12 Méthode *getEnvironnement* de la classe *World* . . . 9
13 Méthode *findRandomPositionInEnvironnement* de la classe *World* . . . 9
14 Représentation 3D d'un *Tore* . . . 10
15 Méthode pour accèder à une case du monde de la classe *World* . . . 10
16 Méthodes d'ajout et *update* de la classe *System* . . . 11

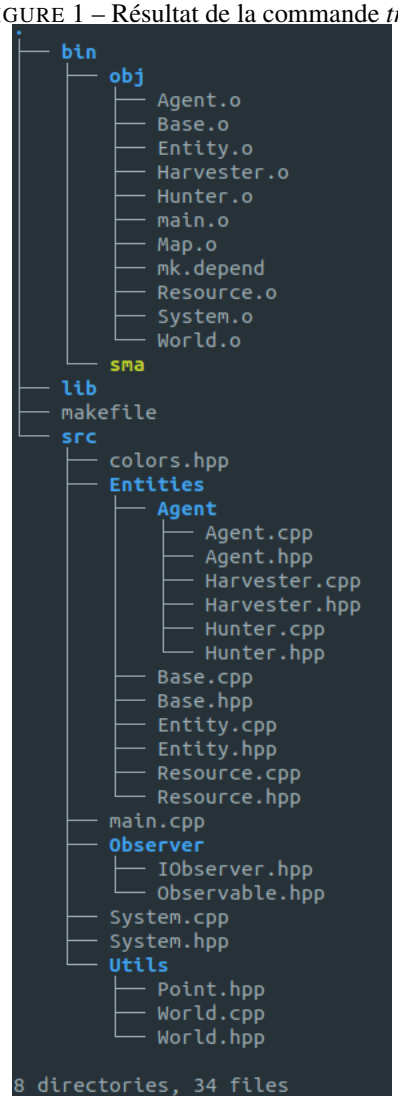
1 Introduction

Ce document fait suite aux cahier des charges concernant un système multi-agents. Il s'agit donc de réaliser un système avec les agents décrit dans ce cahier des charges. Pour cela on utilisera le langage C++, avec la librairie standard fournie avec. Pour la compilation, on utilisera g++ avec un makefile et enfin Doxygen pour la documentation associée à ce projet.

2 Organisation

2.1 Arborescence

FIGURE 1 – Résultat de la commande *tree*



2.2 Fonctionnement

Nous avons vu précédemment dans le cahier des charges le comportement de chaque agent et les relations entre les objets. Ici nous allons nous intéresser au fonctionnement globale qui est un peu différent de ce que nous avions prévu. Par exemple, la classe *System* était censée être à la fois la classe qui "s"occupe" des agents mais aussi représenter leur monde, leur environnement. Cependant nous nous sommes rendu compte après avoir essayé de cette façon que cela compliquait les choses puisque les agents doivent pouvoir obtenir des informations sur leur environnement et le modifier (via le système) et que celui-ci s'occupe de les animer. Cette relation à double sens était trop compliquer à gérer et nous avons décidé de scinder

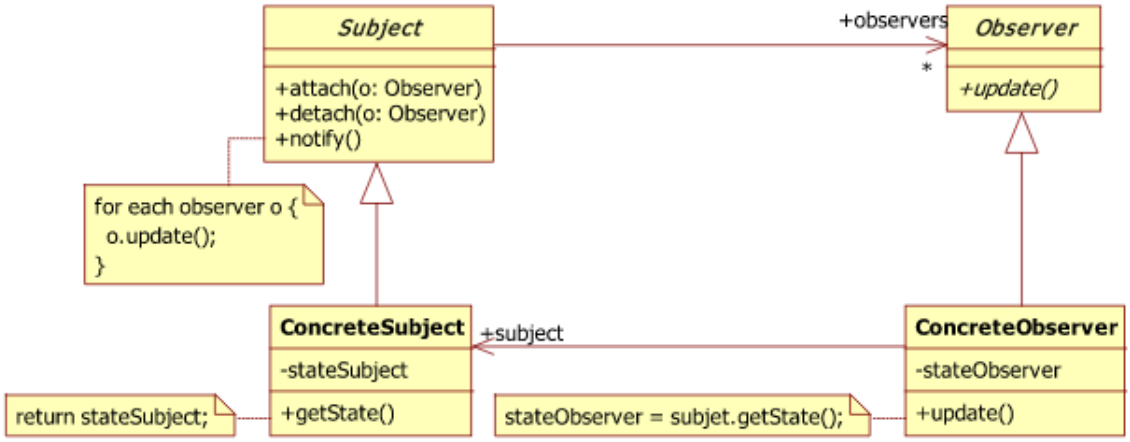
la classe *System* en deux, d'une part le vrai système, qui s'occupe des agents et d'autre part le "monde", l'environnement 2D dans lequel évoluent les agents. Cela nous a permis de bien séparer les fonctionnalités. On a désormais la classe *World* qui gère les déplacements de chaque agent et peut leur fournir des informations sur leur environnement.

2.3 Patrons de conception

2.3.1 Observer

Afin de gérer la séparation des tâches entre la classe *System* et la classe *World*, on a mis en place un patron *Observer*.

FIGURE 2 – Diagramme de classe du patron *Observer* (source : Site de M.Bouhours)



Dans notre cas, les *Observable* sont les agents et les *Observer* sont le système et le world. L'implémentation de ces 2 classes se fait de la manière suivante :

FIGURE 3 – Interface *Observer*

```
class IObserver
{
public:
    /**
     * @brief Détruit l'objet IObserver
     */
    virtual ~IObserver() {}

    /**
     * @brief Méthode virtuelle pure que doit redéfinir un observer dans le cas où une entité change de position
     *
     * @param[in] e l'entité qui bouge
     * @param[in] newPosition nouvelle position de l'entité
     */
    virtual void updateMove(Entity* e, Point const & newPosition) = 0;

    /**
     * @brief Méthode virtuelle pure que doit redéfinir un observer dans le cas où une entité meurt
     *
     * @param[in] e entité qui meurt
     */
    virtual void updateKill(Entity* e) = 0;
};
```

Ce patron permet aux classes *System* et *World* de faire une action en conséquence d'un déplacement ou d'une mort d'une entité provoquée par un agent. Par exemple le *World* bouge ou supprime visuellement une entité dans le monde. Plus tard, on a changé le comportement du système en lui faisant supprimer les agents morts de sa propre façon, il n'a donc plus eu besoin d'être un observer.

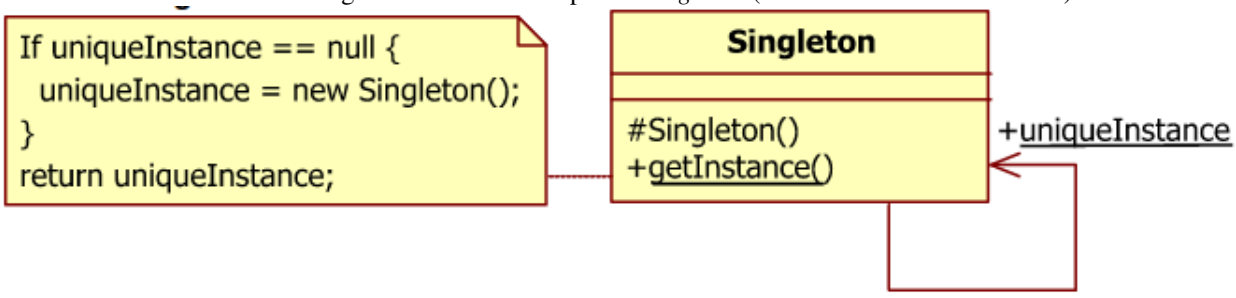
FIGURE 4 – Classe *Observable*

```cpp
class Observable
{
public:
    /**
     * @brief Méthode ajoutant un Observer qui observe l'objet actuel
     *
     * @param[in] o Objet qui possède l'interface IObserver
     */
    void addObserver(IObserver* o)
    {
        _listObserver.insert(o);
    }

    /**
     * @brief Méthode qui notifie tous les Observer d'un mouvement d'entité
     *
     * @param[in] e entité qui bouge
     * @param[in] newPosition nouvelle position de l'entité
     */
    void notifyMove(Entity * e, Point const & newPosition)
    {
        for (auto it = _listObserver.begin(); it != _listObserver.end(); it++)
        {
            (*it)->updateMove(e, newPosition);
        }
    }

    /**
     * @brief Méthode qui notifie tous les Observer de la mort d'une entité
     *
     * @param[in] e Entité qui meurt
     */
    void notifyKill(Entity * e)
    {
        for (auto it = _listObserver.begin(); it != _listObserver.end(); it++)
        {
            (*it)->updateKill(e);
        }
    }

private:
    std::set<IObserver*> _listObserver; /*!< Liste (Set) des Observer qui observe l'objet actuel
    */
};
```

2.3.2 Singleton

Dans notre cas, notre programme ne se compose que d'un sytème et un monde. De plus, on a régulièrement besoin d'accèder à ces classes et en faire des singletons simplifierait leur utilisation.

FIGURE 5 – Diagramme de classe du patron *Singleton* (source : Site de M.Bouhours)



Ce patron s'implémente assez facilement en créant un pointeur static dans la classe et une méthode permettant de créer

l'instance si elle ne l'a pas déjà été et de renvoyer celle-ci. Exemple avec la classe *World* :

FIGURE 6 – Initialisation et méthode du singleton pour *World*

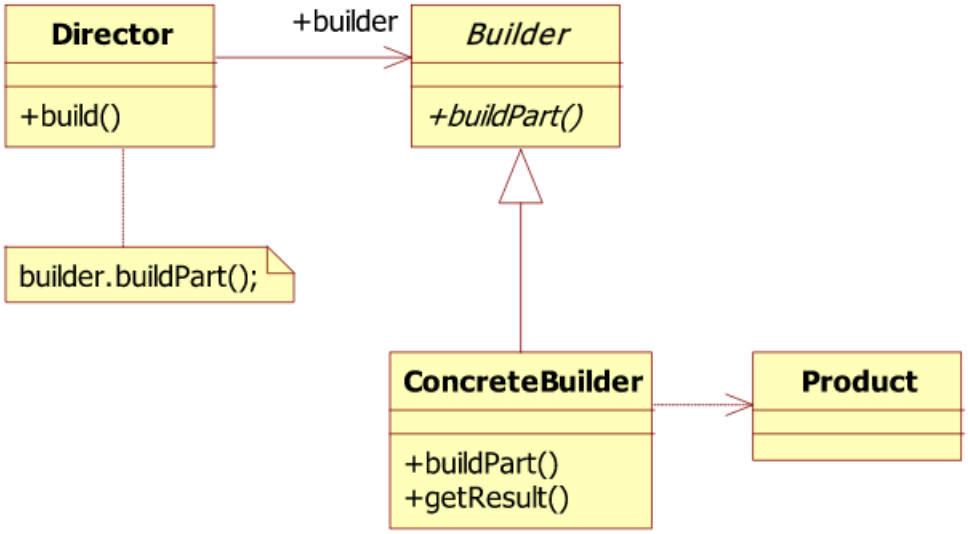
```
World* World::instance = nullptr;

World& World::getInstance()
{
    if (instance == nullptr)
        instance = new World();
    return *instance;
}
```

On a copié la même chose pour la classe *System*, cette duplication de code pourrait être supprimée en faisant un *template* c++ pour généraliser le singleton.

2.3.3 Builder

Le dernier patron de conception utilisé est un *Builder* un peu détourné. Ici notre builder est l'entité *Base* qui a la possibilité de créer des *Harvester* dans son environnement si il possède assez de ressources.

FIGURE 7 – Diagramme de classe du patron *Builder* (source : Site de M.Bouhours)



Ainsi, lorsqu'un *Harvester* apporte une ressource à sa base et que celle-ci a assez de ressource nécessaire, il lui "ordonne" de créer un autre *Harvester* dans son environnement. L'implémentation de la méthode de création d'un *Harvester* par une base a été faite de la manière suivante :

FIGURE 8 – Méthode de la classe *Base* qui génère des *Harvester*

```
void Base::birth()
{
    if (_hasToBirth)
    {
        std::vector<Entity*> env = World::getInstance().getEnvironment(_position, 1);
        Point pos;
        if (World::getInstance().findRandomPositionInEnvironment(env, 1, ENTITY_TYPE::NONE, pos))
        {
            System::getInstance().addAgent(new Harvester(_position + pos, this));
            _hasToBirth = false;
        }
    }
}
```

3 Implémentation

3.1 Les agents

3.1.1 Harvester

On rappelle qu'un agent *Harvester* (récolteur) à un rôle simple. Il parcourt aléatoirement le monde et lorsqu'il trouve une ressource il la prend et la rapporte à la *Base* à laquelle il est lié. Pour l'implémentation de cet agent, on a donc fait en sorte qu'il possède un "état", qui décide de son comportement et a deux possibilités :

— SEARCH, lorsque l'agent recherche une ressource et se déplace donc aléatoirement
— BRING, lorsqu'il possède une ressource et veut la ramener à sa base

FIGURE 9 – Méthode *move* et *update* de la classe *Harvester*

```
void Harvester::moveHarvester()
{
    std::vector<Entity*> env;
    Point nPos;
    switch (_state)
    {
    case STATE::SEARCH:
        env = World::getInstance().getEnvironment(_position, 1);
        if (World::getInstance().findRandomPositionInEnvironment(env, 1, ENTITY_TYPE::NONE, nPos))
            move(nPos);
        break;
    case STATE::BRING:
        nPos = World::getInstance().getDirection(_position, _base->getPosition());
        if (World::getInstance()[_position + nPos] == nullptr)
            move(nPos);
        break;
    }
}

void Harvester::update()
{
    std::vector<Entity*> environment;
    switch (_state)
    {
    case STATE::SEARCH:
        environment = World::getInstance().getEnvironment(_position, 1); // On récupère l'
    environnement du Harvester (voisinage de Moore d'ordre 1 autour de la position de l'agent)
        for (auto &&e : environment)
        {
            if (e != nullptr && e->getType() == ENTITY_TYPE::RESOURCE)
            {
                _state = STATE::BRING;
                notifyKill(e);
                break;
            }
        }
        break;
    case STATE::BRING:
        Point p = World::getInstance().getDistances(getPosition(), _base->getPosition());
        if (p.x <= 1 && p.y <= 1)
        {
            _state = STATE::SEARCH;
            if (_base->addResources(1))
            {
                _base->birth();
            }
        }
        break;
    }
    moveHarvester();
}
```

Ainsi la méthode commune aux différents agents (*update*) pour le *Harvester* n'est en fait que les conditions pour passer d'un état à l'autre, plus le mouvement dans l'environnement. On a différencier ce dernier dans une méthode à part. Ce mouvement dépend lui aussi de l'état du *Harvester*. Si il est dans l'état *SEARCH*, il cherche aléatoirement une case vide dans son environnement de Moore d'ordre 1 et s'y déplace. Si il est dans l'état *BRING*, il se déplace au plus court vers sa base.

3.1.2 Hunter

Le *Hunter* se déplace aléatoirement dans un voisinage de Moore d'ordre 2 si il ne "voit" pas d'agent *Harvester* dans un voisinage de Moore d'ordre 3. Si il voit un *Harvester* et qu'il est à moins de 2 cases, il se déplace sur sa position et le mange, sinon il se déplace au plus proche vers lui.

FIGURE 10 – Méthode *update* de la classe *Hunter* : interactions avec les *Harvester*

```
    // Environnement du Hunter (voisinage de Moore d'ordre 3)
    std::vector<Entity*> environment = World::getInstance().getEnvironment(_position, 3);
    bool moved = false, ate = false;

    for (auto &&e : environment)
    {
        if (e != nullptr && e->getType() == ENTITY_TYPE::HARVESTER)
        {
            Harvester* harvester = dynamic_cast<Harvester*>(e);
            if (!harvester->isDead())
            {
                Point d = World::getInstance().getDistances(getPosition(), e->getPosition());
                if (d.x <= 2 && d.y <= 2) // Si il y a un Harvester dans un voisinage de Moore d'
    ordre 2
                {
                    // On se déplace et on mange le récolteur
                    Point pos(harvester->getPosition());
                    harvester->kill();
                    notifyKill(harvester);
                    setPosition(pos);
                    moved = true;
                    ate = true;
                }
                else
                {
                    // On se déplace vers lui
                    // TODO: regarder si il y a quelque chose entre les deux sur lequel le Hunter
    ne peux pas marcher (Base, ressource)
                    move(World::getInstance().getDirection(_position, e->getPosition()) * 2);
                    moved = true;
                }
                break;
            }
        }
    }

    if (!moved)
    {
        // Si il y a pls position disponible, on bouge sur une aléatoirement, sinon on reste sur
    sa position
        environment = World::getInstance().getEnvironment(_position, 2);
        Point nPos;
        if (World::getInstance().findRandomPositionInEnvironment(environment, 2, ENTITY_TYPE::NONE
    , nPos))
            move(nPos);
    }
```

De plus, le *Hunter* à une "vie", qui diminue lorsqu'il ne mange pas et augmente lorsqu'il mange. Plus exactement cette vie est composée de 3 "stades" :

— vert, si il ne mange pas pendant 5 étapes, il passe au orange. Si il mange il créer un autre *Hunter*
— orange, si il ne mange pas pendant 5 étapes, il passe au rouge. Si il mange il passe au vert
— rouge, si il ne mange pas pendant 5 étapes, il meurt. Si il mange il passe au orange

FIGURE 11 – Méthode *update* de la classe *Hunter* : gestion de sa vie

```
    if (ate)
    {
        Point nPos;
        switch (_lifeState)
        {
        case LIFE_STATE::GREEN:
            environment = World::getInstance().getEnvironment(_position, 1);
            if (World::getInstance().findRandomPositionInEnvironment(environment, 1, ENTITY_TYPE::
    NONE, nPos))
            {
                System::getInstance().addAgent(new Hunter(nPos));
            }
            break;
        case LIFE_STATE::ORANGE:
            _lifeState = LIFE_STATE::GREEN;
            _stepsBeforeChange = GREEN_STATE_DURATION;
            break;
        case LIFE_STATE::RED:
            _lifeState = LIFE_STATE::ORANGE;
            _stepsBeforeChange = ORANGE_STATE_DURATION;
            break;
        }
    }
    else
    {
        if (_stepsBeforeChange > 0)
        {
            _stepsBeforeChange--;
        }
        else
        {
            switch (_lifeState)
            {
            case LIFE_STATE::GREEN:
                _lifeState = LIFE_STATE::ORANGE;
                _stepsBeforeChange = ORANGE_STATE_DURATION;
                break;
            case LIFE_STATE::ORANGE:
                _lifeState = LIFE_STATE::RED;
                _stepsBeforeChange = RED_STATE_DURATION;
                break;
            case LIFE_STATE::RED:
                kill();
                notifyKill(this);
                break;
            }
        }
    }
```

Pour cette gestion de vie, il aurait été intéressant de créer une interface permettant de gérer les durées de chaque stade (vert, orange et rouge) pour observer les changements de comportement.

3.2 World

C'est l'objet *World* qui permet à ces deux types d'agents d'interagir avec leur environnement. On remarque dans les implémentations précedentes plusieurs appels à des méthodes de la classe *World* :

— getEnvironnement, qui permet d'obtenir l'ensemble des cases (sous forme d'un tableau) autour d'un point donné dans un voisinage de Moore dont on précise l'ordre

FIGURE 12 – Méthode *getEnvironnement* de la classe *World*

```
std::vector<Entity*> World::getEnvironment(Point & origin, int range)
{
    std::vector<Entity*> env;
    refactorCoordonates(origin);

    for (int i = origin.x - range; i <= origin.x + range; i++)
    {
        for (int j = origin.y - range; j <= origin.y + range; j++)
        {
            env.push_back((*this)[Point(i, j)]);
        }
    }
    return env;
}
```

— findRandomPositionInEnvironnement, qui prend un "environnement" en paramètre et cherche un type d'entité ou les cases vides. Si il y en a plusieurs, elle choisit aléatoirement entre. Cette méthode renvoie la position de l'entité ou de la case vide trouvée.

FIGURE 13 – Méthode *findRandomPositionInEnvironnement* de la classe *World*

```
bool World::findRandomPositionInEnvironment(std::vector<Entity*> env, int range, Entity::
    ENTITY_TYPE toFind, Point & pos)
{
    std::vector<Point> points;
    for (int i = -range; i <= range; i++)
    {
        for (int j = -range; j <= range; j++)
        {
            points.push_back(Point(i, j));
        }
    }

    std::vector<int> index;
    for (size_t i = 0; i < env.size(); i++)
    {
        if ((toFind == Entity::ENTITY_TYPE::NONE && env[i] == nullptr) || (env[i] != nullptr &&
    env[i]->getType() == toFind))
        {
            index.push_back(i);
        }
    }

    if (index.size() > 0)
    {
        std::shuffle(index.begin(), index.end(), gen);
        pos = points[index[0]];
        return true;
    }
    return false;
}
```

Pour utiliser l'aléatoire, l'objet *World* a une instance d'un générateur Mersenne Twister et utilise *std : :shuffle* fournis par la STL du c++.

Le monde dans lequel évoluent les agents est un "tore". C'est-à-dire que le haut est relié au bas et la gauche est reliée à la droite. Afin de faire cela, on a doté la classe *World* d'accesseurs particuliers. En effet, pour accèder à un point du monde,

on passe en paramètre un *Point* qui, si il dépasse les coordonnées du monde, est ramené aux coordonnées correspondantes en supposant que le monde est un tore. Par exemple, si le monde fait 20 de haut et 20 de large, les coordonnées (4, 3) représentent la même case que les coordonnées (24, 3) ou (4, 43) ou (44, 63).

FIGURE 14 – Représentation 3D d'un *Tore*

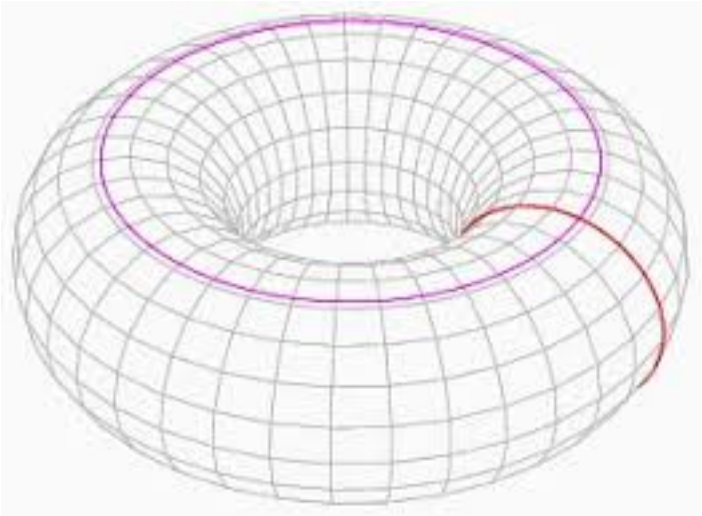


FIGURE 15 – Méthode pour accèder à une case du monde de la classe *World*

```
Entity*& World::operator[](Point const & p)
{
    int x = p.x < 0 ? _width - (-p.x % _width) : p.x % _width;
    int y = p.y < 0 ? _height - (-p.y % _height) : p.y % _height;

    return _world[x][y];
}
```

3.3 System

Depuis que l'on a séparé les fonctionnalités en deux classes, *System* et *World*, la classe *System* est très simple. Elle ne fait qu'ajouter et supprimer des agents d'un tableau et de mettre à jour tous les agents vivants dans ce tableau. Ainsi, avant les "updates" des agents, on mélange l'ordre pour que le sens d'appel ne soit jamais le même. On utilise pour cela le générateur aléatoire Mersenne Twister de *World*.

Pour la suppression des agents qui "meurent" à cette étape, on ne peut pas le faire dans la boucle qui appelle les "updates" puisqu'on modifierait la taille du tableau qu'on parcourt. La solution est que les agents ne se "suppriment" pas mais indiquent qu'ils sont morts grâce à la méthode *isDead*. Une fois la boucle d'updates finie, on utilise le Erase-Remove Idiom pour retirer tous les agents morts du tableau.

Pour l'ajout d'agents (via les *Hunter* lorsqu'ils mangent ou via les *Base* lorsqu'elles on assez de ressources), le principe est le même : on ne peut pas modifier les tableau lorsqu'on le parcourt. On a donc un *buffer* qui contient tous les agents qu'on désire ajouter et lorsqu'on a fini les updates, on ajoute tous le contenu du buffer dans le tableau principal.

FIGURE 16 – Méthodes d'ajout et *update* de la classe *System*

```cpp
void System::addAgent(Agent* a)
{
    _addingBuffer.push_back(a);
    World::getInstance().add(a);
}

void System::update()
{
    // Updates
    std::shuffle(_agents.begin(), _agents.end(), World::gen);
    for (auto it = _agents.begin(); it != _agents.end(); it++)
    {
        auto ptr = *it;
        if (!ptr->isDead())
            ptr->update();
    }

    // Suppression des agents morts pendant cette update
    _agents.erase(std::remove_if(_agents.begin(), _agents.end(), [](Agent * a)
    {
        bool suppr = false;
        if (a->isDead())
        {
            suppr = true;
            delete a;
        }
        return suppr;
    }), _agents.end());

    // Ajout des agents nés dans cette update
    if (_addingBuffer.size() > 0)
    {
        for (auto it = _addingBuffer.begin(); it != _addingBuffer.end(); it++)
            _agents.push_back(*it);
        _addingBuffer.clear();
    }
}
```

A Manuel d'utilisation

Un fois compilé (*make -j*), on execute le programme avec la commande :

```
$ bin/sma --<option>
```

Où les options possibles pour executer le programme sont :

- draw permet d'avoir un visuel plus fluide
- log permet d'avoir chaque étape écrite dans le terminal (equivalent à sans options)
- log-file permet d'avoir la sortie du programme dans un fichier "output.log"
- log-draw permet de combiner –draw et –log-file

B Compilation

Un makefile a été utilisé pour simplifier la compilation de ce TP. Voici son contenu

```
#variables
SRCDIR = src/
OBJDIR = bin/obj/
EXEC    = bin/sma
DEPFILE = $(OBJDIR)mk.depend
INCDIR = -I"."

#les programmes et leurs options
MSQ = @
MSG = $(MSQ)echo
MSG_OK = $(MSG) " Ok"
MSG_CCOK = $(MSG) " a été correctement compilé"
MSG_DEPOK = $(MSG) " *** dépendances calculées"
MSG_BUILDOK = $(MSG) "Le Projet a été construit!"
RM  = $(MSQ)rm -rf
CC  = $(MSQ)g++
SED = sed -e "s/\(.*\.o:\)/$(subst /,\/,$(OBJDIR))\1/g"
CFLAGS = -O3 -Wall -std=c++17 -g -pg
LFLAGS = -O3 -Wall -std=c++17 -g -pg

#variables automatiques
SRC := $(shell find $(SRCDIR) -name '*.cpp')
SRC_H := $(shell find $(SRCDIR) -name '*.hpp')
OBJ := $(patsubst %.cpp,$(OBJDIR)%.o,$(notdir $(SRC)))

all:

ifeq ($(wildcard $(DEPFILE)), )
all: $(DEPFILE)
-include $(DEPFILE)
else
include $(DEPFILE)
all: $(EXEC)
endif

$(EXEC): $(OBJ)
  $(MSG) "==== édition de liens ===== "
  $(CC) $(LFLAGS) $(OBJ) -o $(EXEC)
  $(MSG_BUILDOK)
  #$(EXEC)

$(DEPFILE): $(SRC) $(SRC_H)
  $(MSG) "Calcul des dependances..."
  @mkdir -p $(OBJDIR)
  $(RM) $(DEPFILE)
  $(CC) -MM $(SRC) $(CFLAGS) $(INCDIR) | $(SED) > $(DEPFILE)
  $(MSG_DEPOK)

$(OBJ):
  $(MSG) "---- compilation $*.o ------"
  $(CC) -c $< -o $@ $(CFLAGS) $(INCDIR)
  $(MSG) -n " +++ $*.cpp "
  $(MSG_CCOK)
```

```
54
55 clean:
56   $(RM) $(OBJ) $(DEPFILE)
```

On a seulement besoin de la STL du c++. Notamment de la bibliothèque *<random>* pour générer tous les nombre pseudo-aléatoires du programme, de la bibliothèque *<vector>* et de *<algorithm>* afin de gérer les agents et les entités dans le système et le monde.

C Documentation

Lien vers la documentation Doxygen