

Conception et mise en œuvre d'une interface de
visualisation des résultats de simulation d'une
plate-forme de simulation individus-centrée



Audrey Realini

Stage DUT informatique réalisé sous la direction de Jean Le Fur

Année universitaire 2010-2011

Glossaire

- API (Application Programming Interface) : Interface de programmation.
- CBGP : Centre de Biologie pour la Gestion des Populations.
- Classpath : Paramètre de la machine virtuelle java qui définit le chemin d'accès au répertoire où se trouvent les classes et les packages Java dont a besoin un projet pour fonctionner.
- Cluster : système informatique composé d'unités de calcul (micro-processeurs, cœurs, unités centrales) autonomes qui sont reliées entre elles à l'aide d'un réseau de communication.
- Display : Signifie « écran ». Cela correspond à un onglet dans Repast.
- Dos : Langage de commande Windows.
- Hétérozygotie attendue : estimation de la fréquence des hétérozygotes si les allèles sont associés au hasard pour former les génotypes (ensemble des gènes d'un individu) et ceci pour chaque locus.
- IDisplay : Interface de Repast permettant de créer des onglets personnalisés.
- JAR : Archive Java contenant les fichiers binaires d'un projet et les bibliothèques dont il a besoin pour son exécution.
- Java : Langage de programmation
- Locus : Endroit où est placé un gène sur un chromosome.
- OpenGL : Bibliothèque graphique utilisée dans la conception d'applications 2D ou 3D.
- Output : Signifie « sortie de données »
- Repast Symphony (RS) : Plateforme de programmation multi-agents.
- Shell : Langage de programmation sous Linux. Utilisé pour les scripts.
- SVN : Logiciel permettant d'enregistrer son projet sur un serveur et de revenir facilement à une version antérieure.
- UserPanel : Classe de Repast 2.0 ayant les mêmes fonctionnalités qu'un JPanel. Il a une place réservée dans l'interface graphique de Repast mais reste vide tant qu'il n'est pas spécifiquement instancié par le programmeur.

Table des figures

| | |
|--|----|
| Figure 1 : Diagramme de classes du système épiphyte | 14 |
| Figure 2 : Outline de C_Inspecteur | 15 |
| Figure 3 : Outline C_RenvoiDonnees | 19 |
| Figure 4 : Outline de C_InspecteurGenetique | 21 |
| Figure 5 : Diagramme des classes qui constituent le tableau de bord..... | 22 |
| Figure 6 : Outline de C_Calendar | 22 |
| Figure 7 : Outline de C_Meter | 24 |
| Figure 8 : Outline C_TableauDeBord | 28 |
| Figure 9 : Diagramme de classes pour les onglets personnalisés et leur graphique | 30 |
| Figure 10 : Outline C_Chart..... | 31 |
| Figure 11 : Outline de C_CustomDisplay..... | 35 |
| Figure 12 : Outline de SimMastoInitializer | 36 |
| Figure 13 : Diagramme de classes de la gestion du style des agents | 38 |
| Figure 14 : Outlines de C_StyleAgent et C_SelecteurImage | 39 |

Table des matières

| | |
|--|----|
| Glossaire | 2 |
| Table des figures | 3 |
| Table des matières | 4 |
| Introduction..... | 6 |
| A. Présentation de Repast Symphony | 7 |
| B. Présentation de SimMasto | 8 |
| C. Mise en œuvre..... | 10 |
| 1. Liste des tâches..... | 10 |
| 2. Environnement de travail..... | 12 |
| • Repast..... | 12 |
| • JfreeChart | 12 |
| • Piccolo..... | 12 |
| • OpenGL..... | 12 |
| • SVN..... | 13 |
| • Cluster..... | 13 |
| D. Structure du module présentation..... | 13 |
| I. Le système épiphyte..... | 13 |
| 1. L'inspecteur de données..... | 14 |
| 2. L'inspecteur génétique..... | 16 |
| 1. Les calculs | 17 |
| 2. Les écritures de fichiers | 18 |
| II. Le tableau de bord..... | 21 |
| 1. Le calendrier | 22 |
| 2. Les meters | 24 |
| 3. La console | 26 |
| 4. La gestion du tableau de bord | 27 |
| III. Onglets personnalisés et graphes | 30 |
| 1. Créer un graphique avec JfreeChart..... | 30 |
| 1. Les « Pie Chart » | 32 |
| 2. Les « Line charts » | 33 |
| 2. Créer un nouvel onglet | 34 |
| 3. Ajouter un onglet à la simulation et le mettre à jour | 36 |

| | | |
|-----|--|----|
| IV. | Le Style des agents | 38 |
| 1. | Représentation par une image..... | 39 |
| 2. | Représentation par une ellipse | 40 |
| 3. | Utiliser le nouveau style dans la simulation | 41 |
| V. | Le batch..... | 42 |
| 1. | Lancer un batch sous Eclipse | 42 |
| 2. | Lancer un batch hors Eclipse..... | 42 |
| 3. | Utiliser le multi-run | 43 |
| | Conclusion..... | 45 |
| | Bibliographie..... | 46 |
| | Annexe | 47 |
| | ScriptSimmasto.sh | 47 |
| | SimmastoMulti.bat..... | 48 |
| | Résumé | 49 |

Introduction

Au cours de mon stage de fin d'études, réalisé au Centre de Biologie pour la Gestion des Populations de Montpellier, j'ai eu à réaliser le module « présentation » de la plateforme de simulation du projet SimMasto. Ce module devait contenir une interface de visualisation d'indicateurs sélectionnés dans la simulation, dont les valeurs pouvaient aussi être envoyées dans fichiers. Les indicateurs sont récupérés par un système épiphyte que j'avais à mettre au point. Il fallait aussi pouvoir modifier la représentation graphique des agents pour, par exemple, différencier les mâles et les femelles. Enfin, le projet devait pouvoir être démarré en batch pour l'utiliser sur un cluster et lancé en multi-run pour réaliser des analyses de sensibilité des paramètres du modèle biologique. Enfin, le code devait être bien documenté pour assurer la pérennité du projet puisque d'autres personnes seront amenées à continuer le projet.

Ce document aborde en premier lieu la présentation de Repast Symphony, suivie de celle de SimMasto. Ensuite, nous présenterons la mise en œuvre du travail effectué avant de terminer par l'explication de sa structure.

A. Présentation de Repast Symphony

Repast Symphony (ou RS dans ce document | <http://repast.sourceforge.net/>) est une plateforme de modélisation et de simulation qui utilise Java. Elle a été conçue pour être simple d'utilisation afin de permettre à des non-informaticiens de pouvoir programmer leurs modèles. Elle contient aussi beaucoup de primitives permettant de créer des systèmes multi-agents sans difficulté.

Repast possède une interface graphique déjà programmée et lancée à chaque simulation. Les simulations fonctionnent avec des ticks (pas de temps) et des `scheduledMethods` (actions à réaliser). Une simulation est gérée avec les 5 boutons présentés au centre de l'image qui suit :

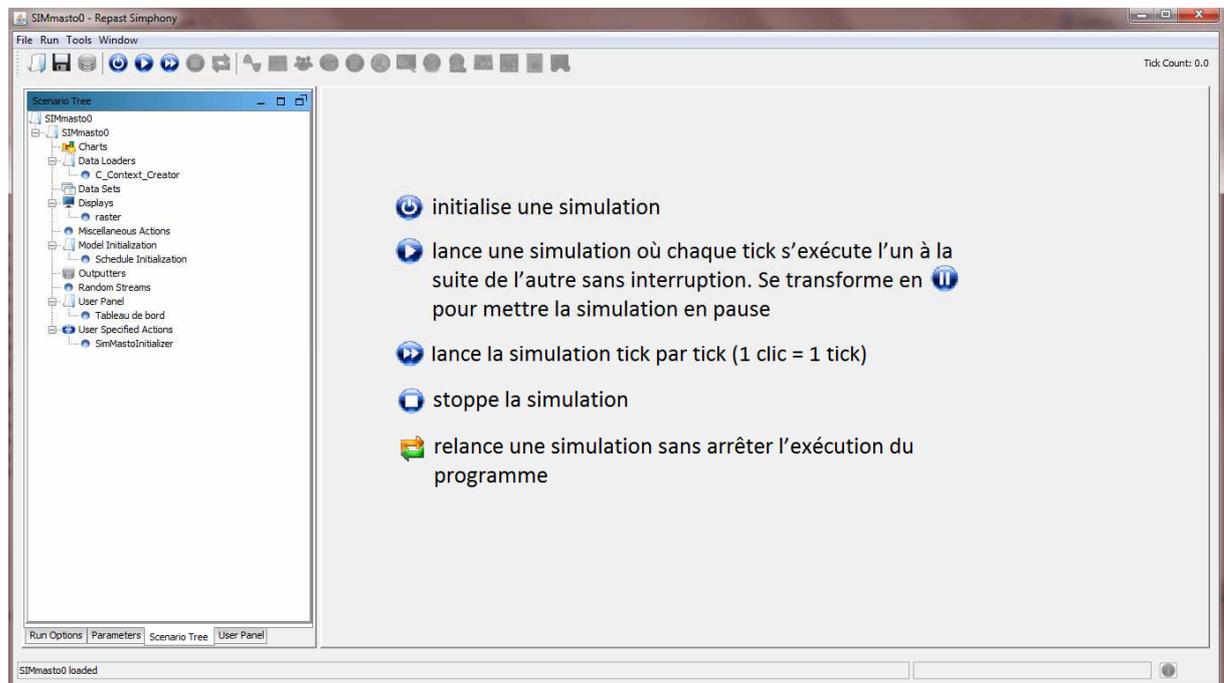


Image 1 : Interface graphique de RS

Les `scheduledMethods` définissent des actions à effectuer à un intervalle de temps régulier. Par convention, on appelle ces méthodes `step()` :

```
@ScheduledMethod(start=0,interval=1,priority=0)
public void step() {...}
```

Elles sont définies par cinq paramètres (mais il n'est pas nécessaire de tous les utiliser):

- `start` correspond au tick auquel on souhaite commencer l'action ;

- *interval* indique l'intervalle de ticks entre deux appels de cette méthode. Par défaut, la méthode est appelée une seule fois (*interval=0*), au moment défini par *start* ;
- *priority*¹ gère l'ordre dans lequel seront appelées toutes les *scheduledMethods* du programme. Par défaut, la priorité est choisie au hasard ;
- *duration* permet de définir le nombre d'actions à exécuter en même temps (sur un tick). Par défaut, une action est lancée quand une autre est finie ;
- *shuffle* active ou non un random sur tous les agents qui utilisent une même *scheduledMethod* (pour que l'ordre d'appel des agents ne soit pas toujours le même). Par défaut, ce booléen est à vrai.

Un projet Repast contient un dossier *nomProjet.rs* qui contient des fichiers XML. Ces fichiers servent à définir des actions à effectuer dans le modèle par Repast. Il y a trois fichiers importants dans la version 2.0 de RS :

- Le « *scenario.xml* » initialise les displays de la simulation (créés par Repast) et peut appeler un *modelInitializer* personnalisé (créé par le programmeur) qui, comme son nom l'indique, permet d'initialiser une simulation ;
- Le « *user_path.xml* » contient toutes les classes qui doivent être ajoutées à la simulation (comme les agents, les classes possédant une *scheduledMethod*, ou implémentant quelque chose en rapport avec le GUI) ;
- Le « *context.xml* » contient tous les paramètres et les projections de la simulation. Les projections sont un système de Repast servant de support et de repère pour situer les agents et leur permettre de se déplacer.

Enfin, le *main* de ce type de projet est géré par Repast. La première classe, créée par le programmeur, à être appelée par RS doit implémenter *ContextBuilder* (interface de RS). Cette dernière implémente une fonction *build(Context context)* qui permet d'initialiser la simulation et le « *Context* ». Ce dernier représente le contenu de la simulation, c'est-à-dire tous les agents qui en font partie. Dans SimMasto, la classe implémentant *ContextBuilder* est *C_Context_Creator*.

B. Présentation de SimMasto

SimMasto² est un projet expérimental qui aspire à être un centre de connaissances dynamiques sur la coévolution bioécologique de petits rongeurs et de leurs parasites. L'objectif est de contribuer à une meilleure compréhension du statut, du rôle et du destin de ces nuisibles en coévolution.

¹ La priorité la plus haute est l'infini (une priorité de 1 sera donc supérieure à une priorité de 0).

² Lien vers le site du projet : www.mpl.ird.fr/ci/masto/index.htm

Ce projet est accueilli par le CBGP, centre de recherche multi-terrains pour la biologie et la gestion des populations. Le projet informatique vise à la production d'une plateforme générique de simulation sur les rongeurs.

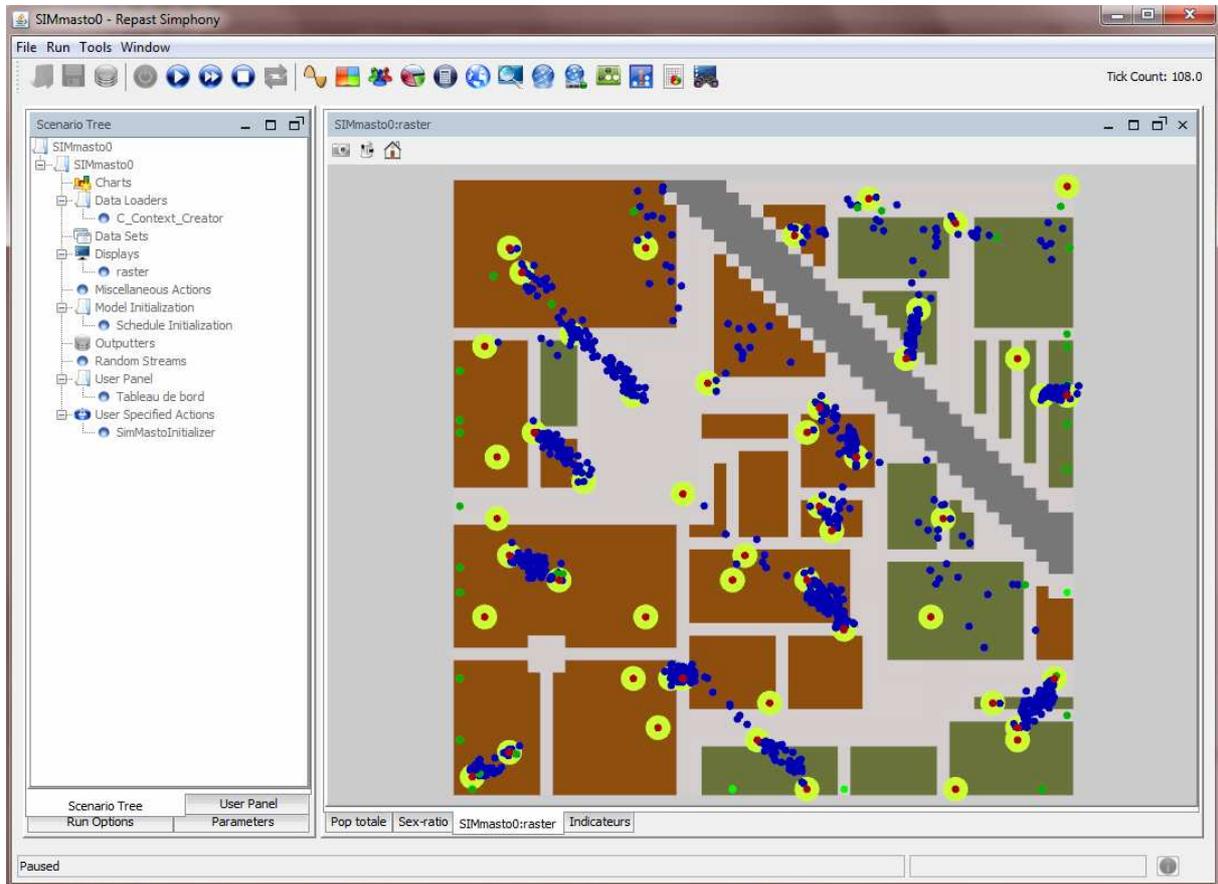


Image 2 : Interface graphique de SimMasto

SimMasto est un projet d'équipe où chaque personne travaille sur un domaine différent du modèle. Le projet global est structuré sur le design Data-Business-Présentation : Il y a deux ans, Q. Baduel a travaillé sur la gestion des espaces du modèle (module Data). Durant mon stage, j'ai travaillé avec J-E Longueville qui s'occupe de coder le comportement des agents rongeurs comme les déplacements, la reproduction, la génétique et les croisements (module Business) ; et je me suis occupée de réaliser différents « outputs » comme des tableaux de données ou encore des graphiques, mais aussi de pouvoir lancer la simulation en batch puis sur le cluster du CBGP (module Présentation).

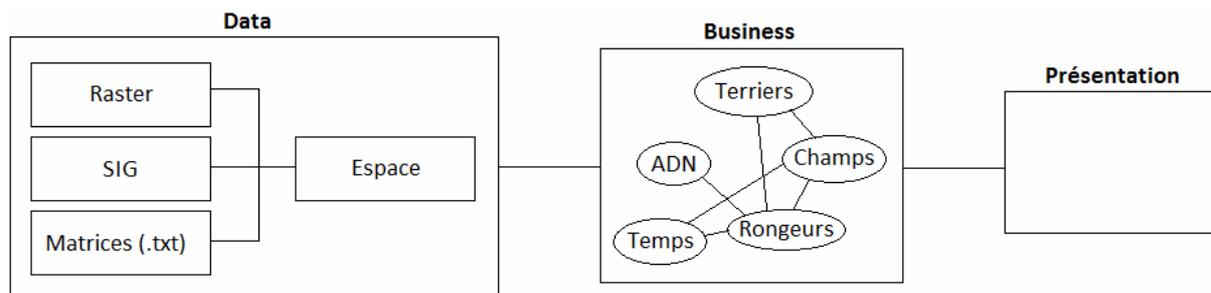


Image 3 : Design Data-Business-Présentation

La structure globale du module présentation s'est élaborée progressivement durant le stage. Pour comprendre un peu mieux comment fonctionne ce projet, voici les différentes étapes du programme (l'ordre correspond à l'ordre d'appel des scheduledMethods) :

- C_Inspecteur récupère la liste des individus et fait des calculs sur la population ;
- C_InspecteurGenetique fait des calculs sur les indicateurs génétiques et écrit les fichiers de sortie ;
- SimMastoInitializer initialise et met à jour les graphiques ;
- C_TableauDeBord affiche les informations demandées ;
- C_StepVariousProcedure augmente d'un jour la simulation (avec C_Calendar) ;
- C_Rodent gère les agents, leurs déplacements et leurs interactions.

C. Mise en œuvre

1. Liste des tâches

SimMasto est un projet fondé sur une approche adaptative en perpétuelle évolution, ce qui inclut qu'il est difficile de connaître à l'avance les prochaines étapes de développement. De plus, certaines tâches qui étaient pendant un temps prioritaires peuvent le devenir moins, notamment dans un travail en équipe. Néanmoins, voici une liste des tâches à accomplir durant mon stage :

| | Date d'identification | Date de réalisation | Date finalisation |
|---|-----------------------|---------------------|-------------------|
| Packages Repast liés au GUI | 16/02/2011 | 18/02/2011 | 18/02/2011 |
| Passage Repast 1,2 vers 2,0 | 10/03/2011 | 22/03/2011 | 22/03/2011 |
| Package epiphyte | 16/02/2011 | | |
| <i>recupération des indicateurs</i> | 16/02/2011 | 23/02/2011 | 18/04/2011 |
| <i>enregistrement indicateurs et GenePop en .csv</i> | 16/02/2011 | 10/03/2011 | 18/04/2011 |
| Système de création/gestion d'onglets | 16/02/2011 | 05/04/2011 | 20/04/2011 |
| graphes des indicateurs | 16/02/2011 | 06/04/2011 | 20/04/2011 |
| Console de sortie | 16/02/2011 | | |
| <i>renvoi de la console vers un onglet</i> | 16/02/2011 | 23/03/2011 | 18/04/2011 |
| Onglet " Tableau de bord " | 16/02/2011 | | |
| <i>vu-mètres</i> | 16/02/2011 | 23/03/2011 | 11/04/2011 |
| Calendrier | 16/02/2011 | | |
| <i>implémentation dans simulateur</i> | 16/02/2011 | 14/04/2011 | 14/04/2011 |
| <i>gestion dans simulateur</i> | 16/02/2011 | 15/04/2011 | 20/04/2011 |
| Affichage de champs icônes (avatars) | 16/02/2011 | 22/02/2011 | 15/04/2011 |
| Sortie de Repast en batch + linux pour sortie cluster | 25/02/2011 | | |
| <i>batch Windows (Eclipse)</i> | 11/03/2011 | 14/03/2011 | 14/03/2011 |
| <i>batch Windows (hors Eclipse)</i> | 11/03/2011 | 04/04/2011 | 04/04/2011 |
| <i>batch multiruns Windows</i> | 18/04/2011 | 20/04/2011 | 20/04/2011 |
| <i>batch multiruns Linux</i> | 18/04/2011 | 20/04/2011 | 20/04/2011 |
| <i>batch multiruns Cluster</i> | 23/03/2011 | 20/04/2011 | 20/04/2011 |

Image 4 : Liste des tâches

Mon travail s'insère dans un protocole multitâche. Pour mieux comprendre le liens entre toutes ces réalisations, voici un image du module présentation :

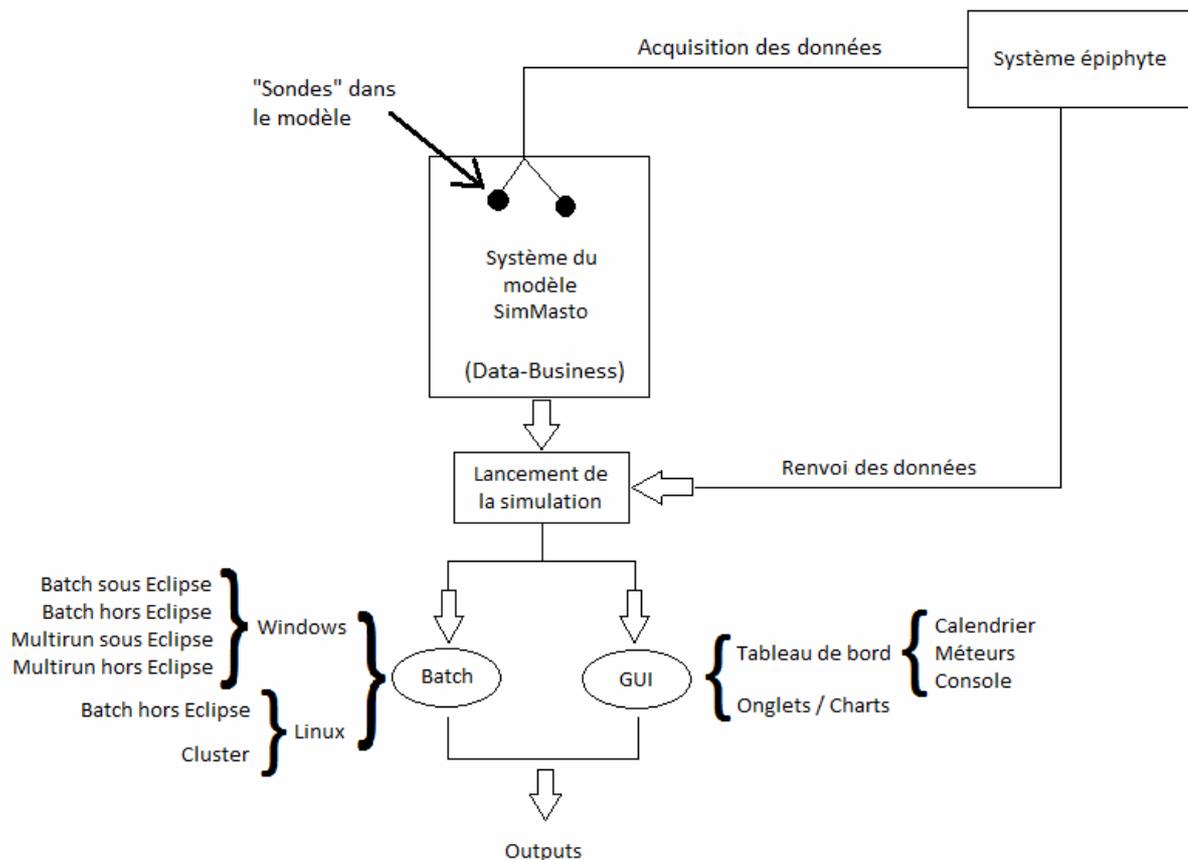


Image 5 : Module présentation

Le paquet « Système du modèle SimMasto » représente le module Data et Business du projet, c'est-à-dire toute la partie du programme qui existait avant que j'arrive. Le

système épiphyte a pour rôle de récupérer des indicateurs dans le système, sans que celui-ci s'aperçoive de son existence, puis de les renvoyer dans la simulation. Ces indicateurs seront renvoyés sous forme de fichiers de données dans tous les cas (lancement en batch ou GUI). Ils pourront aussi servir à construire des graphiques ou des *meters*, insérés dans l'interface graphique de la simulation.

2. Environnement de travail

- **Repast**

Le projet SimMasto est programmé avec Repast. J'ai choisi de programmer sous Eclipse car c'est l'IDE utilisé à l'IUT mais aussi par l'équipe dans laquelle j'ai travaillé.

J'ai commencé à développer avec Repast 1.2 puis je me suis occupée de réadapter le code à la version 2.0 (en beta). Le changement de version s'est avéré nécessaire lorsque quand nous avons voulu passer le modèle sous Linux pour pouvoir l'exécuter sur le cluster du CBGP. En effet, l'ancienne version n'était plus disponible. Nous étions donc obligés de changer de version.

- **JFreeChart³**

JFreeChart est une librairie java gratuite qui permet de créer des graphiques facilement qui pourront être ajoutés à des applications informatiques.

Cette librairie est utilisée par Repast pour créer ses graphiques. Cependant, les options disponibles restent trop minces. C'est pourquoi nous avons décidé de créer nos propres classes dérivées du système JFreeChart. Une autre raison était que nous souhaitions ajouter des *meters* à la simulation et cette librairie permet d'en créer et de les gérer facilement.

- **Piccolo**

Piccolo est une librairie graphique gratuite, utilisée dans la version 1.2 de Repast. Elle permet d'attribuer une image ou une forme géométrique à un agent.

L'utilisation de Piccolo a été abandonnée dans la version beta de Repast, au profit de l'OpenGL, ce qui a nécessité un retour en arrière sur certaines classes du code, notamment celles ayant un lien avec des représentations graphiques (le style des agents et la représentation de leur environnement).

- **OpenGL**

L'OpenGL (Open Graphics Library) est une librairie graphique utilisée dans la conception d'applications générant des images 2D ou 3D. Elle est gratuite et utilisée par de nombreux programmes comme Repast 2.0 par exemple.

³ Lien vers le site de JFreeChart (téléchargement possible à partir de ce site) : <http://www.jfree.org/jfreechart/>

Son utilisation a été imposée par le passage de la version 1.2 à la version 2.0 de Repast. En effet, ce dernier utilise l'OpenGL pour toutes ses classes en rapport avec l'affichage d'images ou les représentations de l'environnement.

- SVN

Le SVN ou subversion est un logiciel qui permet de stocker un ensemble de fichiers en conservant la chronologie de toutes leurs modifications. Son utilisation est très pratique pour sauvegarder un projet sur un serveur mais aussi pour retourner facilement à une version antérieure. Il a été conçu pour permettre le développement collaboratif.

L'utilisation du SVN a été mise en place au milieu du stage avec l'aide de S. Piry. Le logiciel que j'ai utilisé s'appelle *turtoiseSVN*. Je me suis aussi servi du SVN d'Eclipse. L'intérêt d'utiliser le SVN était de pouvoir sauvegarder notre avancement mais surtout de partager ses modifications avec les autres membres du groupe pour que tout le monde puisse travailler sur une version récente du code.

- Cluster

Le cluster est un système informatique composé d'unités de calcul (micro-processeurs, cœurs, unités centrales) autonomes qui sont reliées entre elles à l'aide d'un réseau de communication.

Ce système permet d'effectuer des calculs plus rapidement que sur un ordinateur personnel. Pour le projet SimMasto, son utilisation serait utile pour les calculs de sensibilité que doit réaliser J-E Longueville. Au CBGP, tous les ordinateurs du cluster sont sous Linux. Il a donc fallu écrire des scripts en Shell pour lancer le modèle sur les machines du cluster.

D. Structure du module présentation

I. Le système épiphyte

Le système épiphyte tire son nom des plantes dites épiphytes qui ont la particularité de se fixer sur d'autres plantes pour se développer. Le but du système que nous avons développé est de retirer des informations pour les traiter par la suite. Le cahier des charges spécifiait que le système épiphyte puisse capter des informations dans le simulateur sans modifier son fonctionnement ; c'est-à-dire que si l'on supprime le package « épiphyte », la simulation reste fonctionnelle. Nous avons donc essayé, dans la mesure du possible, de respecter cette contrainte.

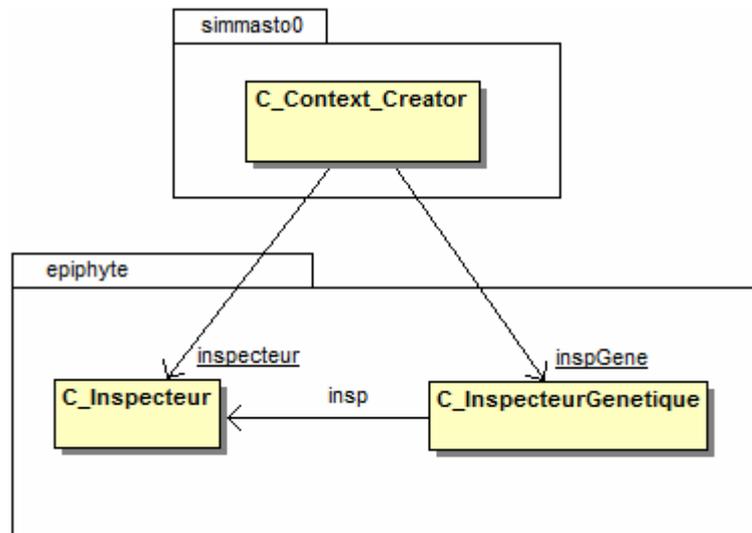


Figure 1 : Diagramme de classes du système épiphyte

Le système épiphyte a été représenté par le package « epiphyte » dans lequel ont été créés deux inspecteurs : la classe `C_Inspecteur` correspond à un inspecteur de données et `C_InspecteurGenetique` est, comme son nom l'indique, un inspecteur génétique. Ces classes sont instanciées dans `C_Context_Creator` afin d'être disponibles dès le lancement d'une simulation. De plus, elles sont placées en statiques pour que leurs variables soient plus simples d'accès par les autres classes, sans devoir instancier un nouvel inspecteur.

1. L'inspecteur de données

L'inspecteur de données correspond à la classe `C_Inspecteur` du package epiphyte. Il s'occupe principalement de relever des informations quantitatives sur la population à partir du contenu du contexte. La classe contient principalement des variables correspondantes à des indicateurs clés, et leur méthode `get` associée.

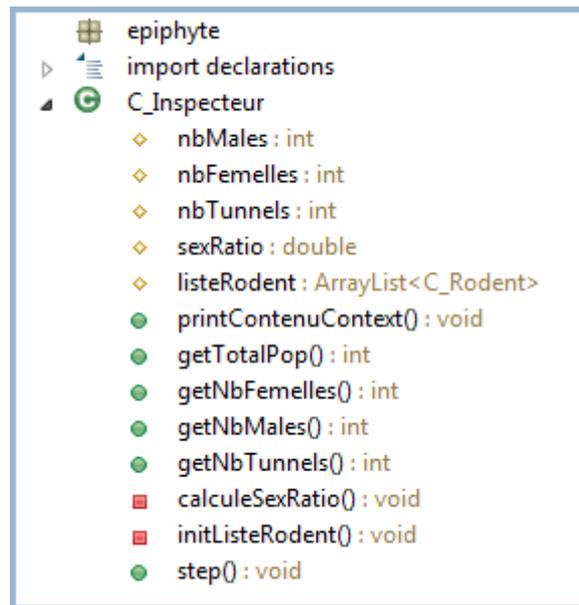


Figure 2 : Outline de C_Inspecteur

Pour récupérer des informations à partir du Context, il faut commencer par le récupérer dans la classe où on se trouve. La meilleure méthode est d'utiliser la commande de RS, `RunState.getInstance().getMasterContext()`. Ensuite, il faut récupérer son contenu. Pour cela, on ajoute `.toArray()` à la commande précédente. On obtient alors une liste d'objets.

A partir de la liste récupérée précédemment, on trie les éléments afin d'en créer une nouvelle. Ce qui nous intéresse est d'avoir une liste de `C_Rodent`⁴, mais on veut aussi connaître le nombre de mâles, de femelles et de colonies (aussi appelées tunnels systems). Les calculs se divisent en trois étapes :

1. On teste les éléments de la première liste puis on ajoute les instances de `C_Rodent` à `listeRodent` ;
2. Si l'élément que l'on est en train d'examiner est un `C_Rodent`, on le teste pour savoir s'il s'agit d'un mâle ou d'une femelle et on incrémente le compteur correspondant ;
3. Si ce n'est pas un `C_Rodent`, on regarde si c'est une instance de `C_TunnelSystem`⁵ et on incrémente `nbTunnels` si c'en est une.

```
protected int nbMales, nbFemelles, nbTunnels;
protected ArrayList<C_Rodent> listeRodent;

/** Initialise une liste de C_Rodent à partir du contenu du context
et calcule aussi le nombre de males, de femelles et de tunnels
présents */
private void initListeRodent()
{
    Object[] liste = RunState.getInstance().getMasterContext()
.toArray();
```

⁴ Classe qui représente les rongeurs étudiés dans la simulation.

⁵ Système de colonies de SimMasto.

```

listeRodent = new ArrayList<C_Rodent>();
nbMales=0; nbFemelles=0; nbTunnels=0;

for(int i=0 ; i<liste.length ; i++)
    if(liste[i] instanceof C_Rodent)
    {
        listeRodent.add((C_Rodent)liste[i]);
        // On calcule le nombre de males et de femelles
        dans la liste de rodent //
        if(((C_Rodent)liste[i]).isMale())
            nbMales++;
        else nbFemelles++;
    }
    else if(liste[i] instanceof C_TunnelSystem)
        nbTunnels++;
}

```

Un autre indicateur important est le sex-ratio. Pour le calculer, on utilise la méthode `calculeSexRatio()` de `C_Inspecteur` qui écrit sa valeur dans la variable `sexRatio`. Il est calculé en divisant le nombre de mâles par le nombre de femelles. Pour éviter la division par 0 dans le cas où toutes les femelles seraient mortes, `sexRatio` vaudra -1.

Enfin, tous ces indicateurs doivent être mis à jour à chaque tick. Pour cela, on crée une `scheduledMethod` dans `C_Inspecteur`. Elle appelle `initListeRodent()` et `calculeSexRatio()` avant de stopper la simulation si la population est morte ou si une des conditions d'arrêt de la simulation est vérifiée :

```

/** Appelle initListeRodent() et calculeSexRatio().
 * Termine la simulation si la population est nulle ou si une des
 conditions établies est vérifiée */
public void step() {
    initListeRodent();
    calculSexeRatio();

    if(listeRodent.isEmpty()) {
        System.out.println("Population is extinct");
        RepastEssentials.EndSimulationRun(); // Terminera la
        simulation une fois que tous les step() seront passés !
    }
    else if(RepastEssentials.GetTickCount()>=
I_sim_constants.TICK_MAX)
    {
        System.out.println("TickMax atteint. Arrêt de la
simulation");
        RepastEssentials.EndSimulationRun();
    }
}

```

2. L'inspecteur génétique

La classe `C_InspecteurGenetique` récupère des informations sur les gènes et les allèles de la population et ceci pour chaque locus⁶. Ces calculs sont inscrits dans

⁶ Un locus est l'endroit où est placé un gène sur un chromosome.

des fichiers de sortie et permettent de calibrer le modèle et de voir quels sont les paramètres sensibles, c'est-à-dire, ceux qui font varier les résultats quand on les modifie.

(: Outline de `C_InspecteurGenetique` disponible à la fin du chapitre)

1. Les calculs

Pour calculer la richesse allélique, il faut connaître le nombre d'allèles différents présent dans la population. Pour cela, on crée une liste contenant tous les allèles différents dans la population. Pour l'initialiser, on récupère les allèles du premier individu. Ensuite, on vérifie que son gène⁷ soit hétérozygote pour ajouter les deux allèles, sinon on en ajoute seulement un (celui de la chromatide gauche dans le code) :

```
/**
 * Initialise la liste d'allèles différents, pour le locus en
 * paramètre, avec le premier rongeur de listeRodent (liste dans
 * C_Inspecteur).
 * @param locus : numéro de locus pour lequel on veut connaître la
 * richesse allélique
 */
private ArrayList<Object> initListeAllDiff(int locus){
    ArrayList<Object> listeAllelesDiffParLocus = new
ArrayList<Object>();
    C_GenePair paireGenes = (C_GenePair)insp.listeRodent.get(0)
.getMicroSat().getLocusAllele(locus);
    listeAllelesDiffParLocus.add(paireGenes.getGene
(C_Chromosome.LEFT).getAllele());
    if(isHeterozygote(paireGenes))
        listeAllelesDiffParLocus.add
(paireGenes.getGene(C_Chromosome.RIGHT).getAllele());

    return listeAllelesDiffParLocus;
}
```

- ☞ Les allèles appartiennent à la classe `Object` de Java puisqu'un allèle peut être un entier, un double ou même une chaîne de caractères. Les calculs sont donc faits de la manière la plus générique possible.

Pour calculer la fréquence d'un allèle, il faut créer une liste contenant tous les allèles de la population pour un locus. Ensuite, il faut calculer le nombre d'occurrences de chaque allèle de ce locus puis enregistrer le résultat dans un tableau :

```
private C_Inspecteur insp;
private ArrayList<Double>[] frequencesAlleliques = new ArrayList
[I_animal_constants.NUMBER_GENES];

/** Calcule les fréquences alléliques pour le locus en paramètre et
les écrit dans la liste frequencesAlleliques */
private void frequenceAllelique(int locus)
{
```

⁷ Le modèle `SimMasto` étant simplifié, les agents ne possèdent qu'un gène par locus.

```

/* Formule : nb occurrences d'un allele / (2*nb indiv) */
ArrayList<Double> freq = new ArrayList<Double>();

if(!insp.listeRodent.isEmpty())
{
    ArrayList<Object> listeAlleles = getListeAll(locus);

    while(!listeAlleles.isEmpty())
    {
        // On récupère le premier allèle de la liste pour
        // le comparer aux autres //
        Object allele = listeAlleles.get(0);
        listeAlleles.remove(0); // On supprime l'allèle de
        // référence

        /* On récupère l'index de la liste où se trouve la
        // première occurrence de l'allèle
        // * de référence. S'il n'y en a pas, l'index vaut -1
        // */
        int cpt=1;
        int index = listeAlleles.indexOf(allele);

        while(index!=-1) {
            listeAlleles.remove(index);
            cpt++;
            index = listeAlleles.indexOf(allele);
        }

        freq.add(convertNumber((double)cpt/
        (2*insp.listeRodent.size()),5));
    }
    frequencesAlleliques[locus] = freq;
}

```

Le calcul de d'autres indicateurs a été codé comme la richesse allélique moyenne, l'hétérozygotie observée et l'hétérozygotie attendue (*sources non présentées*).

Formule de la richesse allélique pour le locus i : nb d'allèles différents pour ce locus

Formule de l'hétérozygotie observée pour le locus i : nombre d'hétérozygotes au locus i / nombre d'individus

Formule de l'hétérozygotie attendue pour le locus i : $1 - \sum (\text{fréquences alléliques au locus } i)^2$

2. Les écritures de fichiers

C_InspecteurGenetique contient trois méthodes afficheDonneesGenePop(), afficheIndicateurs() et afficheIndicateursReducit() qui écrivent les informations demandées dans les fichiers de sortie avec l'aide de la classe C_RenvoiDonnees.

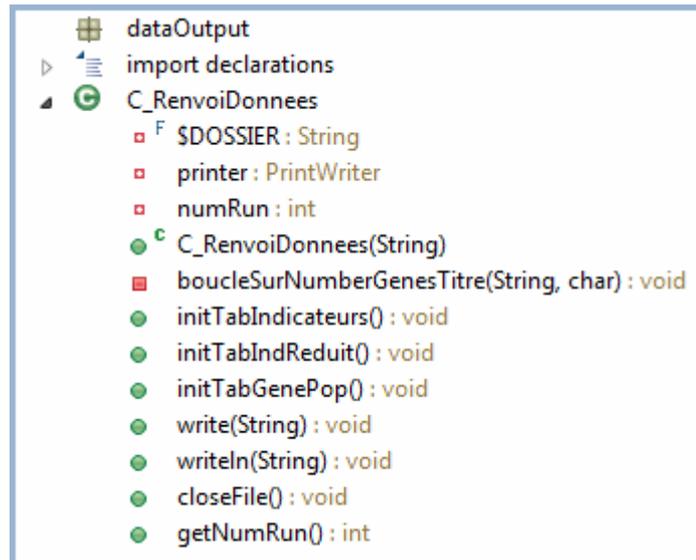


Figure 3 : Outline C_RenvoiDonnees

Il existe deux types de fichiers de sortie :

- un fichier de génétique des populations qui affiche les allèles de chaque individu pour chaque locus ;
- un fichier d'indicateurs, contenant des informations comme la taille de la population, le sexe ratio, le nombre de colonies, mais aussi la richesse allélique moyenne, l'hétérozygotie observée et l'attendue, et ceci pour chaque tick.

Ces fichiers permettent aux chercheurs de traiter les résultats des simulations avec les logiciels spécialisés comme R pour le traitement des fichiers au format CSV, consultables aussi avec Excel, et *genePop*⁸ qui n'as pas besoin d'extension de fichiers.

Tous les fichiers de sortie sont créés dans le dossier « DonneesAlleliques » qui est recréé s'il n'existe plus. Les titres des fichiers commencent par un numéro pour empêcher leur écrasement. En effet, comme le projet peut être lancé en batch, le groupe est souvent amené à lancer plusieurs *runs* à la suite. Il faut donc conserver tous les fichiers de sortie :

```
private final String $DOSSIER = "DonneesAlleliques/";
private PrintWriter printer;
private int numRun=0;

/** Crée un nouveau fichier avec le titre en paramètre (avec
l'extension !) */
public C_RenvoiDonnees(String titre) {
    new File($DOSSIER).mkdir();
    File file = new File($DOSSIER+numRun+titre);

    while(file.exists()) {
        numRun++;
    }
}
```

⁸ Lien vers le site genePop : <http://genepop.curtin.edu.au/>

```

        file = new File($DOSSIER+numRun+titre);
    }

    try {
        printer = new PrintWriter(new BufferedWriter(new
FileWriter(file)));
    } catch (IOException e) {e.printStackTrace();}
}

```

Le contenu des fichiers d'indicateurs possède des titres pour chaque colonne. Ils sont écrits à la main par le programmeur dans une fonction appelée `initTabIndicateurs()` ou encore `initTabIndReduit()` :

```

/** Ecrit les titres pour le fichier d'indicateurs de population */
public void initTabIndicateurs() {
    boucleSurNumberGenesTitre("locus", ';');
    boucleSurNumberGenesTitre("frequenceLoc", ';');
    boucleSurNumberGenesTitre("heteroAttendue", ';');

    printer.println("HeteroObservee;RichAllMoyenne;Tick;TaillePop;
NbMales;NbFemelles;NumRun");
    printer.flush();
}

```

La méthode `boucleSurNumberGenesTitre(String titre, char separateur)` permet d'écrire `n` fois le titre en paramètre en lui ajoutant un indice à la fin et chacun séparé par le séparateur spécifié. Cette dernière variable est très utile car suivant le logiciel qui sera utilisé pour traiter les fichiers de sortie, les séparateurs ne sont pas les mêmes.

Les méthodes de `C_InspecteurGenetique` qui renvoient les informations ne sont pas automatisées et c'est au programmeur de faire attention aux titres des colonnes dans les fichiers quand il écrit les informations. Voici la méthode associée aux titres de `initTabIndicateurs()` vue précédemment :

```

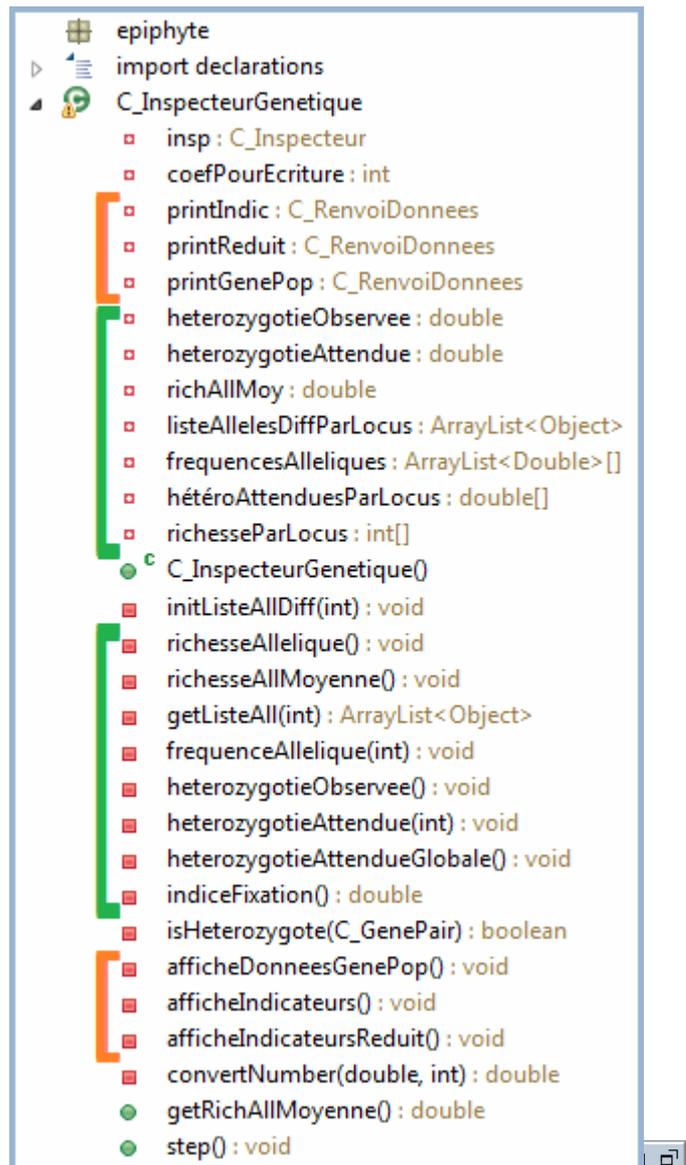
/** Ecrit les données dans le fichier des indicateurs */
private void afficheIndicateurs()
{
    for(int i=0 ; i<I_animal_constants.NUMBER_GENES ; i++)
        printIndic.write(richeParLocus[i]+";");
    for(int i=0 ; i<I_animal_constants.NUMBER_GENES ; i++)
        printIndic.write(frequenceAlleliques[i]+";");
    for(int i=0 ; i<I_animal_constants.NUMBER_GENES ; i++)
        printIndic.write(heteroAttenduesParLocus[i]+";");

    printIndic.writeln(heterozygotieObservee+";"+richAllMoy+";"+
RepastEssentials.GetTickCount()+";"+insp.listeRodent.size()+";"+
insp.nbMales+";"+insp.nbFemelles+";"+printIndic.getNumRun());
}

```

L'inspecteur génétique possède une méthode `step()` de type `scheduledMethod` qui appelle toutes les méthodes de calcul (voir accolade verte) et écrit les fichiers de sortie (voir accolade orange). Comme il y a beaucoup de calculs à faire, la première chose que l'on fait dans `step()` est de vérifier que la liste de rongeurs, `listeRodent` de `C_Inspecteur` n'est pas vide. Auquel cas, il est inutile de faire des calculs. De même, si la population est éteinte, on ferme les fichiers de sortie.

Figure 4 : Outline de `C_InspecteurGenetique`



II. Le tableau de bord

Le tableau de bord est un outil visuel emboîté dans l'interface graphique de Repast. Il doit permettre de vérifier à tout moment que la simulation se déroule correctement, en affichant les variations de divers indicateurs clés comme la date simulée, la population totale mais aussi la richesse allélique moyenne ou encore le nombre de colonies présentes dans l'environnement. L'objectif était d'étendre les

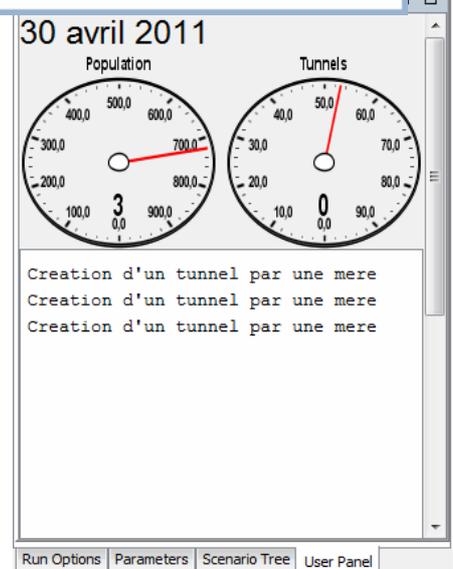


Image 6 : Tableau de bord

fonctionnalités du GUI de Repast.

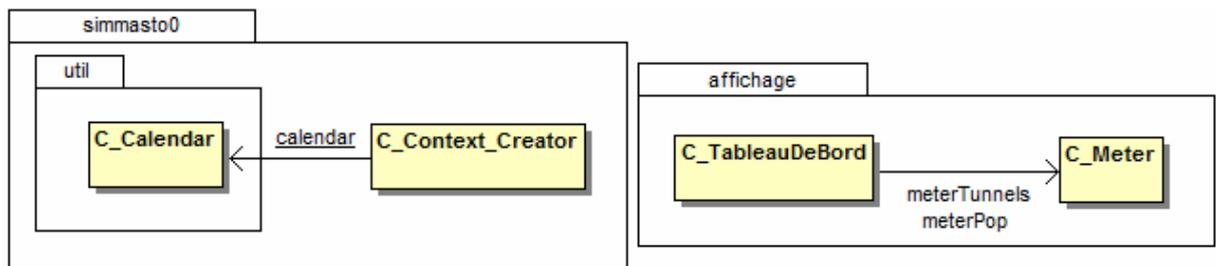


Figure 5 : Diagramme des classes qui constituent le tableau de bord

La classe `C_TableauDeBord` est instanciée par Repast grâce à son implémentation du `UserPanelCreator` qui permet de créer un panneau dans le `userPanel`⁹. Cette interface ne sera pas utilisée si la simulation est lancée en batch, ce qui est logique puisqu'il n'y a pas d'affichage dans ce mode de lancement. Cependant, ceci empêche la création de `C_Calendar` dans le tableau de bord. C'est pourquoi cette classe est instanciée dans `C_Context_Creator` pour être forcément créée et utilisable rapidement, mais elle est aussi placée en statique pour être disponible dans tout le modèle. Enfin, les *meters* donnent des informations sur la simulation mais ne font aucun calcul. De ce fait, ils peuvent être instanciés dans `C_TableauDeBord` et ne seront utilisés qu'avec le GUI.

1. Le calendrier

La classe `C_Calendar` permet de connaître la date courante dans la simulation mais aussi de gérer les mois de reproduction de la population.

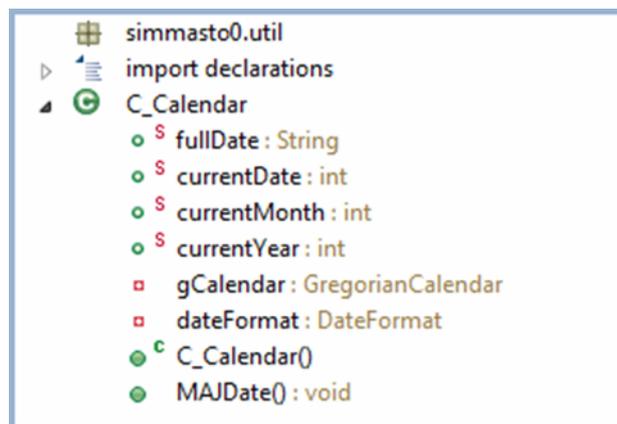


Figure 6 : Outline de `C_Calendar`

Pour utiliser un calendrier, on instancie un `GregorianCalendar` dans le constructeur. Cette classe de Java gère les calendriers grégoriens. De ce fait, il n'est pas nécessaire de créer un nouveau calendrier de A à Z. Il est possible de spécifier l'endroit où on se trouve à l'initialisation mais `GregorianCalendar` récupère

⁹ Onglet de Repast permettant d'ajouter un contenu. Dans `simMasto`, il est utilisé en tant que tableau de bord.

automatiquement les informations de la machine virtuelle java que l'on utilise pour le connaître. Ensuite, il faut définir à quelle date commence la simulation :

```
gCalendar = new GregorianCalendar();
gCalendar.set(2011, Calendar.JANUARY, 1);
```

Comme on souhaite utiliser cette classe à la fois pour afficher la date de la simulation mais aussi pour gérer les périodes de reproduction, il faut créer des variables permettant de récupérer plusieurs informations sur la date. On utilise le type `DateFormat` pour transformer la date en chaîne de caractères pour ensuite l'afficher dans le tableau de bord. De plus, comme il est inutile de connaître le jour de la simulation étant donné que les agents ne réagissent pas à ce paramètre, on le supprime de la variable `fullDate` utilisée pour afficher la date dans le tableau de bord :

```
dateFormat = DateFormat.getDateInstance(DateFormat.FULL,
    Locale.getDefault()); //Affiche la date en toutes lettres et au
format de la localisation
fullDate = dateFormat.format(gCalendar.getTime()); // retourne un String
jour/date/mois/année (ex: jeudi 21 avril 2011)
fullDate = fullDate.substring(fullDate.indexOf(" ") + 1); // Supprime le
jour (ex: 21 avril 2011)
```

La commande `DateFormat.getDateInstance(DateFormat.FULL, Locale.getDefault())` permet de définir le format de la date souhaité. Le premier paramètre peut être `FULL` (ex : jeudi 14 avril 2011) ou `SHORT` (ex : 14/04/2011). Le second est la localisation de l'utilisateur ou du moins, celle qu'il veut utiliser pour son programme. En effet, l'écriture ne sera pas la même en France et aux Etats-Unis par exemple. La commande `getDefault()` permet de récupérer la localisation de la machine virtuelle Java.

La dernière commande cherche à quel indice+1 est présent le premier espace dans la chaîne de caractères `fullDate` pour supprimer tout ce qui est avant.

Pour mettre à jour le calendrier, on crée une fonction `MAJDate()`, appelée par la `scheduledMethod` de `C_StepVariousProcedure` à chaque tick. Pour ajouter un jour à la simulation, on utilise la commande `gCalendar.add(Calendar.DATE, 1)` où le premier argument représente le champ à changer (jour, mois ou année) et le deuxième l'incrément ou la décrémentation à associer au premier. On met ensuite à jour toutes les variables disponibles pour les autres classes :

```
/** Met à jour la date de la simulation ainsi que toutes les variables
associées */
public void MAJDate() {

    gCalendar.add(Calendar.DATE, 1); // Ajoute un jour à la simulation

    fullDate = dateFormat.format(gCalendar.getTime());
    fullDate = fullDate.substring(fullDate.indexOf(" ") + 1);
```

```

    currentDate = gCalendar.get(Calendar.DATE);
    currentMonth = gCalendar.get(Calendar.MONTH);
    currentYear = gCalendar.get(Calendar.YEAR);
}

```

2. Les meters

Les *meters* servent à suivre l'évolution de certains indicateurs. Ils peuvent être créés grâce à la librairie JfreeChart. J'ai donc fait une classe C_Meter qui crée des compteurs.

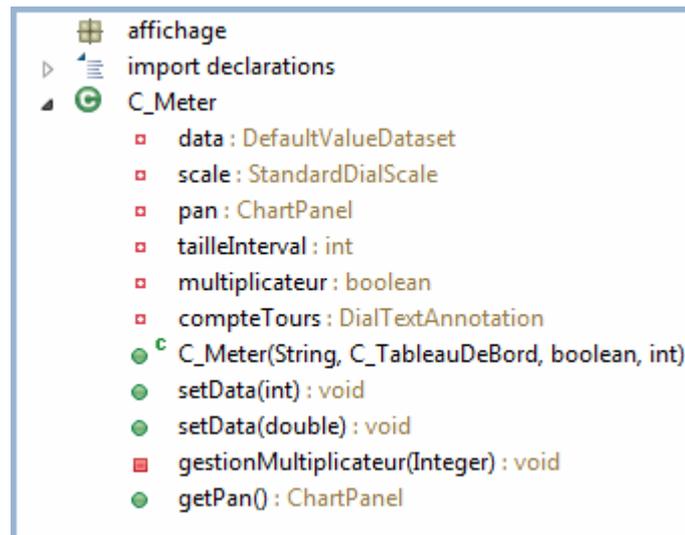


Figure 7 : Outline de C_Meter

Pour créer un *meter*, on instancie un `DefaultValueDataset` qui représente la valeur de l'indicateur représenté par le compteur, puis un `DialPlot` qui contient tous les éléments qui constituent le *meter*. Ensuite, on instancie un `StandardDialFrame` qui sert de fenêtre au *meter*. Enfin, on ajoute le tout au `DialPlot` :

```

private DefaultValueDataset data;
private boolean multiplicateur;

/**
 * Crée un meter
 * @param title : Le nom du meter
 * @param multiplicateur : Active ou non l'indice (le multiplicateur)
 * au centre du meter
 * @param interval : La valeur max du meter
 */
public C_Meter(String title, boolean multiplicateur, int interval) {
    this.multiplicateur = multiplicateur;

    DialPlot plot = new DialPlot();
    data = new DefaultValueDataset(0); // Valeur de départ //
    plot.setDataset(data);

    StandardDialFrame df = new StandardDialFrame(); // Ne pas oublier
    plot.setDialFrame(df);
    df.setVisible(true);
    [...]
}

```

```
}
```

La deuxième étape consiste à instancier un `StandardDialScale` qui permet de modifier les valeurs du *meter*, l'affichage des ticks et le nombre de ticks visibles sur le *meter* et enfin d'ajouter un chiffre qui servira de multiplicateur si la valeur maximum visible est dépassée. Par exemple, le *meter* pour la taille de la population varie de 0 à 1000. Si le nombre d'individus est de 2500, le multiplicateur sera de 2 et l'aiguille sur 500. De même, il sera de 0 si la population est de 200 ou encore de 11 s'il y a entre 11000 et 11999 agents.



```
// Réglages ticks et intervalles //  
scale = new StandardDialScale();  
scale.setLowerBound(0); // Valeur minimum //  
scale.setTickRadius(0.9); // Position des ticks par rapport au centre  
scale.setTickLabelOffset(0.2); // Position des chiffres par rapport  
au centre (TENIR COMPTE DU PARAM PREC) //  
scale.setMajorTickIncrement(interval/10); // Interval entre 2 traits  
  
if(multiplicateur) {  
    scale.setUpperBound(interval-0.01); // Valeur maximum // On  
    enlève 0.01 pour ne pas avoir le dernier chiffre affiché  
    scale.setStartAngle(-90); // Position de la valeur minimum //  
    scale.setExtent(-360); // Position de la valeur maximum //  
    tailleInterval = ((Integer)interval).toString().length();  
    compteTours = new DialTextAnnotation("0");  
    compteTours.setFont(new Font("Arial",Font.BOLD,30));  
    plot.addLayer(compteTours);  
}  
else {  
    scale.setUpperBound(interval); // Valeur maximum //  
    scale.setStartAngle(-120); // Position de la valeur minimum //  
    scale.setExtent(-300); // Position de la valeur maximum //  
}  
  
plot.addScale(0,scale); // scale est ajouté à l'index 0
```

Les méthodes `setStartAngle(double angle)` et `setExtent(double angle)` fonctionnent avec la trigonométrie. Par exemple, dans le cas du *meter* de la richesse allélique moyenne (voir l'image précédente, le compteur du milieu), on commence à -120. Si l'on veut faire un tour complet, on termine à -360. Or, dans cet exemple, on souhaite s'arrêter en -60. Pour cela, on enlève la distance entre -120 et -60, soit -60. On obtient alors -300 comme valeur d'arrivée.

Pour terminer l'initialisation des *meters*, il faut leur ajouter une aiguille de type `DialPointer` et l'ajouter au `DialPlot`. Enfin, on peut aussi interagir avec le cercle au centre du *meter* (le rendre plus ou moins gros par exemple) en instanciant un `DialCap` et en utilisant la fonction `setRadius(double radius)` avant de l'ajouter au `DialPlot`.

Pour finir, il faut instancier un `JFreeChart` en lui ajoutant le `DialPlot` en paramètre, puis instancier un `ChartPanel` comportant le `JFreeChart` :

```
JFreeChart chart = new JFreeChart(plot);
chart.setTitle(title);
pan = new ChartPanel(chart);
pan.setPreferredSize(new Dimension(150,150));
pan.setMaximumSize(pan.getPreferredSize());
```

Lorsque l'initialisation est terminée, il faut mettre à jour les *meters*. Pour cela, on crée deux méthodes, `setData(int value)` et `setData(double value)` qui appellent `data.setValue(value)` pour mettre à jour la valeur de l'indicateur représenté. Ensuite, on vérifie si le `multiplicateur` est activé ou non puis on appelle la méthode `gestionMultiplicateur(Integer value)` s'il l'est. Cette méthode a pour but de mettre à jour le multiplicateur. On commence par vérifier que la taille de la valeur entière en paramètre soit supérieure ou égale à la taille de l'intervalle inscrit en paramètre du constructeur de `C_Meter`. Par exemple, si la population est de 2000 et que l'intervalle est de 1000, la condition est vérifiée et il faudra alors afficher « 2 » au compte tours. A l'inverse, si la population est de 200 et l'intervalle de 1000, le compte tours devra afficher « 0 » et il n'y a donc pas de calcul à faire. Ensuite, si la condition a été vérifiée, on ne garde que le coefficient multiplicateur qui nous intéresse, c'est-à-dire les premiers chiffres de la valeur en paramètre. Pour cela, on supprime ses n-1 derniers chiffres, où n correspond à la taille de l'intervalle :

```
/**
 * Met à jour le multiplicateur au centre du compteur
 * @param value : la valeur entière de la donnée mise à jour avec
 * setData()
 */
private void gestionMultiplicateur(Integer value)
{
    if(value.toString().length()>=tailleIntervall)
        compteTours.setLabel(value/(int)(Math.pow(10,
tailleIntervall-1))+""); // supprime les n-1 derniers chiffres
    else compteTours.setLabel("0");
}
```

3. La console

Le principe est toujours de contrôler la simulation en renvoyant la console d'Eclipse dans le tableau de bord : Quand la fonction `System.out.println()` ou ses dérivées sont utilisées, le contenu à afficher est renvoyé dans la console du tableau de bord au lieu de celle d'Eclipse. L'objectif final était de pouvoir filtrer les

informations, c'est-à-dire de voir uniquement des informations précises (génétiques, quantitatives...). Cependant, je n'ai pas eu le temps de mettre en place ce système.

Pour créer une nouvelle console, on instancie une `JConsole` dans `C_TableauDeBord`. Cette classe possède des flux d'entrées et de sorties qui facilitent la déviation de la console d'Eclipse vers celle-ci. La commande `createVerticalScrollBar()` associée à cette classe permet d'ajouter une barre de défilement vertical très facilement. La commande permettant de renvoyer le flux de la console principale vers celle-ci est `System.setOut(console.getOut())` où `console` est la nouvelle instance de `JConsole`.

Comme tout objet utilisant des flux au cours de l'exécution d'un programme, il faut les fermer avant de le terminer. La difficulté est de penser à tous les moyens qu'a une simulation pour se terminer. Le premier est d'attendre que la population soit éteinte et le second est de stopper manuellement la simulation sans quitter le programme (en appuyant sur le bouton « stop »). A partir de là, il faut trouver quelle est la meilleure classe pour fermer les flux. La classe `C_Inspecteur` sait quand la population est éteinte, cependant aucune classe n'est habilitée à savoir quand l'utilisateur a stoppé la simulation. Seule `SimMastolnitializer` sait quand une simulation est réinitialisée grâce à sa méthode `runCleanup(...)` appelée automatiquement. Les flux de la console seront donc fermés dans cette méthode si la console existe ! En effet, elle n'existe pas en batch puisque le tableau de bord n'est pas instancié.

4. La gestion du tableau de bord

Le tableau de bord contient un calendrier, des *meters* et une console. Il instancie de nouveaux *meters*, appelle leur fonction de mise à jour et s'occupe aussi de rafraîchir l'affichage du calendrier.

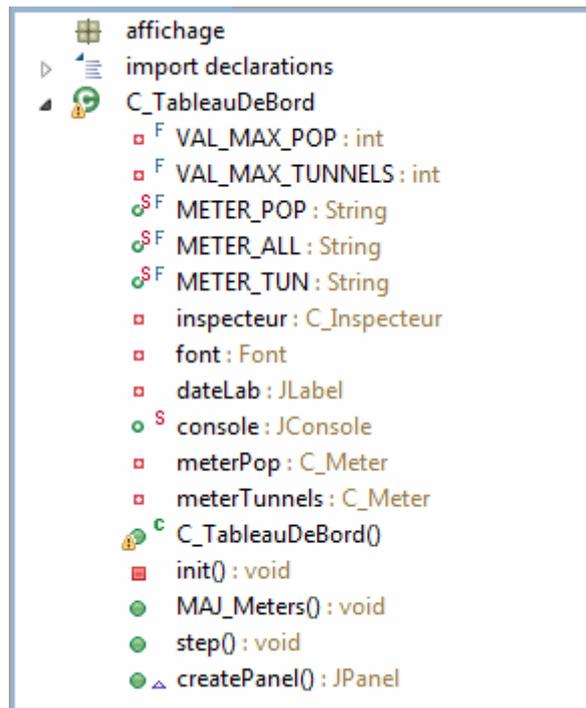


Figure 8 : Outline C_TableauDeBord

La classe `C_TableauDeBord` étend `JPanel` qui la définit comme étant un panneau dans lequel peuvent être insérés divers éléments. De plus, elle implémente `UserPanelCreator` qui ajoute `C_TableauDeBord` au « `UserPanel` » qui est déjà implémenté dans l'interface graphique de `Repast` mais reste vide tant qu'il n'est pas spécifiquement instancié par le programmeur.

Pour ajouter `C_TableauDeBord` à la simulation, il faut l'ajouter au « `user_path.xml` » (voir « *Présentation de Repast Symphony* »). Après avoir lancé le modèle, dans l'onglet « `ScenarioTree` », il faut faire un clic droit sur « `UserPanel` » et sélectionner « `Specify User Panel` ». Une fenêtre s'ouvre et il faut sélectionner une classe qui implémente `UserPanelCreator`.

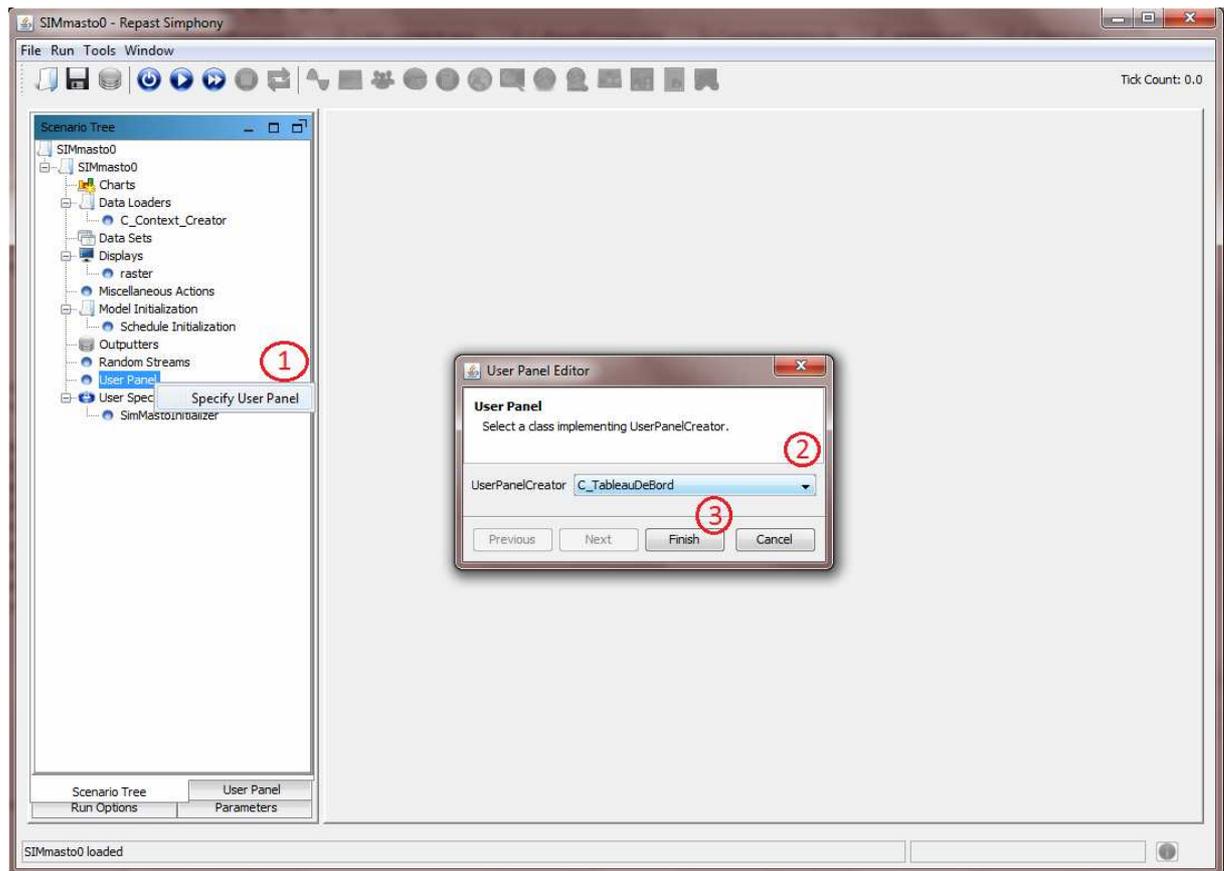


Image 7 : GUI de Repast pendant l'ajout du tableau de bord

Concernant `C_TableauDeBord`, on commence par initialiser un `JLabel` qui servira à afficher la date de la simulation. On utilise la variable statique `C_Calendar.fullDate` pour récupérer la date sous forme de `String` puis la commande `dateLab.setText(C_Calendar.fullDate)` (où `dateLab` est le `JLabel`) pour l'afficher. Ensuite, on initialise tous les *meters* puis on termine en instanciant la `JConsole`.

Il faut penser à ajouter tous ces composants au tableau de bord avec la commande `add(component)`.

Pour la mise en page, j'ai utilisé un `BorderLayout`. La date est placée au nord, les *meters* à l'est et à l'ouest et la console au sud. Pour appliquer cette mise en page, on utilise la commande `setLayout(new BorderLayout())`. Enfin, pour ranger les composants, on peut spécifier leur place au moment de leur ajout au `JPanel`. Pour `dateLab` par exemple, on écrit `add(dateLab, BorderLayout.NORTH)`.

Enfin, la mise à jour de l'affichage des variables de `C_TableauDeBord` se fait dans sa fonction `step()`. La mise à jour des indicateurs pour les *meters*, quant à elle, est faite dans une méthode que j'ai appelé `MAJ_Meters()`, elle-même appelée dans `step()`.

III. Onglets personnalisés et graphes

Repast offre la possibilité de créer des graphiques traçant l'évolution des variables du modèle à partir de son interface graphique. Cependant, les choix disponibles et la maniabilité des graphes sont trop faibles. Notre cahier des charges nécessitait la possibilité d'ajouter des onglets à la demande, par la programmation, c'est-à-dire sans passer par le GUI de RS. De plus, il fallait pouvoir ajouter divers types de graphiques, eux aussi programmables.

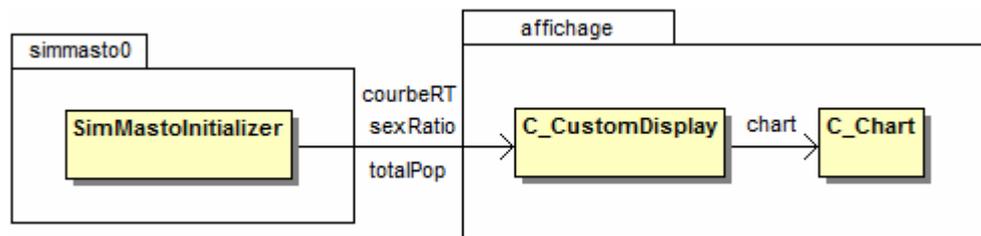


Figure 9 : Diagramme de classes pour les onglets personnalisés et leur graphique

La classe `SimMastolnitializer` que j'ai créée est initialisée par Repast à l'aide du fichier `scenario.xml` (voir « *Présentation de Repast Symphony* »). De plus, elle instancie des `C_CustomDisplay` qui représentent les onglets de la simulation et qui eux-mêmes instancie un `C_Chart` qui sera affiché dans le display.

1. Créer un graphique avec JfreeChart

Créer des graphiques plutôt que d'utiliser les fonctionnalités de Repast permet de modifier beaucoup plus de paramètres. Nous avons donc décidé de créer nos propres graphiques à l'aide de la librairie `JfreeChart`. Pour cela, j'ai créé une classe `C_Chart` qui permet de choisir le type de graphique souhaité, de lui ajouter des variables et de les changer pour faire évoluer les graphes au cours de la simulation.

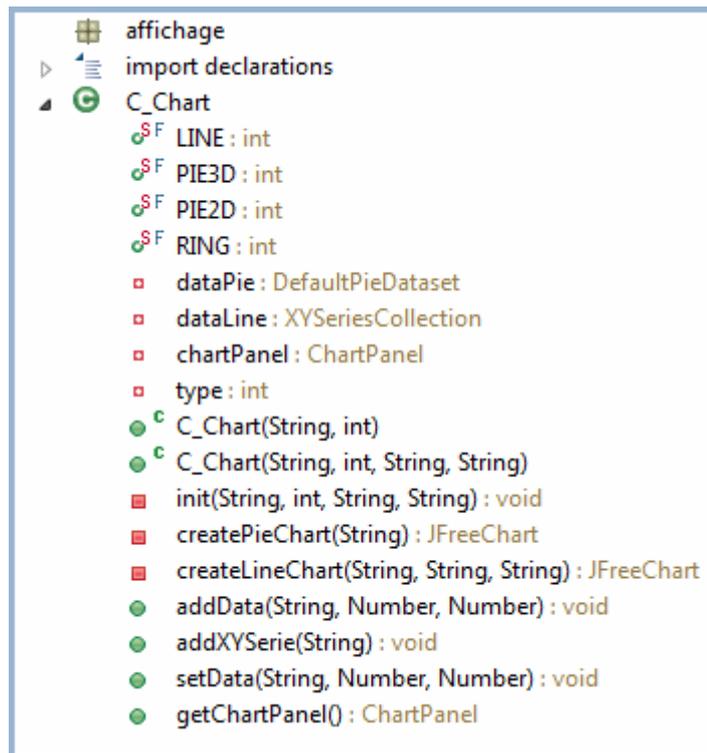


Figure 10 : Outline C_Chart

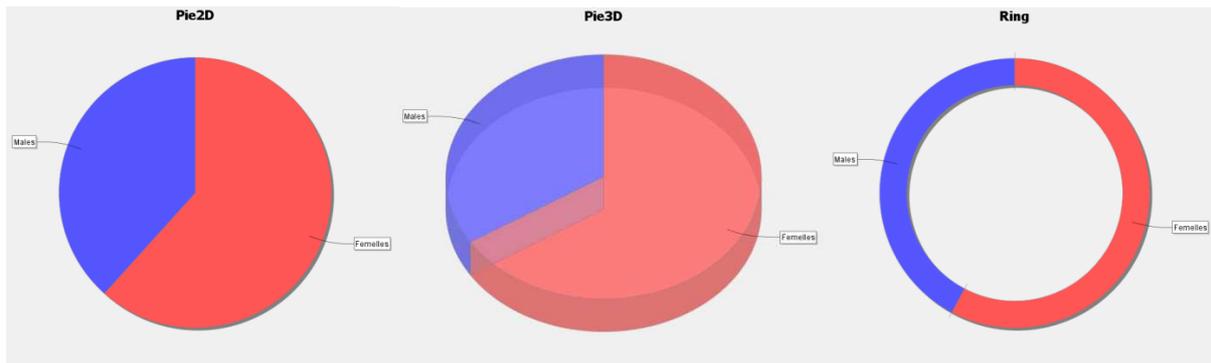
Nous avons décidé de représenter deux catégories de graphes :

- Les « Pie Charts » qui sont des diagrammes circulaires ;
- Les « Line Charts » qui sont des courbes.

Suivant la catégorie de graphique choisie, la création n'est pas la même car JfreeChart contient une méthode spécifique à chaque type¹⁰ pour les créer. De plus, les éléments contenus dans les « Pie Charts » et les « Line Charts » ne sont pas les mêmes. De ce fait, il faut préciser le type de graphique désiré en paramètre du constructeur. Il existe deux constructeurs. Un ayant en paramètre le titre du *chart* et son type et un autre le nom de l'axe des abscisses et celui des ordonnées en plus. Le deuxième constructeur est utilisé pour créer un « Line chart ». Le premier peut être utilisé par n'importe quel type mais les axes auront comme nom par défaut, « X » et « Y ».

¹⁰ Dans SimMasto, le type peut être PIE3D, PIE2D, RING ou LINE. Il en existe beaucoup d'autres dans JfreeChart.

1. Les « Pie Chart »



Si le type est `PIE2D`, `PIE3D` ou `RING`, alors c'est la fonction `createPieChart(String title)` qui est appelée :

```
private DefaultPieDataset dataPie ;

/** Crée un chart de type Pie, Pie3D ou Ring */
private JFreeChart createPieChart(String title)
{
    JFreeChart chart=null;

    if(type==PIE3D) {
        chart = ChartFactory.createPieChart3D(title,dataPie,false,false,
        ,false);
    }
    else if(type==RING) {
        chart = ChartFactory.createRingChart(title,dataPie,false,false
        ,false);
    }
    else if(type==PIE2D) {
        chart = ChartFactory.createPieChart(title,dataPie,false,false
        ,false);
    }

    [...]

    return chart;
}
```

La commande `ChartFactory.createPieChart3D(title,dataPie,false,false,false)` permet de créer un diagramme circulaire en trois dimensions. Les arguments de cette fonction (et des deux autres) sont le titre du graphique, la variable qui contient l'ensemble des données, un booléen pour activer la légende, un pour activer les info-bulles et un dernier correspondant aux URL¹¹.

Pour ajouter des données au « Pie chart », on utilise la méthode `addData(String title, Number Xvalue, Number Yvalue)` de `C_Chart`. Le titre en paramètre correspond au titre que l'on veut attribuer à la donnée. Ensuite, seule `Xvalue` est utilisée. En effet, cette fonction peut aussi être utilisée par les « Line

¹¹ La documentation n'est pas claire sur ce dernier paramètre. Je ne sais donc pas à quoi il sert.

charts » qui possèdent deux variables, x et y, à la différence des « Pie charts » qui n'en possèdent qu'une. La valeur d'Yvalue ne sera alors pas prise en compte :

```

/** Ajoute une donnée à la série en paramètre.
 * @param title : le titre de la série utilisé lors de sa création
 * @param Xvalue : la valeur en X
 * @param Yvalue : la valeur en Y (null pour PIE2D, PIE3D, RING);
 */
public void addData(String title, Number Xvalue, Number Yvalue) {
    if(type==LINE)
        dataLine.getSeries(title).add(Xvalue,Yvalue);
    else dataPie.insertValue(dataPie.getItemCount(),title,Xvalue);
}

```

La commande `dataPie.insertValue(dataPie.getItemCount(),title,Xvalue)` insert la nouvelle donnée dans `dataPie` dans un nouvel emplacement grâce à la commande `dataPie.getItemCount()`, avec le titre et la valeur en paramètre. En effet, si `dataPie.getItemCount()` renvoie 5, c'est qu'il y a déjà 5 données et il n'y aura donc rien à cet indice puisqu'ils commencent à 0.

Pour mettre à jour les données, on utilise la fonction `setData(String serie, Number Xvalue, Number Yvalue)` :

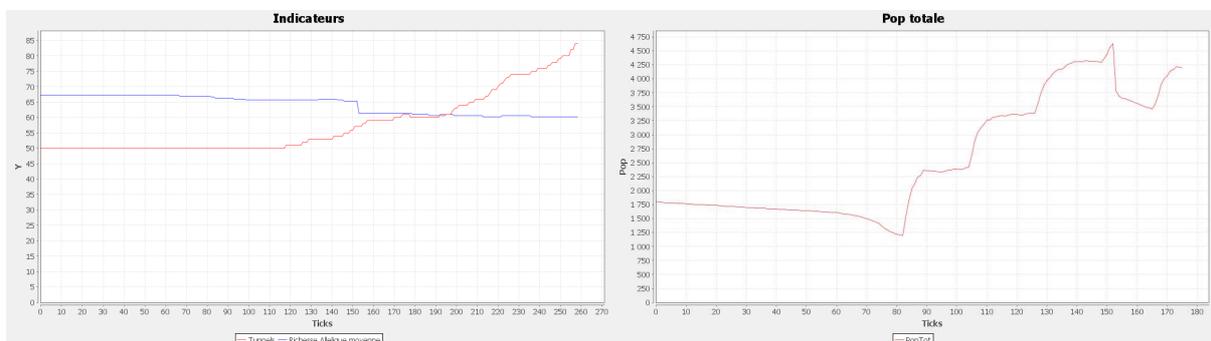
```

/** Met à jour les données d'un chart
 * @param serie : le nom de la série à mettre à jour
 * @param Xvalue : la valeur en X
 * @param Yvalue : la valeur en Y (null pour PIE2D, PIE3D, RING);
 */
public void setData(String serie, Number Xvalue, Number Yvalue) {
    if(type==LINE)
        dataLine.getSeries(serie).add(Xvalue, Yvalue);
    else dataPie.setValue(serie, Xvalue);
}

```

La commande `dataPie.setValue(serie,Xvalue)` modifie la valeur d'une série déjà existante. Comme pour `addData(...)`, la valeur d'Yvalue n'est pas utilisée.

2. Les « Line charts »



Si le type du graphique est LINE, c'est la fonction `createLineChart(String title)` qui est appelée :

```

/** Crée un line chart */

```

```

private JFreeChart createLineChart(String title, String XLabel,
String YLabel) {

    JFreeChart chart = ChartFactory.createXYLineChart(title,XLabel,
YLabel,dataLine,PlotOrientation.VERTICAL,true,true,false);
    [...]
    return chart;
}

```

Cette commande crée une courbe. Les arguments de cette fonction sont le titre du graphique, le nom de l'axe des abscisses, celui de l'axe des ordonnées, la variable qui contient l'ensemble des données du graphique, l'orientation du graphique, un booléen pour activer la légende, un pour activer les info-bulles et un dernier correspondant aux URL.

Il est possible de créer plusieurs courbes sur un même graphique. La méthode `addXYSerie(String title)` permet d'ajouter une série¹². Le titre en paramètre est celui de la série à créer. Cependant, même si on ne souhaite créer qu'une courbe, il faut créer une série au préalable.

- ☞ Cette fonction n'est utilisable qu'avec les « Line charts ».
- ☞ Le nom de la série est important car il sera réutilisé dans la légende mais aussi pour ajouter et modifier des valeurs.

Pour ajouter des données à une courbe, on utilise la même fonction que pour les « Pie charts », c'est-à-dire `addData(...)`. La commande `dataLine.getSeries(title).add(Xvalue,Yvalue)` ajoute un point (X,Y) à la série dont le nom est en paramètre. De plus, c'est cette même commande qui permet de mettre à jour des données. Elle est utilisée dans `addData(...)` et `setData(...)`. En effet, une courbe a besoin d'avoir une liste de points pour être représentée. On ne met donc pas à jour des valeurs mais on en crée de nouvelles.

2. Créer un nouvel onglet

La classe `C_CustomDisplay` implémente `IDisplay`, une interface de `Repast` permettant de créer de nouveaux onglets. Toutes les méthodes de cette interface ne sont pas utilisées mais les plus importantes sont `getPanel()`, `render()` et `addRenderListener()` car ce sont elles qui permettent d'ajouter un contenu à l'onglet et de rafraîchir son affichage.

¹² Une série = une courbe (pour les « Line charts ») ou une donnée (pour les « Pie charts »).

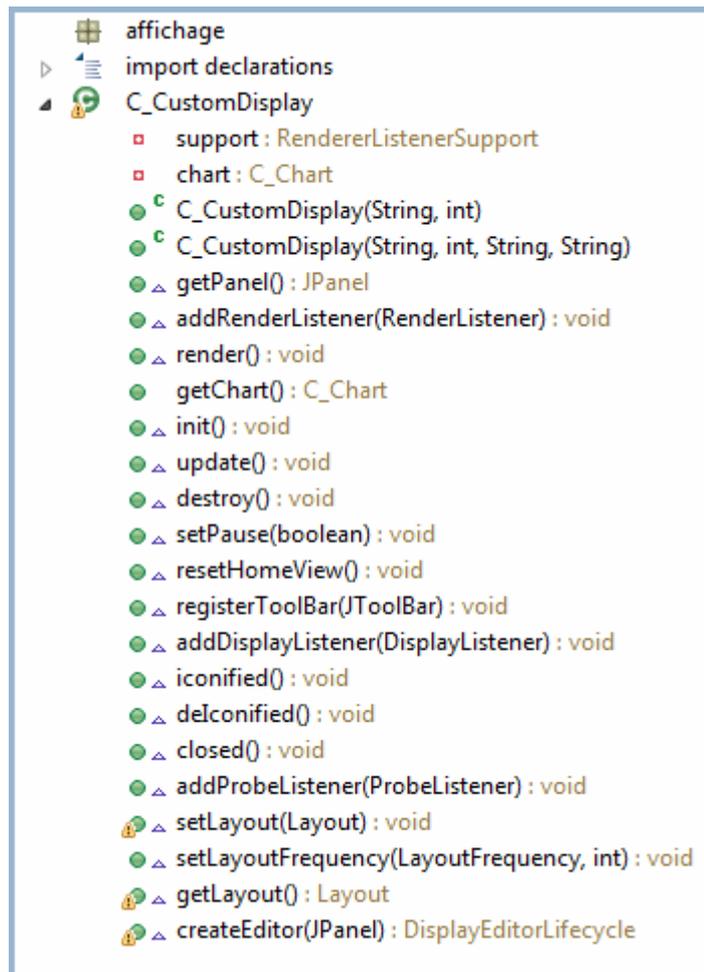


Figure 11 : Outline de C_CustomDisplay

Il existe deux constructeurs dans C_CustomDisplay dont les paramètres correspondent à ceux des deux constructeurs de C_Chart. Leur rôle est de créer une nouvelle instance de C_Chart.

La fonction `getPanel()` est la première à être appelée par Repast. Elle retourne le panneau du graphique créé précédemment.

Pour afficher le contenu de l'onglet dans la simulation, il faut utiliser un `RenderListenerSupport`¹³. Ensuite, pour qu'il soit « écouté » par le système, c'est-à-dire pris en compte, on lui ajoute un `RenderListener` en passant par la fonction `addRenderListener(RenderListener listener)`. Enfin, pour rafraîchir le display, on utilise la commande `support.fireRenderFinished(this)` dans `render()` qui est appelé par Repast à chaque tick. De ce fait, l'affichage sera mis à jour à chaque fois qu'il est susceptible d'avoir changé. Voici l'implémentations des deux méthodes exposées précédemment :

```
/** Support du RenderListener qui rafraichit ce display */
private RenderListenerSupport support;
```

¹³ Classe de Repast permettant d'afficher et de rafraîchir l'affichage du display auquel il est associé.

```

public void addRenderListener(RenderListener listener) {
    support.addListener(listener);
}

public void render() {
    support.fireRenderFinished(this);
}

```

3. Ajouter un onglet à la simulation et le mettre à jour

Pour ajouter un onglet à la simulation, on utilise la classe `SimMastoInitializer` qui implémente `ModelInitializer`. Ce dernier est une interface de `Repast` permettant d'initialiser certains éléments dans la simulation (comme des onglets dans notre cas) et de les réinitialiser correctement à la fin de chaque `run`. `SimMastoInitializer` implémente aussi `IAction` qui est une autre interface de `RS` qui permet de définir les actions (définies dans la méthode `execute()` implémentée par `IAction`) qui devront être répétées à l'intervalle de ticks demandé.

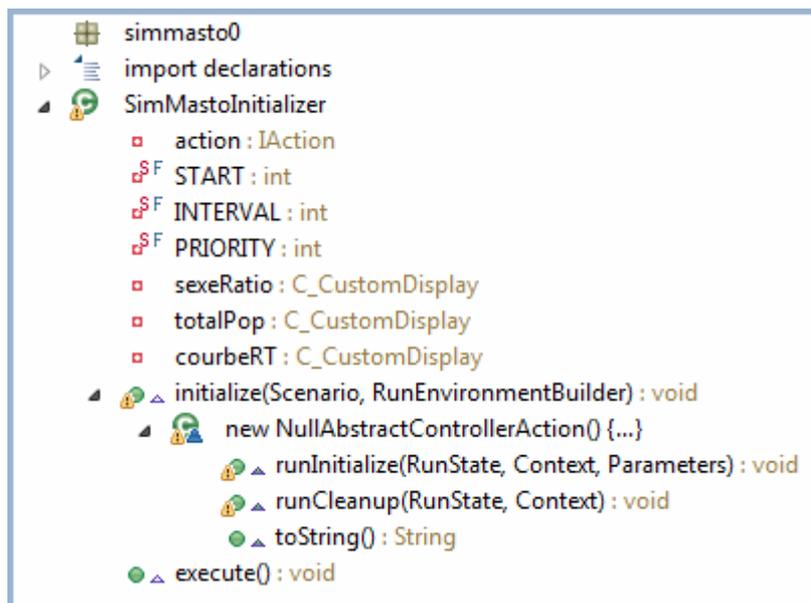


Figure 12 : Outline de `SimMastoInitializer`

Avant de pouvoir ajouter des onglets à la simulation, il faut créer une nouvelle instance de `NullAbstractControllerAction` dans la méthode `initialize(Scenario scen, RunEnvironmentBuilder builder)`:

```
scen.addMasterControllerAction(new NullAbstractControllerAction() {...});
```

Cette classe étend `AbstractControllerAction` qui implémente `ControllerAction`. Cette interface permet d'ajouter des « actions » à la simulation comme des onglets dans le cas de `SimMasto`. `NullAbstractControllerAction` est très pratique puisqu'elle n'oblige pas à réécrire toutes les méthodes de `ControllerAction` mais uniquement `runInitialize(...)`, `runCleanup(...)` et `toString()`.

Pour créer un onglet, on se place dans la méthode `runInitialize(...)` puis on commence par instancier un nouveau `C_CustomDisplay`. Ensuite, on l'ajoute au registre des displays avec la commande `runState.getGUIRegistry().addDisplay("Sexe ratio", GUIRegistryType.OTHER, sexeRatio)`. Le premier argument est le titre de l'onglet, le second est le type d'interface graphique créée (dans notre cas on n'utilise que `OTHER`) et enfin, le dernier est le nom du `C_CustomDisplay` instancié précédemment. En créant plusieurs onglets, on obtient le résultat suivant :

```

/** Ajoute des onglets à la simulation et les initialise */
public void runInitialize(RunState runState, Context context,
Parameters runParams) {
    // Initialisation des onglets //

    /* On crée un nouveau display puis on l'ajoute au registre des
GUI de RunState.
* On recommence ces deux étapes autant de fois que l'on
souhaite ajouter d'onglets. */

    sexeRatio = new C_CustomDisplay("Pie3D",C_Chart.PIE3D);
    runState.getGUIRegistry().addDisplay("Sexe ratio",
GUIRegistryType.OTHER, sexeRatio);

    totalPop = new C_CustomDisplay("Pop totale",C_Chart.LINE,
"Ticks","Taille population");
    runState.getGUIRegistry().addDisplay("Population totale",
GUIRegistryType.OTHER, totalPop);

    courbeRT = new C_CustomDisplay("Indicateurs",C_Chart.LINE,
"Ticks","Y");
    runState.getGUIRegistry().addDisplay("Indicateurs",
GUIRegistryType.OTHER, courbeRT);

    [...]
}

```

👉 Après les avoir initialisés, on utilise la fonction `getChart()` de `C_CustomDisplay` pour récupérer le graphique créé et lui ajouter des données.

Pour afficher ces nouveaux onglets dans la simulation et les mettre à jour, on ajoute la classe `SimMastolInitializer` au registre des plannings à la suite du code précédent avec la commande `runState.getScheduleRegistry().getModelSchedule().schedule(ScheduleParameters.createRepeating(START,INTERVAL,PRIORITY),action)`. Comme cette ligne est un peu longue, je vais la décomposer pour l'expliquer :

- `runState.getScheduleRegistry()` récupère le registre des plannings ;
- `getModelSchedule()` récupère le planning de la simulation ;
- La fonction `schedule(...)` ajoute une action au planning avec comme premier argument le « type » d'action, c'est-à-dire avec ou sans répétition, et en deuxième l'action à exécuter (ou la classe qui implémente `IAction`) ;

- Enfin, `ScheduleParameters.createRepeating(START, INTERVAL, PRIORITY)` est le premier paramètre de la fonction précédente. Il crée une répétition qui commence au tick indiqué par `START`, qui est répétée à chaque `INTERVAL` et dont la priorité est `PRIORITY`.

L'interface `IAction` ne contient qu'une seule méthode appelée `execute()`. C'est elle qui sera appelée dans les conditions décrites précédemment pour mettre à jour les données des graphiques créés dans `runInitialize(...)`.

IV. Le Style des agents

Le style des agents permet de modifier leur représentation graphique. Repast permet d'associer une icône à une classe, à la souris, via son interface graphique. Cependant, un agent `C_Rodent` peut être un mâle ou une femelle, et l'image ne doit pas être la même. De même, on souhaite modifier la représentation d'un individu en fonction de sa maturité. Le cahier des charges spécifiait donc de pouvoir effectuer ces changements par la programmation, puis de pouvoir représenter les agents sous forme d'images (gif, jpg) ou de symboles (ellipses, triangles...) en « switchant » une variable globale.

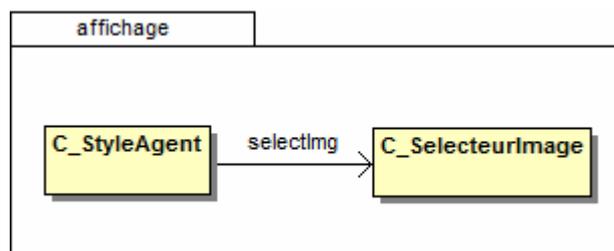


Figure 13 : Diagramme de classes de la gestion du style des agents

La classe `C_SelecteurImage` est un utilitaire permettant de charger des images, de renvoyer le nom de l'image ou la couleur correspondant aux caractéristiques de l'agent en paramètre. `C_StyleAgent` implémente `StyleOGL2D<Object>` qui permet d'associer une image ou une forme géométrique à un agent. C'est Repast qui instancie `C_StyleAgent` grâce à cette interface.

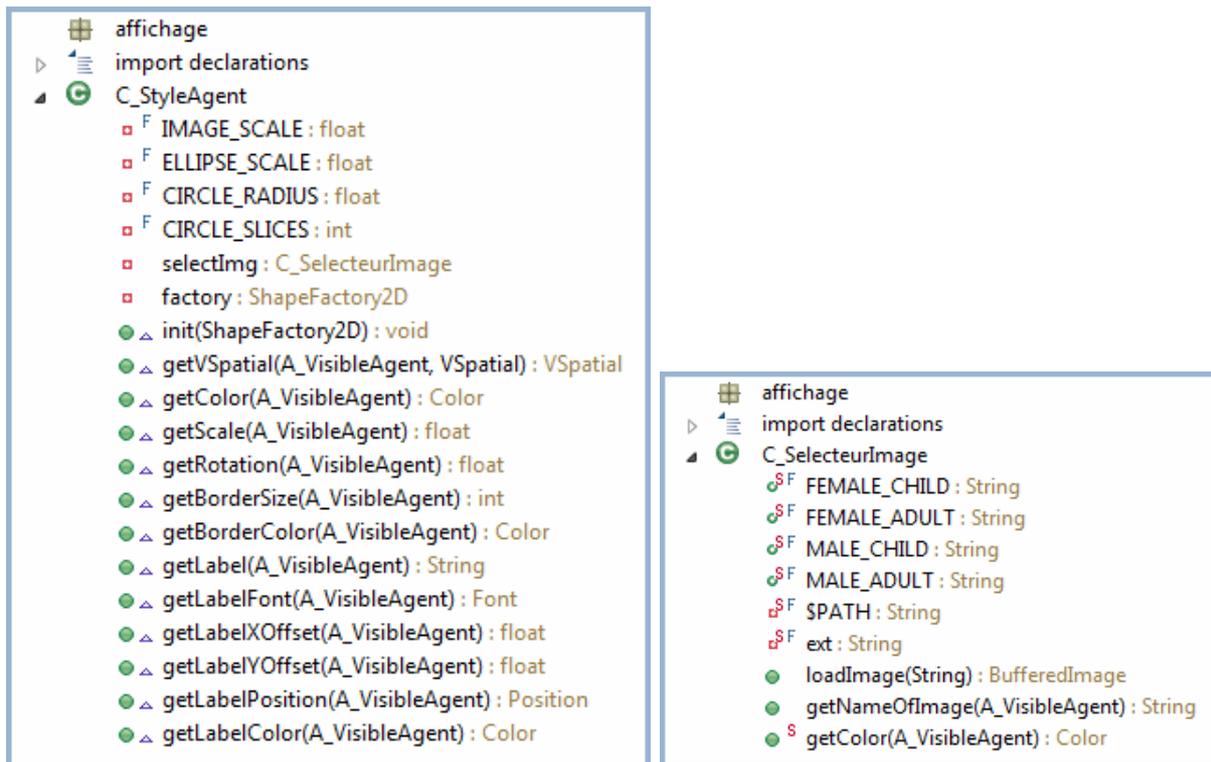


Figure 14 : Outlines de C_StyleAgent et C_SelecteurImage

Pour bien comprendre comment fonctionne C_StyleAgent, il faut commencer par se familiariser avec deux types de variables : le ShapeFactory2D et le VSpacial. Le ShapeFactory2D permet de charger et d'enregistrer toutes les images qui seront utiles à la simulation. Un VSpacial peut être une forme géométrique ou une image. Ces deux classes doivent être combinées pour pouvoir ajouter une représentation graphique à un agent. Un autre point important est de savoir que toutes les fonctions implémentées par StyleOGL2D sont appelées à chaque pas de temps. De ce fait, la gestion est déjà automatisée et il n'est donc pas nécessaire de créer une scheduleMethod. Enfin, toutes les fonctions implémentées ne sont pas réutilisées. Seules `init(...)`, `getVSpacial(...)`, `getColor(...)` et `getScale(...)` le sont.

1. Représentation par une image

Pour enregistrer les images dans le ShapeFactory2D, on utilise la fonction `factory.registerImage(nameOfImage, BufferedImage)` dans la méthode `init(ShapeFactory2D factory)` de C_StyleAgent. La BufferedImage en paramètre sera chargée par la méthode `loadImage(String nomImage)` de C_SelecteurImage. On obtient le résultat suivant :

```
/** Initialise un gestionnaire d'images et enregistre les images qui
seront utilisées au cours de la simulation
 * dans le factory */
public void init(ShapeFactory2D factory) {
    this.factory = factory;
    selectImg = new C_SelecteurImage();
}
```

```

    if(I_sim_constants.SELECT_AFFICHAGE.equals("image")) {
        factory.registerImage(C_SelecteurImage.FEMALE_CHILD,
selectImg.loadImage(C_SelecteurImage.FEMALE_CHILD));
        factory.registerImage(C_SelecteurImage.FEMALE_ADULT,
selectImg.loadImage(C_SelecteurImage.FEMALE_ADULT));
        factory.registerImage(C_SelecteurImage.MALE_CHILD,
selectImg.loadImage(C_SelecteurImage.MALE_CHILD));
        factory.registerImage(C_SelecteurImage.MALE_ADULT,
selectImg.loadImage(C_SelecteurImage.MALE_ADULT));
    }
}

```

Ensuite, pour initialiser l'image d'un agent ou la modifier, on se place dans la fonction `getVSpacial(A_VisibleAgent agent, VSpacial spatial)` de `C_StyleAgent` et on utilise la commande `factory.getNamedSpatial(String name)` pour récupérer l'image de type `VSpacial` qui correspond au nom en paramètre.

- ☞ Ce nom doit être un de ceux utilisé lors de l'enregistrement des images dans le `ShapeFactory2D`.
- ☞ Si l'on souhaite afficher des ellipses plutôt que des images, il n'est pas nécessaire d'enregistrer les images.

Les conditions de changement d'image d'un agent sont définies dans la fonction `haveToChange()` de `C_Rodent`. Si une des conditions est vérifiée, la fonction renvoie « vrai » et on choisit une nouvelle représentation pour notre agent.

- ☞ Le passage de la version 1.2 à la 2.0 de RS ne permet plus de gérer des images de type GIF (la transparence ne fonctionne plus).

2. Représentation par une ellipse

Pour créer une ellipse, on utilise la commande `factory.createCircle(float radius, int slices)` (toujours dans `getVSpacial(...)`). Le premier paramètre correspond au rayon du cercle à tracer et le deuxième au nombre d'arêtes qui le composent :

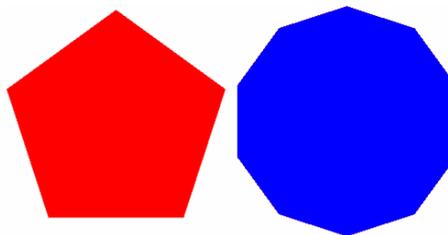


Image 8 : Création de « cercles » avec la commande `createCircle(...)`

En rouge, on a créé un cercle avec 5 slices alors qu'en bleu on en a utilisé 10. On peut voir qu'il ne s'agit pas de cercles parfaitement ronds mais celui en bleu donne un rendu suffisant.

- Plus on augmente le nombre de slices, plus le modèle aura besoin de ressources pour afficher les ellipses. Il est conseillé de ne pas aller au-delà de 12 slices.

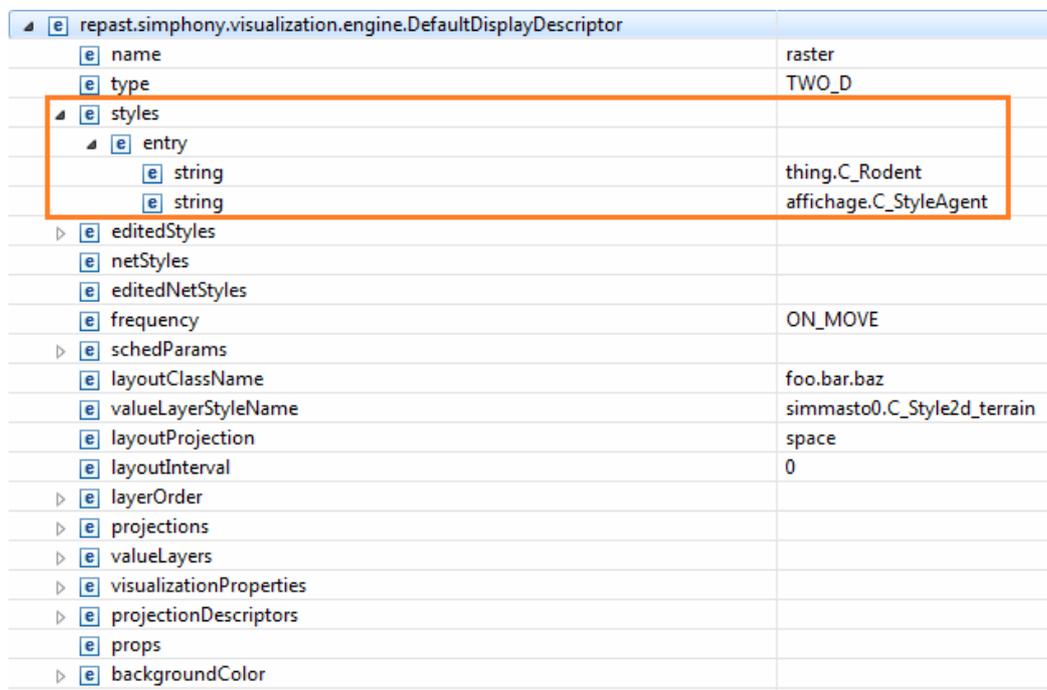
L'affectation et la mise à jour de la couleur se fait dans `getColor(A_VisibleAgent agent)` de `C_StyleAgent` :

```
public Color getColor(A_VisibleAgent agent) {  
    if(I_sim_constants.SELECT_AFFICHAGE.equals("image"))  
        return Color.white; // La couleur n'a pas d'importance  
    else return C_SelecteurImage.getColor(agent);  
}
```

La fonction `getColor(...)` de `C_SelecteurImage` retourne la couleur de l'agent en fonction de son sexe et de son âge, mais d'autres critères pourront intervenir dans l'avenir.

3. Utiliser le nouveau style dans la simulation

Pour attribuer un style à un agent dans un display déjà existant, il suffit de modifier une ligne dans le document « `repast.simphony.action.display_num.xml` » correspondant au display en question (dans le dossier `.rs`). Tout d'abord, il faut aller dans [e] `styles` puis [e] `entry`. Le premier élément [e] `string` correspond au `path package.class` de l'agent intégré au display et le second correspond au style qui lui est attribué. Il faut donc modifier ce dernier et y inscrire le `path package.class` qui correspond à la classe utilisée pour définir le style des agents.



| | |
|---|-----------------------------|
| repast.simphony.visualization.engine.DefaultDisplayDescriptor | |
| name | raster |
| type | TWO_D |
| styles | |
| entry | |
| string | thing.C_Rodent |
| string | affichage.C_StyleAgent |
| editedStyles | |
| netStyles | |
| editedNetStyles | |
| frequency | ON_MOVE |
| schedParams | |
| layoutClassName | foo.bar.baz |
| valueLayerStyleName | simmasto0.C_Style2d_terrain |
| layoutProjection | space |
| layoutInterval | 0 |
| layerOrder | |
| projections | |
| valueLayers | |
| visualizationProperties | |
| projectionDescriptors | |
| props | |
| backgroundColor | |

Image 9 : `repast.simphony.action.display_num.xml`

V. Le batch

Le batch permet de démarrer une simulation en arrière-plan à partir de paramètres définis au préalable et sans avoir la partie graphique du modèle. Pour le projet SimMasto, il est essentiel de pouvoir lancer des batchs afin de faire des calculs de sensibilité plus rapidement. En effet, l'interface graphique consomme beaucoup d'énergie et augmente les temps de calcul.

1. Lancer un batch sous Eclipse

Quand on fait un nouveau projet Repast, ce dernier crée un lanceur de modèle « normal » et un pour le batch. En lançant le batch, une fenêtre s'ouvre pour que l'utilisateur spécifie le fichier qui contient les paramètres de la simulation que l'on souhaite utiliser. Un fichier de ce type est automatiquement créé quand on ajoute des paramètres au *context.xml* (voir « Présentation de Repast Symphony »). Son nom est, par défaut, « batch_params.xml ». Ce fichier est essentiel pour lancer une simulation en batch, avec ou sans Eclipse.

```
<?xml version="1.0"?>
<sweep runs="1">
<parameter name="AGENT_SPEED_M_BY_MN" type="constant" constant_type=
"number" value="1.0"/>
[...]
</sweep>
```

Extrait de batch_params.xml

2. Lancer un batch hors Eclipse

Pour lancer une simulation hors Eclipse, il faut créer un script qui appelle le projet. Le mieux est de créer un JAR (une archive Java) qui contient tout le projet ainsi que les librairies dont il a besoin. Cependant, il n'a pas été possible d'en faire un avec SimMasto. En effet, je n'ai pas réussi à imposer un *main* au classpath¹⁴ ce qui rend le lancement du JAR impossible. La solution a donc été de créer un script qui appelle toutes les classes et sources dont Repast a besoin pour lancer une simulation.

¹⁴ Paramètre de la machine virtuelle java qui définit le chemin d'accès au répertoire où se trouvent les classes et les packages Java dont a besoin un projet pour fonctionner.

Comme le projet peut être utilisé sous Windows et sous Linux, il a fallu créer un script en Dos et un en Shell. Le deuxième n'a été qu'une adaptation de langage. Je vais donc présenter un seul de ces scripts, celui en Shell.

On commence par créer un fichier *ScriptSimmasto.sh* qui charge toutes les classes dont a besoin Repast pour démarrer, puis celles dont le modèle a besoin pour fonctionner. Ensuite, on utilise une ligne de commande qui demande à Java de lancer la simulation suivant les paramètres (*main*, emplacement du dossier .rs...) qu'on lui a spécifiés.

Pour faciliter l'appel des différentes bibliothèques et classes, on initialise plusieurs variables qui font références à l'emplacement de certaines choses : Tout d'abord, on ajoute une variable pour le *path*¹⁵ de Repast, là où sont rangées toutes les bibliothèques Repast, et une pour le *path* du projet, là où sont rangés les binaires, le dossier .rs et d'autres dossiers indispensables au lancement de SimMasto. Il faudra aussi spécifier le *path* du fichier *batch_params.xml*.

- ☞ En Dos, on peut ajouter une variable pour le *path* de Java. Cette variable n'est pas nécessaire en Shell puisqu'il comprend l'appel de la commande « Java ».

Ensuite, on crée une variable CP qui contiendra toutes les classes et sources dont Java a besoin pour lancer le modèle. Elle doit aussi contenir les bibliothèques externes utilisées dans le projet, ainsi que le dossier *bin* (le détail commenté de *ScriptSimmasto.sh* est disponible en annexe).

3. Utiliser le multi-run

Le multi-run peut être interprété de deux manières : on peut faire plusieurs *runs* avec les mêmes paramètres ou faire un *run* par lot de paramètres différents. Dans le premier cas, la marche à suivre est simple, il suffit de modifier le paramètre *sweep* du fichier *batch_params.xml*. Ce paramètre correspond au nombre de *runs* à lancer avec ce fichier. Pour le deuxième cas, il faut créer un nouveau script, appelé *SimmastoMulti.sh* (*SimmastoMulti.bat* disponible en annexe) qui appelle *ScriptSimmasto.sh* et noter en paramètre le nombre de fois que l'on veut le lancer.

¹⁵ Signifie "chemin" ou "emplacement".

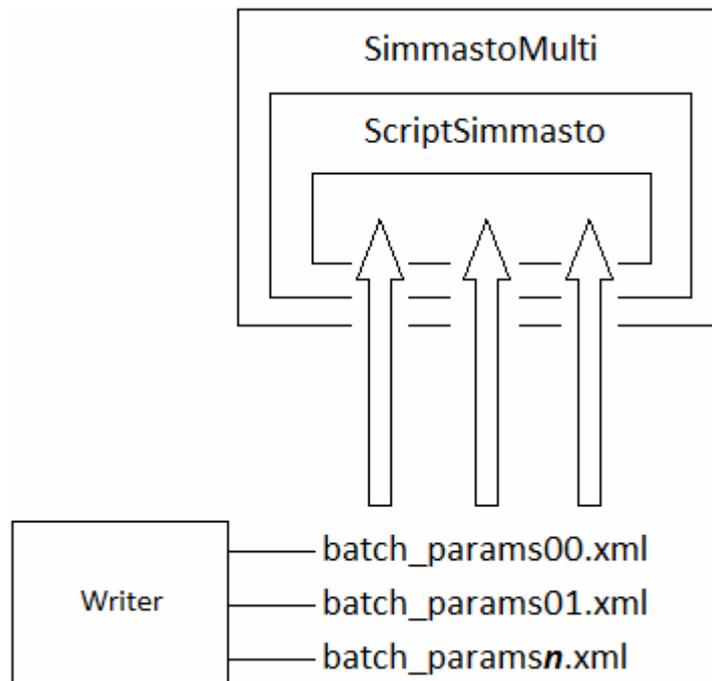


Image 10 : Processus de lancement des scripts et de création des fichiers XML

SimmastoMulti.sh contient une boucle qui appelle n fois *ScriptSimmasto.sh*. Pour pouvoir lancer plusieurs *runs* avec des paramètres différents pour chacun, il faut créer autant de fichiers *batch_params.xml* que de *runs* à lancer. Il serait fastidieux de créer ces fichiers à la main, surtout s'il on veut en créer une grande quantité. Pour cela, j'ai créé un programme Java appelé *Writer* qui écrit ces fichiers dans le dossier *batch* du projet *SimMasto* en leur ajoutant un indice à la fin. De ce fait, les fichiers ne s'écrasent pas. De plus, le numéro du fichier et celui du *run* à lancer sont liés. Par exemple, le 5^e *run* utilisera le fichier *batch_params5.xml* ce qui est pratique pour savoir avec quels paramètres la simulation a été lancée. Par la suite, J-E Longueville a amélioré ce programme pour pouvoir faire évoluer la valeur des paramètres sur un intervalle. Par exemple le paramètre p vaudra 5 pour le premier fichier puis 10 pour le deuxième, puis 15 pour le troisième, etc.

La démarche des scripts est la suivante : *SimmastoMulti.sh* appelle n fois *ScriptSimmasto.sh* qui lui-même utilise un fichier *batch_paramsn.xml* différent pour chaque *run* et dont l'indice est spécifié en paramètre par *SimmastoMulti.sh* (l'indice correspond au numéro du *run* courant).

Conclusion

Ce stage en entreprise fut une excellente expérience. J'ai appris un nouveau langage de programmation, Repast, et aussi à utiliser une librairie permettant de créer des graphiques, JfreeChart. De plus, j'ai appris à utiliser le SVN et le cluster du CBGP. Le développement du module présentation est très diversifié. Les différentes tâches prévues ont été réalisées en fonction de leur priorité. D'autres développements auraient pu être réalisés comme le filtrage d'informations dans la console du tableau de bord, changer les caractéristiques du sol en fonction des saisons, ou encore modifier les informations données par la « carte d'identité » des agents (disponible lorsque l'on clique sur l'un d'entre eux).

Cette expérience m'a permis de travailler au sein d'une équipe de biologistes-modélisateurs ce qui était très intéressant puisque j'ai aussi pu apprendre des petites choses en génétique. J'espère que le travail que j'ai réalisé apportera un vrai « + » au projet SimMasto et que J. Le Fur arrivera au terme de ce vaste projet.

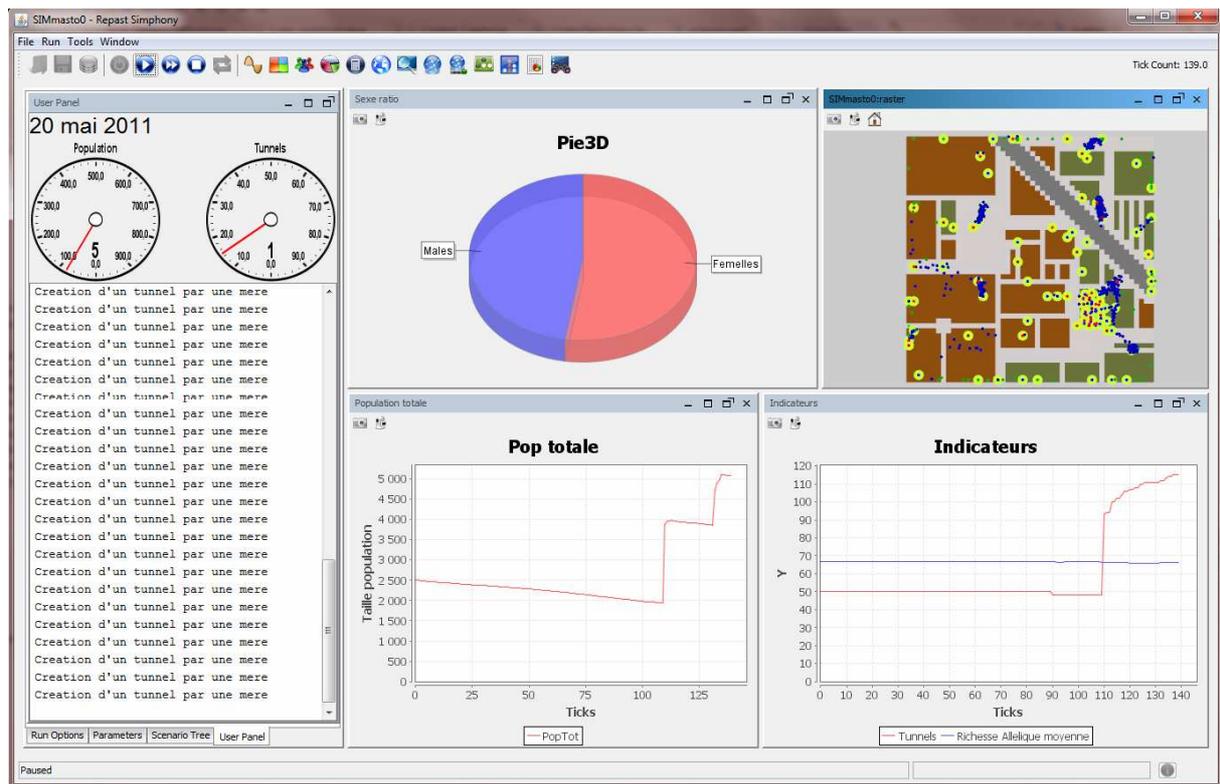


Image 11 : Interface graphique de SimMasto avec un aperçu de mes réalisations graphiques

Bibliographie

- Documentation sur l'API fournie avec Repast
- « Chaînes de traitement et procédures pour l'exploitation de données spatialisées par un simulateur multi-agents » (Module 8 : Mise en place d'une simulation RS), Q. Baduel, J. LeFur, S.Piry.
- Polycopié sur l'utilisation du Cluster (fourni par le CBGP)
- Documentation sur l'installation des logiciels utiles à l'utilisation du cluster à partir de sa machine personnelle : <http://gaia.supagro.inra.fr/wikis/cluster>
- Documentation sur JFreeChart : <http://www.java2s.com/Open-Source/Java-Document/Chart/jfreechart/Catalogjfreechart.htm>
- Le forum sur Repast : <http://old.nabble.com/Repast-f3965.html>
- Documentation sur la conversion d'une date en chaînes de caractères : http://java.developpez.com/faq/java/?page=langage_chaine#LANGAGE_STRING_conversion_string
- Documentation sur StyleOpenGL2D (Repast) : <http://old.nabble.com/Display-Problem-in-Repast-2.0-ts30995650.html#a31004655>
- Documentation sur le batch : <http://www.hotline-pc.org/batch.htm>
- Documentation sur la gestion des dates en Java : http://java.developpez.com/faq/java/?page=langage_date

Annexe

ScriptSimmasto.sh

```
#!/bin/bash

# The version of Repast Simphony being used.
VERSION=2.0.0
# The plugins path of Eclipse (for Repast).
PLUGINS=$HOME/eclipse/plugins
# The name of the model. This might be case-sensitive.
MODELNAME=SIMmasto_0
# The folder of the model. This might be case-sensitive.
MODELFOLDER=$HOME/workspace/$MODELNAME
# The file containing the batch parameters.
BATCHPARAMS=$MODELFOLDER/batch/batch_params$num.xml # $num est un paramètre
de SimmastoMulti
# The repast.simphony.runtime librairie
RUNTIME_LIB=$PLUGINS/repast.simphony.runtime_$VERSION/lib

#### Define the Core Repast Simphony Directories and JARs ####

# Endroit où se trouve la classe principale pour le lancement du batch,
c'est-à-dire BatchMain.class
CP=$PLUGINS/repast.simphony.batch_$VERSION/bin
# Librairies permettant de charger Repast
CP=$CP:$RUNTIME_LIB/saf.core.runtime.jar
CP=$CP:$RUNTIME_LIB/commons-logging-1.0.4.jar
CP=$CP:$RUNTIME_LIB/groovy-all-1.7.5.jar
CP=$CP:$RUNTIME_LIB/javassist-3.7.0.GA.jar
CP=$CP:$RUNTIME_LIB/jpf.jar
CP=$CP:$RUNTIME_LIB/jpf-boot.jar
CP=$CP:$RUNTIME_LIB/log4j-1.2.13.jar
CP=$CP:$RUNTIME_LIB/xpp3_min-1.1.4c.jar
CP=$CP:$RUNTIME_LIB/xstream-1.3.jar
CP=$CP:$RUNTIME_LIB/commons-cli-1.0.jar
# Sources et les classes permettant de charger les paramètres du batch
# + classes utiles au projet comme Context_Utils qui permet de connaître le
tick courant de la simulation
CP=$CP:$PLUGINS/repast.simphony.core_$VERSION/lib/*
CP=$CP:$PLUGINS/repast.simphony.core_$VERSION/bin
# Contient l'essentiel des classes Repast utilisées dans les projets de ce
type
CP=$CP:$PLUGINS/repast.simphony.bin_and_src_$VERSION/*

### Classes et librairies que Java ne parvient pas à trouver seul ###

# Classe de JConsole pour dévier la console vers le tableau de bord
CP=$CP:$PLUGINS/libs.bsf_2.0.0/lib/bsh-2.0b4.jar
# Librairies utilisées dans le projet
CP=$CP:$PLUGINS/repast.simphony.essentials_2.0.0/lib/*
# Path de la librairie externe JfreeChart
CP=$CP:$MODELFOLDER/lib/jfreechart-1.0.13/*
# Fichiers binaires
CP=$CP:$MODELFOLDER/bin
```

```
# Execute in batch mode.
#Commande : Java [args] -cp <classpath> main -params <batchParams>
<model>.rs
java -Xss10M -Xmx400M -cp $CP repast.simphony.batch.BatchMain -params
$BATCHPARAMS $MODELFOLDER/$MODELNAME.rs

echo fin script avec XML $BATCHPARAMS
```

SimmasttoMulti.bat

```
@echo off

rem Run en cours
set RUN=0
rem Nombre de runs à lancer (écrit en paramètre)
set nbRuns=$1
rem Script de lancement de SimMasto
set batch="ScriptSimmasto.bat"

:boucle
call %batch% %RUN%
set /a RUN = %RUN%+1
IF not "%RUN%"=="%nbRuns%" goto boucle
```

Résumé

Ce document présente le module présentation du projet SimMasto réalisé avec la plateforme de programmation Repast Symphony. Ce projet consiste à créer une simulation de rongeurs. Le module présentation permet de récupérer des informations dans le système sans altérer son fonctionnement pour ensuite suivre visuellement l'évolution de certains indicateurs, à partir desquels pourront être réalisés des graphiques. Ils seront aussi envoyés dans des fichiers de sortie pour être analysés.

This document introduce the presentation module of the SimMasto project, realized with the development platform called Repast Symphony. The project consist in creating a simulation of rodents. The presentation module allow to get back information in the system without distorting its functioning to follow visually the evolution of some indicators, from which grphs can be realized. They will also be sent to outputs to be analyzed.