**Java** **agent** **development Framework**

MANUEL D’UTILISATION  
Jules SAUVINET  
Yann CORTIAL



**TABLE DES MATIERES**

[I. Introduction 3](#_Toc452240641)

[II. L’initialisation d’un programme JADE 4](#_Toc452240642)

[III. Jade et les comportements 6](#_Toc452240643)

[IV-1. Les Behaviours simples 6](#_Toc452240644)

[IV-1. Les Behaviours complexes 7](#_Toc452240645)

[IV-1.A Les Behaviours planifiés 7](#_Toc452240646)

[IV. La communication entre les agents 8](#_Toc452240648)

[V. Une simulation de la vie d’un village avec JADE (simplifiée) 9](#_Toc452240649)

[V-1. Présentation de la simulation 9](#_Toc452240650)

[V-2. La description de l’environnement 10](#_Toc452240651)

[V-3. La description des agents villageois 11](#_Toc452240652)

[VI. Conclusion 15](#_Toc452240653)

[VII. Sources 15](#_Toc452240654)

1. Introduction

*JADE* (Java Agent DEvelopment) est un Framework/API basé sur le langage JAVA permettant le développement de systèmes multi-agents (SMA) sur de multiples plateformes, fixes comme mobiles.

On rappellera qu’un système multi-agents est composé d'un ensemble d'[agents](https://fr.wikipedia.org/wiki/Agent_%28informatique%29) situés dans un [environnement](https://fr.wikipedia.org/wiki/Environnement) et [interagissant](https://fr.wikipedia.org/wiki/Interaction_%28sciences_sociales%29) selon des relations prédéfinies. Un agent est une entité caractérisée par le fait qu'elle possède un certain degré d’[autonomie](https://fr.wiktionary.org/wiki/autonomie). Ainsi, un agent peut être caractérisé par un [processus](https://fr.wikipedia.org/wiki/Processus_informatique), un [robot](https://fr.wikipedia.org/wiki/Robot) ou encore un être humain. Les systèmes multi-agents ont comme objectif de créer une intelligence collective afin de faire émerger des structures autonomes et reproductives par interactions.

L’API *JADE* permet par conséquent à travers l’utilisation du paradigme multi-agents :

* la simulation de phénomènes complexes
* la résolution de problèmes
* la conception de programmes

*JADE* est un Projet Open Source, sous License LGPL contrôlée par Telecom Italia Lab, qui reste propriétaire du projet open-source. La plateforme est le résultat des efforts conjoints de différents acteurs réunis au sein du JADE Board (fondé en 2003) dont les missions sont la promotion, la gouvernance et l’implémentation des évolutions de JADE.

*JADE* offre en particulier un support avancé de la norme FIPA-ACL, ainsi que des outils de validation syntaxique des messages entre agents basé sur les ontologies. Les agents créés communiquent via le langage *Agent Communication Language* ou ACL.

De nombreux outils sont mis à la disposition du développeur pour l’aiguiller dans la création du système multi-agents et la communication entre les agents. *JADE* possède notamment une interface graphique permettant au développeur de pouvoir créer des agents issues de classes JAVA.

1. L’initialisation d’un programme JADE

**Pour lancer la plateforme JADE, il est nécessaire de paramétrer les conteneurs dans lesquelles les agents seront contenus.**

**Il faut créer au moins deux conteneurs. Le premier est le « main container ». Le deuxième contiendra une liste des agents qui seront instanciés au moment de leur insertion dans le conteneur.**

**Une manière de procéder pratique pour initialiser les conteneurs et les agents et de créer une classe JAVA qui fera office de « lanceur ».**

**La création du « main container » ressemblera à cela :**

Runtime rt = Runtime.*instance*();  
 Profile p = **null**;  
 **try** {  
 p = **new** ProfileImpl(*PROPERTIES\_FILE*);  
 AgentContainer container = rt.createMainContainer(p);  
 }  
 **catch** (Exception e)  
 {  
 System.***out***.println(e.getMessage());  
 }

Tout d’abord, il faut créer une instance d’exécution. Puis il faut créer le conteneur principal :

Runtime rt = Runtime.*instance*();

Il faut ensuite créer un profil d’implémentation de manière préférable à partir d’un fichier de propriétés contenant les options du lancement de la plateforme JADE:

Profile p = **null**;  
 **try** {  
 p = **new** ProfileImpl(*PROPERTIES\_FILE*);

Le fichier de propriété sera de cette forme :  
 option 1 = <valeur 1>  
 option 2 = <valeur 2>  
 …

Par exemple  le fichier pourra ressemblera à cela:  
 **main**=**true  
 gui**=**true  
 platform-id**=**Village  
 local-port**=**1099**

Il est nécessaire de spécifier si le programme lancera le main conteneur ou non (par défaut oui), si le programme lancera l’interface graphique de gestion des agents (par défaut non), le nom de la plateforme et le port par lequel les agents vont communiquer.

La création des autres conteneurs et des agents se fera par exemple de cette manière :

Runtime rt = Runtime.*instance*();  
 Profile p = **null**;  
 **try** {  
 p = **new** ProfileImpl(*PROPERTIES\_FILE*);  
 ContainerController container=rt.createAgentContainer(p);  
 AgentController app = container.createNewAgent(**"appAgent"**,  
 **"mif24.jadedemo.agent.ApplicationAgent"**, **null**);  
 app.start();  
  
 }  
 **catch** (Exception e)  
 {  
 System.***out***.println(e.getMessage());  
 }

Il faut créer un contrôleur d’agent issu de la classe “AgentController” de JADE en spécifiant le nom de l’agent et l’emplacement de la classe JAVA où l’agent y est décrit, puis une liste d’argument à passer à l’agent (optionnel, ici null).   
 Puis le lancement de l’agent se fait à l’aide de la méthode « start » du contrôleur. La méthode « start » fait appel à la méthode d’initialisation d’un agent « setup ». Chaque classe agent héritant de la classe Agent de JADE doit implémenter cette méthode. Les arguments transmis à l’agent à l’aide de la méthode « createNewAgent » pourront y être récupérés à l’aide de la méthode « getArguments » renvoyant un tableau d’objets.

1. Jade et les comportements

Afin de définir les actions d’un agent en fonction du temps et de son environnement, il est nécessaire d’attribuer des comportements aux agents JADE. Pour qu'un agent exécute une tâche associer à un comportement, il faut lui l'attribuer par la méthode **addBehaviour(Behaviour b)** de la classe **jade.core.Agent**.

Il existe plusieurs types de comportements pour les agents, tous issus de la classe **Behaviour** du Framework. Chacun a un moyen de déclenchement différent. Chaque Behaviour doit implémenter au moins les deux méthodes :

* **action :** qui désigne les opérations à exécuter par le Behaviour;
* **done :** qui exprime si le Behaviour a terminé son exécution ou pas.

Nous présenterons que des comportements « simples » et quelques comportements plus complexes mais nous nous arrêterons à une liste non exhaustive, i.e. les comportements que nous avons utilisés dans notre simulation.

## IV-1. Les Behaviours simples

JADE offre trois types de comportements simples :

* **One-shot Behaviour**

Un **One-shot Behaviour** est une instance de la classe **jade.core.behaviours.OneShotBehaviour**. Il a la particularité d'exécuter sa tâche une et une seule fois puis il se termine. La classe **OneShotBehaviour** implémente la méthode **done** qui retourner toujours **true**.

* **Cyclic Behaviour**

Un **Cyclic Behaviour** est une instance de la classe **jade.core.behaviours.CyclicBehaviour**. Comme son nom l'indique un **Cyclic Behaviour** exécute sa tâche d'une manière répétitive. La classe **CyclicBehaviour** implémente la méthode **done** qui retourne toujours **false**.

* **Generic Behaviour**

Un **Generic Behaviour** est une instance de la classe **jade.core.behaviours.Behaviour**. Le **Generic Behaviour** se situe entre le **One-shot Behaviour** et le **Cyclic Behaviour** du fait que l’implémentation de la méthode **done** laissée au programmeur.

Exemple d’un **Generic Behaviour** :

// l'ajout d'un generic behaviour

// le Behaviour s'arrête quand aléatoire reçoit la valeur 7

addBehaviour(new RandomBehaviour());

Des exemples d’implémentation des comportements seront donnés dans la partie VI portant sur l’exemple d’un programme JADE complet.

## IV-1. Les Behaviours complexes

### IV-1.A Les Behaviours planifiés

Pour planifier une tâche d'un agent JADE offre deux types de Behaviours :

* **Waker Behaviour**

Le **Waker Behaviour** est implémenté de façon à exécuter la méthode **onWake** après une période passée comme argument au constructeur. Cette période est exprimée en millisecondes. Le Behaviour prend fin juste après avoir exécuté la méthode **onWake**.

* **Ticker Behaviour**

Le **Ticker Behaviour** est implémenté pour qu'il exécute sa tâche périodiquement par la méthode **onTick**. La durée de la période est passée comme argument au constructeur.

Des comportements plus complexes existent. Le diagramme de classe suivant résume la hiérarchie des classes Behaviour de JADE :

## C:\Users\Mathilde\AppData\Local\Microsoft\Windows\INetCache\Content.Word\behaviour_diag.png

### 

1. La communication entre les agents

Un des atouts de JADE pour la réalisation de systèmes multi-agents est la capacité de communication des agents. Les agents communiquent entre eux grâce aux messages ***ACL***. Les détails internes sont masqués à l'utilisateur par JADE, et l'API propose des abstractions bien construites à celui-ci.

Chaque agent JADE est identifié par un AID attribué automatiquement par le système. Celui-ci contient un identifiant unique pour chaque agent, son nom et une description personnalisable.

Les messages sont asynchrones et s'accumulent dans une boîte aux lettres aisément consultable, filtrable par le type de message, le protocole du message, etc.

Le langage ***ACL*** est défini par la norme ***FIPA*** (standard internationnal pour l'interopérabilité des agents) et contient un certains nombre de champs, dont :

* **Sender** : émetteur du messages
* **Receivers** : une liste de destinataires
* **Performative** : type informel du message – code l'intention de l'émetteur (sémantique). On trouve ainsi par exemple : INFORM (pour informer), REQUEST (pour demander), PROPOSE, REFUSE\_PROPOSAL, etc.
* **Content**: contenu du message. Il s'agit d'une chaîne de caractères Java. A noter que pour notre projet nous avons souvent eu recours au ContentObject : ie., la possibilité d'attacher un objet java sérialisable en tant que contenu, mais cela ne fait pas partie du standard FIPA.
* **Language** : syntaxe utilisée dans Content. Permet de garantir que l'émetteur et le destinataire se comprennent.
* **Ontology** : vocabulaire des symboles du message et leur sémantique.
* **Protocol** : librement attribué par le développeur. Dans notre projet nous nous sommes appuyés dessus pour étiquetter nos messages.

**Exemple d’envoi d’un message ACL au sein d’un Behaviour**

**try**{  
 ACLMessage info = **new** ACLMessage(ACLMessage.***INFORM***);  
 info.addReceiver(**applicationAgentAID**);  
 info.setProtocol(**"LeaveHouse"**);  
 info.setContentObject(**descriptor**.getEntity().**housePos**);   
 **myAgent**.send(info);  
 System.***out***.println(getLocalName()+**": après ma mort, je lègue ma maison au village..."**);  
 } **catch** (Exception ex){  
 System.***err***.println(**"Erreur - impossible de mettre à jour la**

**maison après décès."**);  
 }

**Exemple de réception d’un message ACL au sein d’un Behaviour**

msg = **myAgent**.receive();  
 **if** (msg != **null**){  
 **try** {  
 **if** (msg.getProtocol().equals(**"kill"**))   
 Object object = (Object) msg.getContentObject();  
 …

1. Une simulation de la vie d’un village avec JADE (simplifiée)

## V-1. Présentation de la simulation

Nous avons eu l’idée de simuler la vie d’un village pour illustrer notre présentation du Framework et pour montrer l’approche que nous avons eu de la création de système multi-agents avec JADE.

Nous avons tout d’abord défini un environnement simpliste composé d’arbre, de baies sauvages, de maisons, de champs et d’une quantité d’eau contenu dans un puits situé au centre du village. Les villageois évolueront continuellement dans cet environnement, le modifiant par leurs actions. Les villageois seront décomposés en 3 catégories :

* Les enfants
* Les adultes
* Les personnes âgées

Chaque villageois est associé en terme d'implémentation à un agent JADE et décrit par la classe **mif24.jade.demo.agent.VillageAgent**, héritant de **jade.core.Agent**.

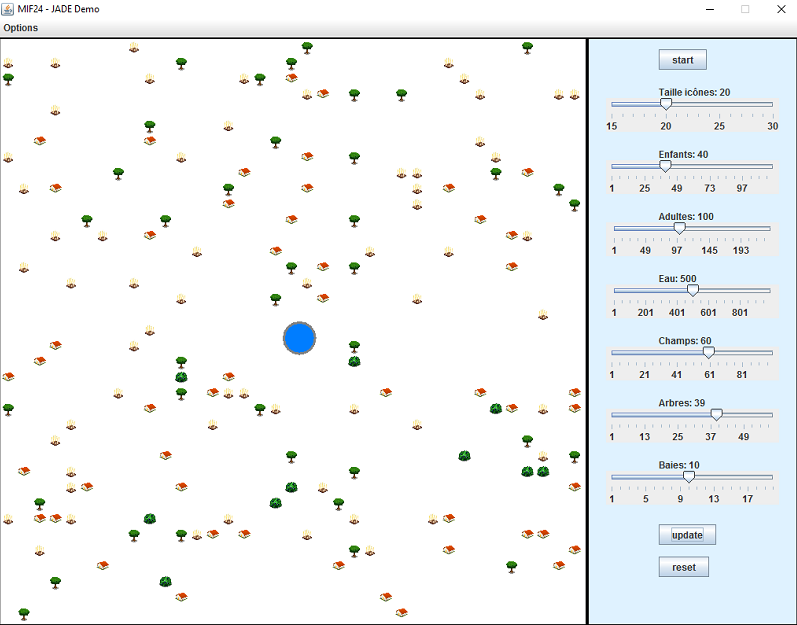
Pour réaliser la visualisation graphique de notre application nous avons eu besoin de créer un autre agent spécialisé et unique, décrit par la classe **mif24.jade.demo.agent.ApplicationAgent**. Il s'agit d'un agent coordinateur et centralisateur, en charge de la fenêtre graphique. Ceci est nécessaire pour éviter les accès concurrents aux ressources Swing de Java. Bien que l'existence de cet agent particulier soit dictée par les besoins techniques de la réalisation (ie. Réalisation d'une application JADE en mode Framework, où l'inversion de contrôle a lieu), l'agent ApplicationAgent est vite apparu pratique pour d'autres besoins : notamment, il permet de mettre en place un mécanisme de Blackboard. A noter aussi que l'ApplicationAgent a accès de manière privilégiée à l'Environnement du système multi-agents, c'est à dire l'état du Village, qu'il maintient en quelque sorte pour le bénéfice des autres agents.

En pratique donc, les agents villageois doivent notifier en continu l’agent application de leurs actions. L’agent application contient une liste des informations relatives à chaque villageois et une description de l’environnement qu’il met à jour. Cela lui permet de faire évoluer la fenêtre graphique.

Les actions des villageois seront associés à différents type comportements selon leur type.

## V-2. La description de l’environnement

L’environnement de départ du village est représenté sur la figure ci-dessous. Il est possible de régler les paramètres de départ (nombre d’arbres, de champs etc…).



Le village (l'environnement) possède un ensemble de propriétés que les villageois vont partager et modifier dans le temps. Parmi ces propriétés figure la quantité d’eau et de nourriture. La ponction des ressources par les villageois sera une fonction stochastique de la population du village. La raréfaction des ressources alimentaires (eau, nourriture) peut entraîner le décès d'un grand nombre de villageois.

Un villageois est potentiellement propriétaire d'une maison, et s'il meurt et qu'il était célibataire cette maison est léguée au village - pour être ensuite probablement attribuée à un villageois sans domicile (mécanisme de blackboard via l'agent application).

Le village est doté d'une politique. Plusieurs politiques pourront être appliquées dans le village en fonction du villageois Chef du village. Le chef du village évalue l'état de l'environnement afin de prendre des décisions : il décide notamment de la natalité (nombre d'enfants maximum par couple), de la construction de champs ou encore de maisons. Dans notre modèle, à la phase de conception, en fonction des caractéristiques du villageois chef, une politique différente devait être appliquée, entraînant une gestion des ressources différentes selon le chef courant, et l’élection d’un chef pouvait être décidée en fonction du bonheur du village - mais cela n'a pas été implémentée (par manque de temps) bien que cela soit aisément réalisable.

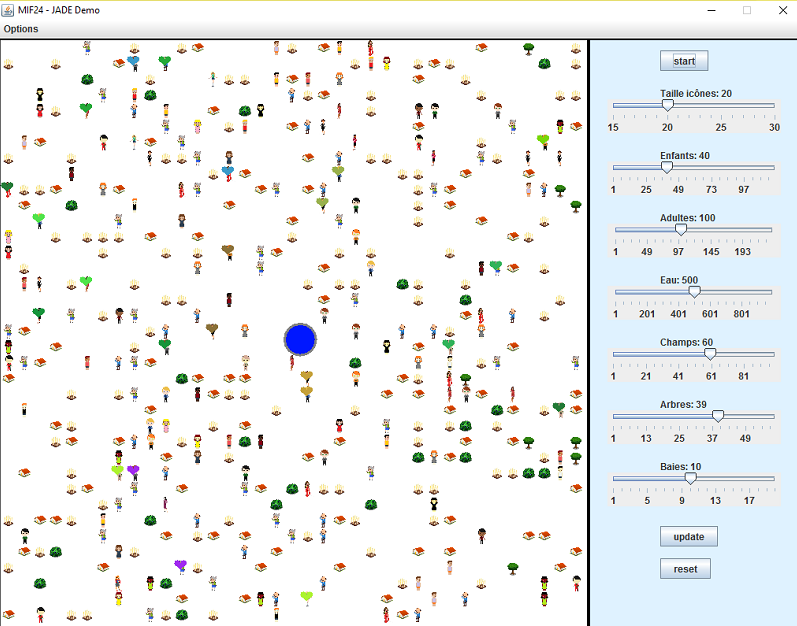
## V-3. La description des agents villageois

Un villageois est symbolisé par une liste de caractéristiques et sa position. Parmi ses caractéristiques, on retrouve : un identifiant, son âge, ses traits de comportements (intelligence, beauté, humour, confiance en soi, ambition), son statut social (argent), son métier, son statut marital (marié ou célibataire), l’identifiant de son conjoint s’il existe, la maison éventuellement détenue, l’identité de ses enfants s’ils existent, et son origine ethnique.

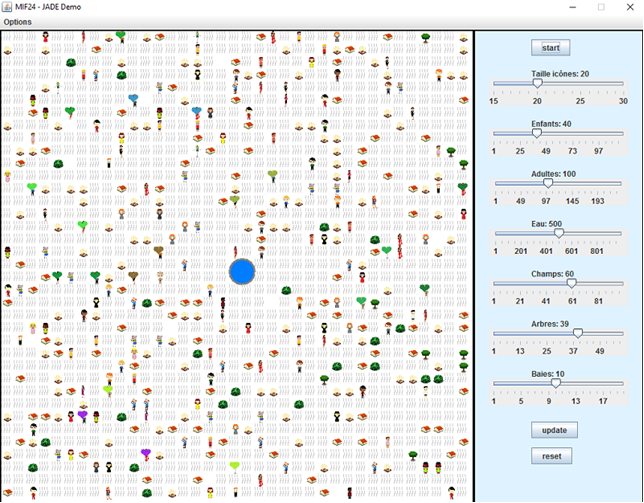
De nombreux comportements sont associés aux villageois afin de simuler certains comportements se rapprochant de ceux que l’on peut observer dans la vie réelle. Tous, à part le polling permanent de la boîte aux lettres des messages, sont implémentés sous forme de TickerBehaviour – ce qui permet de poser des repères de temps suffisamment larges (de l'ordre de quelques secondes) pour permettre la bonne visualisation du système.

* Vieillissement : un Ticker Behaviour est ajouté aux comportements de chaque villageois informant l’agent application du vieillissement du villageois. En fonction de l’âge du villageois, l’image lui étant associée dans la vue changera. De plus, le villageois aura une probabilité de plus en plus forte de mourir. L’application agent peut tuer un agent si la probabilité que celui-ci meurt est supérieure à un certain seuil.
* Recherche d’un conjoint : chaque agent recherche continuellement un conjoint. Celui-ci reçoit régulièrement la liste des villageois près de lui géographiquement. Lorsqu’un agent possède des caractéristiques lui correspondant, le villageois le contacte de manière stochastique afin de lui faire part de son intérêt. L’autre villageois évalue alors sa proposition et lui répond positivement ou négativement. Si les deux villageois se mettent d’accord, alors ils se marient et un nouveau couple est formé. Une villageoise qui se marrie lègue la maison qu'elle possédait éventuellement.
* Conception d’enfant : les villageois en couple peuvent concevoir un enfant. De manière stochastique et en fonction de leur caractéristique et de la politique du village, ils donnent naissance à un nouvel enfant, et en informe l’agent application qui créé un nouvel agent.
* Politique du Village : si le villageois est Chef du village alors il est doté d'un comportement de mise à jour de la politique. Il inspecte l'environnement (ie, la description quantifiée du village) et décide de la construction ou non de champs et de maisons, et de la natalité souhaitée.
* Travail : un villageois régulièrement a une probabilité de se mettre au travail. La probabilité dépend de la politique du village d'une part, et d'autre part de manière opposée, de ses caractéristiques personnelles (notamment l'égoïsme et l'ambition). Si le test probabiliste passe alors il demande une maison s'il n'en a pas (une maison libre peut lui être attribuée, ou si aucune disponible, peut être construite en abattant un arbre) ou construit un champ pour la communauté.
* Polling des Messages ACL : le villageois agent possède ce comportement (un CyclicBehaviour) qui passe en revue les messages qu'il reçoit, soit d'autres agents, soit de l'agent application. Divers types de messages peuvent être reçus : l'agent peut apprendre qu'il a été élu Chef, que son conjoint est décédé, qu'une maison lui a été attribuée, etc. On retrouve aussi tous les échanges « amoureux » dès lors que le villageois est engagé dans une procédure de flirt/mariage.

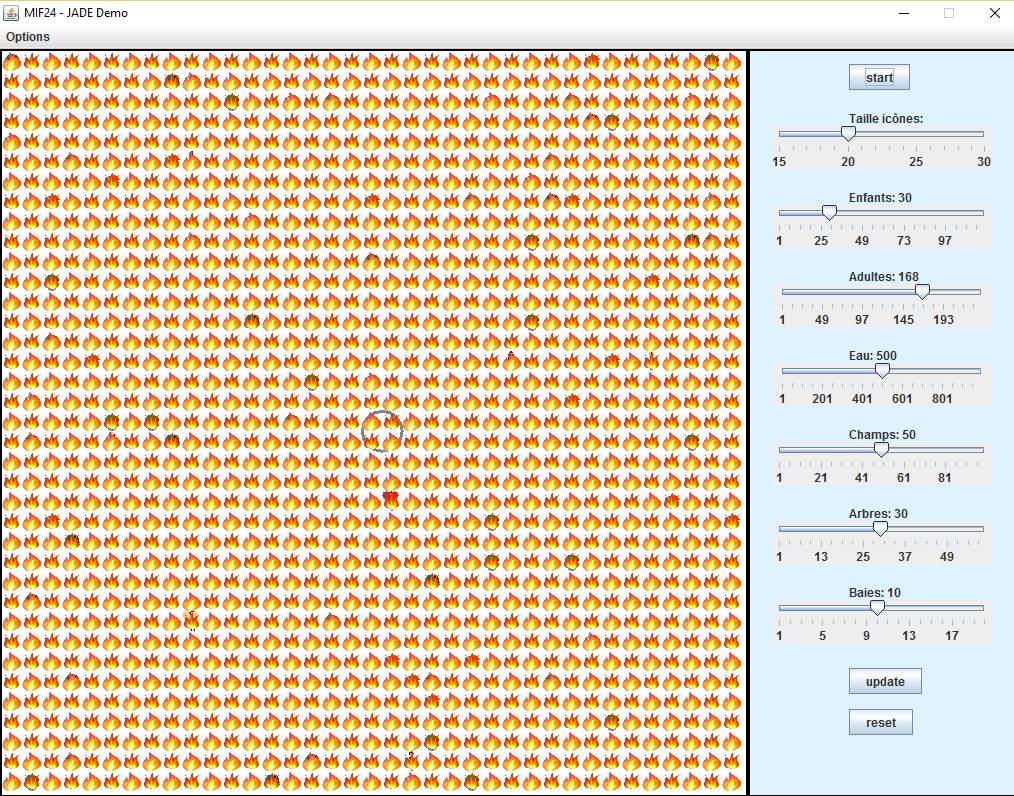
Ci-dessous est représenté l’état graphique du village après plusieurs ticks (le village vit depuis déjà un certain temps).   
 Les villageois sont représentés par des icônes aléatoires voulant symboliser la variété multiculturelle d’une société.   
 L’intensité de la couleur du bleu central symbolise la quantité d’eau disponible dans le village.   
 Deux villageois ayant un cœur greffé sur eux de la même couleur sont en couple.



La pluie permet de réapprovisionner le stock d’eau du village. C’est un paramètre stochastique. Plus les villageois sont égoïstes et consomment d’eau, plus ils mènent l’intérêt commun à sa perte.



Si le village ne possède plus d’eau, le village prend feu et les villageois meurt :



1. Conclusion

La simulation permet de manière simplifiée d’avoir un aperçu d’un comportement d’une part organisé pour la création de structures communes au système multi-agents qu’est le village et d’autre part la création d’un comportement général déstructuré où les agents sont égoïstes et agissent contre l’intérêt général.

1. Sources

* <http://www.emse.fr/~boissier/enseignement/maop12/courses/jade-prog-4pp.pdf>
* <http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf>
* <http://djug.developpez.com/java/jade/behaviours/>