Projet POO : Automate Cellulaire

BRENON Alexis / HOFSTETTER Johann

Date : 29 avril 2012

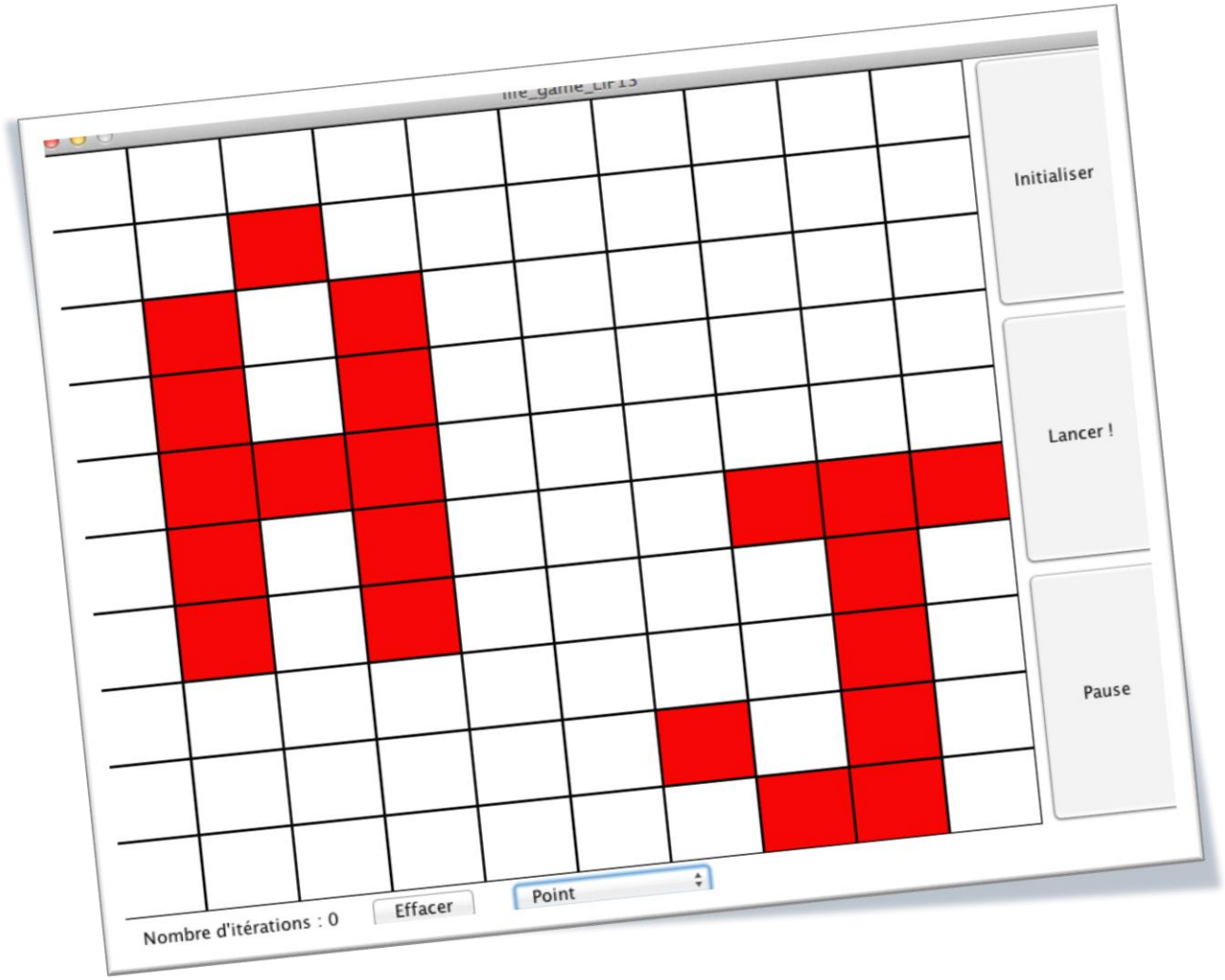


Table des matières

I. Présentation du logiciel

Dans le cadre de l'unité d'enseignement LIF13 « Programmation Orientée Objet », nous avons réalisé un automate cellulaire communément appelé jeux de la vie.
*« Le **jeu de la vie**, automate cellulaire imaginé par John Horton Conway en 1970, est probablement, à l'heure actuelle, le plus connu de tous les automates cellulaires.*
Malgré des règles très simples, le jeu de la vie permet le développement de motifs *extrêmement complexes. »* - Wikipédia

Ce jeu simule l'évolution de cellules dans un monde en deux dimensions théoriquement infini. L'état d'une cellule ainsi que de ses voisines déterminent l'état de la cellule lors de l'itération suivante. Ainsi une cellule morte à l'état n et possédant trois voisines vivantes sera vivante à l'état n+1 (elle nait). De même, une cellule vivante meurt si elle possède aucune, une ou plus de trois voisines.

L'interface permet à l'utilisateur d'interagir simplement avec l'application et la simulation à travers des boutons, ou des items de menu :

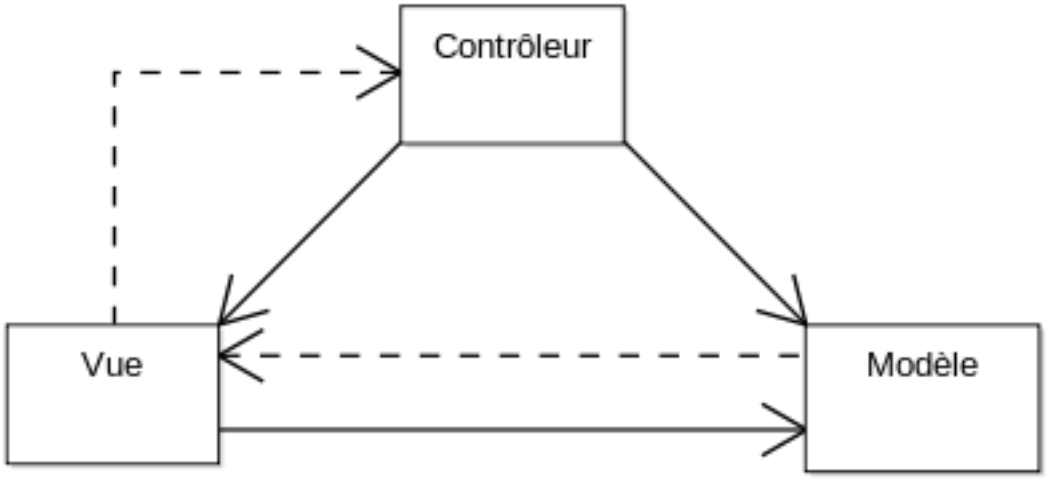
- « Initialiser » : Génération aléatoire d'un état initial.
- « Effacer » : Effacement de la grille.
- « Lancer » : Départ la simulation.
- « Pause » : Mise en pause.
- « Enregistrer Sous… » : Enregistrement de l'état courant.
- « Ouvrir » : Ouverture d'un état sauvegardé.
- « Quitter » : Fermeture de l'application.

De plus, une liste déroulante permet de sélectionner un motif cellulaire à ajouter sur la grille via la souris (une prévisualisation de la forme permet d'anticiper l'apparition des nouvelles cellules).
Enfin, lorsque le motif sélectionné est le point, un clic sur une cellule vivante permet de tuer celle-ci instantanément.

Le développement de l'application est axé sur le respect du modèle MVC. Ainsi, le projet est principalement découpé en 3 classes principales, représentant chacune une partie de ce modèle.

II. Modèle, Vue, Contrôleur (MVC)

Le MVC est un paradigme de programmation où les données manipulées sont indépendantes de la forme sous laquelle elles sont affichées.



Le contrôleur permet de faire la liaison entre la vue et le modèle. En effet le contrôleur reçoit les actions de l'utilisateur et les transmet à la vue et au modèle.
La vue correspond à l'interface utilisateur.
Le modèle traite les données en fonctions des informations données par le contrôleur. Une fois que le modèle a réalisé ces traitements, il notifie alors la vue qui se met à jour.
Dans notre cas, nous avons créé trois classes afin de respecter ce principe :

- « Modele »
- « FenetrePrincipale »
- « Controlleur »

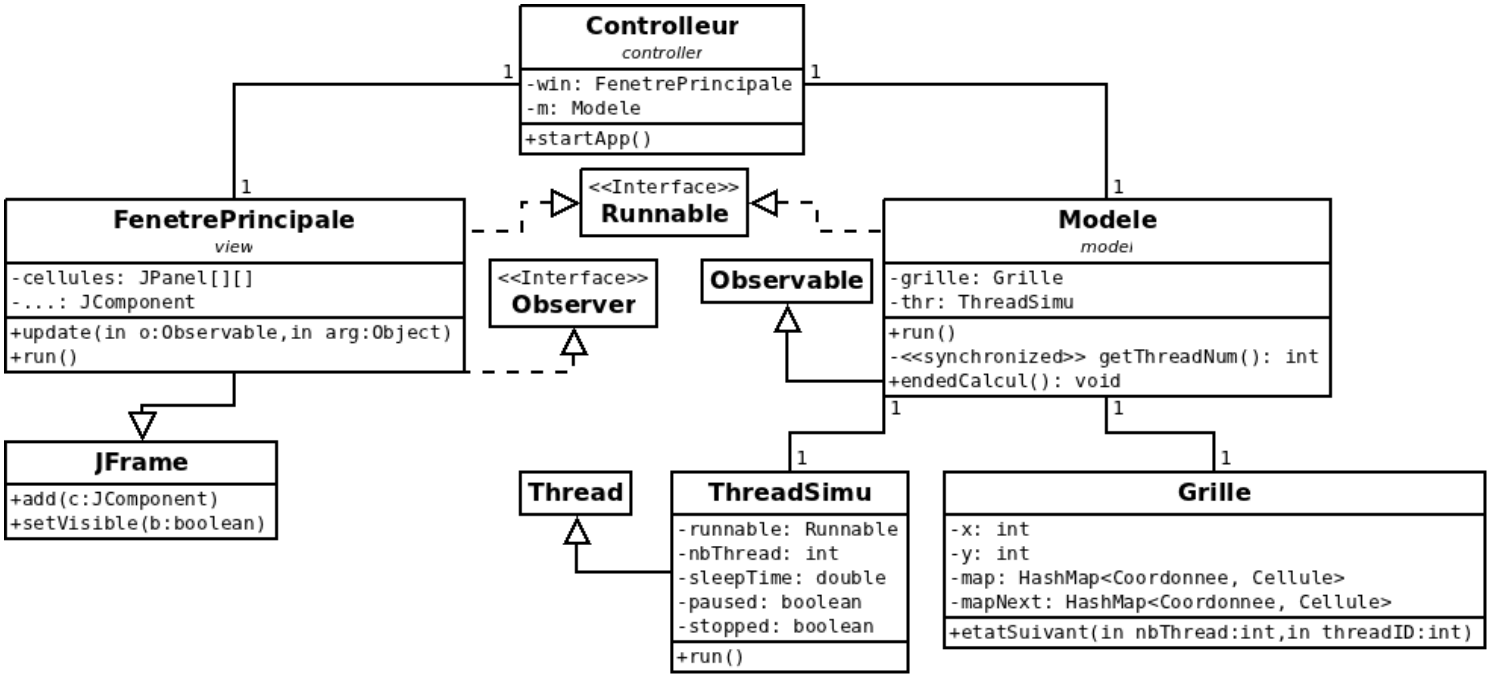
Le lancement de l'application consiste en la création d'un contrôleur qui crée le modèle ainsi que la vue. Celui-ci permet ensuite d'articuler les deux autres modules. Il est le cerveau de l'application et se gère tout seul. Lors de sa création, il crée une instance de chacune des deux autres classes sur lesquelles il garde une référence. Lors de la création du modèle, il lui fournit la taille de la simulation, le temps entre chacune de ces mises à jour ainsi que le nombre de threads qui seront chargés de travailler sur celui-ci. La vue est créée en donnant la référence du modèle qu'elle représente, puis le contrôleur informe la vue qu'elle devra se mettre à jour à chaque fois que le modèle le lui demande. Enfin, devant faire le lien entre la vue et le modèle, le contrôleur relie chacune des actions reçues par la vue à des fonctions chargées de mettre à jour le modèle.

Lors de sa création, le modèle s'initialise comme vide. Il crée aussi un thread de simulation « ThreadSimu » qui sera chargé de gérer les calculs à intervalle régulier. Cependant, le thread n'est lancé qu'à l'appel de la fonction *lancerSimulation().*

La vue est composée d'un ensemble de composants qui sont ajoutés les uns dans les autres de manière hiérarchique. Chacun de ces composants réagit à certains types d'actions, qui sont relayées au contrôleur.

III. Classes

Les trois classes ci-dessus ne suffisent pas à représenter l'ensemble de l'application. Il nous a fallu en créer d'autres.



La classe « Grille » est le cœur de l'application, c'est elle qui contient la représentation des cellules vivantes. Une grille possède une taille (largeur, hauteur) et une table de hachage contenant uniquement les cellules vivantes. De plus, elle possède également une seconde table qui permet de ne pas modifier la table durant les calculs mais seulement à la fin (front et back buffers).
De nombreuse fonctions permettent de manipuler cette grille facilement, sans en connaître réellement le fonctionnement interne :

- public void initGrille() : initialise de manière aléatoire la grille.
- public void clearGrille() : efface le contenu de la grille.
- public void etatSuivant() : calcule l'état suivant en fonction de l'état courant.

La fonction 'etatSuivant' est de loin la plus importante de cette classe. Pour déterminer le prochain état des cellules, nous parcourons la totalité de la grille, case par case. Pour chaque case, nous comptons son nombre de voisins vivants. Un modulo permet, pour les cases en bord de grille de considérer les cases du bord opposé comme voisines. Une fois le nombre de voisins vivants déterminé, il est aisé de déterminer en fonction de l'état courant de la case si une cellule naît, survit, ou meurt. Ce résultat est stocké dans la table de hachage 'arrière'. Une fois la totalité de la grille parcourue, les deux tables sont échangées pour mettre à jour la grille et la grille 'arrière' est réinitialisée, dans l'attente d'une nouvelle phase de calculs.

Dans un souci d'efficacité de calcul et de contrôle de la simulation, nous avons développé la classe « ThreadSimu » qui est un thread mais qui exécute la fonction *run()* de l'objet « Runnable » associé, régulièrement, tous les tant de secondes. De plus, il est possible de le mettre en pause ou de l'arrêter. Pour réaliser cette classe, nous définnissons deux variables booléennes, une première pour la pause, une seconde pour l'arrêt. Tant que la variable

d'arrêt est fausse, le thread teste la variable de pause. Si elle est aussi à fause, alors, il appelle la fonction *run()* de l'objet « Runnable » dans un nouveau thread standard. Puis, quelque soit la valeur de la variable de pause, il s'endort pendant un certain temps, avant de relancer cette suite d'instruction.
Nous avons par la suite modifié un peu cette classe, pour paralléliser les calculs. A chaque passe, le *ThreadSimu* lance plusieurs threads puis attends la fin de chacun d'eux. Une fois les calculs parallèles terminés, la fonction *endedCalcul()* de l'objet « Runnable » est appelée, pour finaliser les calculs (ici, échanger les deux buffers). La gestion du parallélisme est laissée à la charge de l'objet exécutable. Une description plus complète est donnée dans la partie des extensions : calcul distribué.

IV. Extensions

Nous n'avons pas réalisés qu'un simple jeu de la vie, nous y avons rajoutés quelques extensions qui permettent à l'utilisateur d'interagir avec le jeu.

⇒ Environnement interactif :

Le but de cette extension est de permette à l'utilisateur d'interagir avec le modèle à travers la vue. Il peut ainsi modifier la taille de la grille, la vitesse d'exécution et l'état des cellules. Pour cette extension, il a fallu avant tout ajouter des composants à la vue, ainsi, des spinners permettent de modifier les valeurs numériques (taille et vitesse), et la grille réagit au passage de la souris. En effet lorsque l'utilisateur promène sa souris sur le jeu, la case sous le pointeur se fonce pour lui indiquer qu'elle est sélectionnée. S'il clique sur cette cellule en fonction de l'état précédant de celle-ci, une cellule vivante est ajoutée au jeu ou une cellule vivante meurt. Les modifications apportées au modèle doivent être immédiatement visible et il faut donc que la vue se mette à jour juste après les modifications. Cette extension nous a demandé de nombreux mutateurs sur les variables du modèle. Une fois ceux-ci mis en place, il nous a été facile de rajouter un nouveau bouton pour effacer complètement la grille.

⇒ Charger et sauvegarder l'état des grilles de cellules

L'extension permet de sauvegarder ou de charger à n'importe quel moment l'état d'une grille. Pour permettre cela, nous avons ajoutés des items « Ouvrir… » et « Enregistrer… » dans le menu « Fichier » de la barre de menu.
Lorsque l'utilisateur ouvre une grille de jeu, le jeu remplace la grille existante par la nouvelle. De plus si les tailles de la grille du jeu actuel et de la nouvelle grille ne sont pas identiques le jeu s'adapte à la nouvelle grille, ainsi le jeu sera cohérent.
Pour permettre cette extension, nous avons ajouté des fonctions à différentes classes :

- « Controleur» :
  - *private void onSaveAction ()* : cette fonction est appelée par l'item « Enregistrer sous… », elle fait appel à un JFileChooser qui renvoi le nom du fichier dans lequel l'utilisateur veut enregistrer sa grille, ensuite ces informations sont transmise à la fonction save(String file) de la classe « Grille ».
  - *Private void onLoadAction()* : cette fonction est appelée par l'item « Ouvrir… », elle fait appel à un JFileChooser qui permet à l'utilisateur de

sélectionner le fichier qu'il veut charger. Les informations sont transmises à la fonction load(String file) de la classe « Grille »

- « Grille » :
  - *public void save(String file)* : cette fonction génère un fichier à l'aide de la classe « Properties ». Dans ce fichier nous sauvegardons le contenu de notre HashMap « map » qui contient les coordonnées des cellules vivantes sur notre grille ainsi que la taille (x, y) de la grille.
  - *Public void load(String file)* : cette fonction a pour but de charger la grille sauvegardée précédemment par l'utilisateur. La fonction lit le fichier *file* et récupère les informations utiles telles que la taille de la grille(x, y) et les coordonnées des cellules vivantes. Ainsi, l'application peut générer cette grille.

⇒ Motifs préconstruits à placer dans le jeu

Cette extension permet à l'utilisateur d'ajouter des formes préconstruites. Pour cela il sélectionne dans la liste déroulante (en bas de l'application) le motif qu'il veut ajouter, puis il se déplace sur la grille avec la souris pour ajouter le motif et clique ou il veut l'ajouter. Une prévisualisation du motif est fournie à l'utilisateur lorsqu'il se déplace sur la grille. Lorsque l'utilisateur ajoute un motif, il ajoute uniquement des cellules vivantes, il ne supprime pas de cellule sur la grille.

L'utilisateur peut ajouter plusieurs motifs, des motifs classique (tel qu'une croix, un carré, …), des motifs générés aléatoirement (ces motifs ont un taille (x, y) aléatoire mais aussi un nombre de cellule vivante aléatoire) et des motifs qu'il aura lui-même créé à l'aide d'un éditeur de texte, en effet il est possible d'ajouter un motif via le menu « Edition » → « Ouvrir un motif… ».

Pour permettre l'ajout d'un motif, nous avons développé une classe « Motif » qui contient une HashMap où sont stocker les cellules vivantes du motif, la taille du motif « int x, int y » et le nom du motif « String name ». Cette classe contient les fonctions suivantes :

- *public Motif(int x, int y, String name)* : permet de créer un motif.
- *public Motif(File file)* : permet de créer un motif à partir d'un fichier, ce constructeur fait appel à la fonction *private loadMotif(String file)*. Le constructeur initialise les attributs de l'objet en fonction du fichier *file*.
- *public void initMotif()* : initialise un motif aléatoire, c'est à dire que ce motif aura une taille aléatoire et un nombre de cellule vivante aléatoire avec pour chacune une position aléatoire.
- *Public void initMotif(HashMap<Coordonées, Cellule> map)* : initialise la HashMap de l'objet motif en fonction de la HashMap passé en parametre
- *public void loadMotif(String file)* : cette fonction lit un fichier texte formaté (exemple 1) pour générer le motif. Dans ce fichier les 1 correspondent à une cellule vivante et les 0 à une cellule morte, le nombre de ligne et de colonne détermine la taille du motif, le nom du motif est le nom de fichier passé en paramètre.

```
0 0 0 0 0
0 1 0 1 0
0 0 1 0 0
0 0 1 0 0
```

Exemple1

⇒ Calcul distribué, multithreading

Lors du calcul de l'état suivant des cellules, le nombre de tests est proportionnel au nombre de cases de la grille. Dans le cas d'une grille de taille relativement grande, le temps de calcul

devient de plus en plus long. Pour accélérer le processus de calcul il est possible de paralléliser cette phase ainsi, un thread n'analyse qu'une petite partie de la grille pendant que d'autres analysent le reste. Une fois chacun des threads terminés, il ne reste qu'à échanger les deux HashMap. Pour réaliser cette extension, il nous a fallu modifier quelques peu les classes *ThreadSimu*, *Modele,* et *Grille*.
Le thread de simulation doit désormais créer plusieurs threads qui exécuteront la fonction *run()* du modèle. Le nombre de threads à créer est déterminé à la construction et stocké à la fois par le modèle et le thread de simulation.
Il faut désormais prendre en compte que la fonction *run()* du modèle est appelé n fois, par n threads différent. Pour faire travailler efficacement ces threads, il est nécessaire de les différencier, pour ne pas leur faire faire à tous la même chose. Pour ce faire, nous avons implémenté la fonction *private synchronized int getThreadNum()*. Cette fonction est appelée par chacun des threads pour obtenir un identifiant unique compris entre zéro et le nombre de threads moins un. Cette fonction ayant un accès en lecture et en écriture sur une variable de l'instance du modèle, il est nécessaire de la déclarer en tant que fonction « synchronized ». Ainsi, un seul thread peut avoir accès à cette fonction à la fois.
Une fois le numéro d'identification obtenu, le thread appelle la fonction de calcul d'état suivant de la grille en lui fournissant sont numéro de thread et le nombre total de threads qui appelleront cette fonction. Le calcul de l'état suivant est donc divisé par le nombre de threads de calcul, et le thread courant n'analyse qu'une petite partie de la grille. Par exemple, sur une grille de 100 cases, avec deux threads de calcul, le premier analysera les 50 premières cases et le second les 50 dernières. Les deux threads travaillant de manière concurrente, le temps de calcul est théoriquement divisé par deux. Nonobstant, après avoir réalisé des tests, nous nous sommes rendu compte que le gain apporté par le multithreading était dérisoire. En effet, pour une grille de 40000 cases et un pas de calcul de 0.25 seconde, avec un seul thread de calcul, le modèle est recalculé 402 fois en 2 minutes. Dans le même laps de temps et avec la même grille de départ, avec 4 threads de calculs, 408 itérations sont faites. On se rend vite compte que l'on est loin des performances quatre fois supérieures. Cet écart entre la théorie et la pratique dépend de bon nombre de paramètres. En particulier, les ordinateurs récents (avec plusieurs cœurs) ont tendance à paralléliser les calculs même si ceci n'est pas expressément programmé. De plus, la gestion des threads prend du temps, en particulier pour la sauvegarde et restauration du contexte mémoire. On a donc une perte de temps pendant le changement de deux threads que l'on n'a pas lorsque l'on travaille avec un seul thread. Pour améliorer de manière significative les performances, il serait plus judicieux de modifier l'algorithme de calcul. Dans un premier temps en délaissant les zones de la grille n'ayant pas de cellules vivante proche, puis dans un second temps en analysant non plus cellule par cellule mais les motifs récurrents.

V. Conclusion

A travers ce projet, nous avons développé notre connaissance du langage Java en utilisant bon nombre de ses capacités (programmation objet, création d'interfaces graphiques, multithreading). Néanmoins, de nombreuses améliorations pourraient être apportées. Nous citerons par exemple :

- Amélioration de l'algorithme de calcul
- Optimisation de l'affichage (l'utilisation de JPanel pour représenter les cellules n'est, à nos yeux, pas une solution efficace)
- Création d'une classe abstraite MultithreadedRunnable mettant en place le système d'identification des threads et forçant l'implémentation de la fonction *endedCalcul().*

VI. Annexes

Diagramme de séquence :