

Un automate cellulaire est un objet mathématique qui permet de simuler l'évolution d'une population de cellules virtuelles au cours du temps, selon des règles de voisinage. Ces règles sont appliquées à toutes les cellules d'une même génération, produisant ainsi une nouvelle génération, entièrement dépendante de la génération précédente. Le « Jeu de la Vie » du mathématicien John Conway est un exemple d'automate cellulaire à deux dimensions dont les règles d'évolution sont les suivantes :

- Si une cellule possède moins de deux voisines, elle va mourir (solitude) ;
- Si une cellule possède plus de trois voisines elle va mourir (étouffement) ;
- Si un emplacement vide possède trois voisines, une nouvelle cellule va naître ;

Vous allez implémenter le Jeu de la Vie selon un modèle fondé sur cinq *Design Patterns* : *Etat*, *Singleton*, *Observer*, *Command* et *Visitor*. En séparant structures de données, règles de fonctionnement et interface graphique, la modélisation fondée sur les « patrons de conception » simplifie considérablement la maintenance du logiciel.

1. Implémentation des cellules et leurs états :

Vous implémenterez la structure de données du Jeu de la Vie à l'aide d'un tableau de cellules, chaque cellule étant caractérisée par son état (vivante ou morte). Le modèle de référence est la version minimale du DP « State » ci-contre. Son instantiation pour le Jeu de la Vie est donné en Fig. 1. La modification de l'état d'une cellule se fait par invocation des méthodes *vit* et *meurt* de la cellule. Le diagramme d'états des cellules est également donné en Fig. 1.

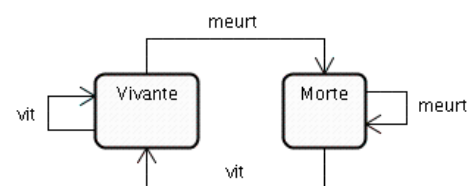
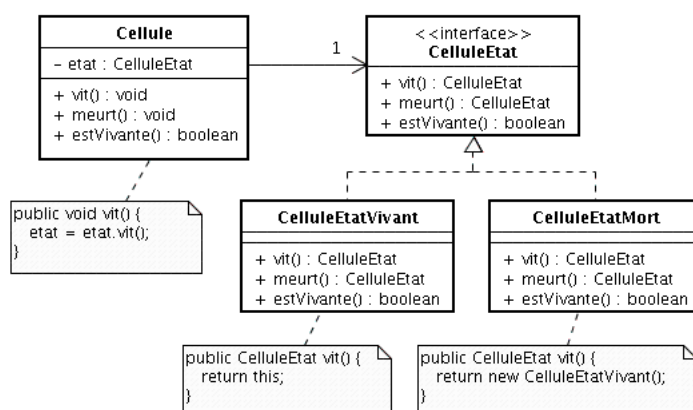
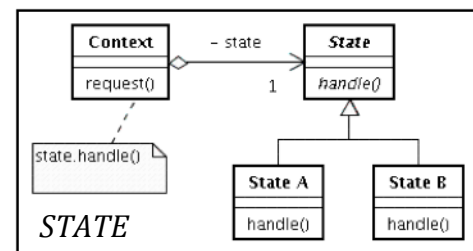


Diagramme d'états des cellules

Fig. 1 : Implémentation des cellules et de leur état à l'aide du pattern « State »

Ce modèle présente un inconvénient : celui de multiplier inutilement les instances des classes *CelluleEtatVivant* et *CelluleEtatMort*, qui sont pourtant immuables. Vous allez y remédier en implémentant le DP « Singleton ». Il s'agit d'associer une instance unique à chacune des deux classes, et de n'utiliser que ces deux instances. L'instance de chaque classe n'est accessible qu'au travers d'une méthode *getInstance*.

2. Implémentation de la grille de cellules :

Le modèle des cellules étant disponible, vous pouvez implémenter la grille de cellules, comme attribut de la classe *JeuDeLaVie*. Complétez votre programme selon le modèle donné Fig. 2. La méthode *initialiseGrille* doit peupler le tableau avec des cellules vivantes ou mortes, en fonction de la méthode *Math.random* qui retourne un entier compris entre 0 et 1. La méthode *nombreVoisinesVivantes* de la classe *Cellule* retourne le nombre de cellules vivantes se trouvant dans le voisinage de la cellule (8 emplacements à tester).

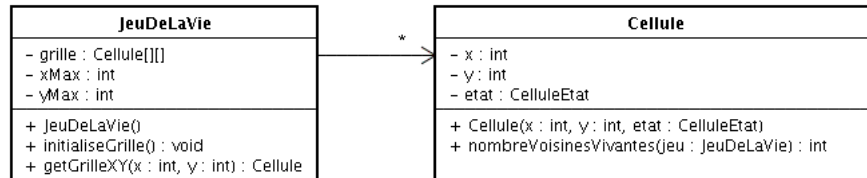


Fig. 2 : Implémentation de la grille de cellules

3. Implémentation de l'interface graphique :

L'interface graphique du jeu sera modélisée selon le DP « Observer » donné ci-contre. Un observateur est un objet qui déclenche un certain comportement sur la demande d'un objet observable. La méthode *notifyObservers* de l'observable envoie le message *update* à tous ses observateurs. Dans notre cas, l'observable est un *JeuDeLaVie* et l'observateur est un *JeuDeLaVieUI*. Implémentez le modèle en Fig. 3, puis écrivez la méthode *main* de *JeuDeLaVie* qui va instancier un *JeuDeLaVie*, un *JeuDeLaVieUI*, et qui va enregistrer ce dernier comme observateur. A la fin de cette étape, vous pourrez visualiser la population initiale aléatoire de cellules.

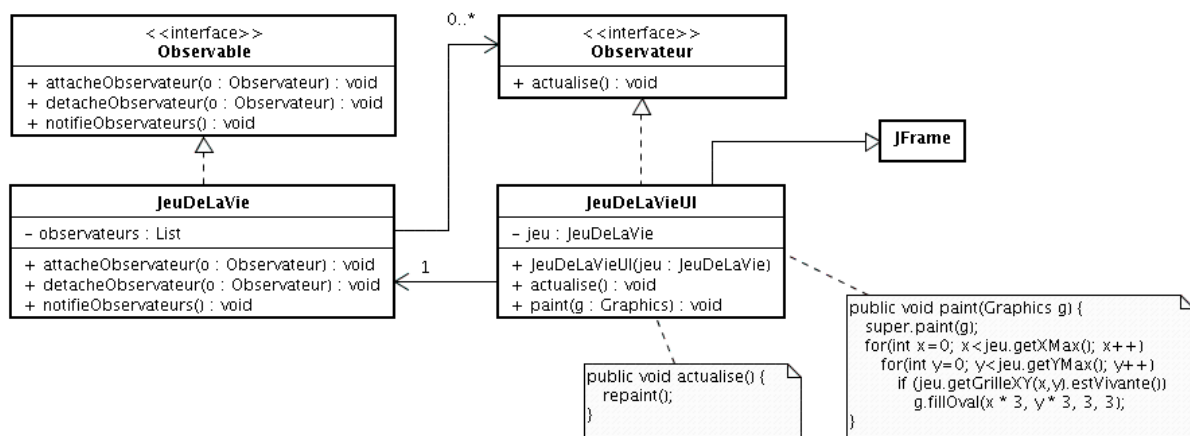
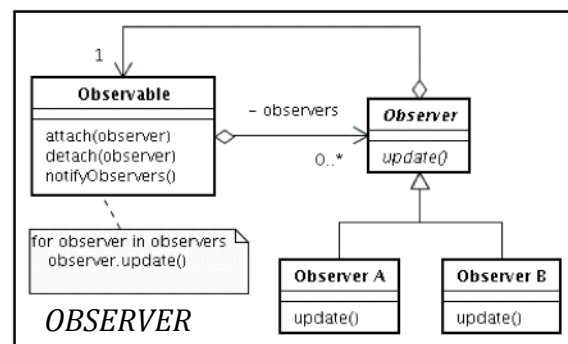


Fig. 3 : Implémentation de l'interface graphique de la grille à l'aide du pattern « Observer »

4. Implémentation des actions sur les cellules :

Dans l'implémentation « classique » du jeu de la vie, la génération suivante est construite dans un deuxième tableau, afin de ne pas modifier la génération courante. Une fois terminée, la génération suivante (qui devient la génération courante) est intégralement recopiée dans le premier tableau. Une autre solution est dans un premier temps de stocker dans une file d'attente toutes les actions à effectuer (modifications des états des cellules), puis dans un deuxième temps d'exécuter les actions. Cette solution n'utilise pas de second tableau, ni d'étape de copie. Elle repose sur l'utilisation du DP « Command » donné page suivante.

Selon ce DP, un « client » instancie des commandes destinées à un « receveur », et un « invoqueur » déclenche l'exécution des commandes. Dans la Fig. 4, l'invoqueur est un *JeuDeLaVie*, les receveurs sont des cellules, et les actions sont *vit* et *meurt* (le client sera défini au cours de l'étape n°5). Vous implémenterez la file d'attente de commandes de la même façon que vous avez implémenté la liste d'observateurs. La méthode *executeCommandes* exécute toutes les commandes et vide la file.

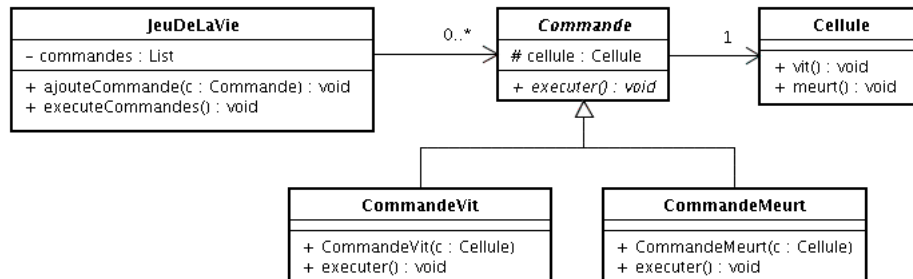
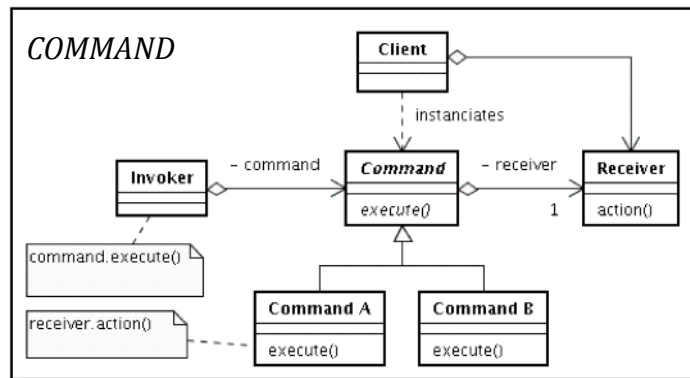
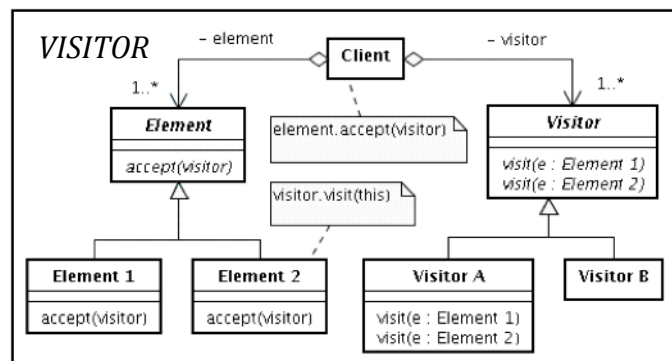


Fig. 4 : Implémentation des actions sur les cellules à l'aide du pattern « Command »

5. Implémentation des règles du jeu :

La cinquième étape consiste à écrire les règles du jeu en utilisant le DP « Visitor ». Un visiteur est une façon de séparer un algorithme d'une structure de données. Le client distribue un visiteur aux éléments, qui demandent à leur visiteur d'exécuter le traitement. Dans la Fig. 5, le client est le jeu de la vie, les éléments sont des cellules, le visiteur est *VisiteurClassique*. La méthode *distribueVisiteur* invoque la méthode *accepte* de toutes les cellules, lesquelles invoquent la méthode *accepte* de leur état.



Enfin, ce dernier invoque la « bonne » méthode du visiteur (*visiteCelluleVivante* ou *visiteCelluleMorte*) qui est susceptible d'instancier et d'ajouter une nouvelle commande (*CommandeVit* ou *CommandeMeurt* selon l'environnement de la cellule) dans la file d'attente de commandes du jeu.

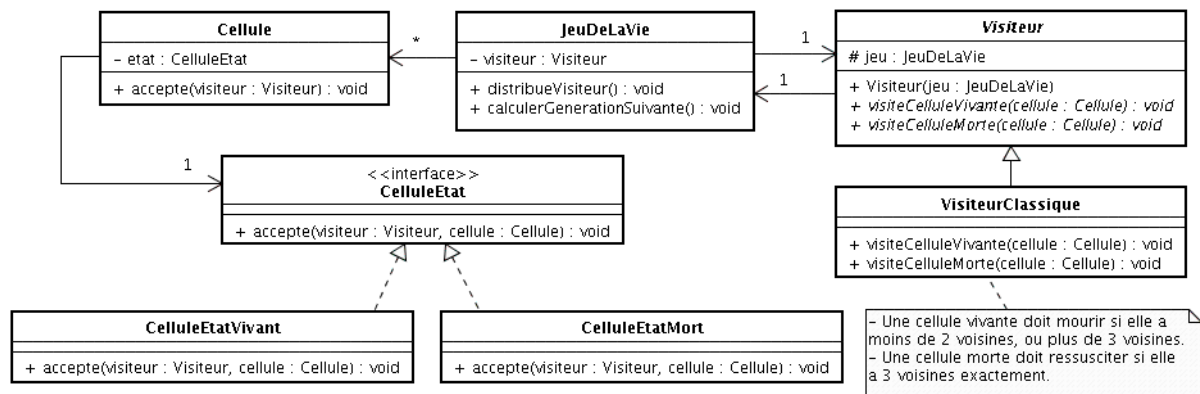


Fig. 5 : Implémentation des règles du jeu à l'aide du pattern « Visitor »

La méthode *calculerGenerationSuivante* fait avancer l'automate d'une génération en trois étapes : 1) distribuer un visiteur ; 2) exécuter les commandes ; 3) actualiser les observateurs. Il ne vous reste plus qu'à instancier le visiteur et à invoquer la méthode *calculerGenerationSuivante* en boucle.

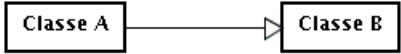
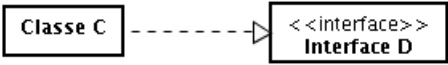
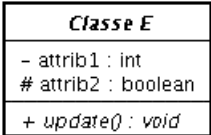
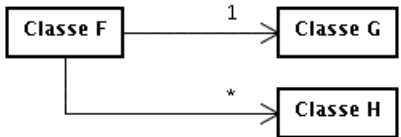


6. Extensions et améliorations :

Plusieurs extensions et améliorations sont vivement encouragées :

1. Écrire un deuxième observateur *JeuDeLaVieConsole* en mode texte qui se contente d'afficher dans la console Java le nombre de cellules actuellement en vie ;
2. Écrire un deuxième visiteur *VisiteurNightAndDay* qui implémente les règles suivantes :
 - Une cellule vivante doit mourir si elle a moins de 3 voisines vivantes, plus de 8 voisines vivantes, ou 5 voisines vivantes exactement.
 - Une cellule morte doit ressusciter si elle a exactement 3, 6, 7 ou 8 voisines vivantes.
3. Écrire une interface graphique qui comporte un certain nombre de contrôles :
 - Bouton pour exécuter/arrêter la boucle d'exécution ;
 - Bouton pour avancer d'une génération lorsque la boucle est arrêtée ;
 - Bouton pour réinitialiser aléatoirement la grille de cellules ;
 - Réglage de la vitesse de la boucle d'exécution ;
 - Sélection de la règle du jeu, *etc.*

◆◆◆

Éléments de notation UML :

Notation	Signification
	La classe A hérite de la classe B ;
	La classe C implémente l'interface D ;
	La classe E est abstraite ; L'attribut attrib1 de type int est privé ; L'attribut attrib2 de type boolean est protégé ; La méthode update est abstraite ;
	Relations structurelles entre objets : <ul style="list-style-type: none"> ▪ Les objets F accèdent à un seul objet G ; ▪ Les objets F accèdent à plusieurs objets H ;
	<ul style="list-style-type: none"> ▪ L'existence de l'objet N ne dépend pas de l'existence de l'objet M (agrégation) ; ▪ L'objet N doit être accessible d'une autre façon qu'au travers de l'objet M ;
	<ul style="list-style-type: none"> ▪ L'existence de l'objet P dépend de l'existence de l'objet O (composition) ; ▪ L'objet P n'est accessible qu'au travers de l'objet O ;