UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

IUT DE VÉLIZY

université PARIS-SACLAY

Rapport de projet

Le Jeu de la vie

BARRAQUÉ Louis
LAGUILLON Céline

Sommaire

Introduction

Le rapport montre les différentes démarches suivies lors de la conception et la réalisation du développement du Jeu de la vie.

La conception a pour but de fournir une structure pour un élément complexe : l'élément est décomposé en parties, chacune avec des responsabilités, et ensemble elles ont pour objectif d'accomplir une tâche bien définie. Toutes les parties interagissent entre elles, et produisent ainsi un modèle. Un modèle est une représentation simplifiée d'une situation problématique réaliste : il structure les informations et les activités d'une organisation.

La conception orientée objet, utilisée dans ce dossier, conceptualise le domaine du problème en termes d'objets qui interagissent, se modifient et réagissent aux événements. Elle est définie par quatre principes fondamentaux : l'abstraction, l'encapsulation, la classification et l'héritage. L'abstraction met en évidence un élément, et néglige tout autre aspect de son environnement. L'encapsulation dissimule la structure de l'objet et les détails de la réalisation des opérations au monde extérieur. La classification définit les classes décrivant un ensemble d'objets qui partagent la même sémantique, les mêmes attributs. L'héritage désigne une généralisation d'une classe par rapport à une autre : une classe A hérite d'une classe B, si B généralise A.

Le diagramme des classes et les diagrammes d'interaction intégrés à ce rapport sont réalisés sur Star UML 2, et suivent les conventions et les partis pris de ce logiciel. De ce fait, les objets ne sont pas soulignés comme l'indique la convention UML.

Pour faciliter la modification du programme, toutes les constantes sont définies dans une interface. Contrairement à une classe, une interface est plus facile d'accès par les autres classes. Bien qu'elle soit intégrée au code, ses relations avec les classes n'apparaissent pas sur le diagramme des classes. Elle est en relation avec toutes les classes qui utilisent des constantes. Les constantes sont utiles si nous souhaitons modifier simplement une valeur pour tout le code sans avoir à le parcourir dans sa globalité.

De plus, toutes les fonctionnalités de l'application développée sont accessibles par des mnémoniques, dans le but de faciliter la navigabilité entre elles.

I. Dossier de conception générale

Ce dossier présente le diagramme des classes, ainsi que les pricipaux diagrammes d'interaction. Les diagrammes ont été réalisé à haut niveau d'abstraction, afin de représenter les multiples interactions entre les classes et les packages.

1. Les diagrammes de classes.

Le diagramme de classes montre un ensemble de classes, d'interfaces et leurs relations entre elles. Le système est illustré d'un point de vue statique. Il sert à modéliser les collaborations simples au sein du système, ainsi que son vocabulaire et un schéma de base de donnée logique. Les diagrammes de classes présentés dans ce dossier contiennent une unique relation entre les classes : l'association.

L'association est la relation structurelle qui spécifie que les objets d'une classe sont connectés aux objets d'une autre. Elle est représentée par un segment qui relie les deux classes : elle peut avoir un nom, un rôle, préciser la multiplicité de chaque classe, et la navigabilité entre les classes par une flèche.

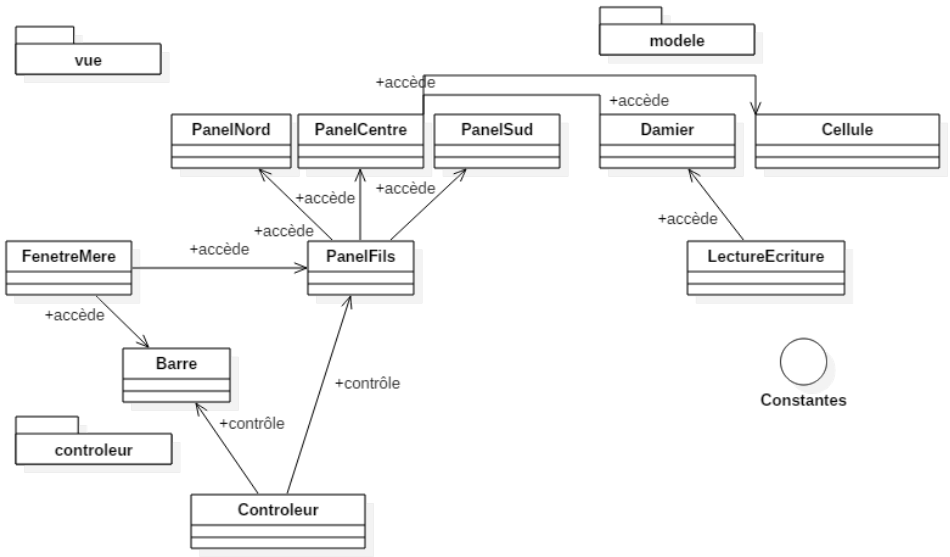


Figure 1 : Diagramme des classes à haut niveau d'abstraction, conception initiale.

La figure 1 représente nos premières idées lors de la conception du projet. Bien que notre idée de conception ait bien changé, elle contient toutes les bases du projet.

Tout d'abord, nous avons créé trois packages pour respecter l'architecture *Modele Vue Controleur*. Puis, nous avons trouvé plus simple de créer trois classes qui construisent le *PanelFils*, et n'avoir tout de même qu'une seule classe à contrôler depuis le contrôleur. Cependant, il s'est révélé compliqué de contrôler tous les panels depuis cette classe. Pour cela, nous avons choisi de contrôler chaque panel indépendamment.

I. Dossier de conception générale

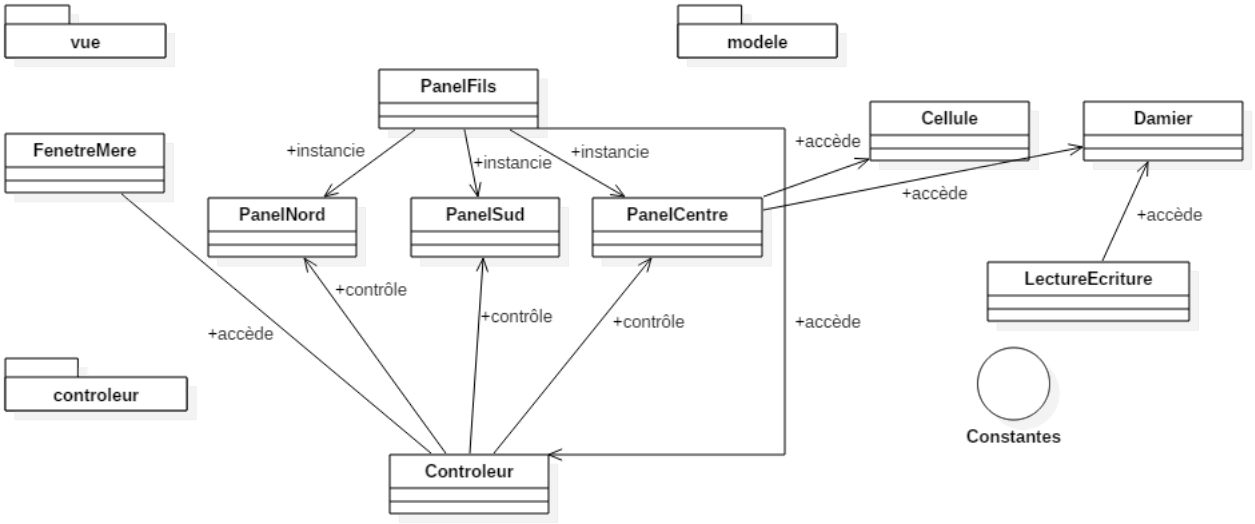


Figure 2 : Diagramme des classes à haut niveau d'abstraction, conception finale.

La figure 2 décrit toutes les relations entre les classes des différents packages que le programme contiendra. La classe *Damier* est un tableau des états de chaque case du damier, et la classe *Cellule*, sa représentation graphique où les états sont représentés par des couleurs. La classe *LectureEcriture* permet la lecture et l'écriture dans un fichier. La classe *FenetreMere* construit la fenêtre qui va s'ouvrir, ainsi que la barre. La classe *PanelNord* contient les étiquettes avec les informations relatives au damier, *PanelSud* les boutons pour faire avancer le jeu, et pour l'enregistrement, et *PanelCentre* le damier. La classe *PanelFils* rassemble les trois classes précédentes. Et enfin, la classe *Controleur* contrôle les interactions entre l'utilisateur et l'interface graphique.

Nous souhaitions créer une classe *Barre* dans le but de ne pas surcharger la classe *FenetreMere*. Mais nous n'arrivions pas à contrôler la barre de menus depuis le contrôleur courant. Nous avons donc créé la barre de menus en même temps que la *FenetreMere*. Dans un premier temps, elle était contrôlée depuis un deuxième contrôleur. Mais, dans un second temps, nous nous sommes rendus compte qu'il était plus facile de tout regrouper dans un seul contrôleur. De ce fait, la classe *Controleur* est reliée à toutes les classes du package vue, à l'exception de la classe *PanelFils*, qui la prend en paramètre cependant.

2. Les diagrammes d'interaction.

Un diagramme d'interaction représente les comportements des objets les uns par rapport aux autres.

Un diagramme de séquence montre les interactions des évènements dans le temps. En ordonnée, le temps est représenté par une ligne de vie, et en abscisse, les objets et les messages sont représentés.

I. Dossier de conception générale

La classe FenetreMere possède la méthode main().

• Les diagrammes d'interaction : Affichage d'un damier vide.

Les objets qui participent à ce scénario sont l'instance de FenetreMere, l'instance de Controleur, l'instance de PanelCentre et damierTemp de type Damier.

L'instance de FenetreMere fait appel à la méthode *actionPerformed()* de l'instance du Controleur avec la taille du damier sélectionné. L'instance du Controleur envoie un message de création de l'objet damierTemp. Puis, elle fait appel à la méthode *afficheDamier()* et *enregistreEcouteur()* de l'instance de PanelCentre. Ensuite, elle fait appel à *affichePopulation(damierTemp)*, met son champ chGeneration à -1, et enfin fait appel à sa méthode *afficheGeneration()*.

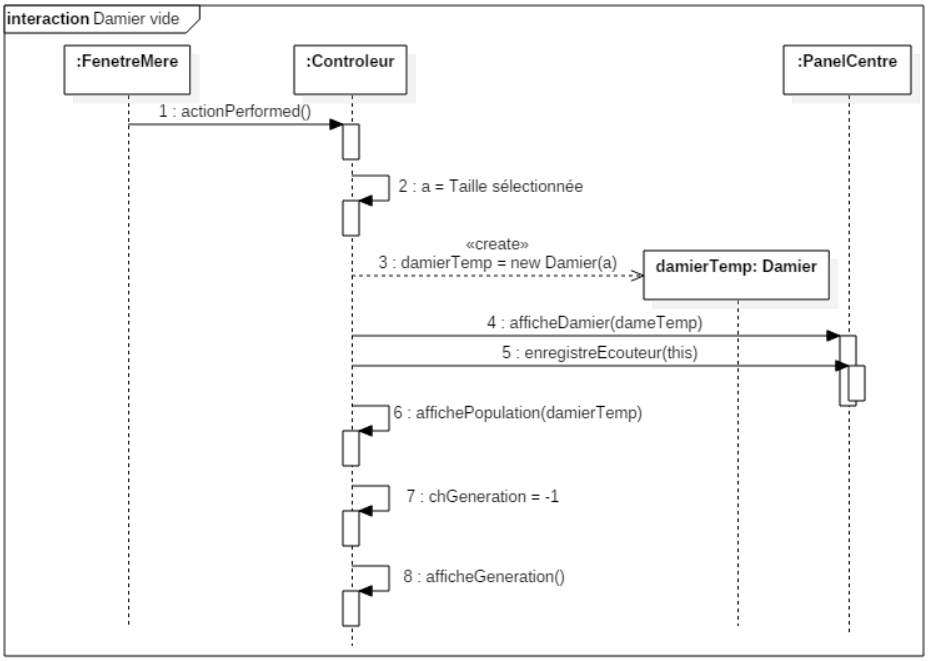


Figure 3 : Diagramme de séquence de l'affichage d'un damier vide à haut niveau.

• Les diagrammes d'interaction : Affichage d'un damier aléatoire.

Les objets qui participent à ce scénario sont l'instance de FenetreMere, l'instance de Controleur, l'instance de PanelCentre et damierTemp de type Damier.

I. Dossier de conception générale

L'instance de FenetreMere fait appel à la méthode ActionPerformed de l'instance du Controleur avec la taille du damier sélectionné. L'instance du Controleur envoie un message de création de l'objet damierTemp et fait appel à la méthode *aleatoire()* de cette même instance. Puis, elle fait appel à la méthode *afficheDamier()* et *enregistreEcouteur()* de l'instance de PanelCentre. Ensuite, elle fait appel à *affichePopulation(damierTemp)*, met son champ chGeneration à -1, et enfin fait appel à sa méthode *afficheGeneration()*.

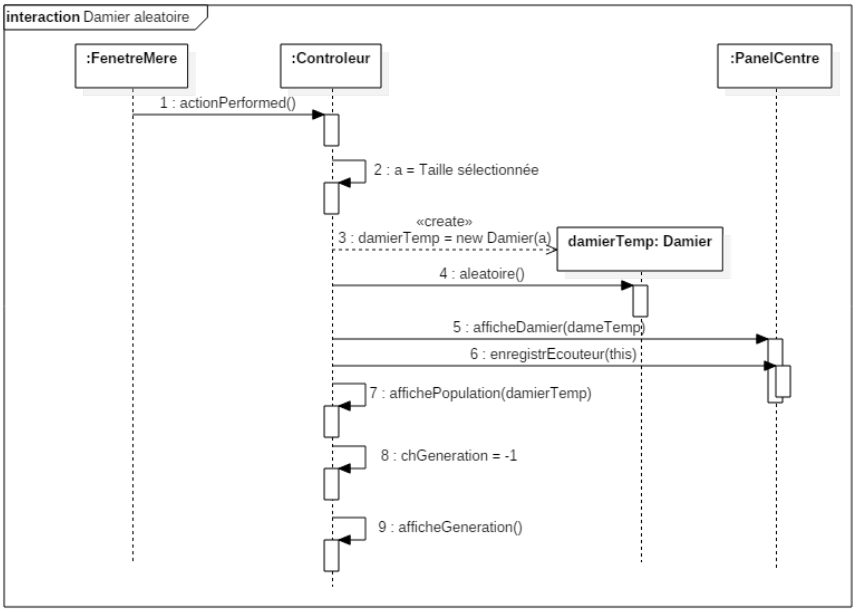


Figure 4 : Diagramme de séquence de l'affichage d'un damier aléatoire à haut niveau.

- Les diagrammes d'interaction : la méthode afficheDamier().

Les objets qui participent à ce scénario sont l'instance de PanelCentre, chCollectionCellule de type ArrayList<Cellule> et cellule de type Cellule.

L'instance de PanelCentre fait appel à la méthode *removeAll()*, *setLayout()*. Puis, elle envoie un message de création d'une ArrayList chCollectionCellule de type Cellule, et un autre à cellule. Ensuite, elle envoie un message qui fait appel à la méthode *add()* de chCollectionCellule. Enfin, elle s'envoie des messages à elle-même : *setBackground()*, *add()*, *revalidate()*, *repaint()*.

I. Dossier de conception générale

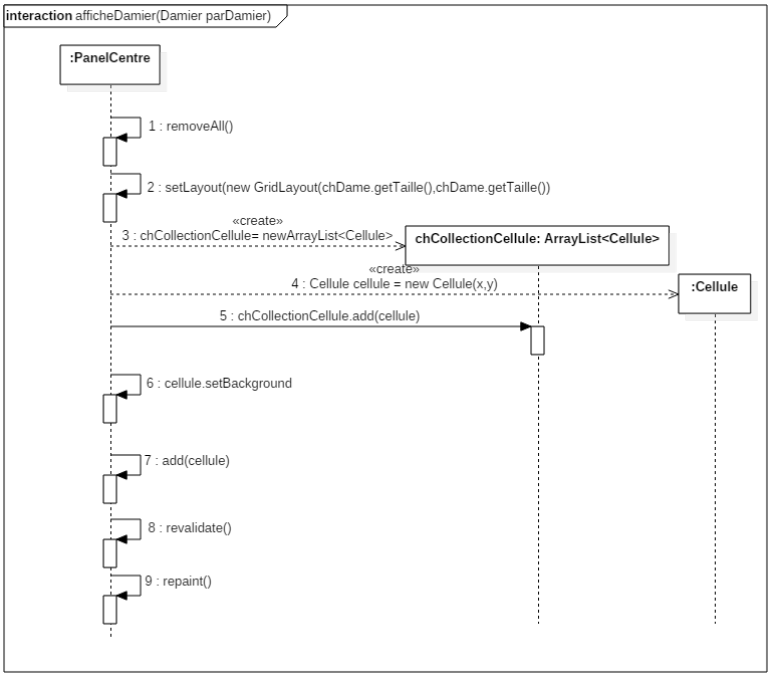


Figure 5 : Diagramme de séquence de la méthode afficheDamier() à haut niveau.

- Les diagrammes d'interaction : Enregistrement d'un damier dans un fichier.

Les objets qui participent à ce scénario sont l'instance de FenetreMere, l'instance de Controleur, l'instance de PanelSud, l'instance de LectureEcriture et fichier de type File.

L'instance de FenetreMere envoie un message d'enregistrement à l'instance du PanelSud, qui elle-même fait appel à la méthode *actionPerformed()* de l'instance du Controleur. L'instance du Controleur envoie un message *getChTexte().getText()* à l'instance du PanelSud, puis envoie un message de création de fichier, et enfin envoie un message *ecriture()* à l'instance de LectureEcriture.

I. Dossier de conception générale

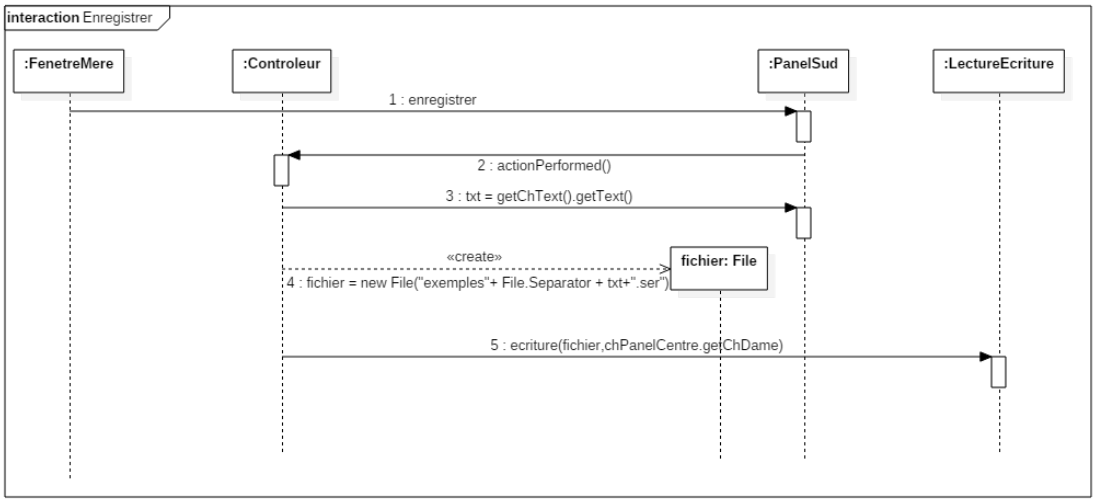


Figure 6 : Diagramme de séquence de l'enregistrement d'un damier à haut niveau.

- Les diagrammes d'interaction : Sortie de l'application.

Les objets qui participent à ce scénario sont l'instance de FenetreMere, l'instance de Controleur, l'objet saisie de type int et l'instance de System.

L'instance de FenetreMere envoie un message *actionPerformed()* à l'instance du Controleur, qui envoie un message de création à saisie, puis envoie un message *exit(1)* à l'instance de System.

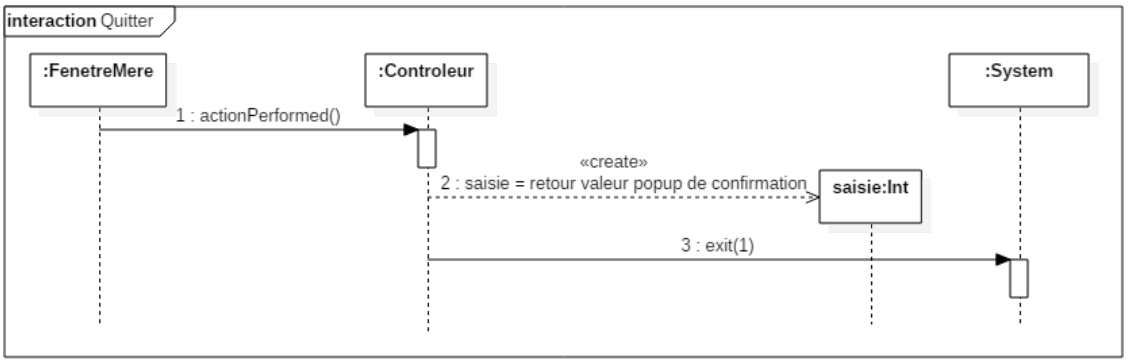


Figure 7 : Diagramme de séquence de la fermeture de la FenetreMere à haut niveau.

II. Dossier de conception détaillée

Le dossier de conception détaillée complète le dossier de conception générale, dans lequel les diagrammes étaient à haut niveau d'abstraction. Le bas niveau d'abstraction permet de s'approcher au maximum du code. Par exemple, les types des attributs, leur visibilité, leur multiplicité sont précisés. Les relations entre les classes ne sont plus représentées par des segments mais par les attributs des deux classes concernées par la relation.

Par souci de logique vis à vis du nom du jeu mathématique étudié, l'état "morte" sera représenté par le booléen false et par la couleur blanc, tandis que l'état "en vie" par le booléen true et par la couleur noir.

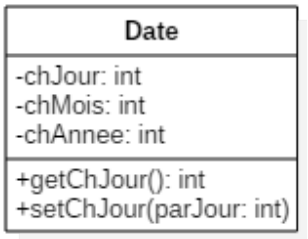


Figure 8 : Représentation UML de la classe Date à bas niveau d'abstraction.

La figure 8 présente la classe Date, selon le langage UML. L'attribut chJour possède un accesseur, getChJour(), et un modifieur, setChJour().
Afin de conserver le principe d'encapsulation des langages de programmation orientée objet, chaque attribut de chaque classe possède des accesseurs et des modifieurs, comme présenté dans la figure 1.

Nous avons décidé d'utiliser un tableau double dimension pour stocker les informations sur les cellules en vie et mortes, pour sa simplicité d'utilisation et sa logique élémentaire. Les informations sont stockées sous forme de booléens pour prendre le moins de place possible. En effet, un booléen ne prend qu'un octet dans la mémoire, contrairement à un entier qui en prend quatre.

Ensuite, nous avons décidé de séparer le stockage des données des éléments graphiques pour réduire la place de stockage des damiers enregistrés. Il s'agit de la raison pour laquelle la classe Damier se situe dans le package modele et non vue. Damier est la classe qui est enregistrée et stockée dans un fichier si l'utilisateur le souhaite, tandis que la classe Cellule est simplement sa représentation graphique, elle ne fait que montrer l'état des cellules par des couleurs.

Puis, la classe Controleur s'occupe de toutes les interactions que peut avoir l'utilisateur avec l'interface graphique. Cette classe est instanciée dès la FenetreMere afin qu'il puisse garder la barre de menu en mémoire. Elle a toutes les classes graphiques, du package vue, en mémoire dans le but d'effectuer des modifications sur celles-ci.

II. Dossier de conception détaillée

Enfin, lors de l'enregistrement, une fenêtre s'ouvre afin de vérifier si l'utilisateur a effectué cette action volontairement, et dans le cas échéant s'il est sûr de lui. Puis, il y a une deuxième vérification, qui s'assure qu'un nom de fichier ait bien été entré, et qu'il n'est pas déjà utilisé dans le dossier exemples.

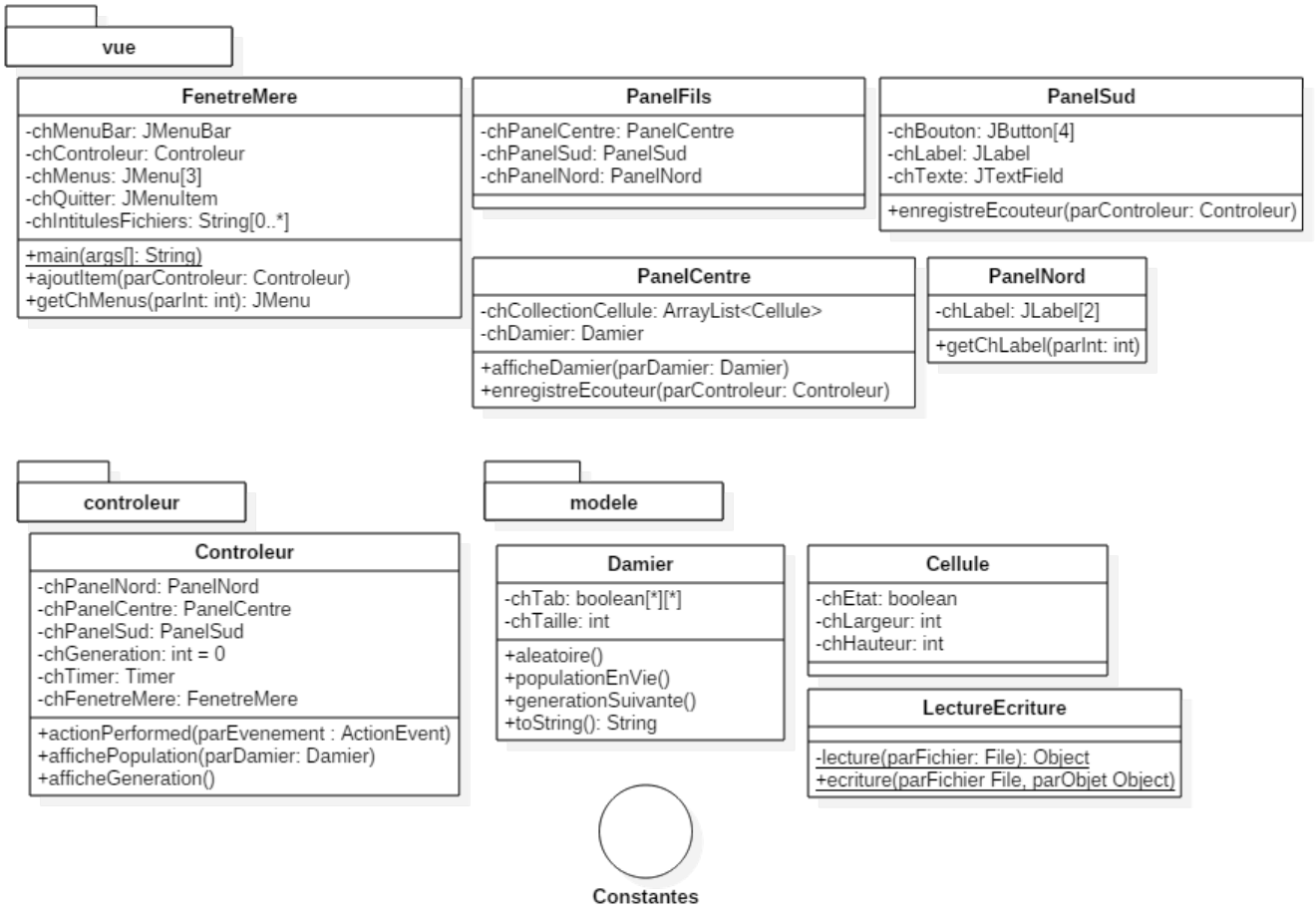


Figure 9 : Diagramme des classes à bas niveau d'abstraction.

La figure 9 est la représentation de la figure 2 à bas niveau d'abstraction. Les associations entre les classes sont représentées par les attributs, non plus par les segments. Si nous prenons comme exemple l'association entre la classe *FenetreMere* et *Controleur* dans la figure 2, dans ce diagramme, elle est représentée par un champ chControleur de type Controleur dans la classe *FenetreMere* et un champ chFenetreMere de type FenetreMere dans la classe *Controleur*.

III. Dossier de tests

L'activité de tests est l'ensemble des approches pour la découverte d'erreurs dans le processus de développement logiciel. Elle a pour objectif de détecter les différences entre le comportement attendu et le comportement observé. Dans l'activité de développement logiciel, on distingue « valider » et « vérifier ». Dans le premier cas, le développeur s'assure que le logiciel répond aux attentes de l'utilisateur, tandis que dans le second cas, il s'assure que le logiciel est conforme à sa spécification.

Dans ce dossier, nous avons conçu et développé des tests unitaires, qui sont effectués par rapport aux artefacts de la conception détaillée. Ils mettent en évidence les défauts d'une partie précise du logiciel. Les tests sont automatisés avec *JUnit*.

Les tests unitaires peuvent être très nombreux, ainsi il est nécessaire de bien choisir les données de test afin de réduire au minimum le nombre de tests à réaliser et de couvrir au mieux les branches du code.

Pour l'ensemble de ce dossier, les tests ont été effectué pour les tailles dites « standards » du damier, soit 25, 35 et 45. Nous n'avons pas testé les valeurs négatives puisqu'un damier ne peut pas avoir une taille inférieure ou égale à 0.

De même, on pose : quels que soient a, x et y trois entiers naturels. De plus, le résultat attendu est supérieur ou égal à 0 et inférieur ou égal au nombre de cellules dans le damier, comme dans le tableau 1.

1. Test de la méthode populationEnVie() dans la classe Damier.

Dans ce test, nous vérifions que le nombre de cellules en vie compté est correct et compris entre 0 et le nombre de cellules dans le damier.

Classe	Nombre de cellules dans le damier	Nombre de cellules en vie	Résultat attendu
P1	25 x 25 = 625	x	0 <= C <= 625
P2	35 x 35 = 1225	x	0 <= C <= 1225
P3	45 x 45 = 2025	x	0 <= C <= 2025

Tableau 1 : Partition d'équivalence du calcul de la population en vie.

Classe	Nombre de cellules dans le damier	Nombre de cellules en vie	Résultat attendu
P1	25 x 25 = 625	x = 204	C = 204
P2	35 x 35 = 1225	x = 351	C = 351
P3	45 x 45 = 2025	x = 906	C = 906

Tableau 2 : Données de test pour le calcul de la population en vie.

III. Dossier de tests

Après éxecution de la méthode de test avec *JUnit*, nous constatons que le test fonctionne correctement.

2. Test de la méthode generationSuivante() dans la classe Damier.

Dans ce test, nous vérifions que le nombre de cellules en vie compté est compris entre 0 et le nombre de cellules dans le damier, puis la méthode testGenerationSuivante() compte le nombre de cellules qui seront en vie à la génération suivante, et celles qui seront mortes. Enfin, la méthode testGenerationSuivante() calcule le nombre de cellules qui seront en vie à la prochaine génération et le compare au nombre de cellules qui le seront selon la méthode generationSuivante().

Classe	Nombre de cellules en vie dans le damier	Nombre de cellules en vie à la génération suivante	Nombre de cellules mortes à la génération suivante	Résultat attendu
P1	0 <= a <= 625	x	y	C = a + x -y
P2	0 <= a <= 1225	x	y	C = a + x -y
P3	0 <= a <= 2025	x	y	C = a + x -y

Tableau 3 : Partition d'équivalence du calcul de la generation suivante.

Classe	Nombre de cellules en vie dans le damier	Nombre de cellules en vie à la génération suivante	Nombre de cellules morte sà la génération suivante	Résultat attendu
P1	a = 313	x = 82	y = 205	C = 190
P2	a = 619	x = 149	y = 398	C = 367
P3	a = 1007	x = 236	y = 641	C = 602

Tableau 4 : Données de test pour le calcul de la generation suivante.

Après éxecution de la méthode de test avec *JUnit*, nous constatons que le test fonctionne correctement.

3. Test de l'ajout d'un fichier dans le répertoire lors de l'enregistrement.

La méthode testAjoutFichier() vérifie que le fichier a bien été ajouté au répertoire prévu à cet effet. Elle compte le nombre de fichier avant l'ajout, puis calcule le nombre de fichiers qu'il devrait avoir après le test, en incrémentant le nombre obtenu prédemment de un.

Classe	Nombre de fichiers avant l'ajout	Résultat attendu
P1	x	C = x + 1

Tableau 5 : Partition d'équivalence de la vérification de l'ajout de fichier dans le répertoire.

III. Dossier de tests

Classe	Nombre de fichiers avant l'ajout	Résultat attendu
P1	x = 2	C = 3

Tableau 6 : Donnée de test pour la vérification de l'ajout du fichier dans le répertoire.

Après éxecution de la méthode de test avec *JUnit*, nous constatons que le test fonctionne correctement.

4. Test de l'ajout d'une cellule lors de la création du damier.

La méthode de test calcule le nombre de cellules que le damier devrait contenir en fonction de la taille de ce dernier. Le nombre de cellules attendu est la taille au carré, puisque le damier est un carré.

Classe	Taille du damier	Résultat attendu
P1	a	$C = a^2$
P2	a	$C = a^2$
P2	a	$C = a^2$

Tableau 7 : Partition d'équivalence de la vérification de l'ajout des cellules au damier.

Classe	Taille du damier	Résultat attendu
P1	a = 25	C = 625
P2	a = 35	C = 1125
P2	a = 45	C = 2025

Tableau 8 : Donnée de test pour la vérification de l'ajout des cellules au damier.

Après éxecution de la méthode de test avec *JUnit*, nous constatons que le test fonctionne correctement.

Les méthodes de tests sont désormais intégrées au code, et seront par conséquent exécutées automatiquement lorsque les méthodes auquelles elles sont référencées seront exécutées.

Jeu de la vie

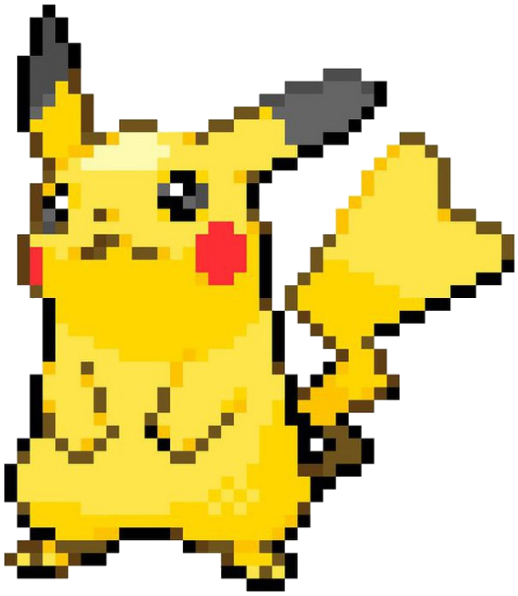
IV. Manuel utilisateur du Jeu de la vie

Histoire

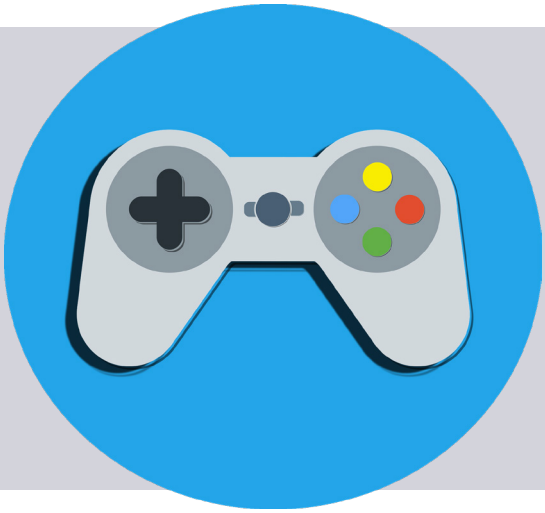
Inventé au Royaume-Uni en 1970 par un professeur de mathématiques à l'université de Cambridge, le jeu de la vie est un jeu mathématique "à zéro joueur". En effet, les cellules du tableau à deux dimensions s'animent par elles-mêmes. C'est pour cela que ce jeu ne se termine jamais.

Lexique

- Une cellule correspond à une case du tableau. Elle peut prendre deux états différents : "morte" ou "en vie". Chaque cellule possède huit voisins, hors cellules en bordure qui n'en possèdent que cinq, ou trois pour les cellules dans les coins.
- Une génération correspond au changement d'état des cellules.



Règles du jeu



Si une cellule a une et une seule cellule voisine, on dit qu'elle meurt par isolement. Si elle possède un nombre de cellules voisines supérieur à trois, il est alors dit qu'elle meurt par étouffement. Et, si une cellule dite "morte" a au moins deux cellules voisines dites "en vie", alors elle "ressuscite".

Ainsi, une cellule qui possède deux ou trois cellules voisines en vie reste dans ce même état.

Fonctionnalités de l'application

L'application comporte un menu déroulant qui permet d'afficher des damiers aléatoires, des exemples de ce qu'il est possible de faire avec ce jeu, ainsi que des damiers vides. Les vaisseaux sont des schémas qui font toujours le même schéma au cours des générations.

Puis, il existe deux étiquettes qui indiquent le numéro de la génération actuelle, ainsi que le nombre de cellules "en vie".

Ensuite, la fenêtre présente des boutons qui permettent respectivement de passer de génération en génération, tel un mode "pas à pas", d'avancer rapidement entre les différentes générations, ou bien de faire une pause.

Enfin, il est possible d'enregistrer le damier dans un fichier extérieur à l'application, afin de pouvoir le retrouver après fermeture de cette dernière.

Conclusion

Bien que le projet soit court, il nous a permis d'apprendre à communiquer et travailler en équipe sur un projet complet. En effet, chaque membre du binôme a participé à la conception du projet, au développement des tests, puis du code et enfin à la rédaction du rapport. De ce fait, nous avons eu l'occasion de mettre en pratique et en relation les connaissances que nous avons acquises dans les cours de conception orientée objet, de programmation orientée objet et d'interface homme machine.

Tout au long de la réalisation de ce projet, nous avons veillé à ce que notre application soit conforme à l'énoncé.

La conception est l'aspect du projet sur lequel nous avons passé le plus de temps, puisqu'il nous permettait d'aborder la phase de développement avec plus d'aisance. Ainsi, la phase de développement ne nous a pas pris beaucoup de temps. Nous avons travaillé en groupe pour faire tout l'aspect de la conception afin de pouvoir travailler chacun de notre côté pour la suite du projet.

Cependant, la conception et le développement des tests a été la phase la plus compliquée. Nous avons eu beaucoup de mal à convertir nos idées en test. Bien que nous ayons des idées sur les méthodes à faire, nous n'arrivions pas à les mettre en œuvre.

Nous nous sommes répartis les tâches au début du projet dans le but d'optimiser au mieux le temps que nous avions. Il s'est avéré que certaines tâches ont été plus compliquées que prévu, et nous ont ainsi demandé plus de temps. Nous avons donc opté pour un programme plus souple, pour lequel nous répartissions les tâches du jour pour le lendemain par messages ou encore messages instantanés. De même, nous tenions le second membre du binôme au courant de ce que nous avions fait, des modifications que nous avions apporté. L'utilisation de *Git* s'est révélée compliquée, c'est pourquoi nous avons utilisé un drive pour partager tous les documents relatifs au projet et toutes leurs versions. Bien évidemment, nous avons utilisé *Eclipse* pour coder l'application.