POO – projet final – PetSaver compte-rendu

Cécile Guitel
Valentina Fedchenko

Principes et fonctionnement

Modèle/vue/contrôleur et généricité

(Voir le schéma des interactions VMC en annexe et les schémas des classes. On notera que les schémas des classes contiennent quelques erreurs dues aux bugs de l'outil utilisé : les noms de classes ne devraient jamais être soulignés, et la relation entre lanceur et vue devrait être une relation d'agrégation et non de composition.)

Vue

La vue est représentée par une classe abstraite, dont héritent une version graphique (IG, en vert) et une version terminale (IT, en violet). Nous avions commencé par des interfaces, mais avons vite basculé sur des classes abstraites.

En effet, nous voulons définir le fonctionnement interne de la classe, pas la façon dont elle est utilisée par d'autres objets. Cette généricité permet de garder le même code pour les parties modèle et contrôleur, qui n'ont pas besoin de savoir si leur vue est une VueIG ou une VueIT.

Modèle

Nous avons séparé le modèle en deux parties : le lanceur (l'environnement, le menu initialisation) et la partie (le menu jouer). Nous aurions pu mettre la partie en attribut du lanceur et passer toujours par le lanceur, mais cela aurait voulu dire rajouter une étape systématique dans l'exécution de toutes les fonctions du menu jouer : contrôleur → lanceur → partie → (plateau →)vue.

Entre un système simple plus compacte vu de l'extérieur mais plus complexe à l'intérieur, et un système double mieux délimité et plus clair qui comporterait deux boucles VMC complètes, nous n'avons pas encore pu prendre de décision. C'est là le plus grand problème de notre programme.

Contrôleur

Le contrôleur est la partie la plus simple, bien qu'il soit passé par de nombreuses versions dans l'évolution de notre programme (version système/communication visible dans notre premier brouillon de structure dans le fichier 'restes d'anciennes versions', classe hybride VueControleurTerminal terminale qui implémentait à la fois Vue et Controleur…). Son rôle principal est de relayer les commandes entre la vue et le système.

Déroulement du jeu :

(Voir le schéma du déroulement du jeu en annexe.)

Principe
Le jeu se fait en deux étapes, l'initialisation (l'environnement, Lanceur, menuInitialisation) puis la partie (Partie, menuJouer). En IT, ces étapes sont gérées entièrement par les deux fonctions menu. En IG, ces deux fonctions créent deux cartes dans un CardLayout avec des Listeners qui correspondent aux options des menus.

Initialisation
Dans le menu d'initialisation, on peut voir les règles et gérer les sauvegardes des joueurs, sauvegardés par sérialisation. Il manque une option pour supprimer les sauvegardes. Une fois un joueur choisi ou créé, une partie est créée et passée au contrôleur, et le menu jouer est lancé.

Jeu
La partie gère un historique de plateaux créés par clonage, ce qui permet d'annuler les coups. L'annulation est définitive.

Les coups (cliquer sur un bloc ou utiliser une fusée) sont gérés par la classe Plateau. MenuJouer passe les instructions à la partie, qui les passe ensuite au plateau.

Fin de partie
Quand l'utilisateur a gagné, le niveau du joueur augmente et est sauvegardé, la partie est supprimée, et on retombe sur le menu d'initialisation. Il faut donc re-choisir le même joueur pour continuer vers le niveau suivant.

Fonctionnement du plateau :
Pour gérer les bords du tableau, on a d'abord ajouté une bordure de cases null. Ensuite, on a opté pour des blocs try/catch qui permettent de gérer les exceptions IndexOutOfBound. Cela facilite l'écriture des boucles de parcours du tableau et allège les conditions.

On a voulu garder la nomenclature de x et y ainsi que l'origine en bas à gauche, afin de faciliter le repérage. C'est un choix personnel : on aurait aussi bien pu parler de hauteur et de largeur, et mettre l'origine dans un autre coin. Notre choix facilite l'écriture des fonctions comme gravite(), avec des boucles d'incrémentation positives, mais on doit alors renverser l'ordre de parcours pour d'autres fonctions, comme par exemple ajouteAnimaux(), et pour générer l'affichage.

Pour introduire un élément de hasard, nous utilisons un objet Random. Cela suffit amplement pour ajouteAnimaux. Cependant, il aurait été intéressant de pouvoir contrôler la probabilité de deux blocs voisins de même couleur. Nous avions pensé à utiliser une distribution de probabilité pour cela, mais n'avons pas trouvé d'outils pour le faire. Nous avons donc décidé de gérer ce facteur de difficulté grâce au nombre de couleurs.

Nous avons fait le choix que la taille minimale d'un groupe de blocs voisins de même couleur pour pouvoir le détruire soit 2. Nous avons aussi écrit une fonction pour si le minimum était de 3, sauvegardée dans le fichier blocDestructible3.

Fonctionnement des interfaces visuelles :
Chaque vue a deux fonctions principales, menuInitialisation et menuJouer, qui correspondent aux deux étapes du programme.

Tous les éléments graphiques importés de fichiers ont été adaptés d’OpenMoji.

VueIT
La vue textuelle utilise un scanner ainsi que des fonctions auxiliaires (demandeIntCheck et autres) afin de demander des informations à l’utilisateur.

Elle affiche les éléments grâce aux fonctions toString des classes du modèle, au lieu d’utiliser des éléments d’autres classes comme VueIG. Elle est donc beaucoup plus compacte.

Le fait que la conversion d’objet du modèle à objet affichable se passe dans le modèle n’est pas idéal pour notre séparation en modèle-vue-contrôleur. Si l’on avait eu plus de temps et plus de connaissances, on aurait probablement créé un principe d’affichage générique, dont le fonctionnement serait calqué sur la structure des classes de l’interface graphique ainsi que la structure du modèle. On aurait ensuite déplacé les fonctionnalités toString des éléments du modèle vers ces nouvelles classes. Cependant, cette structure serait-elle compatible avec toute vue imaginable ?

VueIG
La vue graphique, elle, utilise un système de CardLayout qui remplit toute la fenêtre, avec une carte pour l’initialisation et une carte pour le jeu (CarteMenuInitialisation, gérée par menuInitialisation, et CarteMenuJouer, gérée par menuJouer). Ces cartes sont définies par la classe abstraite mère CarteMenu, et sont séparées en deux zones : à gauche, des boutons et des labels, à droite, un panel pour le contenu. La classe mère met à disposition les fonctions nécessaires pour gérer les éléments qui ne sont pas communs aux deux cartes.

Une de ces fonctions, setContenuDroite, s’est avérée nécessaire car nous avons remarqué que si l'on essayait d’attribuer le contenu directement (attributContenu = nouveauContenu), l'affichage n'était pas mis à jour. La solution que l'on a trouvée est d’ajouter le contenu à un panel préexistant (attributConteneur.add(nouveauContenu)). Il semblerait que la méthode add relance l’affichage, alors qu’appeler validate/revalidate/repaint à cet endroit ne le fait pas. Puisque add le fait, il devrait aussi être possible pour nous de le faire, mais utiliser add donne le résultat escompté.

CarteMenuInitialisation
Une carte menu d’initialisation a pour contenu à sa droite un autre GridLayout avec deux cartes, carteRegles et carteChoisirJoueur. On a cette fois-ci choisit d'utiliser des classes internes, puisque ces cartes ne sont censées être utilisées qu'avec ce menu/cette classe englobante là, mais la longueur du fichier justifierait probablement de les externaliser. Il existe probablement des outils pour mieux gérer les fichiers, et nous avions envisagé d’utiliser des packages, mais notre manque de connaissances sur le sujet ne nous l’a pas permis.

CarteMenuJouer et ContenuPlateau
Une carte menu jouer a pour contenu à sa droite un panel pour afficher le plateau. Celui-ci est géré grâce à une classe interne (ContenuPlateau) qui affiche les éléments grâce à un GridBagLayout. Quand l’utilisateur clique sur le bloc, il active le listener du panel au lieu d’un listener propre au panel du bloc.

Les deux méthodes sont possibles, mais comme on enlève et recrée tous ces petits panels à chaque mise à jour du plateau, il nous a paru plus économe de ne pas aussi avoir à recréer des listeners à

chaque fois. Cependant, si l’on voulait pouvoir afficher sur l’interface quelle case est cliquée (ou quelle colonne si l’on utilise une fusée), l’autre version serait probablement préférable, à moins que l’on puisse récupérer l’élément grâce à ses coordonnées dans le GridBagLayout.

La création du contenuPlateau est séparée de son remplissage, parce qu'on a besoin qu'il soit mis en place dans la carte menu et dans la fenêtre pour pouvoir calculer ses dimensions avec getWidth et getHeight. Il faut donc appeler setContenuDroite avant cela, et lancer setUpPlateau depuis la classe qui ajoute la carte, ici, VueIG.

L’idée de tailleBlocs, yMaxVisible et xMax est de garder la fonctionalité d’un plateau à largeur variable. Il faut donc savoir la largeur des blocs pour savoir leur hauteur et donc pouvoir calculer le nombre de blocs visibles dans une colonne.

On priorise ainsi la forme carrée des blocs, même si on choisit de changer la taille de la fenêtre dans VueIG (et tant que l'on ne la redimensionne pas après création) plutôt que le nombre constant de blocs visibles sur une colonne. Cependant, ce nombre de blocs visibles est fixé à 10 dans l’interface textuelle. Vu la complexité du procédé, le peu d’intérêt du rendu, et le décalage entre les versions, cette décision n’est pas la bonne.

Pour que les colonnes et lignes vides restent vides, on introduit des blocs invisibles.

Reste à faire

Problèmes répertoriés

Démo
Cette fonctionnalité n’a pas été implémentée par manque de temps.

Si on l’avait implémenté, il nous aurait fallu créer un programme qui prend la place du contrôleur pendant la partie menuJouer du jeu. Il aurait fallu un moyen de ralentir les opérations pour donner le temps à l’utilisateur de les voir. Il aurait aussi été utile de pouvoir afficher les actions choisies par le robot.

Suppression des joueurs
Les joueurs créés ne sont pas supprimables depuis l’interface et risquent de s’accumuler. Le seul moyen de les supprimer est de supprimer les fichiers directement. Or, l’utilisateur lambda n’a pas à avoir à trifouiller dans les fichiers du jeu.

Il faudrait ajouter une fonction de suppression dans la classe lanceur, qui prendrait en paramètre le nom du joueur. Ensuite, dans la vue, offrir l’option dans le menu de choix du joueur, et récupérer le nom de la sauvegarde à supprimer. Ce nom serait communiqué au contrôleur qui appellerait la fonction du lanceur.

Conversion élément modèle → élément d’affichage
Voir plus haut, « VueIT ».

Uniformisation des interactions VMC
La façon dont l’information passe d’une partie à l’autre du système VMC n’est pas complètement standardisée. Parfois, on préfère passer par une classe « tête » qui répercute les instructions vers ses

composants et de ses composants vers le reste du système (Partie, par exemple, gère à la fois l'information sortante (quand mettre à jour la vue) et entrante (les actions de l'utilisateur)), parfois, les composants se chargent eux-mêmes d'envoyer des commandes (les différentes cartes menu gardent la référence du contrôleur et lui transmettent les choix de l'utilisateur).

Cela est dû au fait que la différence entre CarteMenuInitialisation et carteMenuJouer est la même que la différence entre Lanceur et Partie, ou menuInitialisation et menuJouer : il s'agit de deux systèmes différents qui alternent mais ne se chevauchent pas. (Voir plus haut dans la partie « modèle » du compte-rendu.) Il faudrait faire un choix clair et s'y tenir.

Si l'on décidait de faire deux systèmes VMC, on pourrait imaginer créer une nouvelle fenêtre au lieu d'une nouvelle carte dans la même fenêtre. Un système de fenêtres multiples permettrait aussi d'implémenter la fonction bravo dans VueIG (voir plus bas, « Fin de partie et messages à l'utilisateur en IG »).

Modificateurs d'accès
À cause de l'héritage des classes abstraites, on est obligés de mettre en protected des fonctions qu'on aurait voulues private à chaque classe fille. Comment faire pour garder la restriction tout en permettant l'héritage ?

De même, on aurait aimé pouvoir restreindre cliqueBloc et utiliseFusee dans Plateau, qui ne sont censées être appelées que par les fonctions du même nom dans Partie. La solution la plus logique aurait peut-être été d'en faire une classe interne, mais ces deux fichiers sont déjà trop longs. Scinder le programme en deux systèmes VMC règlerait aussi ce problème. Quelle serait la meilleure pratique ?

En attendant, nous avons déclaré protected les méthodes dont on aurait voulu restreindre l'accès.

Uniformisation du style de l'IG
Il serait très utile d'avoir un UImanager ou des classes auxiliaires ou quelque chose qui permette de gérer un style au lieu de copier-coller les mêmes lignes de setBackground et setFont. On a fait un pas dans cette voie avec CarteContenu, la petite classe abstraite interne à CarteMenuInitialisation, mais il faudrait élargir le principe à toute la vue graphique.

Dimensions des PanelAnimal
Les PanelAnimal, au contraire des PanelBloc, ne peuvent pas être redimensionnés. Ils ne s'adaptent pas à la taille de la case. Il semblerait qu'il soit compliqué de rendre redimensionnable les ImageIcon, auquel cas une autre implémentation pourrait être envisagée.

Améliorations possibles :

Variabilité du plateau et de la partie
La largeur du plateau est limitée à 10 pièces à cause de l'affichage sur terminal, qui utilise des chiffres pour numéroter l'abscisse. On aurait pu imaginer utiliser des lettres pour les blocs et des chiffres pour les animaux. Dans les faits, notre code ne générera pas de plateau avec xmax < 10, mais on a laissé la possibilité pour les tests et pour une potentielle modification future.

Le nombre de couleurs, d'animaux et la hauteur du plateau augmentent de manière linéaire avant d'atteindre soudainement une limite (cf la fonction Partie.plateauSelonNiveau). Il serait plus intéressant d'avoir une courbe qui tend progressivement vers cette limite.

Le nombre de coups total et de fusées par partie sont constants. Il serait plus intéressant de les rendre variables d'une manière ou d'une autre.

Générer aléatoirement un nouveau plateau en calculant ses caractéristiques avec le nombre du niveau marche bien, mais cela veut dire que les plateaux sont assez semblables. On pourrait implémenter un certain nombre de premiers niveaux pré-faits à désérialisé, puis une fois le dernier niveau atteint, générer les plateaux comme on le fait maintenant.

Score
Le score est une fonctionnalité très importante de tout jeu. Avec un peu plus de temps, on l'aurait implémentée en rajoutant un attribut à Joueur, et en permettant à l'utilisateur d'acheter des fusées avec les points gagnés.

Visuels
Les blocs ne sont pas très jolis, et le nombre d'espèces d'animaux est fixé à 6. On pourrait rajouter plus d'animaux en utilisant d'autres emojis (il en reste beaucoup dans le dossier openmoji-svg-color) et revoir la classe PanelBloc. Rien qu'arrondir les coins serait une bonne amélioration. Un brouillon utilisant Graphics2D se trouve dans le fichier BlocPicture.

À l'heure actuelle, les marges entre blocs sont créées avec des bordures colorées de la couleur du fond. Nous n'avons pas encore réussi à rajouter des marges avec GridBagLayout, mais cela doit être possible. On pourrait ensuite utiliser la fonction setBorder des blocs pour mettre de vraies bordures.

Les éléments graphiques étaient en format svg avant d'être exportés en png afin de pouvoir les utiliser dans le jeu. C'est dommage, et il existe probablement un moyen d'afficher des images vectorielles avec Java.

Mauvais coups et annulations
Pour gérer les coups impossibles, on utilise des catch/try dans les fonctions du Plateau ainsi que des conditions if à plusieurs endroits. Ça marche, mais ce n'est pas forcément la solution la plus élégante.

Le vrai problème est que l'on ne fait pas la différence entre un coup qui marche et un coup qui n'est pas possible ou qui ne fait rien. Les deux sont comptés comme des coups exécutés. On pourrait imaginer utiliser un système d'exceptions pour gérer les mauvais coups, et/ou retourner un booléen vrai si le coup marche, faux sinon, ce qui permettrait de ne pas les comptabiliser et rendrait obsolète ces conditions dans VueIT.

Fin de partie et messages à l'utilisateur en IG
Il faudrait proposer au joueur de continuer sur le niveau suivant au lieu de retomber sur le menu d'initialisation.

La fonction bravo ne fait rien dans la version graphique, et le retour à l'initialisation est particulièrement abrupte. Il serait intéressant de pouvoir créer des fenêtres popup pour communiquer des informations avec le joueur (comme, par exemple, « vous avez gagné ! »). Cela

nécessiterait probablement une révision de la classe VueIG afin de pouvoir ajouter un élément sur les autres, chose que l'on n'a pas réussi à faire pour ajouter le fond derrière les options du menu.

Une autre solution serait de créer un autre type de carte menu, une carte message, avec la seule option de revenir à la carte jouer si la partie n'est pas finie, ou de continuer vers initialisation sinon. On pourrait prendre en constructeur le nom de la carte sur laquelle revenir et le message à afficher.

Une troisième solution serait d'ouvrir une nouvelle fenêtre.

Choix du niveau
On pourrait imaginer demander au joueur quel niveau il veut jouer, avec une limite posée par les niveaux atteints. Cela nécessiterait de vérifier si le niveau joué est le même que le niveau atteint avant d'incrémenter le niveau du joueur, mais cela ne nécessiterait pas de modifications plus profondes du code.

Colonnes vides
En l'état actuel, lorsqu'une colonne est vide, les colonnes à sa droite ne se déplacent pas pour venir se coller aux autres colonnes. Cela rend la destruction des blocs aux bords du plateau délicate.

Pour changer cela, il faudrait ajouter une seconde fonction de gravité au plateau qui déplacerait les colonnes (si possible) après la gravité normale.

Récapitulatif

Notre jeu Pet Saver utilise le principe modèle-vue-contrôleur. Il implémente une interface graphique et une interface terminale. La vue est générique pour le reste du programme grâce à une classe abstraite. Les structures des deux vues ne sont pas génériques entre elles.

La structure VMC forme bien une boucle à sens unique, mais elle correspond en réalité à deux boucles : une pour l'environnement et une pour la partie. Le modèle est représenté par deux classes différentes pour refléter cela, mais le contrôleur et la vue n'ont pas été scindés en deux. Cette question reste ouverte.

L'environnement permet d'afficher les règles, de gérer les sauvegardes des joueurs et de lancer une partie. La partie montre un plateau où l'on peut détruire des groupes de blocs. Le plateau se réorganise automatiquement. Les colonnes vides n'entraine pas de réorganisation du plateau. Une fois les animaux en bas du plateau, ils sont sauvés. Une fois tous les animaux sauvés, la partie est gagnée et la progression du joueur est sauvegardée.

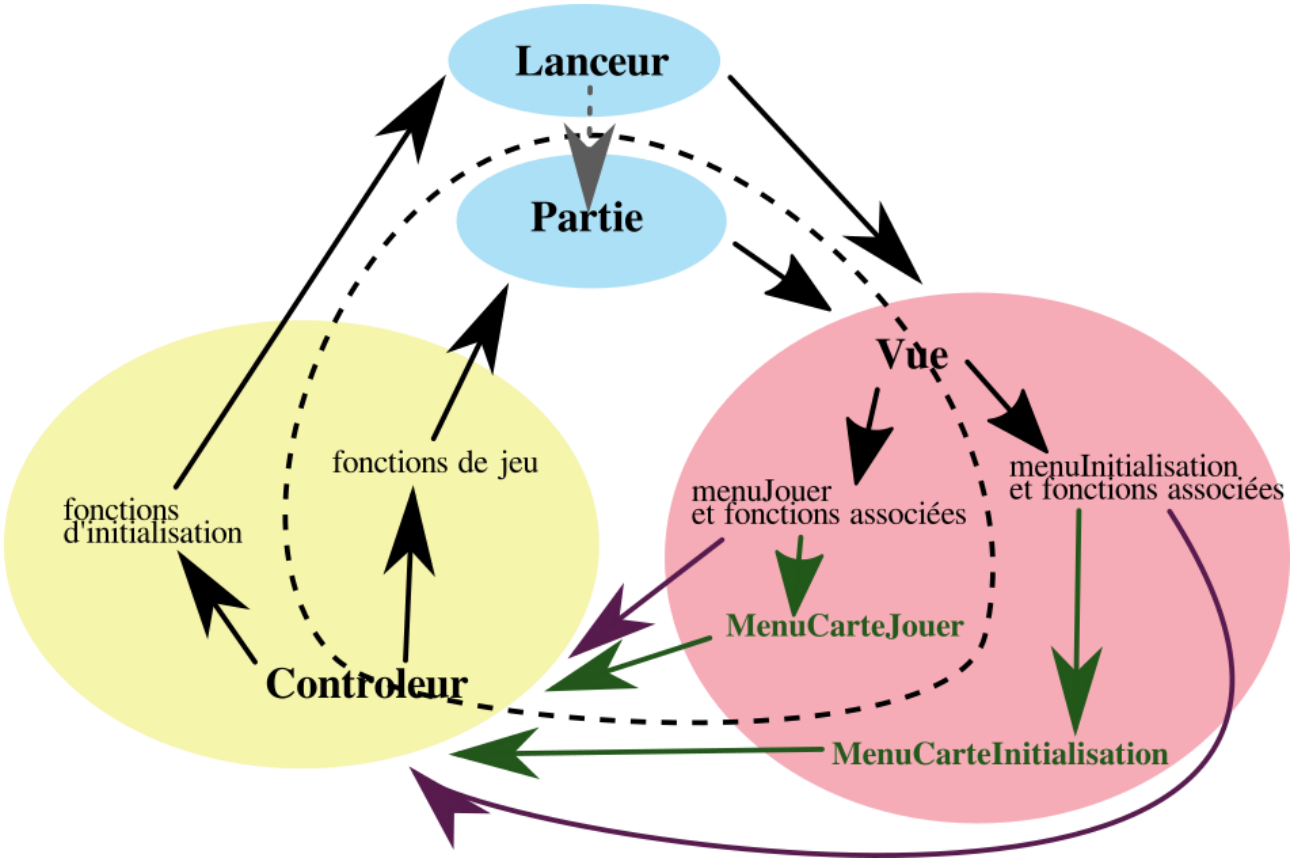
Il est possible d'annuler les coups joués. La difficulté des niveaux est croisante et gérée par des paramètres du plateau calculés selon le niveau. Le plateau est alors généré aléatoirement et non pas chargé depuis un fichier. Cela veut dire que les plateaux ne sont pas fixes ni très différents les uns des autres, mais aussi que l'on ne risque pas d'atteindre la fin du jeu. Il n'y a pas d'obstacles fixes.

Il n'y a pas de joueur robot. L'option démo est présente, mais la fonction ne fait rien.

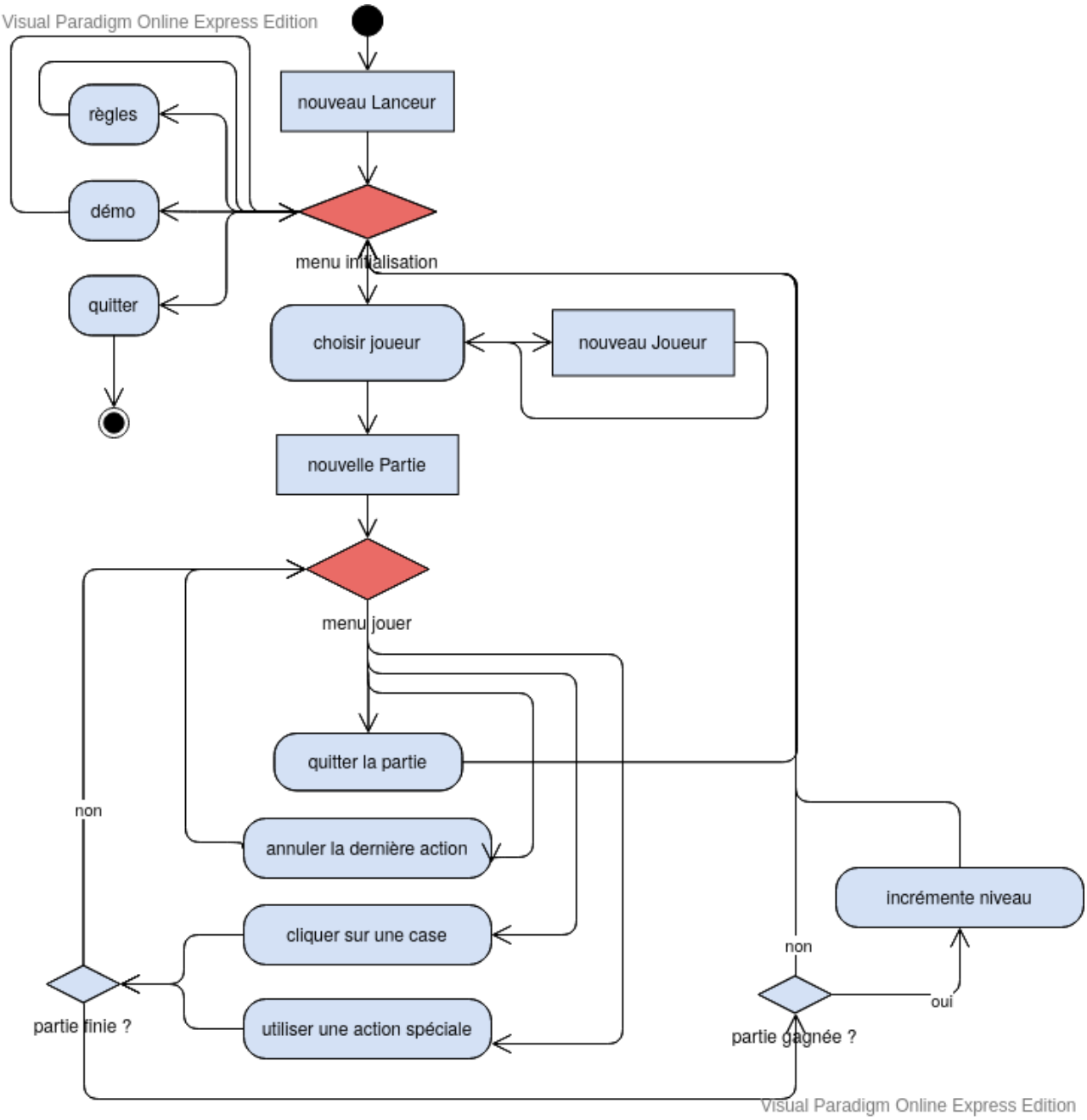
Annexes

Fonctionnement du jeu

Interactions VMC (visualisation/VMC.png)

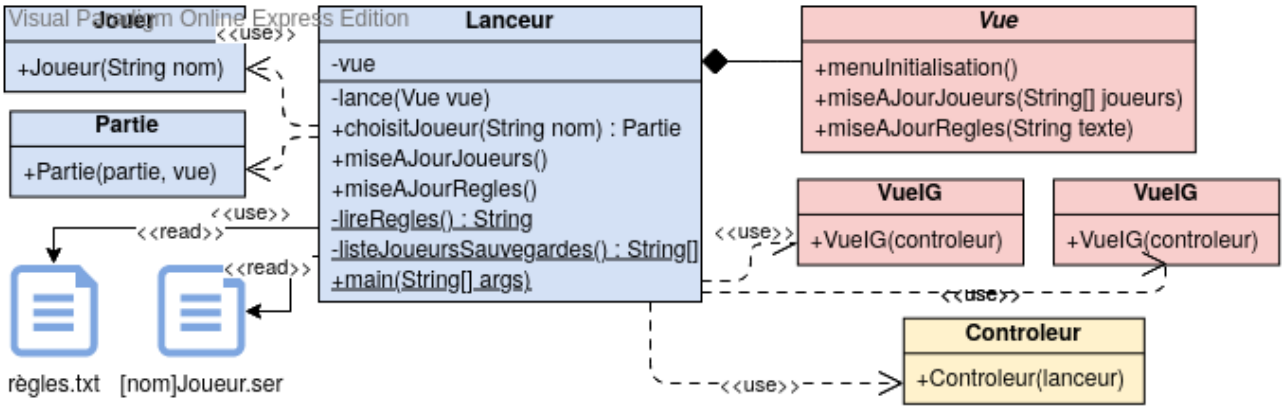


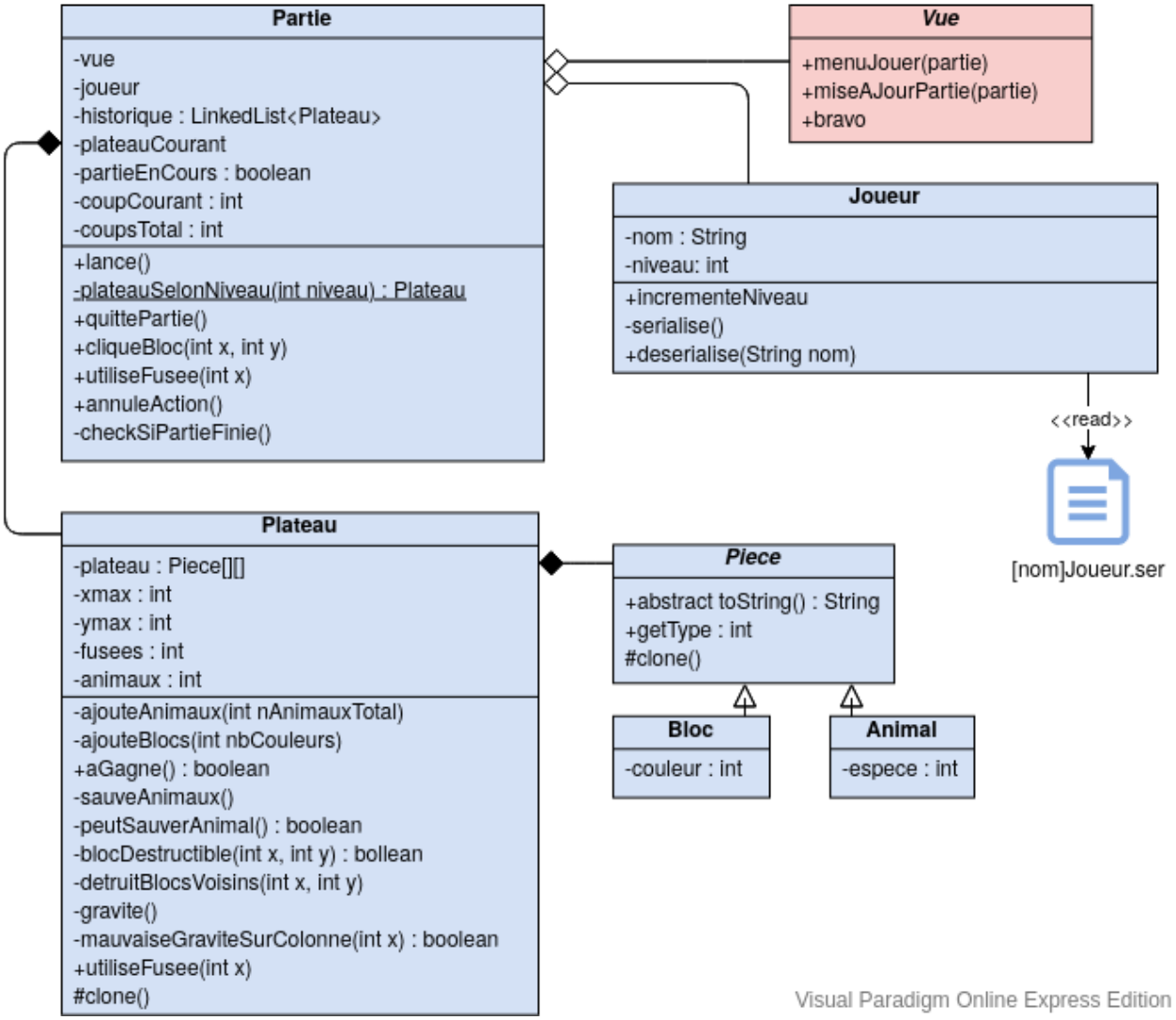
Déroulement du jeu (visualisation/déroulement.png)



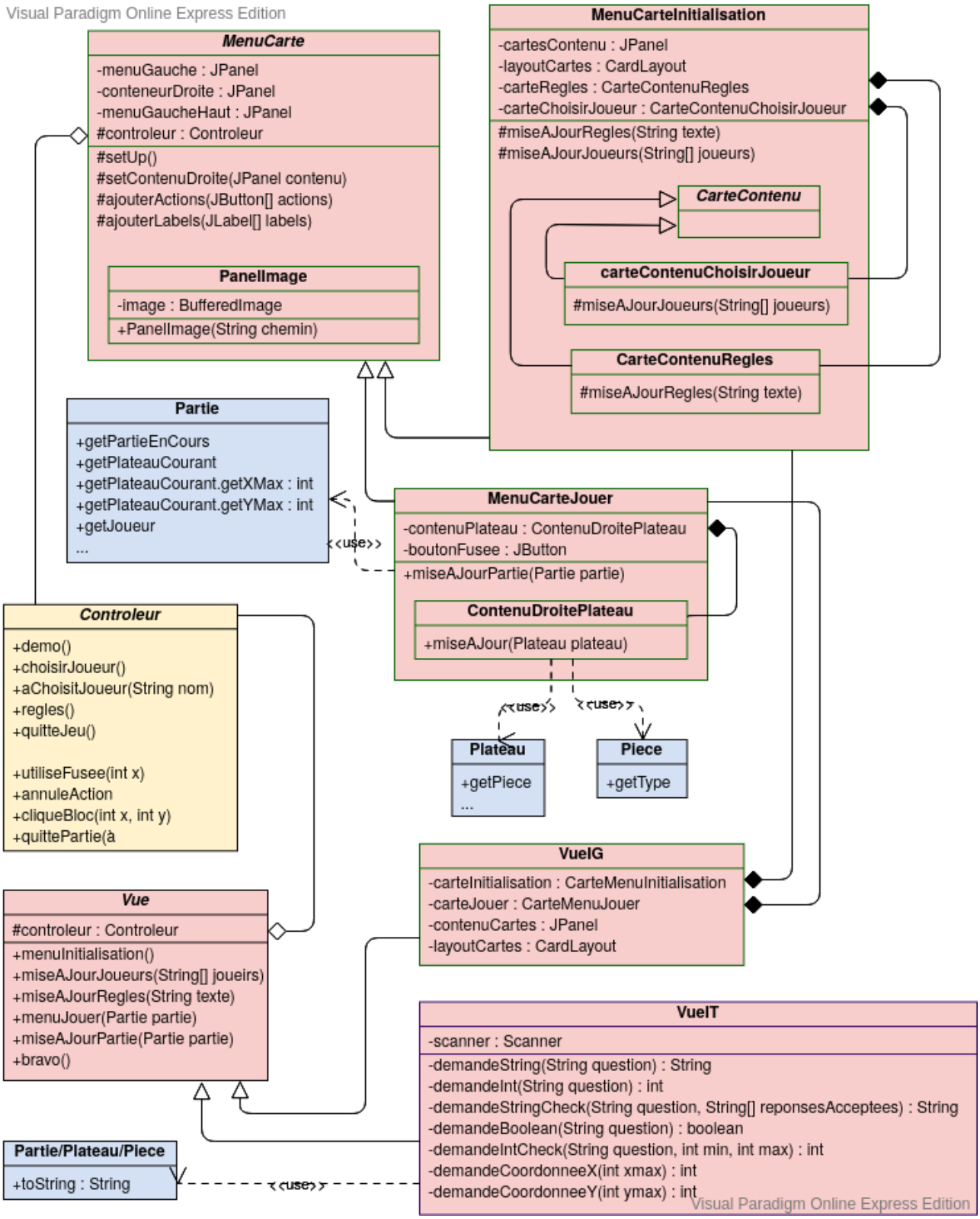
Classes

Modèle (visualisation/classes modèle.png)





Vue (visualisation/classes vue.png)



Contrôleur (visualisation/classes controleur.png)