Soutenance 'Bob Rescue "
POO

1. Comprendre les différentes parties du jeu

Tout d'abord, il a fallu séparer les différentes parties du jeu afin de l'implémenter de la meilleure façon.
Pour ce faire, on a créé différents packages qui représentent chacune de ces parties.
En effet, notre projet comporte 5 répertoires.

D'une part, on a le package « **modèle** ».
Il permet de regrouper notre plateau de jeu qui est composé de blocs.
Les blocs sont de natures différentes :

- -obstacle => .class => *BlocObstacle.java*
- -couleur => .class => *BlocCouleur.java*
- -animal => .class => *BlocAnimal.java*
- -bloc vide => .class => *BlocVide.java*

Chaque type de bloc hérite de la classe *Bloc.java*.

Par ailleurs, on y retrouve notre plateau de jeu avec la classe *Plateau.java*.
Cette classe y donne la représentation de notre plateau.
Ce dernier est composé de blocs (présentés ci-dessus).
Sémantiquement il s'agit d'un tableau bi-dimensionnel de blocs (voir la classe *Bloc.java*).
Chaque plateau possède ainsi un niveau de difficulté.
C'est pourquoi on a décidé d'implémenter la classe *Niveau.java*.
Cette classe permet de récupérer le niveau actuel du jeu afin de lancer la partie adaptée.
En effet, plus le niveau devient élevé et plus le nombre de chats à sauver augmente. De plus, le nombre de blocs de couleurs ainsi que la répartition générale des blocs diffèrent selon le niveau en question.
Évidemment cette classe permet de récupérer la sauvegarde des niveaux sur le disque grâce à la classe *Serialise.java* située dans le package « **sNiveau** » (voir suite).

D'autre part, on peut trouver le package « **sNiveau** ».
Dans ce package se situe la classe *Serialise.java*.
Elle permet de sérialiser chaque niveau afin de les stocker dans un répertoire nommé <NiveauxStockes> de notre disque.
Ainsi, lorsque notre classe *Niveau.java* (située dans le modèle) aura la volonté de lancer la partie adéquate (et donc corrélée avec le bon niveau), elle devra faire l'opération inverse, c'est à dire déserialiser le niveau stocké dans le répertoire <NiveauxStockes>.

Par la suite, on a créé le package « **vue** ». Ce package regroupe toutes les classes liées à l'aspect du jeu.
On a ainsi réutilisé les connaissances apprises sur les interfaces graphiques afin de modéliser au mieux le moteur de jeu.
La classe *Vue.java* permet de démarrer le jeu en affichant une fenêtre principale du jeu « **BOB RESCUE** ».
Pour ce faire, nous avons créé trois panneaux :

- -panneau général => .class => *PanneauGeneral.java*
- -panneau de jeu => .class => *PanneauDeJeu.java*
- -panneau de joueur => .class => *PanneauJoueur.java*

Le panneau général s'occupe de créer le panneau représentant le menu.
Ainsi, on y retrouve trois boutons principaux qui permettent de jouer, de quitter ou de voir les règles du jeu.
Par ailleurs, on a décidé d'y associer des évènements d'écoute.
En effet, lorsque l'utilisateur clique sur le bouton « JOUER », une boite de dialogue s'affiche et lui demande son nom de joueur. (ce nom nous permettra d'afficher le nombre de points à la fin du jeu, voir suite).
Enfin, ce bouton nous permettra de lancer l'interface du jeu avec le panneau de jeu.
Cela est réalisé grâce à l'évènement d'écoute *Mouse Listener.*
De plus, une seconde boite de dialogue propose les niveaux voulus. Le client a le choix de choisir son niveau.
Le jeu se lance ainsi avec l'aide de la fonction *lancePartie(option).* L'option correspond ici au niveau choisi par l'utilisateur.
Dans cette fonction, est instanciée notre nouveau panneau de jeu avec le niveau adéquat et le joueur en question.

Ainsi le lancement du plateau graphique intervient grâce à la classe *PanneauDeJeu.java*.

La représentation graphique de notre plateau est effectuée avec des images provenant du package « **pics** ».

En effet, ces images récupérées grâce aux fonctions situées dans la classe *Images.java*, sont utilisées pour illustrer nos blocs et notre plateau.
Ainsi l'évènement clic associé à notre plateau nous donne la possibilité de réorganiser notre jeu avec les suppressions logiques correspondant aux règles du jeu. (voir 2.fonctionnement du jeu).

Comme dit précédemment, on récupère nos images situées dans notre dossier <*imgEXT*> avec l'aide de la classe *Images.java* située dans le package « **pics** ».
Elle utilise la classe *ImageIo* native de JAVA permettant ainsi la lecture de fichiers images directement grâce à un chemin de notre arborescence.

En définitive, le package qui nous permet de faire le lien entre le modèle et la vue est le « **contrôleur** ».
Dans la classe *Controleur.java*, on y implémente la possibilité de lancer le jeu en mode graphique ou en mode textuel.
L'utilisateur a aussi la possibilité de jouer contre un robot s'il en a l'envie.
Ainsi cette classe a pour principal but de lancer la vue associée et de charger indirectement tous les modèles composant le jeu.

2. Fonctionnement du jeu

Le but premier de jeu est de sauver des « Bobs » en les ramenant tout en bas du plateau.
Ainsi cela est modélise dans notre jeu par notre plateau qui contiendra des blocs de couleurs, des blocs d'obstacles en fonction du niveau ainsi que des « Bobs » évidemment.

La classe Plateau.java contient les méthodes qui vont permettre la réorganisation du jeu.
On y retrouve les fonctions principales :

- ➔ testGroupeVide(x,y)
- ➔ videGroupe(x,y)
- ➔ rearrangement() : vertical()
  horizontal()

Tout d'abord, la fonction *horsLimite(int x, int y)* permet de nous indiquer si on se trouve encore dans les bornes du plateau.
La méthode *testGroupeVide (int x, int y)* nous permet de savoir si on peut regroupe par groupe plusieurs blocs de même couleur.
Pour ce faire, on a créé la fonction *supprime(x,y,bloc).*
Cette fonction permet de nous dire si le bloc actuel est bien de la même couleur que le bloc voisin.
Pour tester tous les voisins, on appelle *supprime(x_voisin,y,bloc)* et *supprime(x,y_voisin,bloc)* dans *testGroupeVide (int x, int y).*
Ainsi si on a au moins 1 voisins de même couleur que le bloc couleur actuel, alors on appelle *videGroupe()* et on réarrange le plateau avec la méthode *rearrangement().*
Avant de réarranger le plateau, il faut d'abord vider les cases de même couleur afin de le faire disparaître.
C'est ce que la méthode *videGroupe(int x, int y)* se charge de faire.
Dans un premier temps, on créer un bloc vide sur le bloc actuel avec la méthode *creerBlocVide()* et on fait de même pour tous les voisins de même couleur. Pour réaliser au mieux ceci,
On utilise la récursivité et on rappelle ainsi la même méthode avec les indices des blocs voisins en argument.
Enfin, on peut maintenant réarranger le plateau avec la méthode *rearrangement().*
Cette dernière contient deux sous-méthodes :

- *vertical()* => réarrangement vertical
- *horizontal()* => réarrangement horizontal

La méthode *vertical()* teste si on peut trouver un bloc vide et si oui grâce à notre index on s'arrête lorsque le bloc est non vide (i.e il contient un bloc couleur ou Bob). Ainsi on peut créer un bloc vide et réorganiser notre plateau en changeant la position du bloc couleur ou Bob.

Par ailleurs, la méthode *horizontal()* prend un booléen doitBouger qui indique si on doit faire un changement en bougeant des blocs.
Ainsi si on trouve sur une même ligne des blocs vides, on peut réarranger notre plateau.
Dans cette méthode, on appelle vertical() pour chaque colonne afin d'optimiser la réorganisation.

La fonction *attrapeBob ()* permet de savoir si « Bob » se trouve tout en haut du plateau.
Ainsi si cette fonction renvoie «true», on a la fonction *bobSecouru()* qui va créer un bloc vide et augmenter le nombre de « Bobs » sauvés grâce à un compteur.

De cette manière, on peut créer une fonction *resultatVictoire()* qui renverra «true» si et seulement si le nombre de « Bobs » sauvés est supérieur au nombre constant de « Bobs » à sauver de chaque niveau.
D'un autre cote, pour savoir si le client a perdu, on a la fonction *resultatDefaite()* qui parcourt notre plateau et qui teste si on peut supprimer des blocs continus. Si on en trouve aucun, alors le client a perdu.

Afin de lancer la partie adaptée, il faut se concentrer sur la classe *Niveau.java*.
En effet, cette classe manipule le niveau actuel de la partie et lance la partie adaptée.
On y retrouve trois organisations différentes en fonction des niveaux :

- ➔ *organisation1()*
- ➔ *organisation2()*
- ➔ *organisation3()*
- ➔ *organisation4()*
- ➔ *organisation5()*
- ➔ *organisation6()*
- ➔ *organisation7()*
- ➔ *organisation8()*
- ➔ *organisation9()*
- ➔ *organisation10()*

La fonction *lancePlateau()* récupère le niveau actuel et lance la disposition adéquate.
Plus le niveau s'élève, plus le nombre de « Bobs » à sauver va augmenter et la disposition diffère évidemment.

Maintenant, pour lancer une partie et pouvoir jouer, il faut s'attacher à la vue.
Dans ce package se trouve l'aspect du jeu et les interfaces graphiques.
La classe *Vue.java* qui dérive de **JFrame**, peut ainsi crée une fenêtre de jeu avec le titre « BOB RESCUE GAME ».
Cette fenêtre possède la dimension de 1320x1080. *(setSize())*
De là, la vue interagit indirectement avec nos trois panneaux qui sont plus ou moins liés entre eux.
Dans *Vue.java*, le panneau général est instancié afin d'obtenir les 3 boutons et l'image principale qu'il contient.
De plus, une musique de fond se lance lorsque le jeu démarre. En effet, on a créé une classe *Sons.java* qui nous permet d'avoir des méthodes statiques afin de les appeler dans notre jeu facilement.
La méthode *joueMusique()* de la classe *Sons.java* lance notre son de fond dans *Vue.java*.

Les classes *PanneauGeneral.java* et *PanneauDeJeu.java* représentant des panneaux, elles dérivent toutes de **JPanel**. Ainsi pour rendre visible le contenu, il est nécessaire d'utiliser la méthode *setVisible(true)* et d'ajouter le panneau dans l'élément qui le contient avec la méthode
panneau_qui_le_contient.add(panneau_en_question).

Donc lorsque le client/le joueur cliquera sur le bouton, avec la gestion d'évènements et d'écoute de notre méthode *addMouseListener()*, on pourra actualiser notre interface graphique et lancer le jeu.

En effet, dans la classe *PanneauGeneral.java*, à chaque bouton est associé un évènement d'écoute.
D'une part, lorsque le client clique sur le bouton **JOUER**, son nom de joueur ainsi que le niveau voulu sont proposés. Cela a été possible grâce à *JOptionPane* et ses méthode *showOptionDialog() & showInputDialog()*.
Ainsi, selon le choix du client, on exécute la fonction *lancePartie(niveau_choisi_par_le_client)* qui lance le plateau de jeu.
C'est pourquoi grâce au niveau passé en argument, notre classe *PanneauDeJeu.java* saura quelle méthode organisation() exécuter.
A partir de ce moment, la musique se coupe et le jeu démarre *=> Sons.stopMusique()*.
D'autre part, lorsqu'il clique sur le bouton quitter, la méthode *dispose()* ferme notre jeu.
Enfin, le dernier bouton lui affiche les règles du jeu avec l'aide de *showMessageDialog()* de la classe *JOptionPane*.

La classe *PanneauDeJeu.java* lance ainsi la partie en faisant appel au constructeur de la classe *Niveau.java* avec le bon niveau choisi en paramètre.
Pour récupérer le bon plateau il suffit d'appeler la méthode *getPlateau()* se trouvant dans la classe *Niveau.java*.
Ainsi à chaque clic du joueur sur un bloc on appelle la méthode *testGroupeVide (x,y)* qui permet de réorganiser notre plateau et de l'actualiser en conséquence.
Pour finir, on teste si le joueur a gagné ou pas.
S'il a gagné, on lui propose de rejouer ou d'avancer au niveau suivant.
Toutefois, ayant créé une classe *PanneauJoueur.java*, ce test nous permet d'ajouter un score au joueur. Selon le niveau du plateau complété, le nombre de points diffère.
Ainsi, on appelle la méthode *initScore(niveau_actuel)*.
Elle a pour but d'actualiser le score du joueur grâce à la méthode *joueur_actuel.setScore(score_du_niveau)*.

Par la suite, il a le choix de passer au niveau suivant, de recommencer ou de finir la partie.
Néanmoins selon le choix du joueur, la suite n'est pas la même.
C'est pourquoi, la méthode *choixVictoire(option)* est appelée.
L'option intéressante ici est de finir la partie.
De là, nous avons décidé de créer un panneau joueur qui permet de récupérer le score du joueur et d'afficher un pop-up lui indiquant son nom choisi au début de la partie ainsi qu'une photo de profil générée aléatoirement dans notre banque d'images située dans notre dossier <profiles>.

S'il a perdu, soit il peut rejouer, soit il peut finir la partie ici.
Cependant, nous avons décidé d'implémenter une règle supplémentaire.
Dans la classe *Panneaujoueur.java* se trouve une méthode *setArgent().*
En effet , chaque joueur possède 15 en argent à chaque début de partie.
Or, si le joueur décide de recommencer la partie il perd 5 en argent.
En d'autres termes, si le joueur atteint 0 ou moins en argent il sera obligé de finir la partie. Cela nous permet de limiter le joueur dans ses choix et e rendre notre jeu plus réaliste.
Ces options sont implémentées dans la méthode *choixDefaite(option_choisie).* (*PanneauDeJeu.java*)
Ainsi, selon son choix, on actualise le plateau et l'organisation en rappelant le constructeur de Niveau avec le paramètre du niveau choisi et grâce à la méthode *getPlateau()* de cette dernière.

Abordons l'aspect personnel et graphique de notre jeu.
Pour cela, nous avons créé une classe *Images.java* qui permet de récupérer les images de notre dossier <imgEXT>.
Des fonctions sont ainsi disponibles pour chaque bloc ; les obstacles, les couleurs et « Bob ».
Ainsi dans notre panneau de jeu, on réutilise ces fonctions pour pouvoir les afficher sur notre plateau.
Des variables pour chaque image sont créés et stockées pour la suite.
Ce processus est maintenu par la méthode native de JAVA *paintComponent(Graphics g).*
On y récupère nos images pour chaque bloc nécessaire et on lui donne la taille appropriée.
Donc si notre bloc à l'indice [i,j] est « bob » on appelle la fonction *getBobImage().*
De façon similaire, on applique les fonctions pour avoir les obstacles et les couleurs.

Il s'agit de la fonction *drawImage()* qui s'occupe du rendu des blocs sur notre plateau.

En réalité, la vue ne s'exécute pas toute seule.
C'est bien le contrôleur situé dans le package « **controleur** » qui a pour rôle de lancer la vue qui elle communique avec le modèle.
Ainsi, on retrouve les commandes dans la classe *Controleur.java*.
Cette classe permet d'exécuter le jeu grâce à une commande et des options spécifiées dans notre terminal.
En effet, selon l'option choisie, il pourra jouer en mode graphique, en mode textuel, contre un bot ou encore afficher une aide.
D'une part, si le client lance « java -classpath src com.controleur.Controleur -g », alors une instance de Vue.java est lancée et le jeu s'exécute en mode graphique.

Toutefois, s'il choisit l'option textuelle, il pourra jouer directement en mode texte dans le terminal.
Pour ce faire, on a créé un constructeur Controleur(booléen b1, booléen b2). Le booléen b1 nous indique si le joueur veut jouer en mode graphique ou en mode textuel. Tandis que le booléen b2, lui, nous indique si le mode robot se déclenche.
D'autre part, si le client lance « java -classpath src com.controleur.Controleur -t », alors une instance de TexteInterface.java est lancée et le jeu s'exécute en mode textuel.
Dans cette classe, on demande au client son nom de joueur ainsi que son niveau en appelant la fonction *updatePrenomNiveauJoueur().*
De là, le jeu se lance automatiquement. On lui demande à chaque étape les coordonnées du bloc sur lequel il veut cliquer afin d'avancer dans le jeu.
Le jeu s'actualise ainsi en récupérant toutes les fonctions déjà crées pour notre jeu graphique (incluant le score du joueur qui s'actualise aussi).

Enfin, si le client lance « java -classpath src com.controleur.Controleur -b », le booléen b2 devient « true » et alors dans le constructeur de TexteInterface.java, on appelle la méthode *updatePrenomNiveauRobot().* Cette méthode demande le prénom au robot et prend aléatoirement un nom de notre tableau initialisé dans cette même méthode.
Le joueur robot est donc crée et le niveau par défaut égal à 1, lance la partie souhaitée.
Par la suite avec la méthode *lanceJeuRobot()* , le robot choisit aléatoirement une position [i,j] pour le bloc qu'il veut effacer.

Ainsi lors de la fin du jeu, on propose au robot s'il veut continuer la partie (avec un choix aléatoire).
Somme toute, selon l'option choisie aléatoirement, le joueur « robot » finit par terminer le jeu. On affiche ainsi son nombre de points ainsi que son nom de joueur généré dès le départ.

3. Implémentation et problèmes rencontrés

Tout d'abord, nous avons eu du mal à comprendre comment on pouvait réorganiser notre plateau sans avoir 200 lignes à coder. Ainsi on a décidé de répartir ces taches sou la formes de plusieurs méthodes et fonctions utiles les unes entre les autres.
Ainsi, le plus gros soucis a été de réorganiser le plateau et même après finition du projet, il reste quelques imperfections…
En effet, la partie délicate fut de comprendre et d'anticiper tous les différents cas (blocs voisins, même couleurs …)
Ce qui nous a permis de rendre cela plus simple fut les méthodes qui nous permettent de regrouper par groupe les blocs de même couleur afin de pouvoir les effacer en créant des blocs vides.
En outre, nous fumes surpris par rapport aux notions de SWING et des interfaces graphiques.
On peut citer la méthode *paintComponent()* qui permet d'appliquer la méthode *drawImage().* Cela nous a donné la possibilité de rajouter une image de fond à notre panneau de Jeu. Toutefois, il s'agissait d'une nouvelle notion pour nous. On a pu mettre en commun nos recherches afin de palier à ces petits soucis.

Par ailleurs, nous sommes bloqués sur le classpath. En effet, on a perdu beaucoup de temps à force d'exécuter notre projet sans qu'il se lance. On a compris par la suite qu'il s'agissait d'un problème de classpath de notre dossier <src>.
Pourtant, même après ajout dans l'environnement de notre classpath avec export classpath= « », on a rencontré le même soucis sur plusieurs ordinateurs.

Enfin, lors de l'implémentation du mode textuel on a rencontrée des problèmes sur les indices i et j choisis.
On a essayé de faire un petit test afin de prendre [i,j] seulement si les deux indices se trouvent dans le plateau mais cela n'a pas été résolu entièrement par manque de temps.

GRAPH UML :
RESUME DES CLASSES DE NOTRES PROJETS