Rapport CSC 2

Pour comprendre comment marchait l'élaboration d'un code en entreprise, nous avons été amené à travailler sur le Projet Siam, un programme déjà en parti écrit par une autre personne, pour nous initier à partager notre code et surtout à découvrir le code écrit par d'autres personnes (avec tous les avantages et inconvénients que cela engendre, comme par exemple comprendre des algorithmes ou devoir écrire des commentaires pour que les autres personnes participant au projet puisse comprendre le code). Cela nous initiait aussi à travailler sur un code avec plusieurs fichiers et beaucoup de fonctions, beaucoup de variables, variables et fonctions que l'on ne connaissait évidemment pas et qu'on devait s'approprier pour coder efficacement.

Nous travaillions toujours, globalement, tous les deux sur les fonctions à faire, les tests, etc. Ce n'était pas le plus rapide, mais au moins nous savions comment fonctionne le code (si on s'était partagé les taches en deux, on aurait pas su exactement comment marche telles ou telles parties de code par exemple).

Le code est fonctionnel et permet de jouer au jeu selon les règles du jeu Siam. Pour vérifier cela, nous avons utilisé des tests au sein des fonctions, et des tests d'intégration pour vérifier les mouvements pouvant être effectués (nous y reviendrons plus en détail dans la suite du rapport).
Cela nous a permis de vérifier que les fonctions fonctionnait convenablement et comme on le souhaitait. S'il y a un problème avec les retours de la fonction et les paramètres modifiés, ont corrige le tout pour que la fois suivante, tout marche comme on le veut. On répète cette démarche pour toutes les fonctions et ce tant que la fonction ne fonctionne pas correctement.

Le mieux est de commencer les tests pour chaque fonction dès le début (ce que nous avons fait). En effet faire les fonctions de test au début, pour les fonctions les plus basiques et de bas niveau, permettait de s'assurer que les prochaines fonctions les utilisant (de plus haut niveau), ait les bons retours de fonctions pour fonctionner correctement. En plus, en étant sur que les fonctions de bas niveau fonctionnent, cela voulait dire que les prochains bugs que l'on rencontreraient dans des fonctions de plus haut niveau seraient du à un problème dans les fonctions elles mêmes et non dans les appels de fonctions plus bas niveau, donc plus facile à trouver.
Cela nous permettait d'avancer sur les fonctions suivantes tranquillement, sans devoir revoir tout le code fait précédemment pour savoir s'il fonctionne ou non, on était sûr que oui.
A la fin, une fois que les fonctions ont été finies, nous avons utilisés des tests d'intégration, c'est à dire qui testent des mouvements particuliers plus globalement.

Nous avons aussi fait quelques fonctions en récursif (la fonction poussee_realiser notamment). Nous avions pensé que c'était le plus simple au départ, car nous pouvons aisément aller à la fin de la chaîne de poussée à l'aide d'appels récursifs, en se déplaçant de case en case jusqu'à une case vide ou le bord du plateau, pour ensuite faire les actions voulues (ici déplacer la pièce d'une case en avant), et une fois la fonction terminée, on revient à l'appel précédent de la fonction, c'est à dire à la pièce précédente, celle de la case

d'avant, pour refaire l'action et ainsi de suite jusqu'à remonter à la première pièce poussée. Nous pouvions faire autrement, puisqu'il y a une fonction pour inverser l'orientation. Nous aurions donc pu aller jusqu'à la fin du plateau à l'aide d'une boucle, puis revenir en arrière à l'aide de cette fonction, en appliquant la poussée sur chaque pièce. Nous avions vu cette fonction pour inverser l'orientation après, nous sommes donc rester sur notre première implémentation récursive, puisqu'elle fonctionnait en plus très bien.
On voit à l'aide de cette exemple, qu'il peut être judicieux, quand on découvre un programme, d'aller fouiller un peu dans les fonctions déjà écrites pour voir s'il n'y a pas certaines d'entre elles qui pourraient nous être utiles pour éviter de nous répéter et de refaire (complètement ou même partiellement) des fonctions déjà écrites.

Nous avons, tout au long de notre projet, créer des fonctions de tests, pour tester les fonctionnalités de notre programme et voir s'il fonctionne correctement. Nous les avons créé après chaque fonction implémentée, cela permettait de débuguer les fonctions si nécessaire et ce, le plus rapidement possible. Cela est surtout utile pour des fonctions qui sont utilisées par d'autres fonctions, car il est plus difficile de voir où est l'erreur dans une fonction, si celle ci en appelle plusieurs autres (est ce que le problème provient de la fonction elle même ou des renvoies des appels des autres ?).
Ces fonctions de test permettent de voir si une fonction fonctionne comme on le souhaite, en envoyant des paramètres précis. On sait ce qu'elle va faire et les valeurs modifiées, que l'ont vérifient :
Si jamais elles concordent avec ce que l'on attendait, tout va bien, on passe à la suite.
Si jamais la fonction ne fait pas ce que l'on attend d'elle, on affiche un message d'erreur précisant quel fonctionnement n'est pas attendu, pour le corriger en conséquence.
A chaque test de fonctions, il faut faire plusieurs appel à celle ci, avec la totalité des paramètres différents que l'on peut envoyer (sauf cas similaires), pour voir si chaque traitement est effectué comme il faut, donc cas valides et cas non valides.
Des tests d'intégration (et unitaires) sont aussi fait, pour tester cette fois la globalité d'un mouvement (ou plusieurs, et qui font appellent à plusieurs fonctions ! Par exemple le déplacement appelle plusieurs fonctions, dont le déplacement, mais aussi la poussée, la vérification des pieces, etc...) , ce qui permet de voir si les fonctions fonctionnent bien entre elles et non individuellement.
Ceci est une bonne base pour tester le jeu, mais reste insuffisant, il faudrait faire beaucoup de tests pour chaque configuration de jeu, sans compter qu'il est possible (et facile) d'en oublier. Il faut cependant être sûr d'éviter tout problème de mémoire, ou quoique ce soit qui pourrait faire planter le programme. Il vaut mieux avoir un problème d'une piece qui ne va pas dans la bonne direction ou qui ne suit pas exactement les règles du jeu, qu'un programme qui plante pour une erreur mémoire !

Pour la gestion de la Victoire, nous avons ajouté une fonction dans la poussée, nommée « piece_etre_derniere_poussee », et qui nous permet de connaître, lors d'une poussée, le type de la dernière piece à être poussée, plus exactement, on retourne le type de la piece uniquement si elle est au bord du plateau. Si jamais elle n'est pas au bord du plateau, c'est le type de la case suivante qui est retourné , c'est à dire une case vide. Cette

fonction étant utilisé lors de la vérification de la victoire, elle ne nous sert qu'à vérifier qu'un rocher est présent au bord du plateau, pour voir s'il est éjecté. Qu'elle nous renvoie un type case vide si la dernière piece n'est pas au bord du plateau n'est donc pas gênant, bien au contraire, puisque comme ça, on est sur que le rocher est bien au bord ! (et donc qu'il y a un gagnant).

Pour cette victoire, nous avons aussi créé deux nouveaux fichiers (un source et le header associé) appelé victoire_siam, avec deux fonctions :
« verification_condition_victoire_valide » et
« premiere_piece_meme_orientation_poussee ».

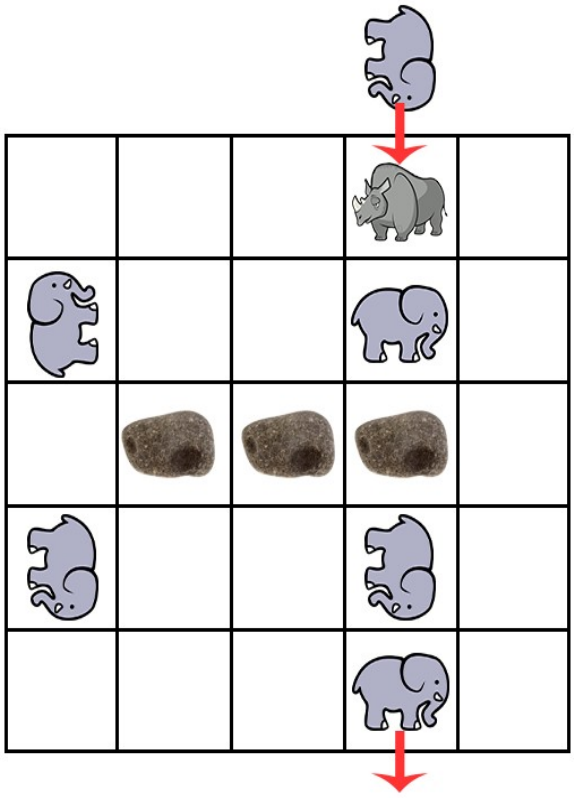
La première nous sert à vérifier si un joueur est victorieux. On utilise la fonction de la poussée citée précédemment pour connaître la dernière pièce de la poussée. Si c'est un rocher, alors il est forcement au bord (si la dernière pièce n'est pas au bord, c'est la case_vide après celle ci qui est renvoyé!).
On a donc un gagnant, on utilise alors la deuxième fonction, qui nous sert à obtenir un pointeur sur la pièce qui a la même orientation que la direction de la poussée et qui est le plus proche du rocher, car ainsi sont définies les règles du jeu. On peut alors déclarer le joueur de la pièce vainqueur.

Nous avons choisit de vérifier cette victoire au sein de l'Api (pour déplacement et introduction, puisque l'orientation ne permet en aucun cas de gagner), bien qu'il soit aussi possible de le vérifier dans la poussée puisque c'est celle ci qui va permette la victoire.

Ces nouvelles implémentations de fonction nous ont d'abord fait réfléchir à où les mettre :
Quel fichier correspondait le mieux à cette fonction ? Avait elle des points commun avec d'autres ? Est il logique de mettre cette fonction dans ce fichier ?
Par exemple, est il judicieux de mettre une fonction concernant la poussée dans le fichier de l'écriture/lecture d'un fichier texte ? Évidemment non.
Incluait elle les bons fichiers d'en tête ? Avait elle accès aux bonnes structures, ou énumération ?
Ainsi nous avions tout d'abord mis la fonction «piece_etre_derniere_poussee» dans le fichier « plateau_modification » car il contenait les bons fichiers d'en tête et permettait donc de faire la fonction convenablement. Nous avons cependant jugé mieux de la mettre dans le fichier de la poussée, car, tout d'abord elle ne modifie pas le plateau, et ensuite parce qu'elle concernait une action uniquement valable lors d'une poussée.
Pour la vérification de la victoire d'un joueur, nous avons choisit de la mettre dans un nouveau fichier car elle compte plusieurs fonctions qui sont assez différentes des fonctions des autres fichiers. Nous avons choisit les fichiers d'en tête qui sont utiles à la fonction (et uniquement ceux ci).

Nous avons aussi ajouté une spécificité au règle du jeu :
Le nombre maximum de pièce d'un joueur sur un plateau est de 5. Mais nous avons permis le fait que s'il y a déjà 5 pièces d'un joueur sur le plateau, et qu'en introduisant une pièce de ce même joueur la poussée fait qu'une de ses pièces est éjecté, alors le coup est quand même

permis, puisqu'il reste, après le coup, 5 pièces. Pour mieux illustrer cette règle, voici un schéma :



En effet, ici, il y a 5 éléphants, mais en introduisant un 6e éléphant, on en fait sortir un, ce qui ramène à 5 le nombre total d'éléphant après le coup.

Nous avons rajouté une IA qui joue le deuxième joueur automatiquement. Cette IA ne permet que de jouer des coups aléatoires. Nous avons tout d'abord réfléchit à un algorithme, comment allait jouer l'IA, comment allait elle choisir son coup, dans quelle fonction implémenter le tour de l'IA, etc...
Nous avons donc créé dans deux nouveaux fichiers (source et en-tête) appelé simplement AI, les fonctions permettant à cette IA de jouer. Nous avons aussi rajouté et modifié le code du mode interactif pour qu'il puisse prendre en compte le tour de l'IA, puisque c'est dans ce fichier que le jeu tourne et fait jouer les joueurs, puis s'arrête quand il y a un gagnant.
Nous avons modifié comment la fonction du mode interactif fonctionnait, en prenant en compte que le 2e joueur était une IA :
Si le joueur est le joueur 0 (l'utilisateur) on prend la sortie du clavier (celle écrite de base)
Si le joueur est le joueur 1 (l'IA) alors on va tout d'abord faire jouer l'Ia (on verra en détail plus bas comment), puis chercher les informations utiles de l'action à faire dans un fichier nommé «AI_instructions.txt».
Dans les deux cas on récupère une chaîne de caractère, et le reste de la fonction ne change pas du programme de base.
Nous avons aussi créer plusieurs fonctions pour l'IA, une pour la faire jouer, globalement, qui appelle plusieurs sous fonctions comme par exemple le tirage au sort des actions à effectuées, et les actions en elles même. Nous avons choisit d'enregistrer l'instruction de l'IA

dans un fichier texte car il était ensuite facile de récupérer la chaine de caractère dans la fonction du mode_interactif.

Concernant le jeu de l'IA :
Elle va d'abord compter le nombre de pièce qu'elle possède sur le plateau. Le but est de choisir aléatoirement son action entre introduire, déplacer ou orienter une pièce. Nous avons pour cela créer une énumération pour ces 3 types d'instructions, cela permet de voir plus aisément quelle action est effectuée.
Quand aucune pièce n'est sur le plateau, nous avons choisit de faire obligatoirement une introduction de pièce, cela évite de faire des tours inutiles en espérant tomber aléatoirement sur la bonne action c'est à dire l'introduction.
S'il y a déjà des pièces, alors elle appelle une fonction de tirage au sort des instructions, puis on appelle la sous fonction correspondant à l'instruction tirée.
Dans chacune des sous fonctions déplacer et orienter, on se déplace dans le plateau et on choisit aléatoirement une pièce, on choisit alors aléatoirement une orientation (intègre évidemment à un déplacement).
Pour introduire, on tire aléatoirement deux coordonnées que l'on prendra au bord du plateau (on fera pour cela une saisie protégée) et une orientation.
Pour les trois actions, on ouvre ensuite le fichier «AI_instructions.txt», et on inscrit dedans l'instruction correspondante, avec le caractère, les deux coordonnées et la ou les orientation(s).
Ensuite, au niveau du mode interactif, on fera comme pour le programme de base : on mettra la chaine de caractère, cette fois pris dans le fichier «AI_instructions.txt» (et non depuis le clavier), dans la même variable que pour la saisie au clavier, le reste étant le même programme que pour la saisie au clavier.
Malheureusement par manque de temps, nous n'avons pas pu faire les contrats et aucun test pour l'IA.

Finalement, ce projet nous as permis de voir globalement comment fonctionne un projet en entreprise, à petite échelle : Nous étions seulement deux, avec quelques fichiers et quelques fonctions. Mais cela nous a appris à coder de façon à ce que les autres programmeurs, qui seront forcément amenés à voir notre partie de code, puissent comprendre et utilisés notre code. Nous avons nous, dans le même temps, appris à lire et à comprendre ce que les autres programmeurs ont codés, en commentant nos programmes, en créant aussi bien des contrats des fonctions, en écrivant ce que fais la fonction, ce qu'elle attend en paramètre, et ce qu'elle retourne ou modifie, mais aussi en écrivant les algorithmes de fonctions plus compliquées.
Cela nous a aussi appris quelques bonnes pratiques de codage, comme par exemple à donner des noms de variables et de fonctions cohérentes et lisibles avec ce qu'elles font ou ce qu'elles représentent, ou encore à utiliser des énumérations et des structures pour simplifier aussi bien notre façon de coder que notre lecture, ainsi que d'utiliser des fonctions de vérification pour voir si'il est possible de faire le coup avant de l'effectuer.
Nous avons aussi appris à coder selon une méthodologie de tests et de vérification constante de notre programme à l'aide d'assertion, de test au sein des fonctions, et de test d'intégration plus global, qui permettent de débuguer le code le plus rapidement possible. Cela nous permettait de nous assurer le bon fonctionnement des futures fonctions qui seraient amenées à les appelées, mais aussi un débug plus facile car le fonctionnement des fonctions sont encore frais dans nos têtes (il est moins facile de faire un test d'une fonction écrite il y a plusieurs semaines, même si les commentaires sont là pour aider) et qui offre donc un réel gain de temps.