**Travail pratique 2**

**Laborat – Rapport de projet**

**Analyse textuelle**

Laborat est un projet *pygame* ou le Laborat (personnage joueur), doit s’échapper, trouver le fromage (condition de victoire) et éviter des zombies (condition de défaite). Les quatres types d’acteur dans le jeu sont représentés par des carrés de différentes couleurs. Laborat et les zombies sont les seuls acteurs pouvant bouger. Il y a également des murs que ni Laborat ni les zombies ne peuvent traverser. Tous les types d’acteurs doivent pouvoir enregistrer des collisions avec Laborat. Les collisions entre Laborat et le fromage ou les zombies terminent le jeu. L’arrière-plan doit changer de couleur dépendamment de l’état du jeu (blanc si on gagne, rouge si on perd). La fenêtre est une grille de 9 lignes et 16 colonnes, chaque acteur occupe une case.

**Conceptualisation**

Notre version de Laborat implémente une architecture de type *MVC*. *MVC* nous semblait être approprié dans pour le type de jeux à construire comme on peut facilement trancher les éléments venant du model des éléments de contrôle usager et visuels.

Le programme comprend 7 modèles principaux : *game*, *actor*, *actor\_factory, etat, tile et command*.

*Game* est le modèle central du jeu. C’est dans cette classe que l’on retrouve l’initialisation des éléments centraux du jeu à son bon fonctionnement (*window*, *listeners*, *actors*, clock, ect.). Comme cet objet central s’avère absolument essentiel à la bonne coordination des opérations du logiciel et qu’un seul et nécessaire, un patron de type singleton a été choisi pour sa conceptualisation. C’est aussi la classe *game* qui contient la boucle de jeu principale (*run*), applique les *input* capturer par le *game\_listener* et gère les conditions de victoire.

Actor représente tous les éléments du jeu. Cette classe contient les informations sur la position et état de l’acteur et ses méthodes de type *get/set*. De *actor* découle 4 sous-classes : *Player* pour le personnage joueur, *Fromage* pour l’objectif, *Zombie* pour les ennemis et *Wall* pour les obstacles. Chacune de ces sous-classes ont un chiffre permettant d’identifier de quel type d’acteur il s’agit. *Player* a une méthode *update* qui permet de gérer les déplacements liés aux saisies du joueur. *Zombie* a deux méthodes qui permettent aux zombies de se déplacer vers le joueur : *update* et *is\_further\_to\_player*. *Wall* et *Fromage* n’ont pas de méthodes puisque ce sont des acteurs statiques.

*actor\_factory* est la classe chargée de l’initialisation des objets dynamiques des objet des sous-classes actors et les retourne avec ses méthodes get.

Laborat ne comporte qu’un seul contrôleur, *game\_listener*. Cette classe est chargée de pousser les événements de clavier vers le modèle *game.* Par contre, les événements de clavier sont retournés à l’état sous forme de commandes qui servent de patron général et qui peuvent être générés par tous types d'acteurs.

Le programme un élément : *Window* . *Window* prendra les informations de l’**état** et les rendra à l’écran, elle est aussi chargée de l’affichage et de la boucle de mise à jour des icônes. Sa méthode *end\_game* est responsable de refléter les victoires et défaites. *IconesActeur* quant à elle hérite de la classe *pygame sprite*. Elle contient l’objet *Pygame* *rect* qui définit nos acteurs visuellement et définit leur collision. Elle est malheureusement aussi devenue obsolète au fur à mesure du développement.

*état* est la carte du jeu, c’est là que sont enregistrés des tuiles qui contiennent les acteurs et leurs positions. Le fichier *maps.txt* contient plusieurs cartes prédéfinies, *etat* en sélectionne une au lancement du programme de manière aléatoire.

*tile* est une classe représentant les tuiles. Les tuiles servent de contenant pour les acteurs. Quand l’**état** est généré, il contient une grille de tuiles.

Command contient 3 données : un acteur, une position de départ et une position d’arrivée. Les commandes sont des objets permettant une interprétation uniforme des événements peu importe leur provenance que ce soit de Player ou Zombie.

**Exécution**

L’exécution du travail cette fois s’est prouvée plus laborieuse que la première partie.

À notre grande surprise, l'implémentation du patron de conception de commande s’est avérée complexe dû à notre précédente implémentation. Par contre, grâce à l’habileté et à l’acharnement des membres de notre équipe, nous avons su mener à bien ce projet.

Nous avons d’abord commencé par ajouter les éléments nécessaires au diagramme de conception pour faciliter la division du travail. Ensuite nous nous sommes divisés le travail en fonction de la quantité de programmation estimée: Tom et Olivier ont travaillé ensemble sur la mise à jour du “package” modèle et sur l’implémentation de la classe **État** , Tymotheï à travaillé sur la mise à jour du package “view” suite au modification du modèle et Clément s’est chargé du package “**Controller”** ainsi que de la documentation.

Nous avons rencontré un heurt lors de la mise en place de **l’état** qui nous a fait perdre un temps considérable de temps et c’est pourquoi nous avons jugé utile et nécessaire d’implémenter la classe **tile** comme conteneur d’acteur et d’autres informations utiles à la mise à jour de la grille **d’état.**