RAPPORT DE PROJET

GARNIER Romain
HARDOUIN Armel

Licence 2 INFORMATIQUE
2021-2022

—

INFO0402

—

Programmation Orientée
Objet et C++

SOMMAIRE

PARTIE I : PRESENTATION

Au terme du module d'INFO0402 Programmation Orientée Objet en C++, nous avons la tâche de réaliser un projet. Il s'agit ici de recréer le célèbre jeu des années 1980, Bomberman, en version simplifiée.

Dans ce rapport, nous allons vous présenter une description du projet et du jeu. Nous allons ensuite présenter et argumenter l'ensemble de notre modélisation ainsi que les concepts POO utilisés afin de répondre aux compétences visées du module.

Nous présenterons enfin quelques fonctionnalités qui auraient pu être implémentées afin d'optimiser notre jeu.

PRESENTATION DU JEU

Bomberman est un jeu d'action labyrinthe paru pour la première fois en 1983. Depuis, des suites et de nombreuses variantes virent le jour, toutes plus originales que les autres.

Le principe du jeu est simple : Vous incarnez un personnage qui doit tuer l'intégralité des ennemis en posant des bombes, et ainsi atteindre le point de victoire, en se frayant un chemin à travers un labyrinthe de murs, qui peuvent être détruit grâce à ces mêmes bombes.

Différents ennemis sont présents comme des monstres basiques, des fantômes qui peuvent traverser les murs, ou encore des archers qui vous attaquent à distance.

Afin de renforcer votre personnage, différents bonus sont présents afin d'améliorer les différents aspects de votre gameplay et ainsi vous donner un coup de pouce. On retrouve par exemple de quoi améliorer les dégâts des bombes, leurs portées, ou encore de quoi améliorer votre vitesse ou vos points de vie.

PRESENTATION DU PROJET

Le projet final Bomberman est un jeu entièrement pensé objet, et codé en C++, la version minimale étant C++14 ou plus. Le jeu se déroule sur terminal, avec entrée clavier et sortie standard.
L'objectif est de pouvoir, à partir d'un fichier texte formaté au préalable, de pouvoir générer une carte de jeu en console, et de pouvoir réaliser l'intégralité des actions vues dans plus tôt (section PRESENTATION DU JEU). Chaque fichier est un niveau à part entière.

Une carte apparait en console de cette façon :

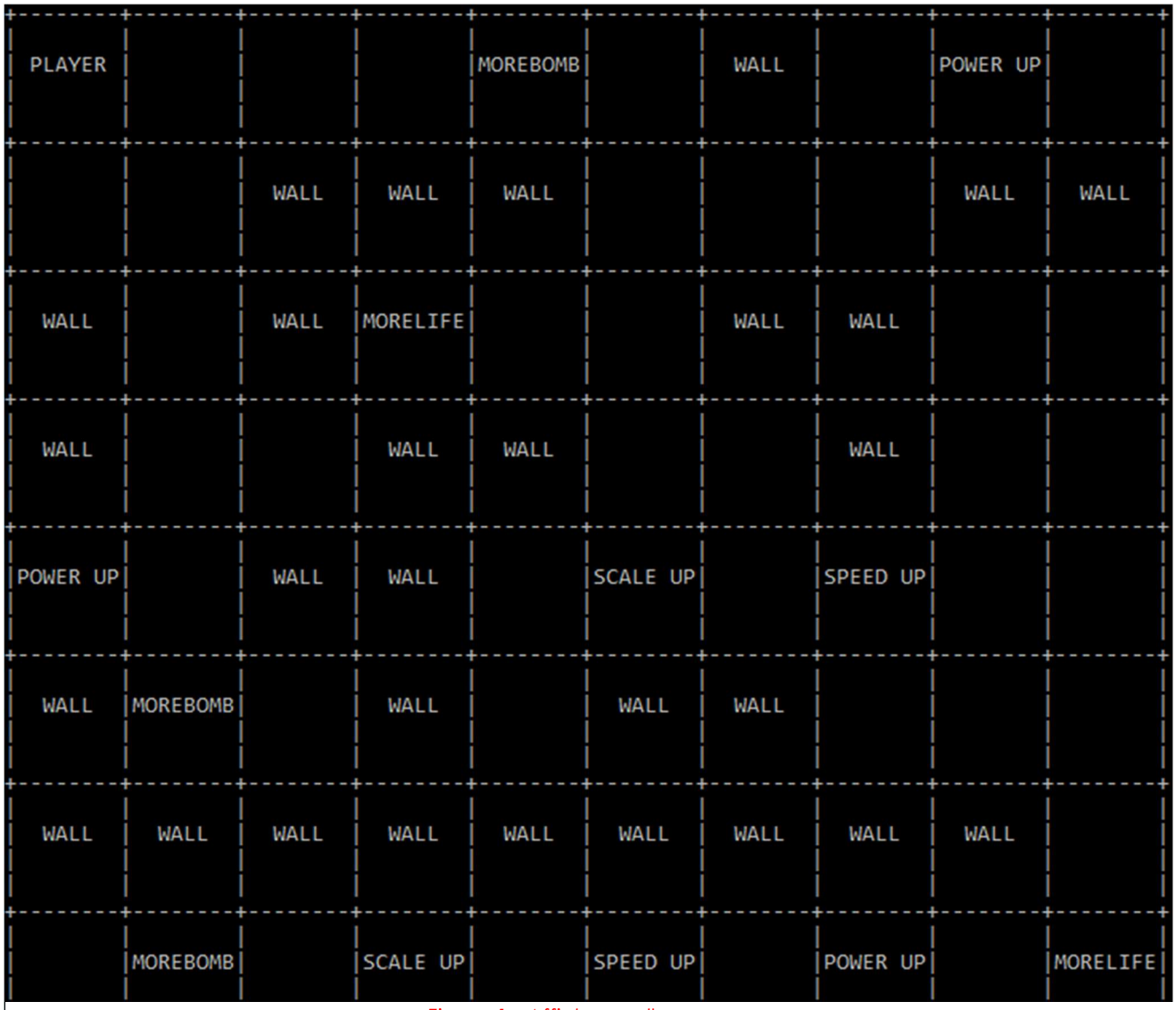

Figure 1 - Affichage d'une map

Un fichier texte est formaté de la manière suivante :

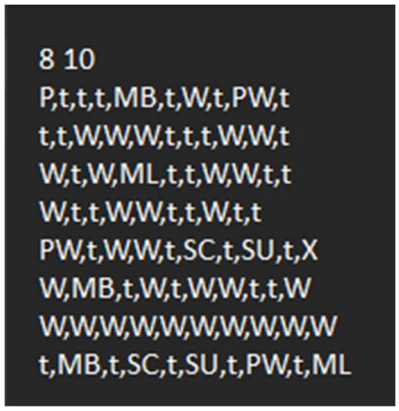

Figure 2– Exemple d'un fichier texte

Les 2 premiers chiffres sont la taille de la carte (par exemple ici : 8 pour les lignes et 10 pour les colonnes).
Les autres caractères visibles sont les différents éléments pouvant apparaître au cours du jeu, ils sont séparés par des virgules. Nous verrons plus en détail comment est gérer l'affichage de la carte.

Nous allons à présent lister les différents éléments visibles sur la carte au cours d'une partie :

- **PLAYER** : Le joueur (utilisateur)

- ****espace vide**** : Les tuiles sur lesquelles se déplacent le joueur que les ennemis, et sur lesquelles se tiennent les murs et les objets consommables

- **WALL** : Les murs qui servent de rempart au joueur pour atteindre son objectif

- **MORELIFE** : objet consommable par le joueur pour améliorer ses points de vie

- **MOREBOMB** : objet consommable par le joueur pour augmenter son nombre de bombe

- **SPEED UP** : objet consommable par le joueur pour améliorer sa vitesse de déplacement (petite subtilité ici, on augmente le nombre de case dont il se déplace et non sa vitesse)
- **SCALE UP** : objet consommable par le joueur pour améliorer la portée des bombes

- **POWER UP** : objet consommable par le joueur pour accroitre la puissance des bombes

Le dossier Bomberman regroupe un dossier src qui contient tout le code source, ainsi qu'un dossier res, contenant lui les fichiers textes pour générer les niveaux.

L'arborescence est structurée 2 en arbres :

-includes : ensemble des fichiers .h, qui contiennent chacun une classe éponyme.

-src : ensemble des fichiers contenant le code source des classes.

Les fichiers sont répartis par thème dans des sous-répertoires : les personnages, la carte de jeu, les objets et la gestion du jeu.

Et pour terminer, un makefile est présent afin de faciliter la compilation des fichiers sources dans un exécutable.

COMMANDES

Comme vu dans la section précédente, ce projet utilise un makefile pour factoriser les différentes compilations des fichiers source. La commande *make* ainsi que la commande « *./game* » suffisent à lancer le programme.

Figure 3 - Exemple d'exécution

PARTIE II : MODELISATION

Afin de mettre en ordre nos idées, nous avons procéder à une partie modélisation.

Le diagramme de classe, le plus important selon nous car ce dernier permet de planifier des classes avec leurs attributs et méthodes respectifs, et de les faire interagir entre elles.

Ce diagramme a été choisi de façon à mettre en évidence les éléments du projet.

DIAGRAMME DE CLASSE

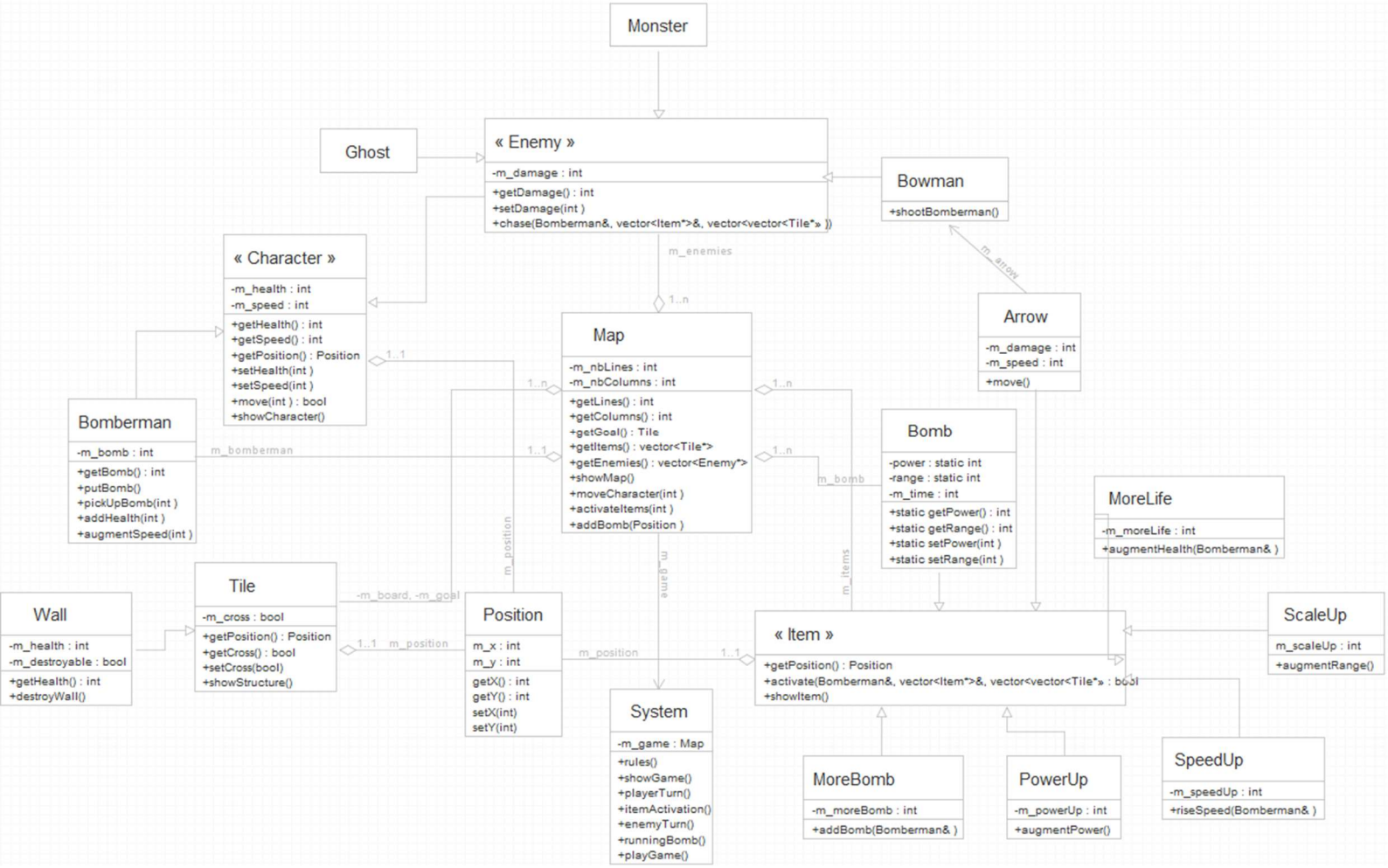

Figure 3 - Diagramme de classe

Ci-dessus se trouve le diagramme de classe de notre projet. Dans ce diagramme nous retrouvons une classe prédominante qui est la classe *Map*. Cette dernière est en effet le berceau de notre jeu, c'est ici que prennent vie les différents éléments du jeu. On remarque une première relation entre la classe *Map* ainsi que la classe *Tile*, il s'agit d'une **agrégation**. D'un point de vue purement logique, nous pensions en premier lieu à une **composition** ; En effet, si on détruit la map, logiquement, les tuiles le sont aussi, mais pour des raisons d'implémentation, nous avons opté pour une **agrégation**, nous reviendrons sur cette opération ultérieurement.
Ensuite, la map agrège différentes autres classes, comme la classe *Bomberman* qui représente le joueur, il peut y avoir qu'un seul

bomberman comme le montre bien la cardinalité « 1..1 ». D'autres agrégation sont présentes auprès de Map comme la classe *Enemy* qui représente les ennemis, la classe *Item* pour les items consommables par le joueur, et la classe *Bomb* pour les bombes utilisables par le joueur. *Enemy, Item* et *Character* sont trois classes particulières, car celles-ci sont des classes dites **abstraites**, elles ne peuvent être instanciées, la raison de ce choix étant de ne pas vouloir créer d'item, de personnage ou d'ennemi à proprement parlé (section CLASSE ABSTRAITE).

C'est pour cela qu'intervient une autre notion très intéressante de la programmation orientée objet, l'**héritage** (section HERITAGE).
Nous avons trois axes d'héritage différents :

- Entre les personnages, les tuiles/murs et les items. Une classe mère pour les personnages, nommée *Character*, permet d'apporter les caractéristiques primaires communes des personnages, que ce soit bomberman ou les ennemis, *Enemy* est elle aussi une classe mère pour les classes des différents ennemis, en plus d'être abstraite.
- La classe *Wall* hérite de *Tile* car, on permet grâce à un seul attribut, que la tuile soit traversable ou non, si c'est non alors c'est un mur.
- Quant aux items, c'est le même procédé que les personnages, sachant que *Bomb* est également un item.

Etant donné que le jeu nécessite de se déplacer sur les tuiles, nous avons décidé d'implémenter une classe *Position* qui compose toutes les classes qui sont positionnables sur la map, telles que *Character*, *item*¸ ou bien *Tile*. Deux entiers x et y suffisent pour se positionner dans la matrice qu'est la map.
Map réunit une grande partie de ce qui se passe dans le jeu, c'est pourquoi nous avons importé une dernière classe afin d'y implémenter la gestion du jeu, la classe *System*. Elle instancie la classe *Map* et gère le système en entier.

Comme dit précédemment, certaines opérations ne sont pas logiques d'un point de vue modélisation, c'est pourquoi nous allons aborder par la suite l'implémentation en C++ et ainsi justifier ce choix.

PARTIE III : CONCEPTION

Afin de mener à bien ce projet C++, nous avons dû implémenter certaines notions de la programmation orientée objet mais aussi d'autres concepts comme la manipulation de fichiers.

Nous allons donc à présent lister ces différents concepts, et donner pour chacun une définition ainsi que des exemples de notre propre projet.

L'ensemble des classes sont définies dans des fichiers header et les méthodes, elles, le sont dans les fichiers source correspondants.

CLASSE MAP

Comme préciser plus tôt (section DIAGRAMME DE CLASSE), le noyau du projet est la classe *Map*. En effet, pour que le jeu soit jouable, il faut commencer par créer une map sur laquelle va se tenir l'ensemble des autres éléments. Nous avons également vu que les différents niveaux sont générés grâce à des fichiers .txt (section PRESENTATION DU PROJET).

La classe *Map* possède plusieurs attributs correspondant à ses relations. Elle possède plusieurs vecteurs de pointeurs pour les items, les bombes et les ennemies.

Le constructeur commence donc par ouvrir le fichier grâce à un entier représentant son ID passé en paramètre, puis génère la map. Cette map est créée grâce à une matrice de plusieurs instances de *Tile*, l'attribut correspondant est un vecteur à 2 dimensions, noté **std::vector<vector<Tile>>**. Nous avons bien évidemment pris soin d'inclure la classe *vector* pour utiliser ce conteneur d'objets séquentiels.
Cela serait ainsi si nous faisions une composition mais, comme vu plus tôt, nous avons dû remplacer par une agrégation. Cela devient donc **std::vector<vector<Tile*>>**. Le constructeur de *Map* instancie plusieurs **pointeurs** de *Tile* grâce à une allocation dynamique, et ces dernières sont introduites dans le vecteur2D grâce à la méthode **push_back()** de la classe *vector*. Nous parcourons ensuite le fichier, on compare le caractère courant avec la fonction **std::string::compare()** et en fonction du caractère, on instancie la classe souhaitée et on la « pousse » dans son le vecteur correspondant. Une tuile est à nouveau instanciée également à chaque fois que l'on importe un élément, mais cette tuile devient intraversable, car celle-ci comporte du coup un obstacle.

Nous avons introduis au sein de la classe des méthodes permettant la gestion des éléments présent sur la map, comme par exemple une méthode **moveCharacter()** permettant d'utiliser la méthode **move()** de *Character*, ou encore une méthode **activateItem()** qui appelle la méthode **activate()** de *Item*. Nous verrons ainsi pourquoi lors de l'explication de la classe *System* (section CLASSE SYSTEM).

CLASSE ABSTRAITE

Une **classe abstraite** à la particularité de ne pas pouvoir être instanciée, toutefois elle peut avoir des attributs et un constructeur. Pour rendre une classe abstraite, il suffit de déclarer une **méthode virtuelle pure**. Nous verrons plus tard ce qu'est la virtualité et à quoi sert-elle (section POLYMORPHISME ET VIRTUALITE).
Une méthode virtuelle pure possède le prototype suivant :

virtual void myMethod() const = 0 ;

Elle ne peut être définie dans le fichier source de classe qui la déclare. Nous avons plusieurs classes qui utilisent cette notion afin de ne pas être instanciable car ces classes représentent un concept trop abstrait de la réalité pour être façonner. Les classes *Character, Item, Enemy* sont toutes des classes qui ne peuvent donner naissance à des objets concrets. C'est pourquoi nous les rendons **abstraites**. On pourrait alors se demander à quoi servent-elles ? Nous avons déjà donné une idée à ce sujet dans l'explication du diagramme de classe, nous avons tout simplement réuni au sein de ces classes abstraites, les membres d'instance qui sont communs à plusieurs de classe instanciables, et nous réalisons ainsi une relation d'**héritage** entre elles.
La classe *Character* possède comme méthode virtuelle pure la méthode **showCharacter()** qui ne renvoie rien et qui ne modifie pas l'objet courant. Cette dernière sera utilisée par l'ensemble de ses filles, chacune à leur manière.
La classe *Item* fait de même avec une méthode **showItem()**.
Petite subtilité avec pour la classe *Enemy*, celle-ci ne déclare aucune méthode de se genre. Il faut savoir que si une classe fille ne redéfinit pas la méthode virtuelle pure de sa mère, elle est **abstraite** à son tour, et nous avons donc permis cela.

Item possède également une deuxième méthode virtuelle pure qui permet d'activer l'item, celle-ci prend en paramètre une référence de *Bomberman*, une référence de vecteur de *Item* ainsi qu'un vecteur2D de *Tile*,* elle ne retourne rien.

HERITAGE

L'héritage est un concept incontournable pour notre projet, afin de pouvoir condenser notre code et de factoriser les membres d'instances communs dans la hiérarchie.
L'héritage se fait en premier lieu dans la classe mère : les attributs qui sont sous héritage doivent être **protected**.
Dans la classe fille, on réalise la déclaration suivante :

class Fille : public Mère {….} ;

Tout élément hérité de manière identique n'a pas besoin d'être redéfini dans la classe fille.

Le constructeur de la classe fille n'a donc pas besoin d'initialiser chaque attribut, il suffit d'appeler le constructeur parent puis d'initialiser les attributs qui la spécialise.

MyClass::MyClass(int x) : ClassParent(), m_x(x){…}

Nous avons énoncé 3 branches d'héritage, pour les items, les personnages et les tuiles. Ces différentes classes mères ont des membres d'instance qui sont donc hérités pour chacune des classes filles. Nous prenons, pour la suite, l'initiative de ne pas énoncer l'héritage des différents accesseurs (getters) et modificateurs (setters) de chaque classe mais ils sont bien hérités dans notre programme.

- Pour les personnages :

  La classe mère *Character* possède les attributs protégés **m_health** qui représente les points de vie du personnage, **m_speed** qui lui représente la vitesse du personnage et **m_position** qui représente la position.
  On retrouve également les méthodes **move(int)**, une méthode qui ne renvoie rien et qui prend en paramètre un

entier symbolisant la direction. Elle permet de faire bouger le personnage. La classe *Enemy,* qui hérite d'abord de la classe précédente, est aussi classe mère des trois classes des ennemis. Elle apporte à ses filles, en plus des membres hérités de *Character*, un attribut, toujours en protected, pour les dégâts des ennemis, un entier appelé **m_damage**.

- Pour les items :

  Les items du jeu héritent d'une classe mère *Item* qui elle donne en héritage l'attribut protégé **m_position**.

- Pour les tuiles :

  Le ciment de notre map, la classe *Tile* possède, elle, deux attributs protégés qui sont une position **m_position** et un booléen **m_cross**. Si ce booléen est vrai, la tuile est traversable, sinon elle ne l'est pas, et le mur prend vie à partir de là.

  Comme précisé plus haut, les méthodes virtuelles pures des classes mères sont héritées et doivent être redéfinies afin que les classes filles ne soient pas elles aussi **abstraites**. Elles sont alors redéclarées avec le mot-clé **override** afin de montrer au compilateur qu'il s'agit d'une redéfinition. Le mot-clé **virtual** n'est plus nécessaire.

POLYMORPHISME ET VIRTUALITE

Afin de répondre à un besoin particulier de notre projet, nous avons dû implémenter le Polymorphisme et la Virtualité.

Nous avons parlé plus d'une fois d'un passage d'une composition à une agrégation pour la classe *Tile*, et la raison est simple :

Lorsque nous souhaitons afficher les instances de *Tile* stockées dans le vecteur correspondant, il n'y a aucune distinction de faite entre les tuiles classiques et les murs. En effet dans la classe map lorsque nous souhaitons afficher la *Tile* un certain emplacement du vecteur, s'il s'agit d'une instance de *Wall*, alors un affichage de *Tile* est quand même produit. Cela est dû au fait que nous n'avons pas de pointeur ainsi que de méthode virtuelle, les deux ingrédients principaux, pour y remédier.

EXCEPTION

Afin de gérer des erreurs, nous avons défini une classe pour une **exception**. Cette exception, nommée *BombermanException* est utilisée pour les déplacements du personnage.

Cette classe hérite de la classe *Exception*. Elle possède un attribut char* pour le message d'erreur, et redéfinie la méthode **what()**.
Elle est utilisée dans la classe *Map* dans la méthode **moveCharacter(int)**, lorsque le Bomberman essaie de sortir de la map, lorsqu'il essaie de passer sur une tuile avec mur. Un bloc try catch est présent dans la classe *System* afin d'assurer la bonne exécution de la méthode **moveCharacter(int)**.

CLASSE SYSTEM

Le but de ce projet, est certes d'avoir un programme fonctionnel, mais avec un minimum d'optimisation. Utiliser nos classes dans un main et y manipuler le système serait bien trop lourd, c'est pourquoi nous avons pour consigne de créer une ultime classe, la classe *System*.
Cette classe doit donc gérer le jeu en lui-même, le tour du joueur, le tour des ennemis, les explosions des bombes et bien d'autres.

Nous avons donc seulement **m_game** de type *Map* comme attribut privé. Nous savons que la map gère la totalité des actions du jeu, il suffit d'en instancier une dans cette nouvelle classe.

Nous avons également précisé dans la section CLASSE MAP que nous des méthodes avaient été écrites spécialement pour les méthodes d'« action » des différentes classes. Tout l'intérêt est là, si nous voulons utiliser les méthodes des personnages ou des items, nous devons pouvoir y avoir accès. Certes, elles sont publiques, mais nous n'avons pas d'objet *Character* ou *Item* dans cette classe *System*, nous devons donc les appeler dans des méthodes de *Map* pour pouvoir les appeler avec l'attribut **m_game**, ainsi nous pouvons faire interagir les objets dans la classe *System*.

CONCLUSION

Pour conclure ce rapport au sujet de notre projet BOMBERMAN, nous pouvons déjà parler du fait que nous ne pouvons pas réellement faire une partie. En effet, la map s'affiche bien, les murs, les items et le joueur aussi ; Le joueur peut se déplacer, les collisions sont respectées, il peut récupérer les items qui changent donc ses caractéristiques, mais le principal intérêt du jeu, les bombes qui explosent, n'a pu être implémenté à temps. On peut en poser avec une limite de stock, mais elle n'explose pas, elles ne disparaissent pas, elles ne font aucun dégât. La tuile de victoire ne peut donc pas être trouvée, et la partie ne se termine jamais. Les ennemis ne sont également pas entièrement implémentés.

Nous avons malgré tout essayé de retranscrire le plus de notions étudiées ce semestre, comme l'héritage, le polymorphisme, les exceptions ou encore les conteneurs d'objets et la STL.