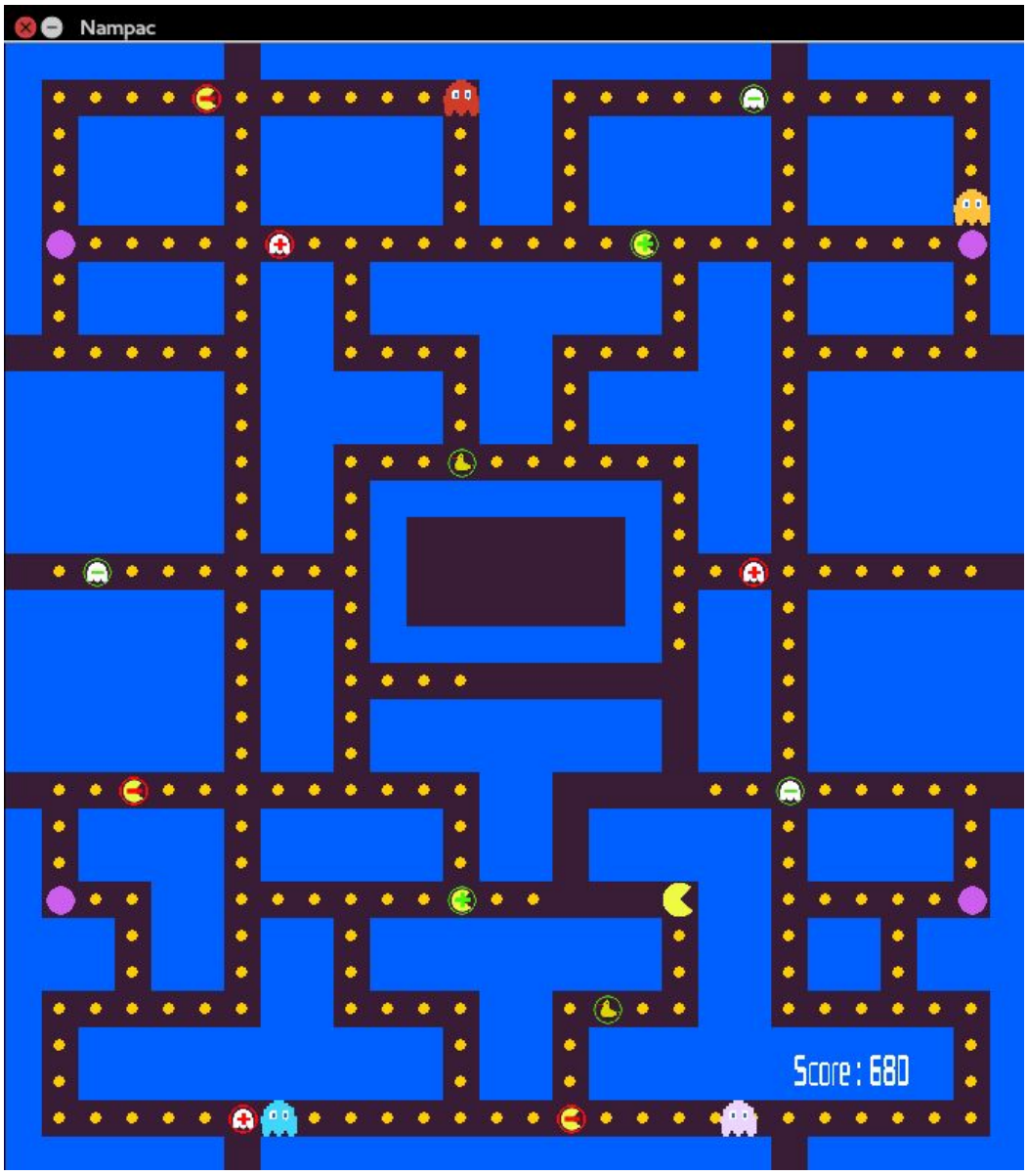
ALAPETITE
Florent

LATOUCHE
Dorian

Projet Nampac



Objet et développement d'applications

Année Scolaire 2016-2017

UNIVERSITÉ DE NANTES

Sommaire

I. Présentation du jeu et règles

A. Présentation du Nampac :

Le Nampac est une réinterprétation du jeu Pacman original. Le joueur contrôle un petit personnage jaune à l'aide des touches directionnelles du clavier et doit récupérer tous les bonus gum (petites boules oranges) présents sur le plateau de jeu. Comme dans le Pacman classique, le joueur doit faire face à des fantômes de couleurs et d'intelligence artificielles différentes. Le jeu s'arrête lorsque le joueur a récupéré tous les bonus gum ou qu'il est entré en contact avec un fantôme.

Le jeu est développé en C++ et utilise la bibliothèque graphique SDL2 pour C/C++. Il est distribué sous licence GPLv3.

B. Règles du jeu :

1. Règles basiques :

- Jeu basé sur un système risk/reward (risque/récompense).
- Pour gagner : récupérer tous les petits bonus oranges (gum).
- Toucher un fantôme entraîne la fin de la partie (sauf si le bonus Hunter est actif).
- Le joueur perd 1 point par tour de jeu.
- Ramasser un bonus gum ajoute 50 points.
- Ramasser un bonus ajoute 100 points et facilite un peu plus le jeu.
- Ramasser un malus ajoute 500 points et complexifie un peu plus le jeu.
- Objectif : trouver la meilleure stratégie de prise de bonus pour maximiser le score.

2. Bonus / Malus / Fantômes :

Bonus :

Bonus **Gum** : +50 pts. (Victoire si plus aucun sur le plateau)

Bonus **SlowGhost**: + 100 pts.
Ralentit tous les fantômes (75 % de la vitesse actuelle).

Bonus **SpeedPacman** : + 100 pts.
Accélère Pacman (125 % de la vitesse actuelle).

Bonus **Hunter** : + 100 pts.
Pacman peut manger les fantômes pendant 200 déplacements.

Bonus **StupidGhost** : + 100 pts.
Les fantômes sont stupides pendant 250 déplacements.

Malus :

Malus **SpeedGhost** : + 500 pts.
Accélère tous les fantômes (125 % % de la vitesse actuelle).

Malus **SlowPacman** : + 500 pts.
Ralentit Pacman (75 % de la vitesse actuelle).

Fantômes:

Fantôme bleu.
I.A. : Imprévisible (Implémentation : change de direction tous les 8 tours de jeu.)

Fantôme orange.
I.A. : Stupide (Implémentation : change de direction tous les tours de jeu).

Fantôme rose.
I.A. : Ambuscade (Implémentation : change de direction tous les 15 tours de jeu).

Fantôme rouge.
I.A. : Chasseur (Implémentation : change de direction tous les 30 tours de jeu).

3. Génération de map :

Le Nampac comporte un système de génération dynamique de plateau de jeu. Pour cela, il suffit de modifier le fichier PacMap.laby présent dans le dossier map/ du projet avec les informations ci-après.

La génération de plateau de jeu obéit à certaines règles :

- Taille du plateau de jeu : 31 lignes x 28 colonnes.

- Le rectangle central vide doit rester identique.

- Eléments de la génération :
    - p : Pacman (exactement 1)
    - R : fantôme rouge (0 à n)
    - B : fantôme bleu (0 à n)
    - P : fantôme rose (0 à n)
    - O : fantôme orange (0 à n)
    - 0 : couloir contenant un bonus Gum (bonus classique +50 points)
    - 1 : mur
    - . : couloir vide
    - + : couloir contenant un bonus SpeedPacman (accélère Pacman +100 points)
    - - : couloir contenant un bonus SlowPacman (ralenti Pacman +500 points)
    - ~ : couloir contenant un bonus SlowGhost (ralenti tous les fantômes +100 points)
    - # : couloir contenant un bonus SpeedGhost (accélère tous les fantômes +500 points)
    - $: couloir contenant un bonus HunterBonus (permet à Pacman de manger les fantômes pendant 200 déplacements)
    - ! : couloir contenant un bonus StupidGhost (rend les fantômes stupides pendant 250 déplacements).

II. Description des classes :

A. Personnages :

1. Character :

La classe Character est une classe abstraite qui définit les bases d'un personnage. Elle comprend divers attributs liés à l'affichage graphique de celui-ci et notamment les objets graphiques de la bibliothèque SDL (comme SDL_Texture, SDL_Surface_ SDL_Rect). La classe contient également des attributs liés au déplacement du personnage sur le plateau (speed_ la vitesse du personnage et direction_ la direction du personnage).

2. Ghost :

La classe Ghost est une classe abstraite qui définit les bases des fantômes tout en héritant des spécificités de la classe Character. Elle est composée des différents états de mouvements que peuvent prendre les fantômes (voir III.A.1 GhostMovementState).

3. Fantômes concrets :

Les différents fantômes du jeu (rouge, bleu, orange, rose) héritent directement de la classe abstraite fantôme. Ils ont chacun une intelligence artificielle différente (représentée par un GhostMovementState).

4. Pacman :

La classe Pacman hérite de la classe abstraite Character. En plus des éléments graphiques classiques d'un personnage, le Pacman est composé de quelques éléments graphiques supplémentaires comme des sprites pour réaliser une animation et un sprite de particule. La classe comprend également les différents états possible du Pacman (voir III.A.2 PacmanState).

B. Bonus :

1. Bonus :

La classe Bonus est une classe abstraite qui définit le comportement classique d'un bonus. Elle comprend les éléments graphiques nécessaires au bon affichage du bonus. Deux méthodes abstraites sont redéfinies dans les bonus concrets : getPoint() et getBonusType(). Ces deux méthodes permettent respectivement de récupérer le nombre de points offerts par la prise du bonus et le type du bonus.

2. Bonus concrets :

Les bonus concrets (Gum, SpeedPacman, SlowPacman, Hunter, SpeedGhost, SlowGhost, StupidGhost) héritent de la classe abstraite Bonus et redéfinissent les méthodes getPoint() et getBonusType() en fonction du nombre de points offerts par le bonus et du type du bonus.

C. Éléments de map :

1. MapElement :

La classe MapElement est une classe abstraite qui définit le comportement classique d'un élément du plateau de jeu. Elle est composé d'un Bonus et des divers éléments graphiques nécessaires à l'affichage de l'élément de map sur la fenêtre du jeu.

2. Elements de map concrets :

La éléments de map concrets (Lane et Wall) héritent de la classe abstraite MapElement. La particularité de la classe Wall est qu'elle ne puisse pas posséder à proprement parler de bonus (bonus pointe vers nullptr). Les deux classes redéfinissent la méthode abstraite canBeCrossed() : renvoie un booléen qui indique si un personnage peut traverser l'élément de map (Vrai pour Lane, Faux pour Wall).

D. Moteur du jeu (GameEngine) :

Classe principale du jeu. Le moteur de jeu comprend tous les éléments qui composent le plateau de jeu :

- Une matrice de MapElement (des murs et des couloirs).
- Un Pacman.
- Une liste de fantômes.

Il est également composé de quelques éléments graphiques supplémentaires (pour l'affichage du score du joueur par exemple).

Le moteur de jeu se charge de la création dynamique des différents éléments du plateau via l'utilisation d'un design pattern de création particulier dont nous parlerons plus loin : le pattern Prototype.

Le moteur de jeu se charge également du rendering (affichage) des différents éléments du jeu ainsi que du déroulement du jeu via des appels à toutes les méthodes de calcul du prochain tour de jeu.

Un tour de jeu classique se comporte comme suit : calcul du déplacement des personnages, effacement de l'écran de jeu, affichage du plateau de jeu, affichage des personnages, affichage du score du joueur, calcul des collisions Pacman/Fantômes, calcul de la condition de victoire du joueur (toutes les Gum mangées).

Les bonus du jeu permettent de déclencher des modifications sur les personnages, c'est pourquoi la gestion de ceux-ci est réalisée par le moteur du jeu via la méthode handleBonus() qui prend en paramètre le type du bonus (qui est alors fourni via la méthode getBonusType() des bonus).

III. Design Patterns implémentés :

A. State :

1. GhostMovementState :

Voir diagramme de classe Annexe A.1

L'intelligence artificielle des fantômes est gérée par un design pattern State. Lorsque nous avons réfléchi aux fonctionnalités que nous aimerions mettre en place dans notre Nampac, nous avions évoqué la possibilité de modifier pendant l'exécution du programme

l'intelligence artificielle des fantômes. De plus, nous avions pensé à "bloquer" les transitions d'états dans certains cas particulier. Le design pattern de comportement State nous a paru le plus adapté puisqu'il permet de modifier dynamiquement le comportements d'objets tout en définissant des transitions différentes entre les états.

Les fantômes possèdent tous des attributs des états de déplacement qu'ils pourront prendre pendant l'exécution du jeu ainsi qu'une variable de l'état de déplacement courant de type GhostMovementState (classe abstraite). C'est sur cette variable qu'ils vont pouvoir déclencher une demande de calcul de direction via la méthode calculateDirection() (abstraite dans la classe GhostMovementState, redéfinie pour chaque état) ainsi qu'une demande de déplacement en fonction de la direction calculée via la méthode moveCharacter() (redéfinie dans certains états).

Chaque type de fantôme est lié lors de sa création à un état de déplacement qui lui est propre (affecté à la variable d'état courant) parmi les états suivants : GhostMovementUnpredictable (mouvement imprévisible), GhostMovementChase (mouvement de chasse du Pacman), GhostMovementAmbush (mouvement d'embuscade du Pacman), GhostMovementStupid (mouvement stupide).

Un dernier état particulier est défini : GhostMovementDead. Il s'agit d'un état simulant la mort d'un fantôme lorsque celui-ci est "mangé" par Pacman. Dans cet état, le fantôme ne peut pas se déplacer ni calculer de nouvelle direction. Le fantôme retrouve son état classique à la fin d'un décompte de tours de jeu.

Lorsque le moteur de jeu fait une demande de calcul de direction au fantôme, cette demande est redistribuée à l'état de déplacement courant du fantôme qui se charge de réaliser l'action. Il en est de même pour le déplacement.

Le moteur de jeu a la possibilité de faire une demande de transition d'un état vers un autre au fantôme (via des méthodes askChangeMovement non représentées sur le diagramme simplifié). La demande est alors redistribuée à l'état courant du fantôme qui va vérifier si la transition est possible (par exemple : tant qu'un fantôme est dans l'état mort (GhostMovementDead), il ne peut pas changer d'état). Lorsqu'une transition est réalisée vers

un état qui n'est pas l'état classique d'un fantôme, l'état ainsi modifié déclenchera un retour vers l'état classique du fantôme au bout d'un certain nombre de tours de jeu.

Par exemple, le Bonus StupidGhost va déclencher par le biais du moteur de jeu une demande de transition des fantômes vers l'état de déplacement stupide GhostMovementStupid (modification d'intelligence artificielle). Cette demande sera traitée différemment selon l'état dans lequel sont actuellement les différents fantômes.

2. PacmanState:

Voir diagramme de classe Annexe A.2

Comme dans le jeu officiel, notre Pacman a la possibilité de pouvoir "manger" les fantômes. Il est naturel de penser qu'il y ait deux états qui puissent être échangés au cours de l'exécution du jeu: un état dans lequel le Pacman peut être mangé par les fantômes (état proie) et un état dans lequel le Pacman peut manger les fantômes (état chasseur). Le design pattern de comportement Stratégie pouvait répondre à ce type de problématique mais nous avions également besoin de transitions différentes en fonction de l'état du Pacman. Par exemple, quand le Pacman est déjà dans l'état chasseur et qu'il mange un bonus pour devenir chasseur (HunterBonus), le temps du bonus est ajouté au temps restant du bonus précédent. En revanche, dans le cas où il est dans l'état proie lorsqu'il mange un bonus chasseur, il change d'état pour devenir chasseur. C'est pourquoi nous avons décidé d'utiliser le pattern de comportement State.

Nous avons ainsi modélisé deux états : le premier qui est l'état dans lequel le Pacman est la proie (PreyState) et le second dans lequel le Pacman est le chasseur (HunterState). L'état par défaut est celui de la proie.

Le Pacman possède donc les états qu'il lui est possible de prendre pendant le jeu (attributs hunterState_ et preyState_ héritants de la classe abstraite PacmanState) et un état courant (currentState_) qui pointe à l'initialisation vers l'état de proie (preyState_). Ces états contiennent la méthode canEatGhost() qui renvoie un booléen, vrai pour l'état HunterState et faux dans l'état PreyState. Les méthodes decrementRemainingMovement() et addRemainingMovement(m: int) ne font rien dans l'état "proie" mais dans l'état "chasseur"

font respectivement la décrémentation du nombre de tours restants dans l'état chasseur (ainsi quand le nombre de tours passe à zéro, on change d'état) et l'ajout d'un nombre de tours du bonus au compteur de tours restants (appelé quand le pacman mange un bonus alors qu'il est déjà dans l'état chasseur grâce à la méthode changeStateHunter()). Il y a dans chaque état des méthodes qui permettent de changer l'état courant comme changeStatePrey() pour passer de l'état chasseur à l'état proie, et la méthode changeStateHunter() pour passer de l'état proie à chasseur. La méthode changeStatePrey() dans l'état proie ne fait rien car nous sommes déjà dans l'état proie.

B. Decorator :

Voir diagramme de classe Annexe B.1

Lorsque nous avons réfléchi aux modifications que pourraient engendrer les bonus sur les différents objets composant notre Nampac, nous avions pensé mettre en place une modification de la vitesse de déplacement des différents personnages du jeu. Une des caractéristiques principales de cette modification de vitesse devait être le cumul de la prise de bonus. En effet, prendre plusieurs bonus de modification de vitesse devait entraîner plusieurs modifications de vitesse du personnage. Pour traduire cette idée de cumul, d'"empilement" de modification de comportement, nous avons mis en place le design pattern de structure Decorator qui nous a paru le plus adapté. En effet, le Decorator permet d'ajouter dynamiquement des comportements à un objet tout en conservant cette idée de "pile d'appel" de l'objet décoré lors de l'appel d'une de ses méthodes.

Le Decorator que nous avons mis en place (CharacterDecorator) décore la classe Character : il hérite de Character et contient un Character (l'objet décoré). Les méthodes de Character non modifiées par le décorateur se contentent d'appeler les même méthodes sur l'objet décoré. De cette manière, quels que soit le nombre de décorations réalisées sur l'objet, le comportement du Character décoré sera le même pour les méthodes non modifiées par le Decorator.

Deux classes concrètes héritent du CharacterDecorator : SpeededCharacter et SlowedCharacter. Ces classes représentent les modifications de vitesses qui peuvent être réalisées sur un personnage : une augmentation de vitesse ou une diminution de vitesse.

C'est le getter getSpeed() (qui renvoie la vitesse du personnage) qui est décoré par ces deux classes : la méthode getSpeed() définie dans les deux classes concrètes appelle le getSpeed() de l'objet décoré et lui applique un pourcentage (75% pour la diminution de vitesse, 125% pour l'augmentation). De cette manière, quel que soit le nombre de décorations que l'on applique sur un personnage, sa vitesse sera modifiée dynamiquement.

La décoration des personnages est réalisée lors de la récupération des bonus SpeedPacman, SlowPacman, SpeedGhost, SlowGhost.

Prenons un exemple de scénario de récupération de bonus : Pacman se déplace à une vitesse de 5 pixels/tour de jeu. Il ramasse un bonus SpeedPacman et se déplace maintenant à une vitesse de 5*1.25 soit environ 6 pixels/tour de jeu. Il ramasse un autre bonus SpeedPacman et se déplace à une vitesse de (5*1.25)*1.25 soit environ 8 pixels/tour de jeu. Il ramasse un bonus SlowPacman et se déplace alors à une vitesse de ((5*1.25)*1.25)*0.75 soit environ 6 pixels/tour de jeu. Sur cet exemple on voit bien le principe d'appels récursifs/empilements que nous voulions mettre en place et qui est parfaitement traduit grâce au pattern Decorator.

C. Prototype :

Voir diagrammes de classes Annexe C

Notre plateau de jeu étant créé dynamiquement avec la lecture d'un fichier, la création des différents objets du jeu devait également être dynamique. Nous avions dans un premier temps pensé à mettre en place le design pattern de création Factory mais nous nous sommes rendu compte que celui-ci n'était pas forcément adapté puisque la diversité de classes à instancier ne justifiait pas l'utilisation d'une Factory. Nous nous sommes donc intéressés à d'autres patterns de création et c'est le pattern Prototype qui nous a paru le plus adapté à répondre à notre problème.

Le pattern de création Prototype permet d'alléger en temps et en mémoire la création d'objets très similaires par le biais d'une méthode clone() réalisée sur un objet instancié en tant que prototype d'objet de la même classe. La méthode clone() appelle le constructeur de recopie de l'objet pour créer un nouvel objet du même type et réalise sur cet objet des modifications spécifiques pour le transformer en un objet utilisable (les objets sont très

similaires mais pas identiques). Le pattern Prototype est particulièrement adapté à notre Nampac puisque les objets que nous devons créer sont nombreux et très similaires (bien souvent les seules choses qui diffèrent sont les positions des objets sur l'interface graphique).

Nous avons mis en place le pattern Prototype sur la création de 3 éléments du jeu : les fantômes, les bonus et les éléments de la map. Afin de simplifier l'utilisation du pattern, nous avons créé une classe template abstraite Prototype<T> contenant une méthode template clone() prenant en paramètres les positions des objets sur l'interface graphique. Les différents objets à créer étendent cette classe template paramétrée avec leur type respectif.

Les différents clonages sont réalisés dans des classes clientes Factory (MapElementFactory, BonusFactory, GhostFactory). Ces classes clientes sont composés d'un objet prototype de chaque type d'objet qu'il sera possible d'instancier et sur lesquels sont appelées les méthodes clone(). La méthode clone renvoie un objet modifié avec les attributs qui ne sont pas les mêmes que l'objet prototype (position de l'objet sur l'interface graphique par exemple).

Malheureusement l'intérêt du pattern Prototype a été limité par une contrainte technique imposée par la bibliothèque graphique que nous utilisons : SDL2. En effet, une très grosse optimisation en terme de mémoire et de temps d'exécution aurait pu être réalisée en appliquant une recopie des pointeurs vers les éléments graphiques de SDL (qui sont identiques pour tous les objets d'une même classe), très gourmand en terme de mémoire et de temps de création. Cependant, les objets graphiques SDL sont des objets très particuliers qui ne réagissent pas bien avec le partage de pointeurs (et qui le signalent par un "Segmentation Fault"). Il est donc **obligatoire** de recréer ces éléments graphiques dans chaque objet ce qui réduit quelque peu l'intérêt du pattern Prototype.

IV. Difficultés rencontrées et améliorations :

A. Librairie SDL2 :

La mise en place de la bibliothèque SDL2 a été un point central de notre projet. Nous n'avions jamais développer d'interface graphique en C++, c'est pourquoi la compréhension de la bibliothèque SDL2 et de ses spécificités a été l'un des point complexe du projet.

De plus, nous nous sommes heurtés à une spécificité des objets graphiques de SDL2. En effet, tout au long du développement du jeu, nous avons tenté de nous familiariser avec les smart pointers qui nous étaient jusqu'alors inconnu. La particularité des objets graphiques de SDL2 (comme SDL_Texture, SDL_Surface, SDL_Renderer, ...) est qu'ils utilisent des méthodes d'instanciation (comme SDl_CreateTexture()) et de destruction (comme SDL_DestroyTexture()) à la place des constructeurs et des destructeurs classiques. Or les smart pointers sont naturellement incompatibles avec les objets de ce genre. Des solutions semblent exister pour régler ce problème (notamment des surcharges des opérateurs des smart pointers) mais nous n'avons pas pu nous pencher dessus faute de temps.

B. Langage C++ :

Durant notre cursus précédent nous étions plutôt amenés à développer en Java. Nous avions déjà développé en C++ mais rien d'aussi conséquent. Il a fallu se réadapter, apprendre les mécanique du C++, revoir les bases. De plus, nous avons tenté de mettre en place des outils différents de ceux que nous avions utilisé comme les smart-pointers ou encore Doxygen.

Tout au long du développement du projet, nous avons eu quelques problèmes de "Core Dump" / "Segmentation Fault" un peu obscures qui nous ont parfois bloqués et l'utilisation de GDB n'était pas toujours d'une grand aide.

C. Problème de temps :

Le développement du projet a pris plus de temps que prévu car nous avons utilisé beaucoup d'outils que nous n'avions jamais utilisé auparavant comme la bibliothèque SDL2 ou les smart-pointers par exemple.

Ce projet étant un projet centré autour de l'utilisation des Design Pattern, nous avons été obligé d'accorder plus de temps à la mise en place des différents patterns que nous avions identifié lors de la conception du projet qu'à d'autres éléments du jeu.

A l'origine, nous avions prévu de mettre en place un algorithme d'inondation couplé à un algorithme de calcul de plus court chemin travaillant sur notre matrice d'éléments de jeu afin de réaliser des intelligences artificielles de fantômes plus convaincantes (et fonctionnelles). Malheureusement, le manque de temps nous a contraint à développer des I.A. très basiques pour les fantômes. Cependant, le design pattern State que nous avons mis en place pour l'I.A. des fantômes rend le code ouvert à l'ajout de nouvelles choses et facilitera sans aucun doute la mise en place de véritables I.A.

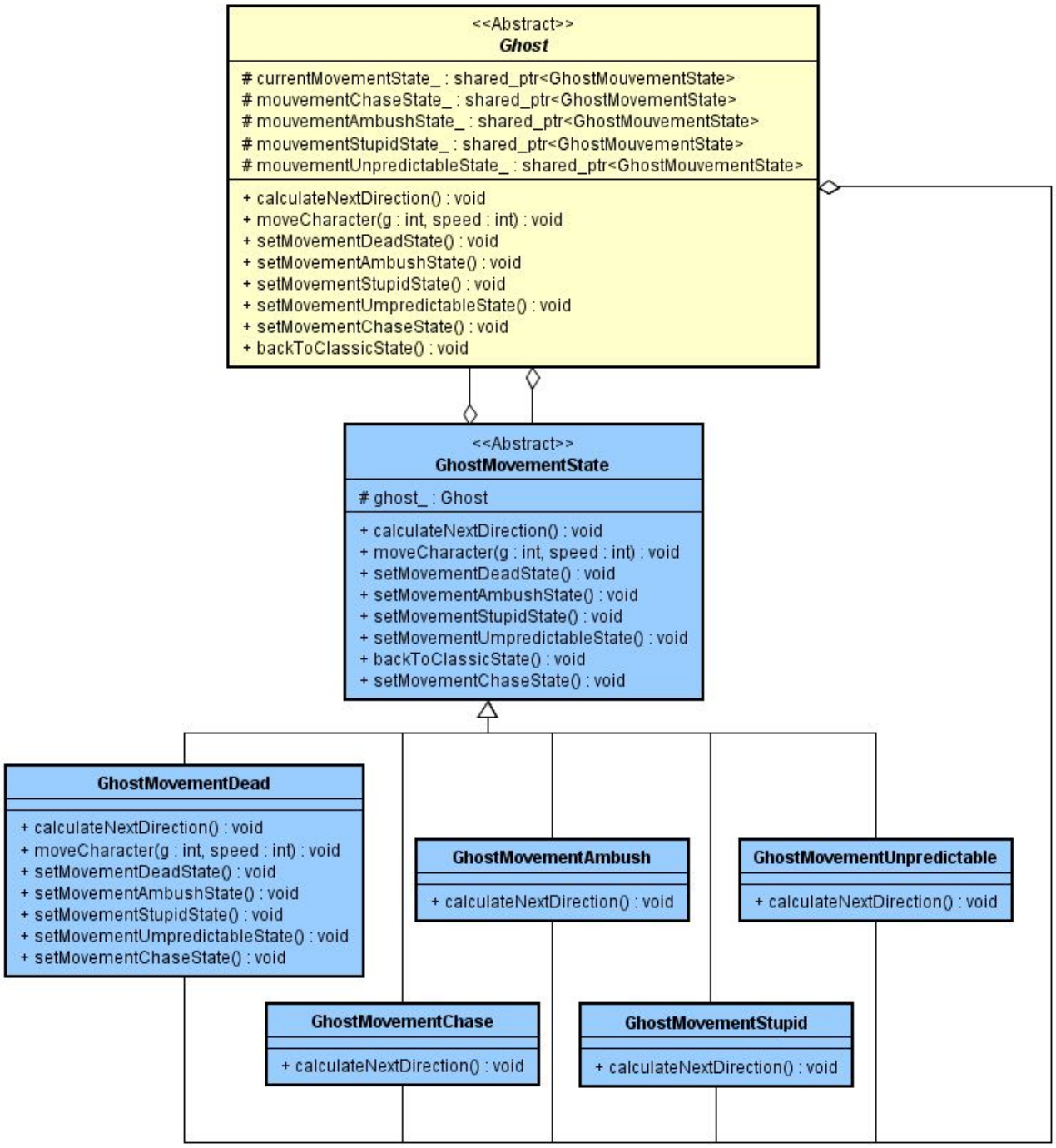
De plus, nous n'avons pas pu consacrer tout le temps que nous aurions souhaité à la gestion de la mémoire utilisée par le programme. Nous avons essayé dans la mesure du possible d'utiliser les outils fourni par C++ (smart-pointers, objets STL, ...) mais certaines contraintes techniques (imposées notamment par la bibliothèque SDL2) nous ont empêché de réaliser un gestion mémoire optimale dans les temps.

V. Annexes :

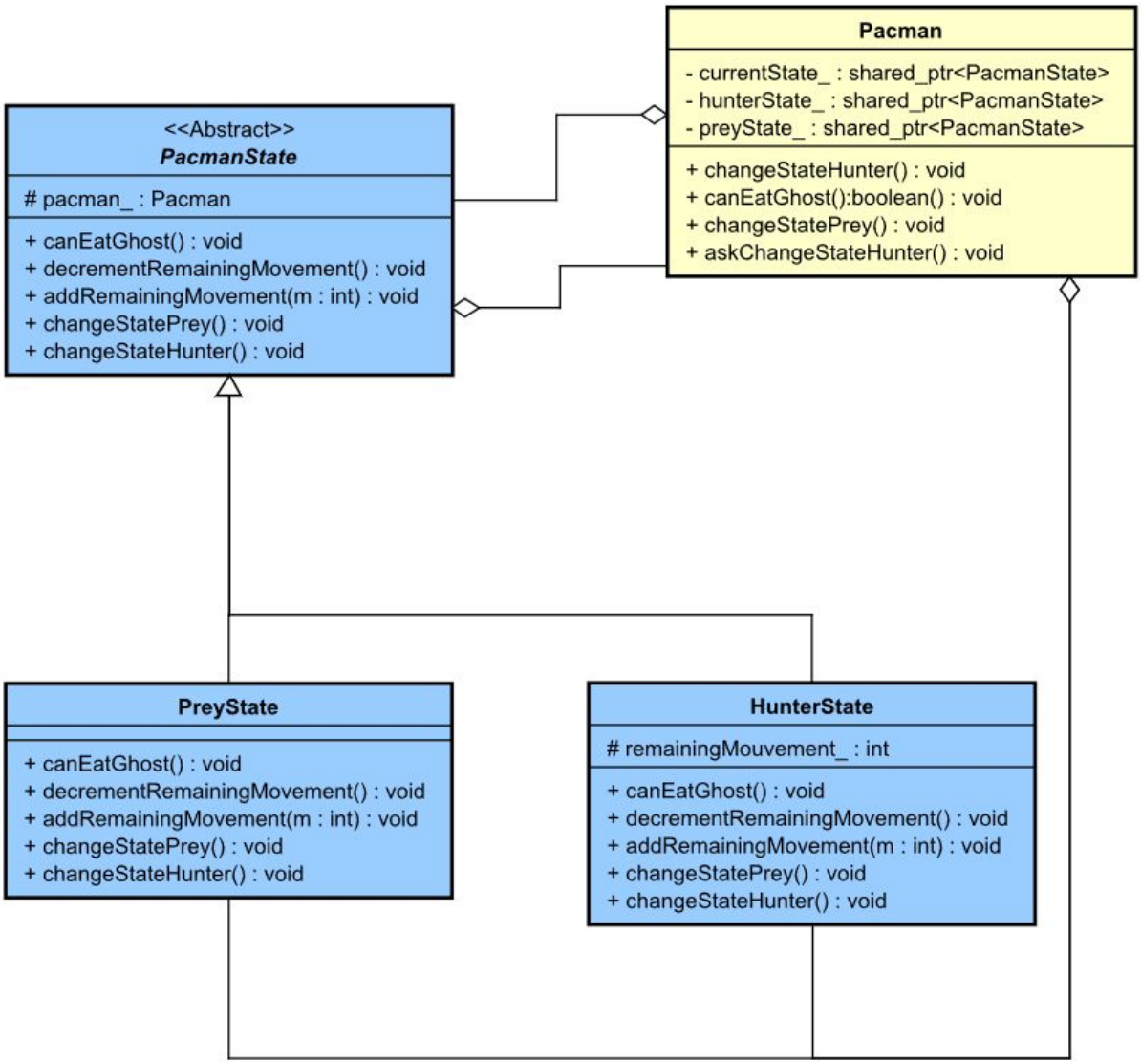
Les diagrammes UML (complets et simplifiés) sont disponible dans le dossier diagram/ du projet.

A. Les patterns State:

1. GhostsMovementState simplifié :

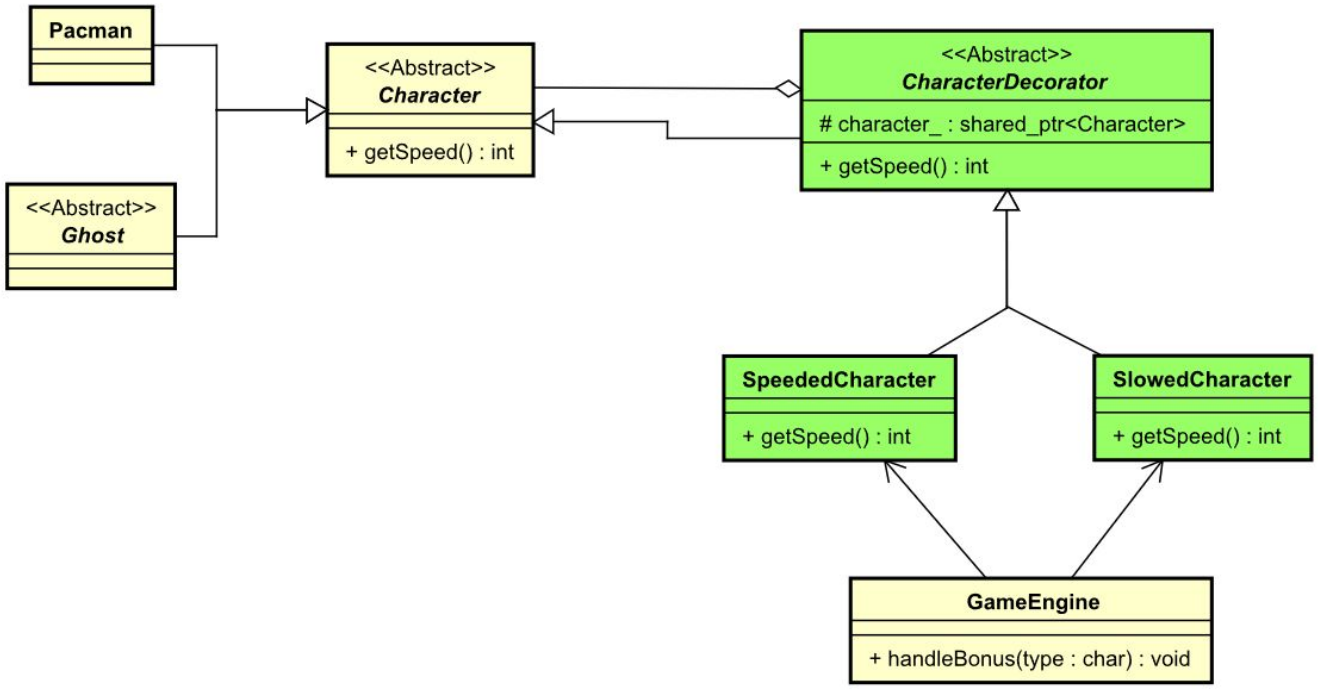


2. PacmanState simplifié :



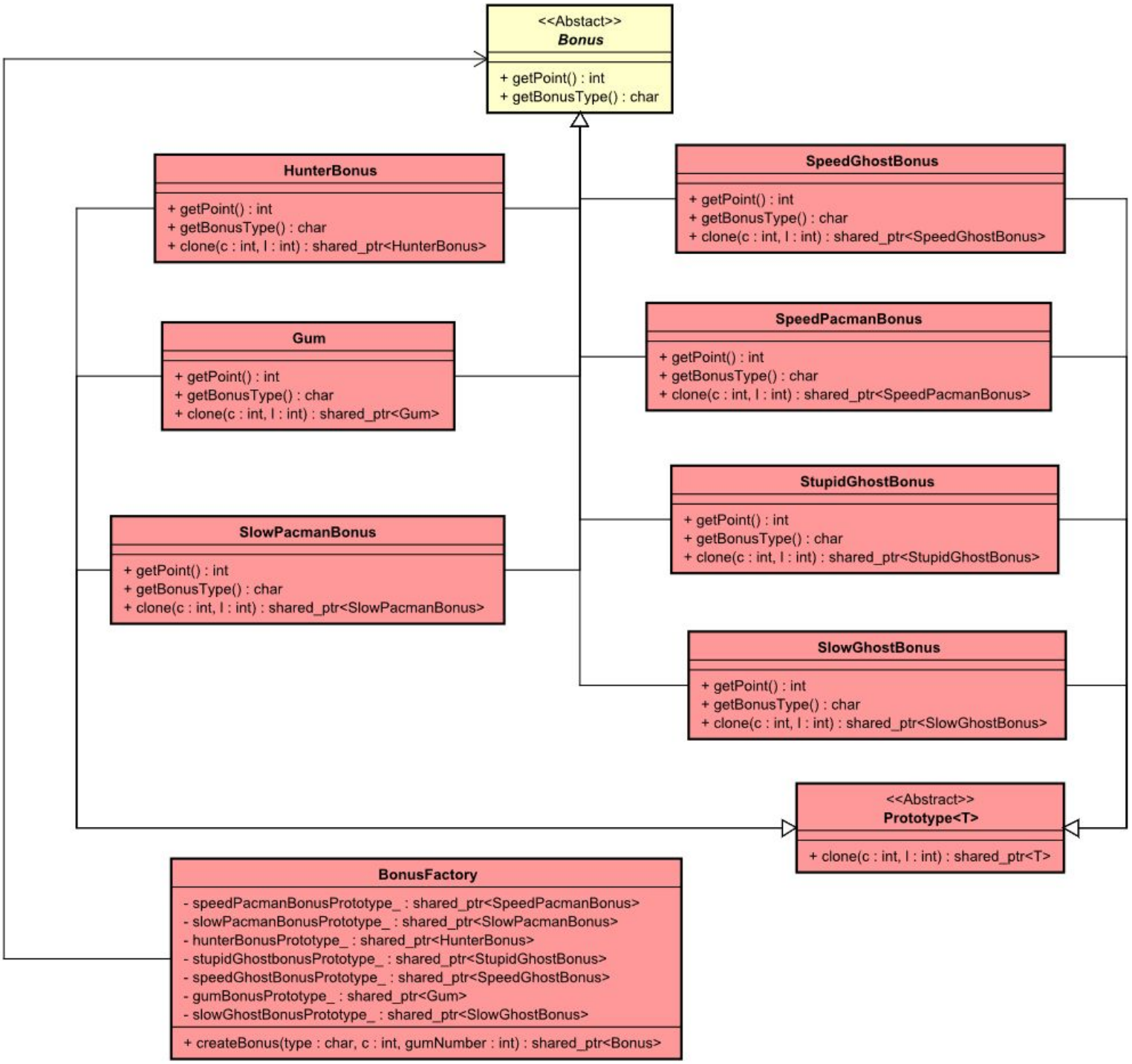
B. Le pattern Decorator :

1. CharacterDecorator simplifié :

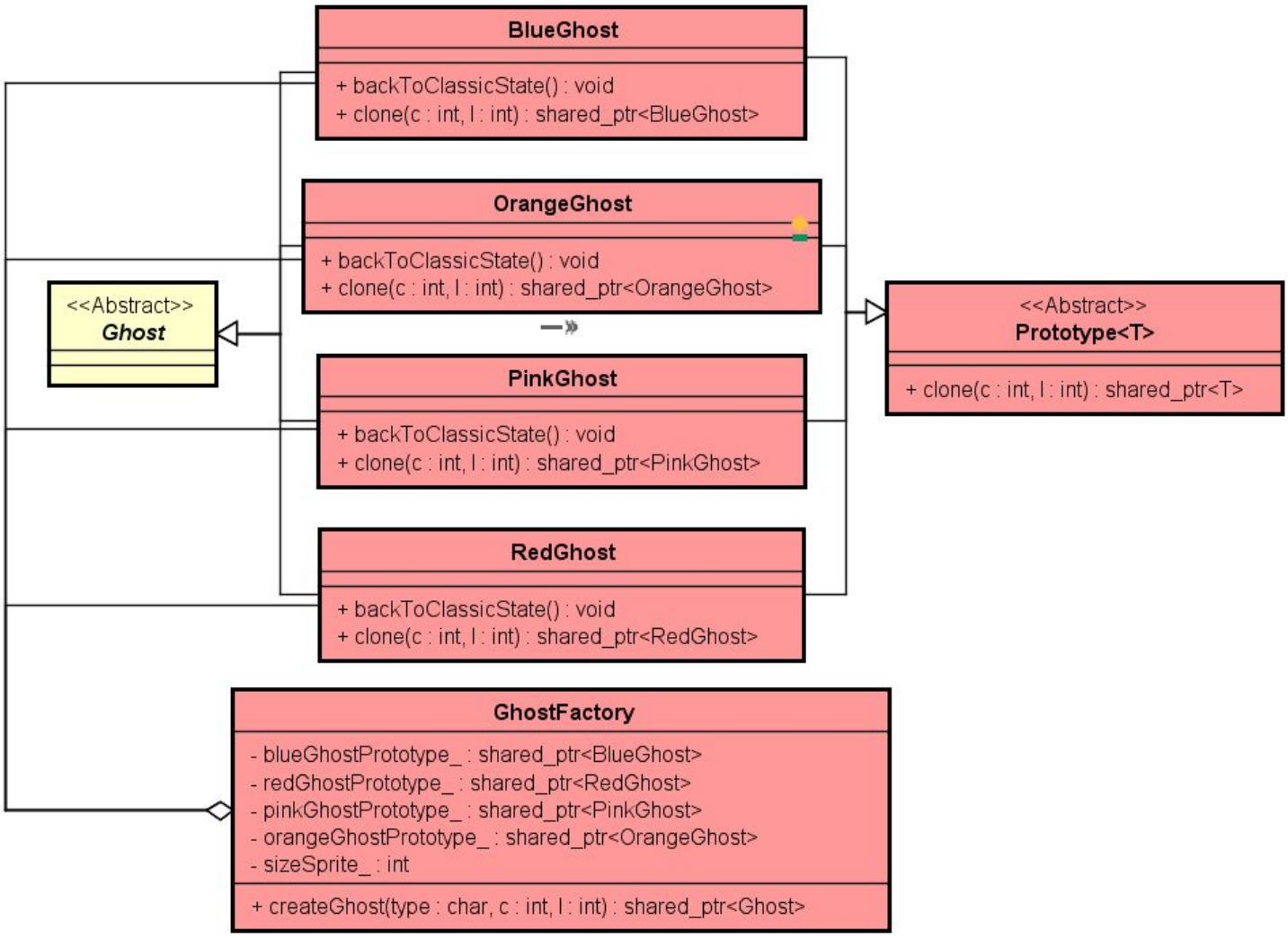


C. Les patterns Prototype:

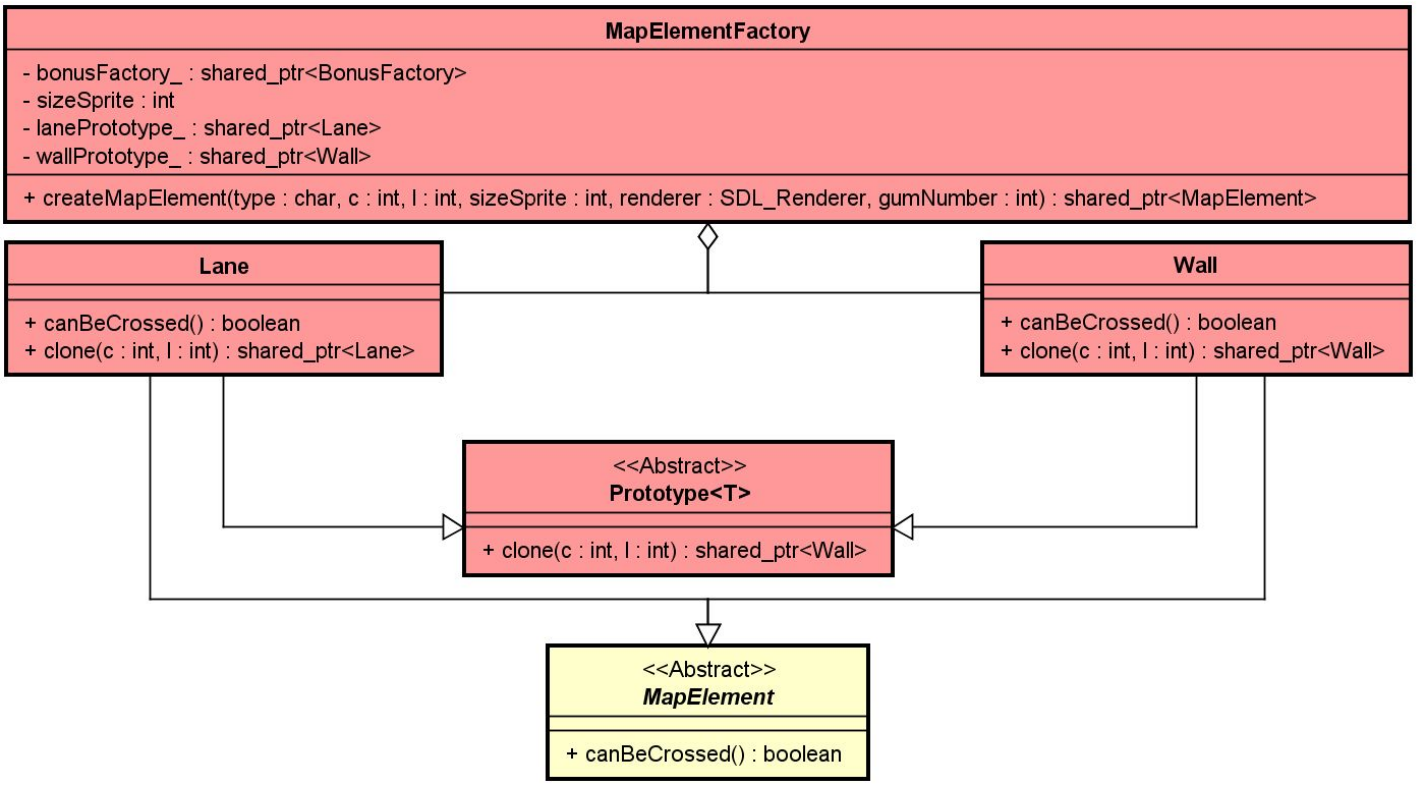
1. Appliqué à la création des Bonus (simplifié) :



2. Appliqué à la création des Ghost :



3. Appliqué à la création des MapElement :



D. Diagramme complet:

Le diagramme UML complet n'étant pas du tout clair et lisible une fois imprimé, nous vous encourageons à le consulter ainsi que les autres diagrammes (complets et simplifiés) dans notre dossier projet (sous-dossier diagram/).