Projet Logiciel « Pacman »

Philippe-Henri Gosselin

Ce document est un exemple de rapport attendu pour votre projet logiciel, mais également un guide qui vous fournira un certain nombre de conseils pour bien réussir. N'oubliez pas que ce document est le rapport ***(presque)*** *final – en conséquence, les premières sections sont dans un état bien plus avancé que ce que vous pourrez présenter au début du projet.*

Note importante : cet exemple de rapport n'est terminé que pour les modules « Génie Logiciel », « Algorithmique » et « Programmation Parallèle ». Les thèmes du module « App. Mobile et Web services » ne sont pas abordés !

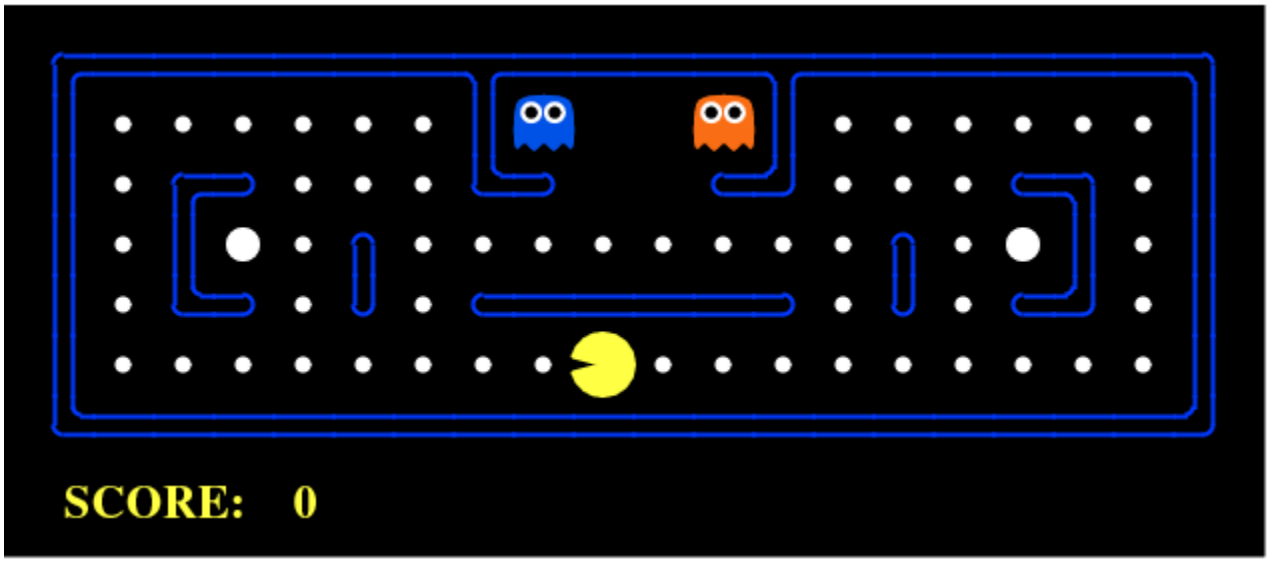


Illustration 1: Exemple du jeu Pacman

Table des matières

1 Objectif

1.1 Présentation générale

Présenter ici une description générale du projet. On peut s'appuyer sur des schémas ou croquis pour illustrer cette présentation. Éventuellement, proposer des projets existants et/ou captures d'écrans permettant de rapidement comprendre la nature du projet.

L'objectif de ce projet est la réalisation du jeu pacman, avec les règles les plus simples. Un exemple est présenté en Illustration 1.

1.2 Règles du jeu

Présenter ici une description des principales règles du jeu. Il doit y avoir suffisamment d'éléments pour pouvoir former entièrement le jeu, sans pour autant entrer dans les détails . Notez que c'est une description en « français » qui est demandé, il n'est pas question d'informatique et de programmation dans cette section.

Le joueur déplace pacman dans un labyrinthe, et doit manger toutes les pastilles pour réussir le niveau. Si les fantômes touchent pacman dans son état normal, celui-ci perds une vie. Si pacman n'a plus de vie, cela met immédiatement fin à la partie. Enfin, si pacman mange une grande pastille, il devient capable de dévorer les fantômes.

1.3 Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.

L'ensemble des principaux packages de la solution finale est présenté en Illustration 2 :

Package state. C'est le package dont dépend tous les autres : il contient toutes les informations permettant de représenter un état du jeu.
Package engine. Ce package contient les éléments qui permettent d'appliquer les règles du jeu. Plus précisément, ces fonctionnalités permettent de modifier un état de jeu en fonction d'événements, comme les touches du clavier ou un paquet réseau. Notons que la notions d'événements est purement abstraite dans ce package, et sera implanté au sein des packages enfants.
Package ai. Le rôle de ce package est la création de joueurs artificiels. Contrairement à un joueur humain qui se fit au rendu graphique pour prendre ses décisions, les joueurs artificiels ont besoin d'un moteur de jeu. Etant donné que tout est connu dans le cas du jeu Pacman (par exemple, il n'y a pas de « brouillard de guerre »), les joueurs artificiels n'ont pas besoin d'approximer ou simuler le moteur de jeu. On peut utiliser directement le moteur réel, sans offrir un avantage stratégique aux joueurs artificiels.
Package server. Le premier rôle de ce package est de faire fonctionner le moteur de jeu, i.e. le mettre à jour régulièrement. Le second rôle de ce package est de de recevoir les commandes, soit de manière directe (appel méthode), soit de manière indirecte (paquet réseau).
Package render. Ici, il s'agit de représenter la « conversion » d'un état de jeu en version graphique. Notons bien que tout ici est formel, et que les liens vers des librairies graphiques sont abstraits.
Package client. Ce package gère l'interaction entre un serveur de jeu, le rendu de son état, et les différentes entrées possibles (clavier, réseaux, …).

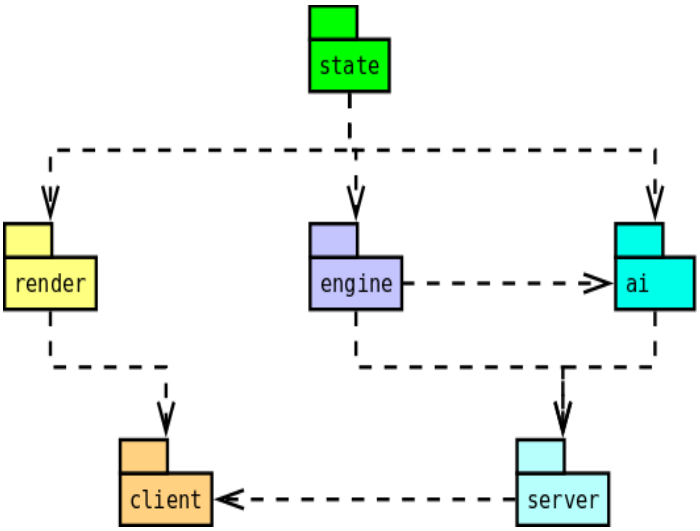


Illustration 2: Diagramme des packages

2 Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très appriécé. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (le labyrinthe) et un ensemble d'éléments mobiles (pacman et fantômes). Tous les éléments possèdes les propriétés suivantes :

- Coordonnées (x,y) dans la grille
- Identifiant de type d'élément : ce nombre indique la nature de élément (ie classe)

2.1.1 Etat éléments fixes

Le labyrinthe est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :

Cases « Mur ». Les cases « mur » sont des éléments infranchissable pour les éléments mobiles. Le choix de la texture est purement esthétique, et n'a pas d'influence sur l'évolution du jeu.

Cases « Espace ». Les cases « espace » sont les éléments franchissables par les éléments mobiles. On considère les types de cases « espace » suivants :

- Les espaces « vides »
- Les espaces « pastille », qui contiennent une pastille
- Les espaces « grande pastille », qui contiennent une grande pastille
- Les espaces « cimetière », qui servent uniquement à définir les lieux où les fantômes apparaissent au début du jeu, mais également les lieux où ceux dévorés par pacman peuvent reprendre une forme normale.
- Les espaces « départ », qui définissent une position initiale possible pour pacman

2.1.2 Etat éléments mobiles

Les éléments mobiles possède une direction (aucune, gauche, droite, haut ou bas), une vitesse, et une position. Une position à zéro signifie que l'élément est exactement sur la case ; pour les autres valeurs, cela signifie qu'il est entre deux cases (l'actuelle et celle définie par la direction de l'élément). Lorsque la position est égale à la vitesse, alors l'élément passe à la position suivante. Ainsi, plus la valeur numérique de vitesse est grande, plus le personnage aura un déplacement lent. Ce système est choisi pour pouvoir être toujours synchronisé avec une horloge globale (cf section suivante sur les changements d'états).

Element mobile « Pacman » . Cet élément est dirigé par le joueur, qui commande la propriété de direction. Pacman dispose également d'un « compteur super », qui sert à déterminer le temps restant avant de repasser en normal. Enfin, on utilise une propriété que l'on nommera « status », et qui peut prendre les valeurs suivantes :

- Status « normal » : cas le plus courant, où pacman peut se déplacer dans le labyrinthe, et doit éviter les fantômes.
- Status « super » : cas où pacman peut manger les fantômes

- Status « mort » : cas où pacman a été pris par un fantôme.

Elements mobiles « Fantôme ». Ces éléments sont également commandés par la propriété de direction, qu'elle proviennent d'un humain ou d'une IA. Ces éléments possèdent deux propriétés particulières. La première est la « couleur », qui est purement esthétique. La seule règle concernant cette couleur est qu'elle est unique. La seconde propriété particulière est le « status », qui peut prendre les valeurs suivantes :

- Status « normal » : cas le plus courant, où le fantôme peut tuer pacman
- Status « fuite » : cas où le fantôme peut être tué par « super » pacman
- Status « mort » : cas où le fantôme a été dévoré par « super » pacman

2.1.3 Etat général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :

- Epoque : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tic » de l'horloge globale depuis le depuis de la partie.
- Vitesse : le nombre d'époque par seconde, ie la vitesse à laquelle l'état du jeu est mis à jour
- Compteur de pastille : le nombre de pastille qu'il reste sur le plateau

2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Illustration 3, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Element. Toute la hiérarchie des classes filles d'*Element* (en jaune) permettent de représenter les différentes catégories et types d'élément. C'est un exemple très classique d'utilisation du ***Polymorphisme***. Etant donné qu'il n'y a pas de solution standard efficace[1] en C++ pour faire de l'introspection, nous avons opté pour des méthodes comme *isStatic()* qui indiquent la classe d'un objet. Ainsi, une fois la classe identifié, il suffit de faire un simple cast.

Fabrique d'élements. Dans le but de pouvoir fabriquer facilement des instances d'Element, nous utilisons la classe *ElementFactory*. Cette classe, qui suit le patron de conception ***Abstract Factory***, permet de créer n'importe quelle instance non abstraite à partir d'un caractère. L'idée est de l'utiliser, en autres, pour créer un niveau à partir d'un fichier texte. Par exemple, au caractère « - », on fait correspondre une instance de *Wall* avec la propriété « wallTypeId » égale à HORI.
Dans le but de faciliter l'enregistrement des différentes instancation possible, nous avons créé le couple de classes *AElementAlloc* et *ElementAlloc<E,ID>*. La première est une classe abstraite dont le seul but est de renvoyer une nouvelle instance. La seconde sert à créer des implantations de la première pour une classe et une propriété d'identification particulière. Par exemple, pour créer le créateur de mur horizontal et l'enregister dans une factory, il suffit de faire :

```
factory.registerType('-',new ElementAlloc<Wall,WallTypeId>(HORI));
```

Conteneurs d'élément. Viennent ensuite les classes *State*, *ElementList et ElementGrid* qui permettent de contenir des ensembles d'éléments. *ElementList* contient une liste d'éléments, et *ElementGrid* étends ce conteneur pour lui ajouter des fonctions permettant de gérer une grille. Enfin, la classe *State* est le conteneur principal, avec une grille pour le niveau, et une liste pour

1 Il existe « Run-Time Type Information (RTTI)”, cependant cela est peu portable et souffre de sérieux problèmes de performance.

pacman et les fantômes.

2.3 Conception logiciel : extension pour le rendu

Observateurs de changements. Dans le diagramme de classes, nous présentons en mauve les classes permettant à des tiers de réagir lorsqu'un événement se produit dans l'un des éléments d'état. Les premiers intéressés par cette fonctionnalité sont les clients de rendu, qui pourront mettre à jour les textures/sprites en fonction des changements au sein de l'état actuel.
Les observateurs implantent l'interface *StateObserver* pour être avertis des changements de propriétés d'état. Pour connaître la nature du changement, ils analysent l'instance de *StateEvent*. La conception de ces outils suit le patron ***Observer***.

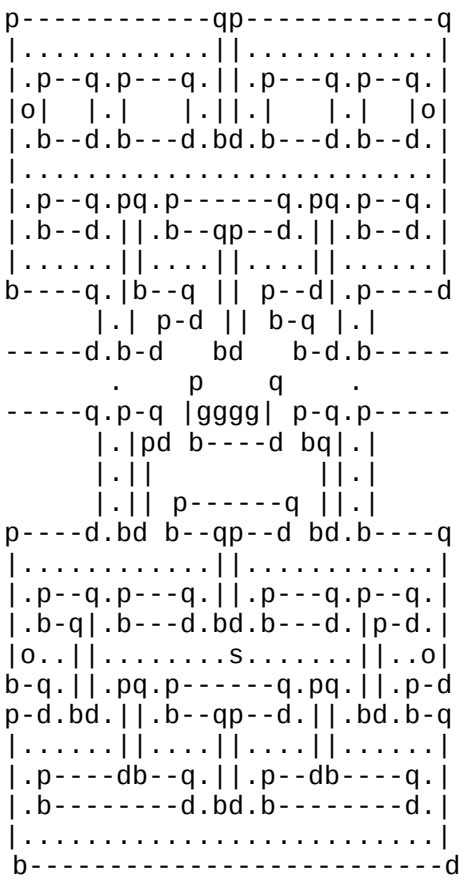
2.4 Conception logiciel : extension pour le moteur de jeu

Au sein des différentes classes d'*Element*, ainsi que leurs conteneurs comme *ElementList* et *State*, nous avons ajouté des méthodes de clonage et de comparaison :

- Méthodes *clone()* : elles renvoie une copie complète (les sous-éléments sont également copiés) de l'instance.
- Méthodes *equals()* : elles permettent de comparer l'instance actuelle avec une autre, et renvoie *true* si les deux instances ont un contenu identique.

2.5 Ressources

Les niveaux sont codés en fichier texte, puis traduits en instance d'éléments grâce à la factory. Voici un exemple :



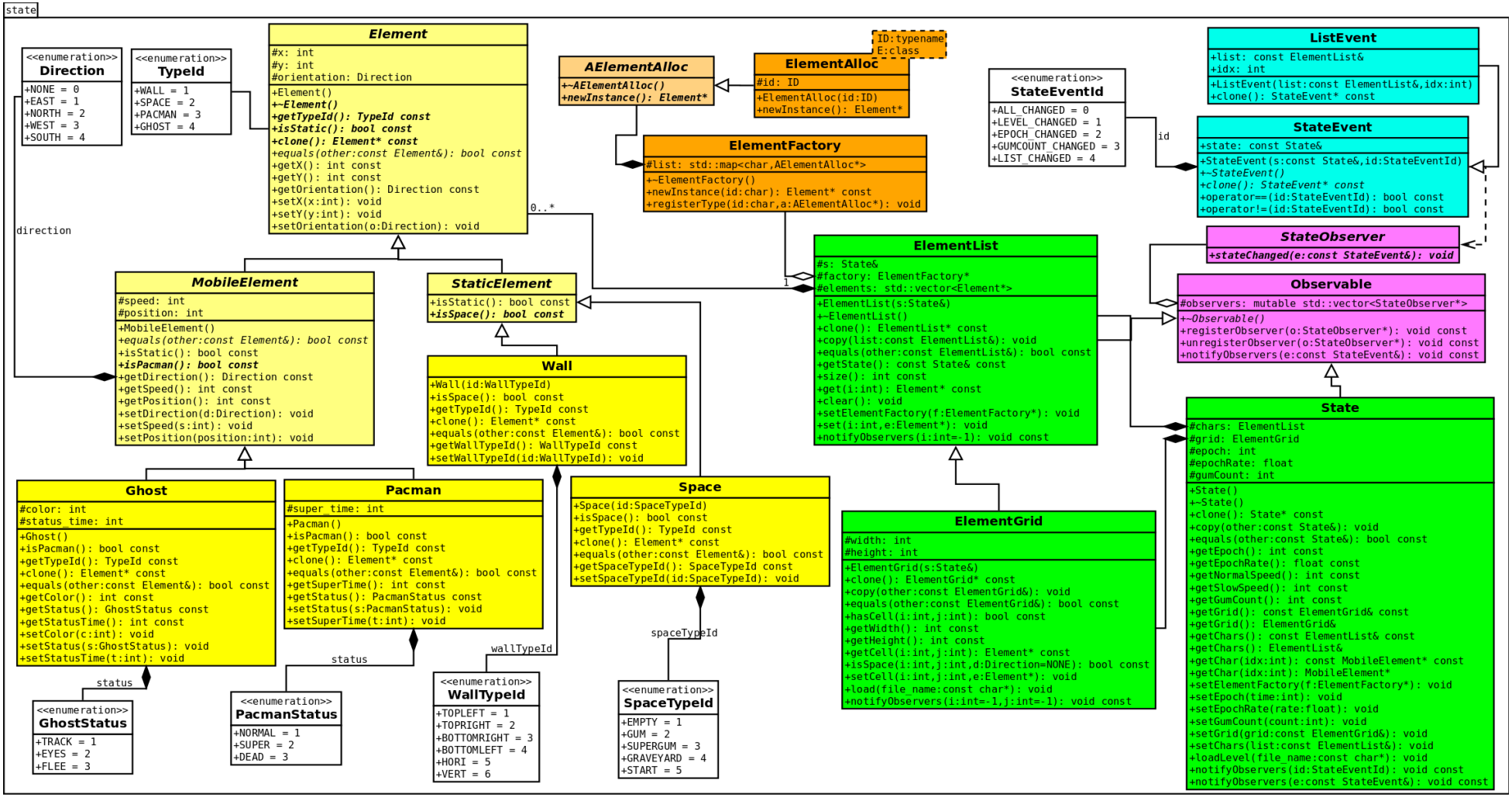


Illustration 3: Diagramme des classes d'état

3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons opté pour une stratégie assez bas niveau, et relativement proche du fonctionnement des unités graphiques. En effet, les cartes graphiques actuelles, tout comme celles des années 80, sont plus efficace si le CPU prépare les éléments à rendre au sein de structures élémentaires, avant de tout envoyer au GPU.

Plus précisément, nous découpons la scène à rendre en plans (ou « layers ») : un plan pour le niveau (mur, pastilles, etc.), un plan pour les éléments mobiles (pacman, fantômes) et un plan pour les informations (vies, scores, etc.). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles (ou « tiles »), et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

Pour la formation de ces informations bas-niveau, la première idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanent, comme les animations et/ou les éléments mobiles, nous tiendrons à jour une liste d'éléments visuels à mettre à jour (= modifier la matrice du plan) automatiquement à chaque rendu d'une nouvelle frame.

En ce qui concerne les aspects de synchronisation, nous avons deux horloges : celle des changements d'états, et celle de la mise à jour du rendu à l'écran. Chaque horloges pourra tenir le rythme qui lui convient, avec pour seule hypothèse que l'horloge des changements d'états sera plus lente que celle des rendus. En général, l'horloge des changements d'états sera dans les 4-12Hz alors que celle des rendus dans les 30-60Hz. En conséquence, il faut interpoler entre deux changements d'états pour pouvoir obtenir un rendu lisse :

- Pour les animations qui n'ont aucun lien avec l'état (comme le clignotement des super pastilles), nous avons choisi de définir une fréquence pour chaque animation. A partir de cette information, les animations tournent en boucle, sans corrélation avec les deux horloges.
- Pour les animations qui ont un lien avec l'état (comme le déplacement de Pacman), on produit un rendu équivalent un à sous-état fictif, produit d'une horloge fictive des changements d'états synchronisée avec celle du rendu. Par exemple, pour le déplacement d'un élément mobile, on calcule la position intermédiaire entre celle de l'état courant, et celle de l'état à venir.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique, est présenté en Illustration 4.

Plans et Surfaces. Le cœur du rendu réside dans le groupe (en jaune) autour de la classe *Layer*. Le principal objectif des instances de *Layer* est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une implantation de *Surface*. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique choisie. L'ensemble classe *Surface* avec ses implantations suivent donc un patron de conception de type ***Adapter***. La première information donnée est la texture du plan, via la méthode *loadTexture()*. Les informations qui permettrons à l'implantation de *Surface* de former la matrice des positions seront données via la méthode *setSprite()*. Notons que, dans un soucis d'efficacité, nous indexons tous les éléments graphiques (« sprites »). Ainsi, la surface sait qu'il faut gérer un nombre fixe de sprites, chacun identifié par un numéro unique.
La classe *Layer* est le cas général, que l'on peut spécialiser avec des instances de *StateLayer* et *ElementListLayer*, chacun étant capable de réagir à des notifications de changement d'état via le mécanisme d'***Observer***.

Scène. Tous les plans sont regroupés au sein d'une instance de *Scene*. Les instances de cette classe seront liées (« bind ») à un état particulier. Une implantation pour une librairie graphique particulière fournira des surfaces via les méthodes *setXXXSurface()*. Notons bien que les instances ont pour rôle de remplir les surfaces, mais pas de les rendre : cela restera le travail de la librairie choisie.

Tuiles. La classe *Tile* ainsi que ses filles (en bleu clair) ont pour rôle la définition de tuiles au sein d'une texture particulière, ainsi que les animations que l'on peut former avec. La classe *StaticTile* stocke les coordonnées d'une unique tuile, et la classe *AnimatedTile* stocke un ensemble de tuiles. Etant donné qu'*AnimatedTile* est à la fois une classe fille de *Tile* et un conteneur de celle-ci, nous sommes donc face à un patron de conception de type ***Composite***. Enfin, les implantations de la classe *TileSet* stocke l'ensemble des tuiles et animations possibles pour un plan donné. Notons que nous ne présentons pas dans le diagramme des implantations, uniquement la classe abstraite *TileSet*. En outre, on peut voir ces classes comme un moyen de définir un thème graphique : lors de la création d'instance de *Layer*, on pourra choisir n'importe quel jeu de tuile, pourvu qu'il contiennent des tuiles cohérentes avec le plan considéré.

3.3 Conception logiciel : extension pour les animations

Animations. Les animations sont gérées par les instances de la classe *Animation*. Chaque plan tient une liste de ces animations, et le client graphique fait appel aux méthodes de mise à jour pour faire évoluer ses surfaces. Nous faisons ce choix car certaines évolutions dans l'affichage ne dépendent pas d'un changement d'état, ou sont d'une fréquence différente. Pour les animations, on peut distinguer deux cas (non exclusifs) :

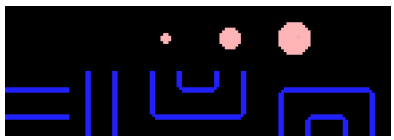
- Animations sans mouvement : Par exemple, le fait que Pacman ouvre et ferme sa bouche en continue, ou bien le clignotement des super-pastilles. Dans tous les cas, on suppose que ces animations n'ont aucun rapport avec l'état du jeu. En conséquence, à chaque rafraîchissement de l'affichage (généralement 60 fois par seconde), le client graphique appel la méthode *update()* de *Scene*, qui fera de même pour tous les plans. Ces appels modifient la

texture des éléments concernés au rythme défini par le sprite, ie *AnimatedTile::getRate()*.

- Pour le cas des animations de mouvement, par exemple Pacman qui se déplace, nous avons ajouté toutes les informations permettant d'afficher le déplacement (direction et vitesse) sans dépendre de l'état. Ainsi, lorsque une information de mouvement parvient au plan, elle est pleinement définie dans une instance d'*Animation* et peut se prolonger de manière autonome. En outre, cette animation est synchronisée avec les changement d'état grâce à la méthode *sync()*, qui permet de transmettre le moment précis où un état vient de changer. C'est grâce à ces astuces que l'on peut voir les personnages se déplacer à 60 images par secondes même si le jeu évolue bien plus lentement, par exemple à 4 changements d'état par seconde.

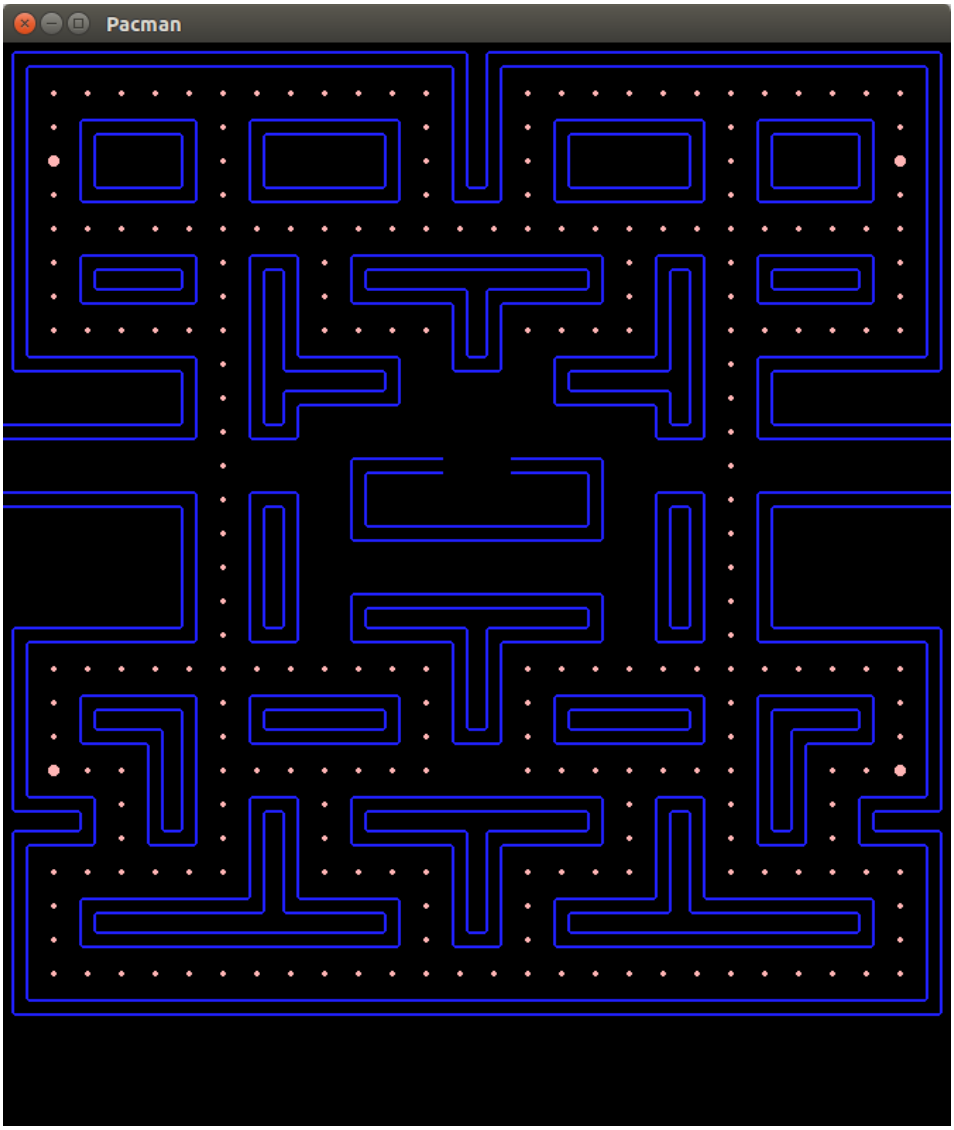
3.4 Ressources

Voici un exemple de textures pour le plan grille :



3.5 Exemple de rendu

Voici un exemple de rendu avec une reproduction pas totalement fidèle du niveau le plus classique :



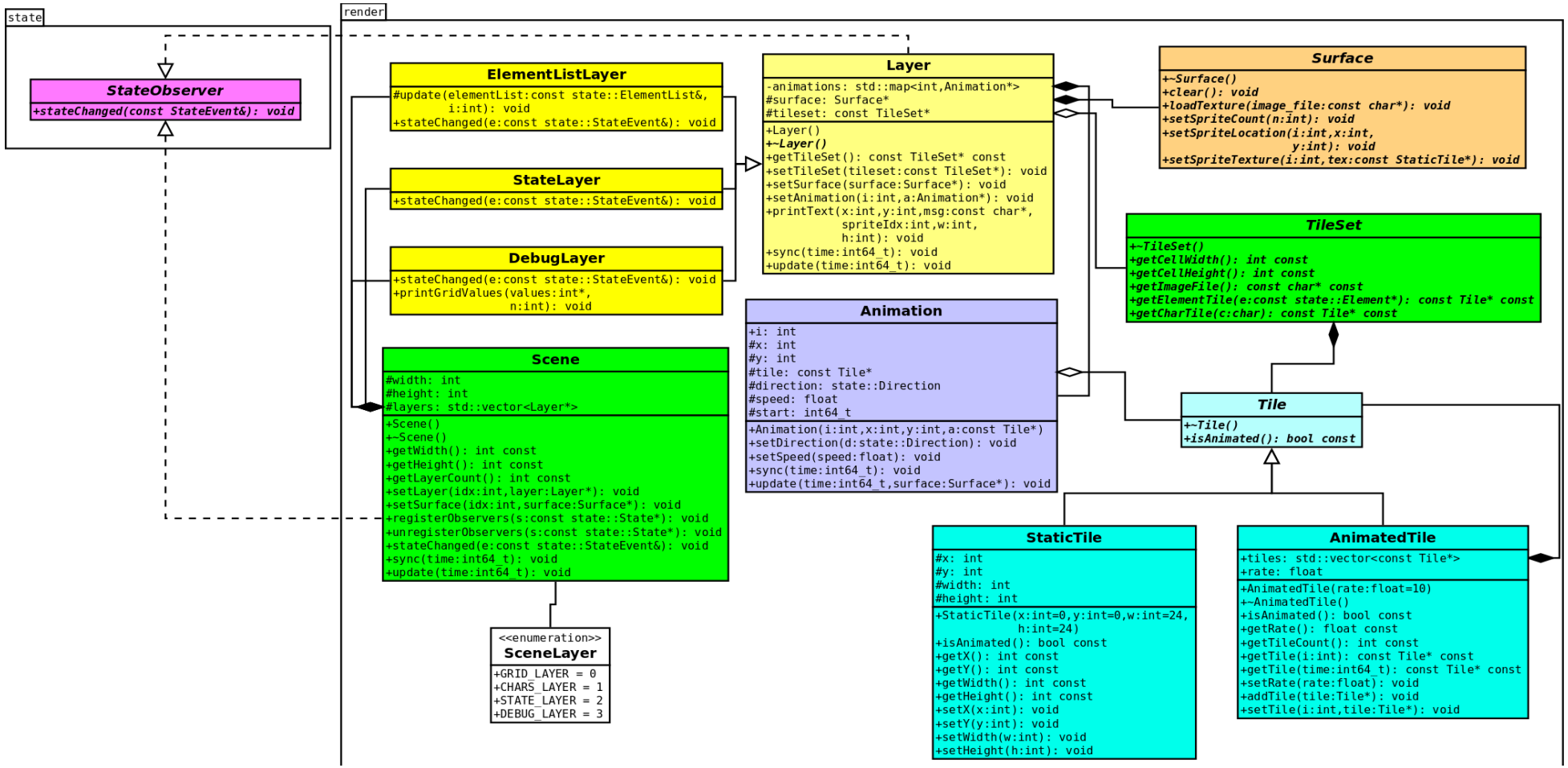


Illustration 4: Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

Les changements d'état suivent une horloge globale : de manière régulière, on passe directement d'un état à un autre. Il n'y a pas de notion d'état intermédiaire. Ces changements sont calibrés sur le temps qu'il faut pour un élément mobile à vitesse maximale pour passer d'une case à une autre. En conséquence, tous les mouvements auront une vitesse fonction de cet élément temporel unitaire.
Notons bien que cela est décorrélé de la vitesse d'affichage : l'utilisateur doit avoir l'impression que tout est continu, bien que dans les faits les choses évoluent de manière discrète. Par exemple, s'il y a 60 images par seconde, et que le temps unitaire est de 200ms, alors pendant un temps unitaire, 12 étapes d'animation seront affichées pour les éléments en mouvement.

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un ordre provenant du réseau :

1. Commandes principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier ; « Nouvelle partie » : Tout est remis à l'état initial
2. Commandes « Mode » : On modifie le mode actuel du jeu, comme « normal », « pause », « rejouer la partie », etc.
3. Commandes « Direction personnage », paramètres « personnage », « direction » : Si cela est possible (pas de mur), la direction du personnage est modifiée.

4.3 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état, après les changements extérieurs. Ils sont exécutés dans l'ordre suivant :

1. Si le « compteur super » est non nul, le décrémenter. S'il passe à zéro, pacman perd le statut « super » et tous les fantômes en fuite obtiennent le status « normal ».
2. Appliquer les règles de mouvement pour pacman
3. Si pacman est sur une case « pastille », on décrémente le compteur de pastilles.
4. Si pacman est sur une case « grande pastille », on décrémente le compteur de pastilles et pacman obtient le status « super » avec un compteur super au maximum. En outre, tous les fantômes obtiennent le status « fuite ».
5. Si le compteur de pastilles est nul, pacman deviens super pour toujours !
6. Si pacman et un fantôme sont sur la même case (dès qu'on réunit les conditions pour un cas, on ne cherche plus à réaliser les suivants):
   a) Si le fantôme a le status «mort», il ne se passe rien
   b) Si pacman a le status « super », le fantôme obtient le status « mort »
   c) Sinon, pacman devient un fantôme, et est condamné à errer sans fin dans le niveau.
7. Pour tous les éléments mobiles sauf pacman, appliquer les règles de mouvement
8. Répéter les opérations en 6 (ie retester si pacman rencontre un fantôme)

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 5.
L'ensemble du moteur de jeu repose sur un patron de conception de type ***Command***, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

Classes Command. Le rôle de ces classes est de représenter une commande extérieure, provenant par exemple d'une touche au clavier (ou tout autre source). Notons bien que ces classes ne gère absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes. A ces classes, on a défini un type de commande avec *CommandTypeId* pour identifier précisément la classe d'une instance. En outre, on a a défini une catégorie de commande, dont le but est d'assurer que certaines commandes sont exclusives. Par exemple, toutes les commandes de direction pour un personnage sont exclusives : on ne peut pas demander d'aller à la fois à gauche et à droite. Pour l'assurer, toutes ces commandes ont la même catégorie, et par la suite, on ne prendra toujours qu'une seule commande par catégorie (la plus récente).

Engine. C'est le coeur du moteur. Elle stocke les commandes dans une instance de *CommandSet*. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode *update()* après un temps suffisant, le principal travail du moteur est de transmettre les commandes à une instance de *Ruler.* C'est cette classe qui applique les règles du jeu. Plus précisément, et en fonction des commandes ou de règles de mises à jour automatiques, elle construit une liste d'actions. Ces actions transforment l'état courant pour le faire évoluer vers l'état suivant.

Action. Le rôle de ces classes est de représenter une modification particulière d'un état du jeu. Notons bien que ce ne sont pas les règles du jeu : chaque instance de ces classes applique la modification qu'elle contient, sans se demander si cela a un sens.

4.5 Conception logiciel : extension pour l'IA

Nous exploitons tout le potentiel du patron ***Command*** en ajoutant les fonctionnalités suivantes.

Rollback. Nous avons ajouté aux classes *Action* une méthode *undo()* ainsi que les attributs nécessaires pour permettre d'annuler une action. Cela permet de revenir en arrière (« rollback »), par exemple pour remonter dans le graphe d'état sans avoir à stocker un état complet à chaque nœud.

Record. Ces mécanismes nous permettent d'enregistrer toutes les actions, et par conséquence de rejouer, à l'endroit ou à l'envers, tout ce qui a été enregistré. Notons que cette fonctionnalité nous permet de valider l'implantation des classes actions. Par exemple, si un retour en arrière n'a pas reconduit à l'état initial, on peut en déduire qu'il y a un problème dans l'implantation des actions.

4.6 Conception logiciel : extension pour la parallélisation

Engine. Dans le but de pouvoir utiliser le moteur de jeu dans un thread séparé, nous avons ajouté un deuxième jeu de commandes appelé *waitingCommands* qui récupère les commandes envoyés au moteur de jeu lorsque celui-ci met à jour l'état du jeu. Plus de détails dans la section 6.1.1.

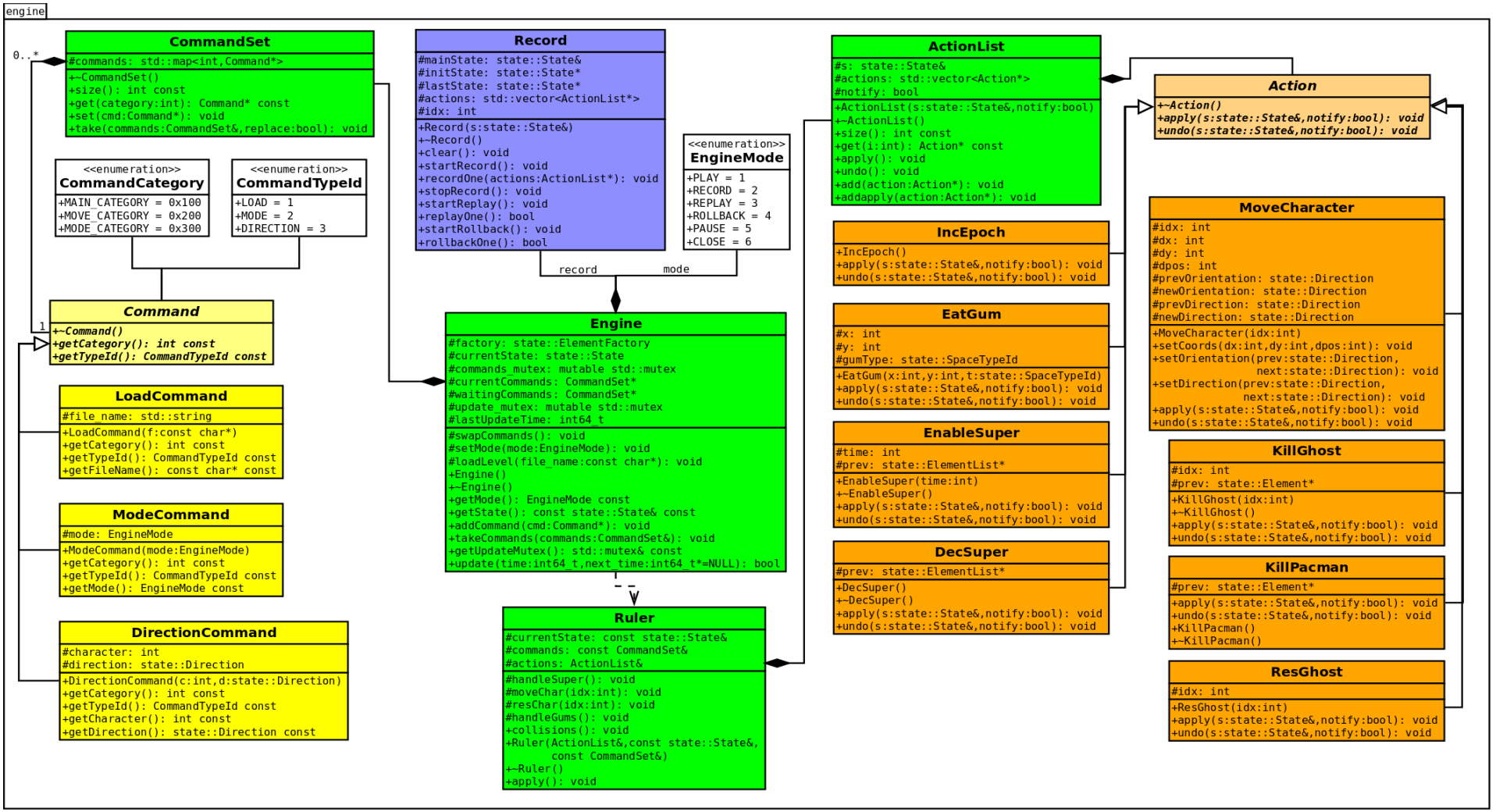


Illustration 5: Diagrammes des classes pour le moteur de jeu

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

L'ensemble de notre stratégie d'intelligence artificielle repose sur le principe des poupées russes. L'ensemble est décomposé en différents niveaux d'intelligence, de la plus sommaire à la plus avancée. Les niveaux supérieurs font appel aux niveaux inférieurs pour réduire les possibilités à étudier, en éliminant les comportements absurdes ou « dangereux ».

5.1.1 Intelligence minimale

Dans le but d'avoir une sorte d'étalon, mais également un comportement par défaut lorsqu'il n'y a pas de critère pour choisir un comportement, nous proposons une intelligence extrêmement simple, basée sur les principes suivants :

- Tant que c'est possible on avance : on ne fait demi-tour que si on est dans un cul de sac. Cela supprime les mouvements erratiques.
- Lorsqu'on arrive à une bifurcation, on choisit aléatoirement l'une des possibilités. Cela permet d'assurer l'exploration complète du niveau, tout en rendant les mouvements imprévisibles.

5.1.2 Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard, et avec une chance notable de résoudre le problème complet (ie, manger toutes les pastilles sans mourir) :

- Si Pacman est trop près d'un fantôme, on s'en éloigne le plus possible
- Sinon on se dirige vers la pastille la plus proche

Nous proposons également des heuristiques pour les fantômes, afin d'offrir un peu plus de challenge au joueur, tout en restant raisonnable. Rappelons que dans la conception de notre moteur de jeu, les fantômes sont dirigés comme Pacman : il n'y a pas de changement notable dans l'exploitation d'une intelligence pour ces personnages. Voici les heuristiques que nous avons choisi :

- Si le fantôme est normal, il se déplace simplement (ie, stratégie intelligence minimale)
- Si le fantôme peut être mangé par Pacman, il s'éloigne le plus possible de Pacman
- Si le fantôme a été dévoré par Pacman, il se dirige vers le cimetière

La plupart des heuristiques proposées sont mis en œuvre en utilisant des cartes de distance vers un ou plusieurs objectifs. Pour calculer ces cartes, nous utilisons l'algorithme de Dijkstra.

5.1.3 Intelligence basée sur les arbres de recherche

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états finis. Dans cette configuration, un état est un état du jeu à une époque donnée, tel que défini en section 2. Les arcs entre les sommets du graphe d'état sont les changements d'états, définis en section 4. Passer d'un sommet du graphe d'état à un autre reviens à passer d'une époque à une autre du jeu, fonction de l'ensemble des commandes reçues (clavier, réseau, IA,…). L'évaluation/score d'un sommet/état du jeu est déterminé par le nombre pastille restantes si Pacman est vivant. Un sommet/état du jeu avec Pacman mort a un score infini, et n'a qu'un seul arc (sommet/état juste

avant la mort de Pacman). Le meilleur choix de mouvement pour Pacman est donc celui du plus court chemin dans le graphe d'état qui mène vers un score nul (toutes les pastilles ont été mangées).

Pour trouver ce chemin, nous suivons des méthodes basées sur les arbres de recherche, avec une propriété importante. En effet, nous n'envisageons pas de dupliquer l'état du jeu à chaque sommet du graphe d'état : compte tenu du nombre de nœuds que nous allons traiter, nous aurions rapidement des problèmes de mémoire. Nous n'allons considérer qu'un seul état que nous modifions suivant la direction choisie par la recherche. Si le sommet suivant est à une époque suivante, i.e. on descend dans l'arbre de recherche, on applique les commandes associées, et notre état gagne une époque. Si le sommet suivant est à une époque précédente, i.e. on remonte dans l'arbre de recherche, on annule les commandes associées, et notre état retrouve sa forme passée. Ces contraintes interdisent les recherches en largeur d'abord, ainsi que A^* . Cependant, même si dans l'ensemble une recherche par profondeur d'abord doit être suivi, il reste néanmoins possible d'utiliser quelques idées de A^* , comme privilégier certaines branches et ne pas analyser celles qui n'ont aucune chance d'offrir un meilleur résultat. Pour ce faire, il va donc nous falloir des heuristiques : nous utilisons tout simplement celles proposées pour les intelligences précédentes.

5.2 Conception logiciel

Classes AI. Toutes les formes d'intelligence artificielle implantent la classe abstraite *AI*. Le rôle de ces classes est de fournir un ensemble de commandes à transmettre au moteur de jeu. Notons qu'il n'y a pas une instance par personnage, mais qu'une instance doit fournir les commandes pour tous les personnages. La classe *DumbAI* implante l'intelligence minimale, telle que présentée ci dessus. De même, la classe *HeuristicAI* implante la version améliorée.

PathMap. La classe *PathMap* permet de calculer une carte des distances à un ou plusieurs objectifs. Plus précisément, pour chaque case « espace » du niveau, on peut demander un poids qui représente la distance à ces objectifs. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids. Même si cela n'est pas optimal, on peut également utiliser ces poids pour s'éloigner des objectifs, en choisissant une case avec un poids supérieur. L'interface *PathMapTarget* nous permet de fabriquer des cartes de distances pour de nombreux types d'objectifs (patron de type ***Strategy***). Nous avons implanté un certain nombre, qui ne sont pas présentés sur le diagamme :

- *GraveyardTarget* : Objectif cases cimetières
- *GumTarget* : Objectif pastilles (normales ou super)
- *PacmanTarget* : Objectif Pacman
- *GhostsTarget* : Objectif n'importe quel fantôme capable de manger Pacman

5.3 Conception logiciel : extension pour l'IA composée

ComposedAI et Behavior. Le but de cette première extension est de permettre de former librement des combinaisons de méthodes d'intelligence. Par exemple, on peut imaginer que certains fantômes sont plus agressifs que d'autres, ou que leur comportement change avec le temps, selon des humeurs. Pour ce faire, on réuni un ensemble d'implantation de *Behavior* au sein de la classe *ComposedAI*, avec un comportement pour chaque personnage. Le rôle de chaque instance de *Behavior* est de proposer un ensemble de commandes pertinentes pour un personnage. Puis, une commande est choisie aléatoirement par ComposedAI. La première et la plus simple des implantations est *AnyBehavior,* qui propose toutes les commandes possibles (i.e. éviter les directions inutiles, qui vont vers un mur). La classe *RandomBehavior* propose l'ensemble des commandes d'une stratégie d'intelligence simple.

Catégories de comportements. Nous avons réuni les comportements similaires en sous-classes abstraites de *Behavior.* La classe *TargetBehavior* gère les heuristiques basées sur les cartes de distances, où il faut s'approcher ou s'éloigner d'objectifs. Nous retrouvons sur le diagramme les implantation des heuristiques présentées précédemment. La classe *CompositeBehavior* permet le changement de comportement en fonction de la situation. C' est un patron de conception de type ***Composite***. Nous présentons sur le diagramme les implantations pour Pacman et les fantômes, avec les règles présentées précédemment. Notons toutefois que bien d'autres règles peuvent être imaginées.

5.4 Conception logiciel : extension pour IA avancée

DeepAI. Nous proposons ici une implantation pour une IA basée sur la résolution de problèmes à état fini, tel que défini dans la section 5.1.3. Pour ce faire, nous avons tout d'abord créé la classe *SearchTree*. Les instances de cette classe contiennent l'état actuel, ainsi qu'une instance de *ComposedIA*. L'IA composée nous permet de définir l'ensemble des choix de mouvement que peuvent faire les personnages. Le cas le plus exhaustif est de choix *AnyBehavior* pour tous les personnages : on teste toutes les possibilités, sauf celles totalement inutiles. Ce choix de conception permet également de limiter drastiquement les possibilités. Par exemple si on connaît l'heuristique suivie par les fantômes, il suffit de choisir la même dans l'instance de *ComposedIA* de notre IA avancée.

Extension de Behavior. Nous avons modifié la classe *Behavior* pour qu'elle renvoie l'ensemble des choix (classe *Choice*) que peut faire un personnage qui suit un comportement particulier. Puis, c'est au client de ces classes de choisir un choix parmi ceux proposés. En outre, nous offrons la possibilité de donner un score à chaque choix. Par exemple, pour le comportement de type *TargetBehavior*, le score est la distance à l'objectif pour la direction choisie. Dans l'exploration de notre arbre de recherche, nous utilisons ces scores pour estimer la pertinence des sommets du graphe d'état à explorer.

Extension moteur de jeu. Comme décrit dans la section 4.5, nous avons étendu les fonctionnalités du moteur de jeu pour pouvoir effectuer des retours en arrière dans l'état du jeu.

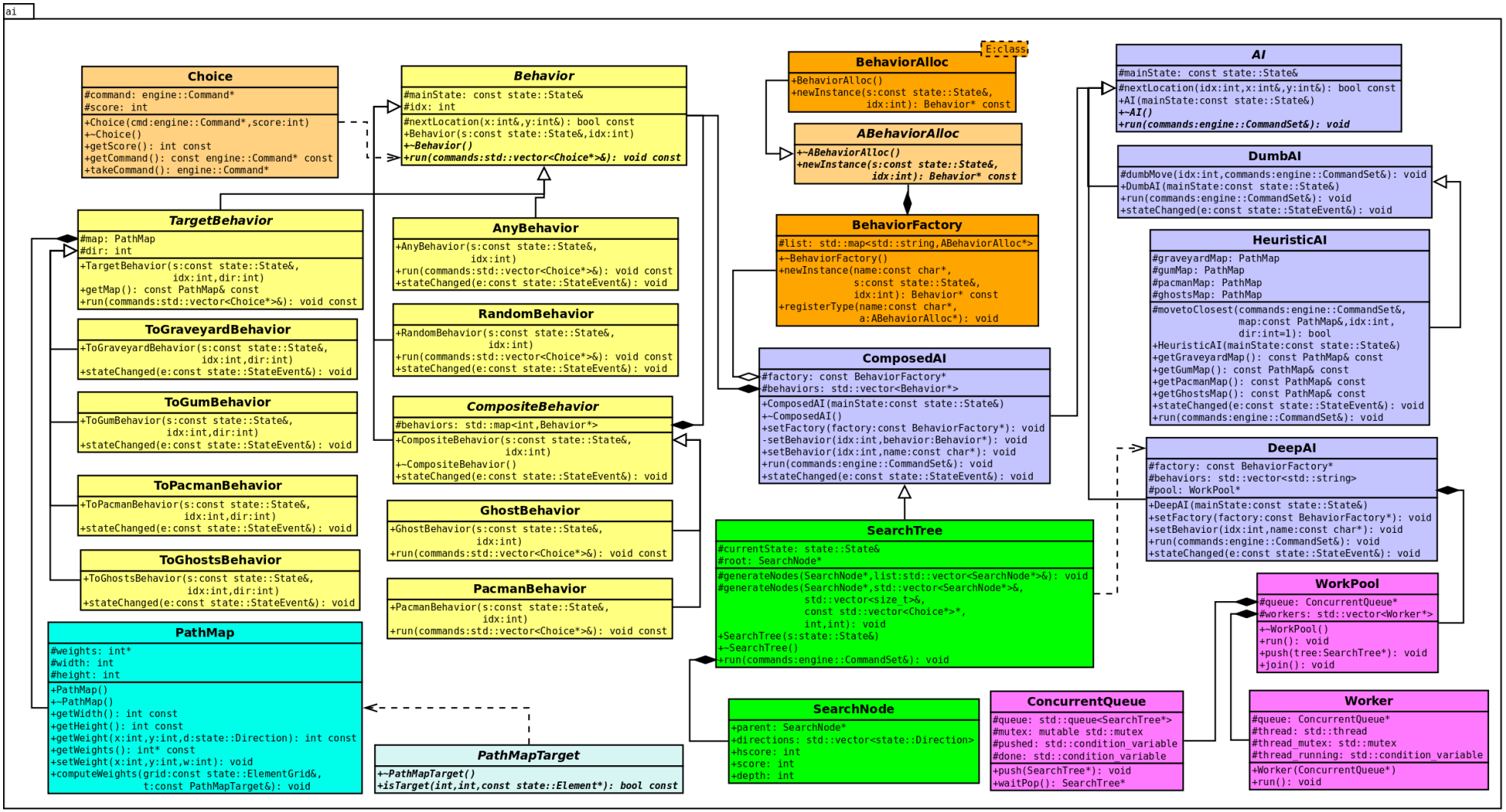
Algorithme. Pour réaliser le parcours de notre arbre de recherche, nous suivons une version légèrement modifiée de *Branch & Bound*. Nous tenons une liste des nœuds restant à observer. Ces nœuds sont représentés par des instances de la classe *SearchNode*. Puis, nous procédons de manière itérative, en sélectionnant le nœud le plus pertinent dans cette liste. Compte tenu des contraintes de parcours que nous avons présentés précédemment, nous ne choisissons que des nœuds qui sont fils ou parent du nœud précédemment observé. Puis, une fois le nœud choisi, les opérations sont les mêmes que celle de l'algorithme standard :

- Un nœud fils non feuille génère de nouveaux nœuds ; les nœuds qui ne peuvent donner lieu à un meilleur score global sont éliminés
- Un nœud parent remonte le meilleur score de ses fils

5.5 Conception logiciel : extension pour la parallélisation

La parallélisation du parcours de l'arbre de recherche pour l'IA avancée nécessite d'être capable de dupliquer entièrement une instance de *SearchTree* ainsi que tous les éléments qu'elle contient. Pour l'état, nous nous reposons sur les mécanismes de clonage précédemment créé pour l'enregistrement

de l'activité du moteur de jeu. Pour l'IA composée, nous souhaitons dupliquer la stratégie suivie, et non une instance particulière. Pour ce faire, nous avons rajouté les mécanismes de patron ***Factory*** avec les classes *BehaviorFactory*, *BehaviorAlloc* et *AbehaviorAlloc*. Grâce à cette approche, nous pouvons fabriquer autant d'IA composée uniquement sur la base des noms des comportements attendus pour chaque performance.
Nous exploitons ensuite cette capacité à dupliquer *SearchTree* de la manière suivante. A la racine de l'arbre de recherche dans l'espace d'état, on dresse la liste des nœuds possibles, de la même manière qu'auparavant. Cependant, au lieu de traiter chacun de ces nœuds fils l'un après l'autre, nous fabriquons une instance de *SearchTree* correspondante. Puis, nous traitons ces instances en parallèle, en utilisant une queue de tâches (***multithreaded work queue***). Cette queue est implanté par les classes en mauve sur le diagramme : *WorkPool*, *Worker* et *ConcurrentQueue*. La classe *WorkPool* est la classe principale, qui lance les *workers* (autant que de coeurs de processeur) et reçoit les instance de *SearchTree* à traiter. La classe *ConcurrentQueue* est une queue classique, à laquelle on ajoute des processus de synchronisation, afin de pouvoir l'utiliser dans un contexte multi thread. Chaque instance de la classe *Worker* est un thread qui tourne tant qu'il reste des tâches à réaliser. Si le worker n'a pas de tâche, il en demande une à la queue concurrente, puis l'exécute. S'il n'y plus de tâches, il s'arrête.


ai
Choice
#command: engine::Command*
#score: int
+Choice(cmd:engine::Command*,score:int)
+~Choice()
+getScore(): int const
+getCommand(): const engine::Command* const
+takeCommand(): engine::Command*
Behavior
#mainState: const state::State&
#idx: int
#nextLocation(x:int&,y:int&): bool const
+Behavior(s:const state::State&,idx:int)
+~Behavior()
+run(commands:std::vector<Choice*>&): void const
BehaviorAlloc
E:class
+BehaviorAlloc()
+newInstance(s:const state::State&, idx:int): Behavior* const
ABehaviorAlloc
+~ABehaviorAlloc()
+newInstance(s:const state::State&, idx:int): Behavior* const
AI
#mainState: const state::State&
#nextLocation(idx:int,x:int&,y:int&): bool const
+AI(mainState:const state::State&)
+~AI()
+run(commands:engine::CommandSet&): void
DumbAI
#dumbMove(idx:int,commands:engine::CommandSet&): void
+DumbAI(mainState:const state::State&)
+run(commands:engine::CommandSet&): void
+stateChanged(e:const state::StateEvent&): void
TargetBehavior
#map: PathMap
#dir: int
+TargetBehavior(s:const state::State&, idx:int,dir:int)
+getMap(): const PathMap& const
+run(commands:std::vector<Choice*>&): void const
AnyBehavior
+AnyBehavior(s:const state::State&, idx:int)
+run(commands:std::vector<Choice*>&): void const
+stateChanged(e:const state::StateEvent&): void
BehaviorFactory
#list: std::map<std::string,ABehaviorAlloc*>
+~BehaviorFactory()
+newInstance(name:const char*, s:const state::State&, idx:int): Behavior* const
+registerType(name:const char*, a:ABehaviorAlloc*): void
HeuristicAI
#graveyardMap: PathMap
#gumMap: PathMap
#pacmanMap: PathMap
#ghostsMap: PathMap
#movetoClosest(commands:engine::CommandSet&, map:const PathMap&,idx:int, dir:int=1): bool
+HeuristicAI(mainState:const state::State&)
+getGraveyardMap(): const PathMap& const
+getGumMap(): const PathMap& const
+getPacmanMap(): const PathMap& const
+getGhostsMap(): const PathMap& const
+stateChanged(e:const state::StateEvent&): void
+run(commands:engine::CommandSet&): void
ToGraveyardBehavior
+ToGraveyardBehavior(s:const state::State&, idx:int,dir:int)
+stateChanged(e:const state::StateEvent&): void
RandomBehavior
+RandomBehavior(s:const state::State&, idx:int)
+run(commands:std::vector<Choice*>&): void const
+stateChanged(e:const state::StateEvent&): void
ComposedAI
#factory: const BehaviorFactory*
#behaviors: std::vector<Behavior*>
+ComposedAI(mainState:const state::State&)
+~ComposedAI()
+setFactory(factory:const BehaviorFactory*): void
-setBehavior(idx:int,behavior:Behavior*): void
+setBehavior(idx:int,name:const char*): void
+run(commands:engine::CommandSet&): void
+stateChanged(e:const state::StateEvent&): void
DeepAI
#factory: const BehaviorFactory*
#behaviors: std::vector<std::string>
#pool: WorkPool*
+DeepAI(mainState:const state::State&)
+setFactory(factory:const BehaviorFactory*): void
+setBehavior(idx:int,name:const char*): void
+run(commands:engine::CommandSet&): void
+stateChanged(e:const state::StateEvent&): void
ToGumBehavior
+ToGumBehavior(s:const state::State&, idx:int,dir:int)
+stateChanged(e:const state::StateEvent&): void
CompositeBehavior
#behaviors: std::map<int,Behavior*>
+CompositeBehavior(s:const state::State&, idx:int)
+~CompositeBehavior()
+stateChanged(e:const state::StateEvent&): void
ToPacmanBehavior
+ToPacmanBehavior(s:const state::State&, idx:int,dir:int)
+stateChanged(e:const state::StateEvent&): void
GhostBehavior
+GhostBehavior(s:const state::State&, idx:int)
+run(commands:std::vector<Choice*>&): void const
SearchTree
#currentState: state::State&
#root: SearchNode*
#generateNodes(SearchNode*,list:std::vector<SearchNode*>&): void
#generateNodes(SearchNode*,std::vector<SearchNode*>&, std::vector<size_t>&, const std::vector<Choice*>*, int,int): void
+SearchTree(s:state::State&)
+~SearchTree()
+run(commands:engine::CommandSet&): void
WorkPool
#queue: ConcurrentQueue*
#workers: std::vector<Worker*>
+~WorkPool()
+run(): void
+push(tree:SearchTree*): void
+join(): void
ToGhostsBehavior
+ToGhostsBehavior(s:const state::State&, idx:int,dir:int)
+stateChanged(e:const state::StateEvent&): void
PacmanBehavior
+PacmanBehavior(s:const state::State&, idx:int)
+run(commands:std::vector<Choice*>&): void const
PathMap
#weights: int*
#width: int
#height: int
+PathMap()
+~PathMap()
+getWidth(): int const
+getHeight(): int const
+getWeight(x:int,y:int,d:state::Direction): int const
+getWeights(): int* const
+setWeight(x:int,y:int,w:int): void
+computeWeights(grid:const state::ElementGrid&, t:const PathMapTarget&): void
SearchNode
+parent: SearchNode*
+directions: std::vector<state::Direction>
+hscore: int
+score: int
+depth: int
ConcurrentQueue
#queue: std::queue<SearchTree*>
#mutex: mutable std::mutex
#pushed: std::condition_variable
#done: std::condition_variable
+push(SearchTree*): void
+waitPop(): SearchTree*
Worker
#queue: ConcurrentQueue*
#thread: std::thread
#thread_mutex: std::mutex
#thread_running: std::condition_variable
+Worker(ConcurrentQueue*)
+run(): void
PathMapTarget
+~PathMapTarget()
+isTarget(int,int,const state::Element*): bool const

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Nous avons deux type d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

Commandes. Il s'agit ici des touches du clavier pressées par l'utilisateur. Celle-ci peuvent arriver à l'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous proposons d'utiliser un double tampon de commandes. L'un contiendra les commandes actuellement traitées par une mise à jour de l'état du jeu, et l'autre accueillera les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on permute les deux tampons : celui qui accueillait les commandes devient celui traité par la mise à jour, et l'autre devient disponible pour accueillir les futures commandes. Ainsi, il existe toujours un tampon capable de recevoir des commandes, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.

Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter une approche par double tampon. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés donc vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible.

Pour ce faire, nous ajoutons un tampon qui va « absorber » toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. Puis, lorsqu'une mise à jour est terminée, le moteur de jeu envoie un signal au moteur de rendu. Celui-ci, lorsqu'il s'apprête à envoyer ses données à la carte graphique, regarde si ce signal a été émis. Si c'est le cas, il vide le tampon de notification pour modifier ses données graphiques avant d'effectuer ses tâches habituelles. Lors de cette étape, une mise à jour de l'état du jeu ne peut avoir lieu, puisque le moteur de rendu a besoin des données de de l'état pour mettre à jour les siennes. Nous avons donc ici un recouvrement entre les deux processus. Cependant, la quantité de mémoire et de processeur utilisée est très faible devant celles utilisées par la mise à jour de l'état du jeu et par le rendu.

6.1.2 Répartition sur différentes machines

TODO

6.2 Conception logiciel

Nous présentons en Illustration 6 le diagramme de classes pour la modularisation du projet. Deux packages sont présentés. Le package *server* regroupe les opérations relatives au moteur de jeu, et le package *client* les opérations relative au moteur de rendu.

Classes Server. La classe abstraite *Server* encapsule un moteur de jeu, et lance un thread en arrière plan pour effectuer les mises à jours de l'état du jeu. L'implantation *LocalServer* est le cas où le moteur de jeu principal est sur la machine où est lancé le programme. Dans tous les cas, les instances de *Server* peuvent notifier certains événements grâce aux classes *Observable*, *ServerObserver* et *ServerEvent* qui forment un patron de conception ***Observer***.

Classes Client. La classe abstraite *Client* contient tous les éléments permettant le rendu, comme la scène et les définitions des tuiles. Un maximum d'opération communes à tous les clients graphiques ont été placés dans cette classe, afin de rendre la création d'un nouveau type de client plus rapide. La classe *SFMLClient* est un exemple de client graphique avec la librairie SFML.

CacheStateObserver. Cette implantation particulière de *StateObserver* nous permet de récupérer toutes les notifications émises par un moteur de jeu, dans le but de les appliquer plus tard, tel que décrit dans la section précédente.

6.3 Conception logiciel : extension réseau

TODO

6.4 Conception logiciel : client Android

TODO

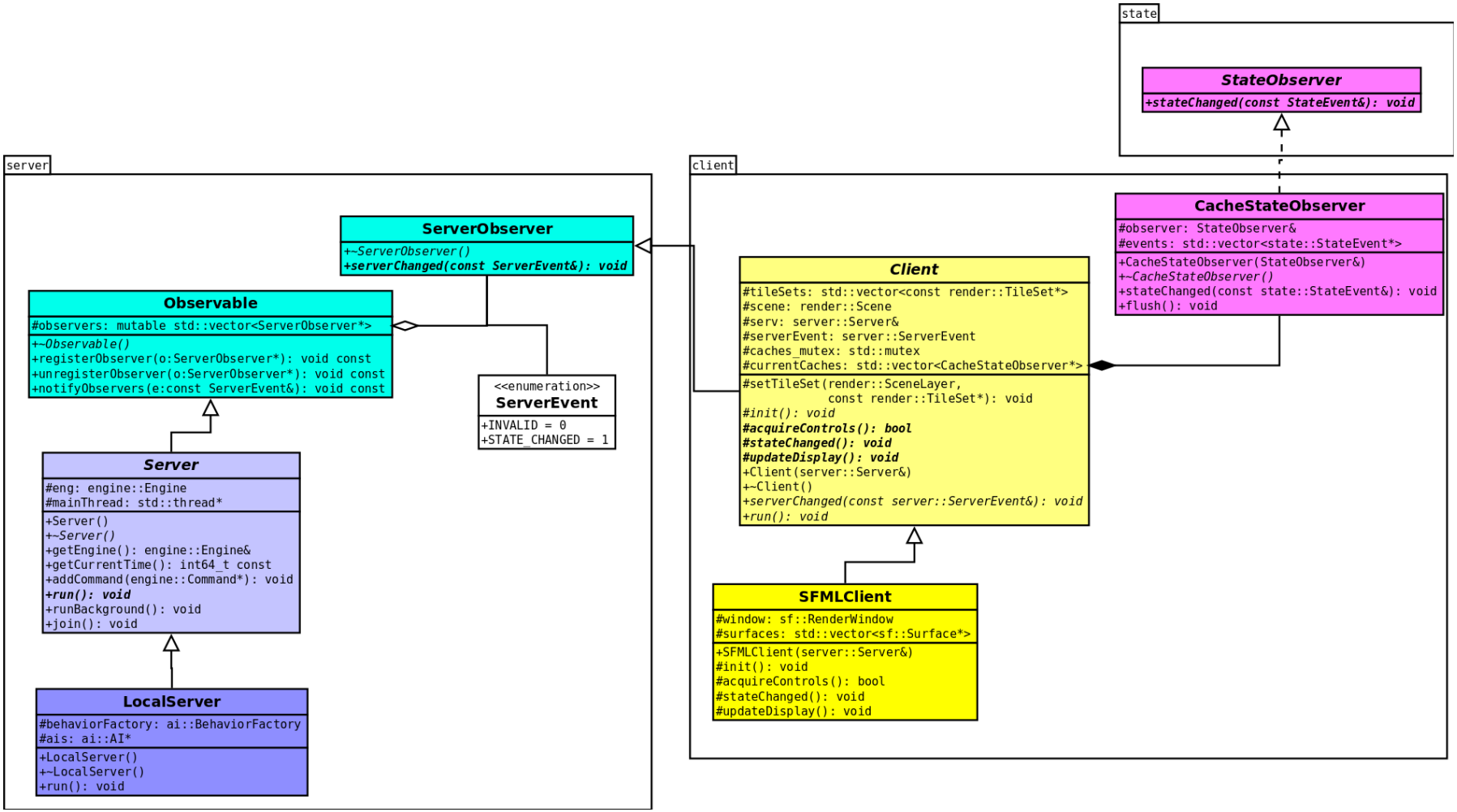


Illustration 6: Diagramme de classes pour la modularisation