23/12/2017

Projet Snake

Design Pattern

FACULTÉ
DES SCIENCES
Unité de formation et de recherche

Guillaume HUET | Hugues DUMONT | Rachid AIT SIDI HAMMOU | Mohamed Achraf HAROUACH SLASSI
MASTER 1 INFORMATIQUE – UNIVERSITE D'ANGERS

Table des matières

1. Introduction

Dans le cadre du cours sur les Design Pattern, il nous a été demandé de travailler sur la conception d'un jeu de type Snake à la façon RPG et jouable en réseau. Il ne s'agit aucunement ici d'implémenter tout le jeu dans son intégralité mais de réfléchir uniquement à l'architecture de notre programme en faisant ressortir certains Design Pattern abordés durant les cours. Pour ce premier semestre, ce projet consiste en la rédaction d'un rapport renseignant les spécifications de notre jeu. Deux grandes parties sont décrites :

- Une partie réseau décrivant les échanges entre des hôtes distants.
- Une partie Design Pattern décrivant l'architecture logicielle de notre programme ainsi que tous les Design Pattern utilisés permettant de concevoir au mieux notre jeu.

Pour réaliser ce projet, la classe a été divisée en plusieurs petits groupes. Notre groupe quant à lui est composé des personnes suivantes :

- Guillaume HUET
- Hugues DUMONT
- Mohamed Achraf HAROUACH SLASSI
- Rachid AIT SIDI HAMMOU

2. Objectif

Créer un jeu Snake en réseau multiplateforme (Windows, Linux, MacOs).

3. Les spécifications du projet

3.1. PARTIE RESEAU

3.1.1. Contexte

Dans cette partie, nous allons nous intéresser à la partie réseau du projet Snake. En effet, il ne s'agit pas d'un simple jeu à jouer seul sur son ordinateur mais d'un jeu en réseau où plusieurs personnes peuvent venir jouer. Il s'agit donc d'un Snake en réseau et multijoueur.

Concernant le côté réseau du jeu, nous nous limitons ici au réseau local, le Local Area Network (LAN). En effet, pas d'objectif ici, en tout cas pour le moment, d'étendre le jeu au-delà du LAN. Cependant, l'idée n'est pas exclue.

Pour le côté multijoueur, il faut être au minimum 2 joueurs. Pour plus de facilité et de simplicité, nous allons restreindre à 2 le nombre de joueur pour notre jeu Snake. Chaque joueur jouera sur son propre ordinateur et chaque joueur contactera le même serveur par un protocole réseau qui sera définit dans la suite.

3.1.2. Architecture du programme

Dans le cadre de notre jeu en réseau, l'architecture qui nous parait la plus naturelle et logique est l'architecture de type Client/Serveur, un modèle bien connu des utilisateurs d'internet. Dans le cas d'un exemple basique, un programme qui n'est autre qu'un client va demander un service ou une ressource à un autre programme qui lui est en fait le serveur. Le client effectue ce que l'on appelle des requêtes et le serveur va quant à lui envoyer des réponses en fournissant s'il le peut les ressources demandées. Ce modèle est représenté de la manière suivante (figure 1) :

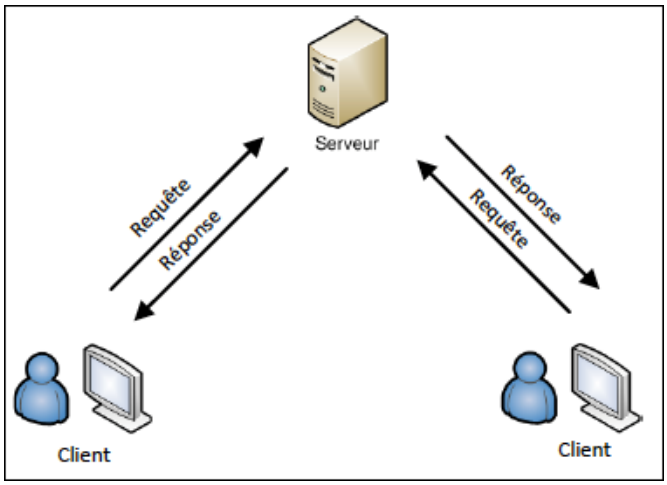

Figure 1 - Modèle Client/Serveur

Dans notre jeu, nous aurons donc au minimum deux clients et un seul serveur.

3.1.3. Définition du protocole de transmission

Nous allons nous intéresser ici à la couche transport du modèle TCP/IP et plus précisément aux protocoles de cette fameuse couche qui vont nous permettre de transporter les données de notre jeu. Il y a donc deux protocoles capables de transporter nos données entre nos clients et notre serveur :

- **Le protocole TCP** :
  - Avantages :
    - Orienté connexion.
    - Fiable.
    - Pas de contrôle à faire.
  - Inconvénients :
    - Lent si traitement de beaucoup de données volumineuses.

- **Le protocole UDP** :
  - Avantages :
    - Rapide dans un réseau non saturé.
    - Plus adapté généralement pour le multimédia (jeux, streaming…)
  - Inconvénients :
    - Non orienté connexion.
    - Non fiable.
    - Pas de contrôle d'erreurs ni de gestion de la congestion et donc il faudra mettre en place ce contrôle.

Suite à cette petite comparaison, nous avons décidé de nous orienter dans un premier vers l'utilisation du protocole TCP. En effet, ce dernier nous permettra de bénéficier d'une transmission fiable des données avec gestion des erreurs ce qui nous facilite grandement l'implémentation. De plus, nous ne prévoyons pas de transmettre une grande quantité de données volumineuse donc nous supposons que notre jeu sera fluide. Dans le cas où l'on constaterait d'énormes ralentissements alors nous envisagerons d'utiliser le protocole UDP. Cependant, il faudra pour ce dernier mettre en place la gestion d'erreurs à moins que la perte de certains paquets ne soit pas si grave dans le cas de notre jeu ?

3.1.4. Gérer la communication réseau en JAVA

Cette partie consiste en une rapide description des outils que nous allons utiliser pour gérer la communication entre nos clients et notre serveur en JAVA. En effet, chaque action des clients sera transformée en un ou plusieurs objets qui seront traités par des classes spécifiques JAVA afin de les transmettre sur le réseau vers le serveur. De même, une fois que le serveur a effectué tous les traitements nécessaires, les réponses à envoyer aux clients seront également traitées par des classes spécifiques.

L'API java.net

Concrètement, pour gérer les communications entre les clients et le serveur, nous allons utiliser l'API réseau de JAVA qui est java.net. Cette API fournit un ensemble de classes permettant l'implémentation d'applications en réseau.

L'API peut être divisée en deux sections, une de bas niveau et une de haut niveau. Nous allons nous intéresser uniquement à la partie bas niveau. En effet, celle-ci traite des abstractions suivantes :

- Les adresses, qui sont les identifiants de réseau telles que les adresses IP.
- Les Sockets, qui sont des mécanismes de communication de données bidirectionnels de base.
- Les interfaces, qui décrivent les interfaces réseau.

Les Sockets

Pour faire communiquer nos clients avec le serveur et vice-versa, nous allons donc utiliser des adresses IP, des ports et des protocoles et pour échanger les différentes informations à travers le réseau, les clients et le serveur utiliseront des sockets.

Dans notre architecture, nous n'auront pas le choix, il faudra utiliser deux types de sockets :

- **Socket côté client** : permet la connexion à une machine distante afin de communiquer avec elle. Dans ce cas, nos clients établiront une connexion avec le serveur, enverront et recevront des données vers et depuis ce dernier et une fois que tout est terminé, la connexion se terminera.
- **Socket côté serveur** : connexion qui attend qu'un client vienne se connecter afin de communiquer avec lui. Ici, notre serveur sera en attente de connexion des clients. Pour le reste du fonctionnement il y aura également des échanges et une fois le tout terminé, on ferme toutes les connexions.

Concernant les numéros de port à utiliser, nous utiliserons ceux présents dans la plage des ports disponibles.

3.1.5. Sécurisation des échanges d'informations

Un point important concerné la sécurité au niveau des communications qui s'effectuent entre les clients et le serveur. En effet, la question qui se pose actuellement est : doit-on chiffrer les communications clients <-> serveur ? Cela représenterait-il un risque dans notre architecture si tous les messages s'affichent en clair dans les trames Ethernet ?

Si vraiment on souhaite crypter toutes les communications alors on pourra s'orienter vers les SSLSockets et SSLServerSockets qui nous permettent d'avoir des Sockets sécurisées crypter en SSL ou TLS.

Cependant, accordant une importance sur la fluidité et la rapidité du jeu et donc des échanges d'informations, le fait déjà d'avoir choisi le protocole TCP fait naître en nous un petit doute quant à la validité de ces paramètres. Ajouter en plus du chiffrage sur toutes les communications augmenterait considérablement les temps de traitement de chaque paquet étant donné qu'il faut crypter et décrypter.

En revanche, il serait tout à fait envisageable de chiffrer uniquement certaines communications dans lesquelles passent des données sensibles. Par exemple, durant l'inscription d'un joueur ou de sa connexion au serveur où là il y a utilisation de login et de mot de passe, on cryptera les échanges à ce niveau. Une fois inscrit/connecté et donc In-Game, les communications pourront s'effectuer normalement sans que celles-ci soient chiffrées.

3.2. PARTIE DESIGN PATTERN

3.2.1. Contexte

Dans cette section, nous allons plus nous tourner vers l'architecture logicielle de notre jeu en nous aidant et en utilisant notamment les Design Pattern. Cette partie se composera de manière générale de diagramme de classes représentant la modélisation de notre jeu.

L'objectif est donc de réaliser un jeu de type Snake en mode RPG c'est-à-dire qu'il ne s'agit pas ici de contrôler un serpent qui ne fait que grossir en mangeant de la nourriture mais d'ajouter des fonctionnalités supplémentaires tels que des bonus qui font changer nos serpents de comportement, des obstacles qu'il faudra éviter, des retours dans le temps etc.

3.2.2. Architecture globale : un premier pattern, le MVC

Ici, nous aurons une vue très généraliste et globale de notre jeu. Par ailleurs, un premier pattern vient ici faire son entrée. Il s'agit du pattern MVC (Modèle, Vue & Contrôleur). Le schéma représentant notre programme est le suivant :

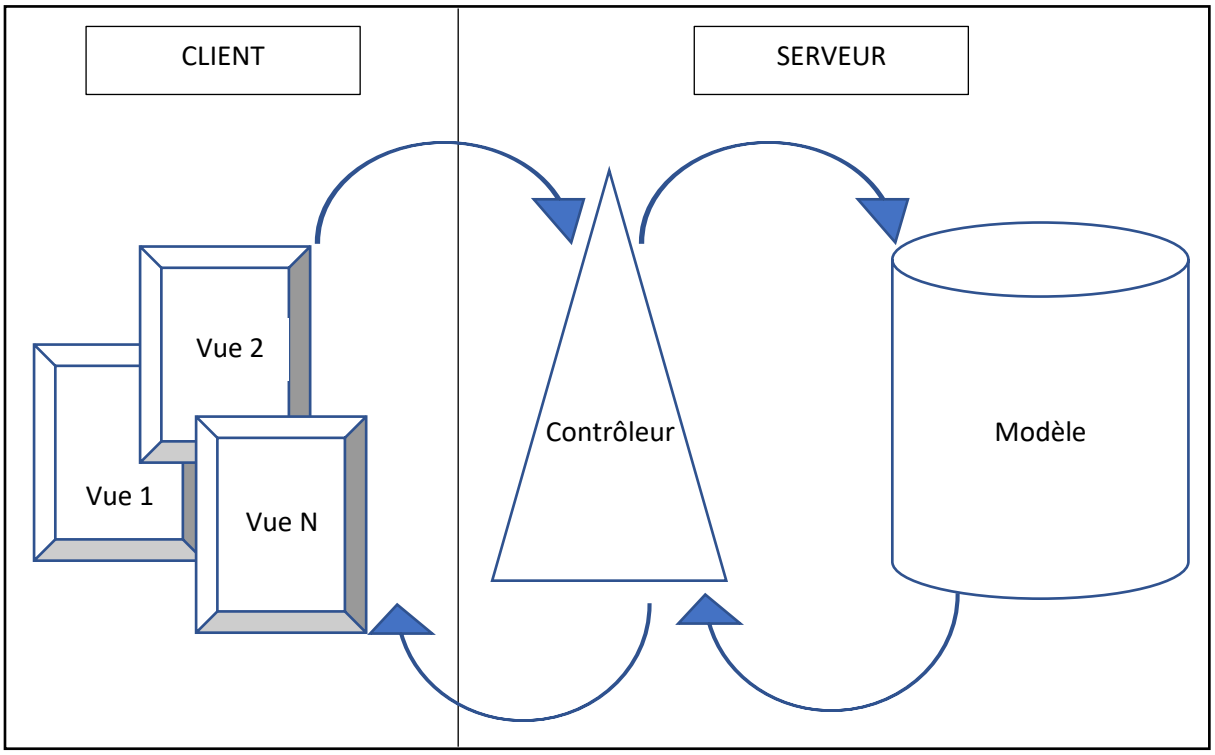

Figure 2 - Représentation sous forme MVC

Comme on peut le voir sur la figure numéro 2, nous avons une architecture MVC permettant de distinguer 3 grandes parties distinctes de notre logiciel. Par ailleurs, nous pouvons également constater une séparation entre les vues qui est en fait côté client et donc des joueurs et le contrôleur ainsi que le modèle qui sont côté serveur donc sur une machine distante accessible dans le réseau local.

Pour des raisons de sécurité, de maintenabilité et d'extension, nous avons décidé qu'il n'y aurait pas de communication du modèle vers les vues. En effet, nous avons préféré dans notre cas, une fois le

modèle mis à jour, de notifier le contrôleur qui lui transmettra toutes les informations nécessaires aux vues afin que ces dernières se mettent également à jour. Obligation donc de passer par le contrôleur. Ce dernier peut nous faire penser à un mandataire. Enfin, tous ces éléments sont en fait représentés sous forme d'objets et dans notre cas d'objets JAVA.

Qui dit MVC dit également pattern composé et dans notre cas, nous verrons plus loin les différents patterns qui composent donc notre architecture. En revanche, nous pouvons d'ores et déjà faire apparaître dans notre architecture un design pattern chargé de mettre des objets au courant de changement et se mettre à jour en conséquence. Il s'agit bien entendu du pattern Observateur.

Le pattern Observateur dans notre MVC

Voici donc le schéma représentant l'idée que nous avons du pattern observateur dans notre architecture globale :

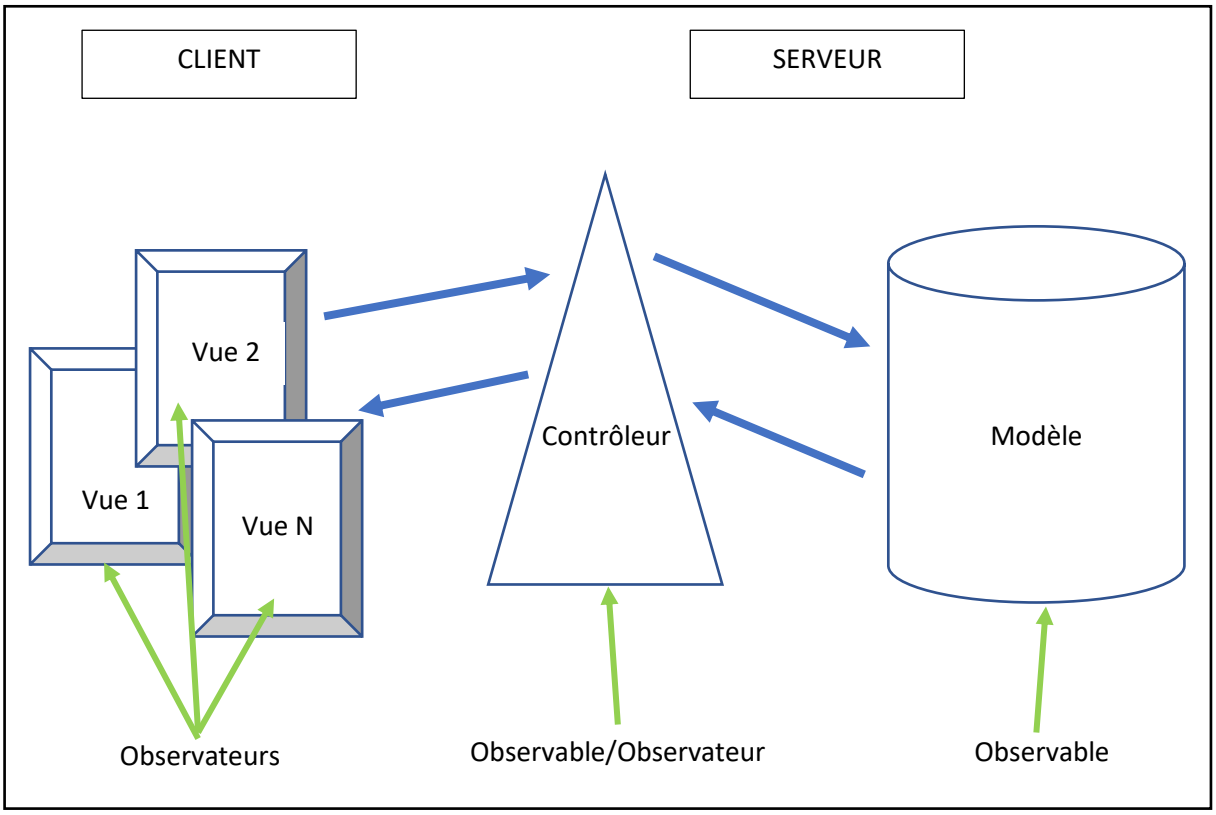

Figure 3 - Pattern Observateur dans MVC

Que se passe-t-il ici ? En fait, nous allons partir du modèle qui lui est observé par le contrôleur donc le contrôleur est un observateur du modèle. Cela veut dire que dès qu'un changement s'effectue dans le modèle, le contrôleur en sera notifié. Ensuite, le contrôleur est lui aussi un observable et il observé par les vues qui sont donc des observateurs du contrôleur. En effet, nous avons proscrit toute communication entre le modèle et les vues. De ce fait, le contrôleur se chargera de notifier les vues lorsque lui-même sera notifié par le modèle.

Dans la suite de ce rapport, nous allons parler de chaque grande partie de notre architecture globale à savoir en premier lieu des vues puis viendra ensuite la partie contrôleur et enfin nous terminerons avec le modèle.

3.2.3. Les vues du jeu (côté client)

Description générale des vues

En ce qui concerne la partie vue, elle contient les éléments visuels de notre jeu et elle communique à distance avec le contrôleur via un protocole réseau pour répondre aux actions déclenchées par les joueurs et les modifications effectuées sur le modèle, dont elle est notifiée par le contrôleur, ainsi nos différentes vues ne traitent aucune donnée directement et ne sert qu'à effectuer de l'affichage et à transmettre les informations au contrôleur. L'utilisation du pattern Composite est ici implicite.

Interface de connexion/inscription

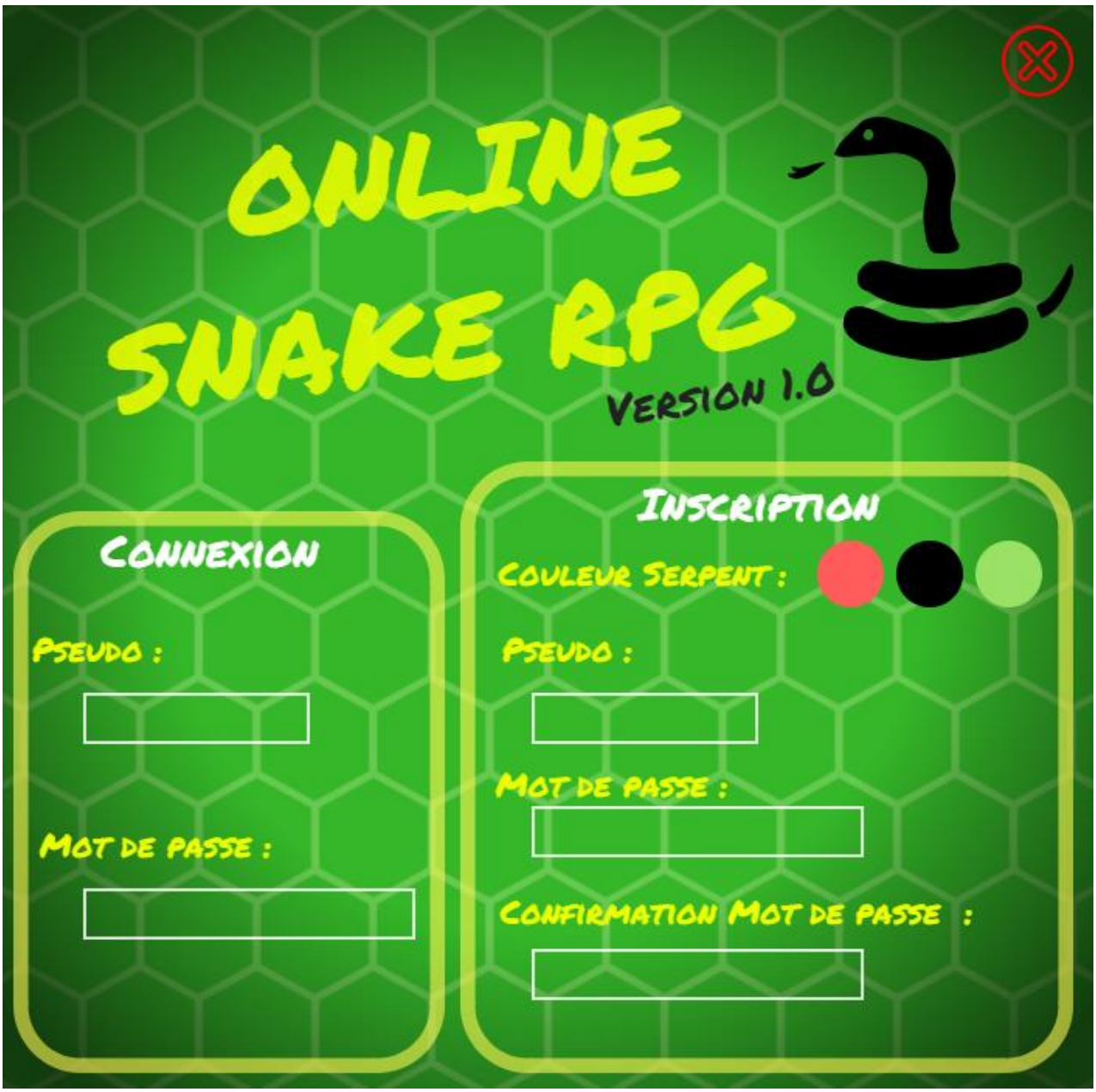

Figure 4 - Vue : Interface de connexion/Inscription

La première vue de notre jeu consistera en l'affichage d'une fenêtre de connexion ou d'inscription (les deux seront faisables via la même fenêtre pour simplifier l'expérience utilisateur).

Cette première vue doit donc permettre 2 choses :

- S'inscrire, en créant son pseudo (qui sera unique pour chaque joueur) et son mot de passe (qui sera à confirmer) et la couleur de son serpent.
- Se connecter, pour les personnes déjà inscrites, en saisissant tout simplement le pseudo et le mot de passe.

Dans les deux cas, deux résultats sont possibles :

- Une erreur de connexion/inscription (Pseudo existant pour l'inscription, mot de passe erroné, pseudo inexistant pour la connexion, etc…)
- Une redirection vers le menu principal du jeu.

Menu principal

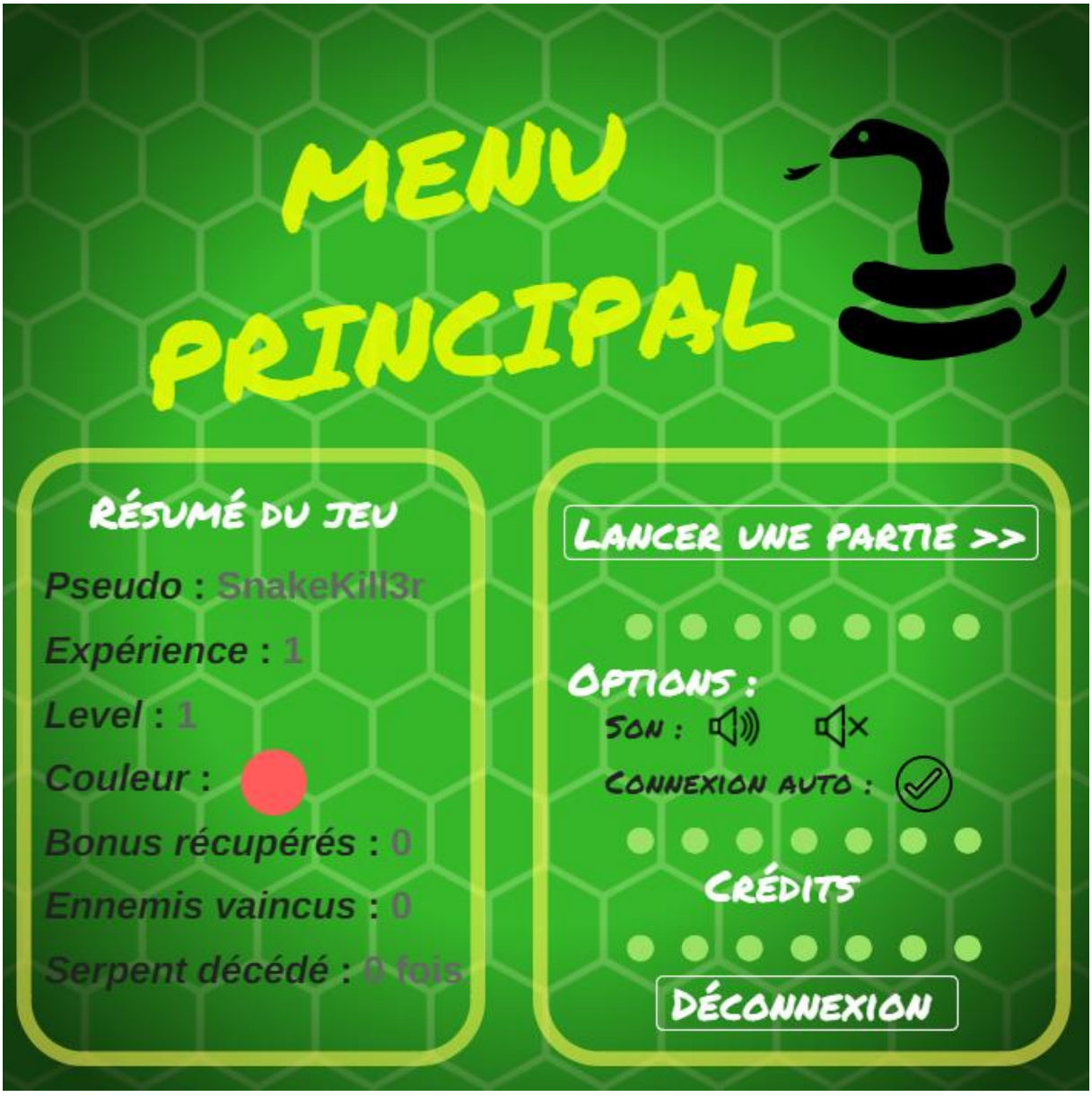

Figure 5 - Vue : Interface du menu principal

Suite à l'inscription ou la connexion (sauf erreur dans la tentative de connexion ou d'inscription), la vue suivante est en quelque sorte le menu principal de l'application. Celle-ci résume les caractéristiques du joueur connecté (en affichant son expérience, son niveau, ses bonus permanents, le nombre d'ennemis vaincus, son nombre de morts, le nombre de bonus récupérés, etc…). Elle est également la transition avec le lancement d'une partie, qui ouvrira une nouvelle vue, et permet également la gestion des options (sélection du volume de la musique, du volume des bruitages, choix d'une connexion automatique ou non, choix du lancement d'une partie automatique ou non) et enfin fait également le lien avec la vue des crédits du jeu (le descriptif du rôle de chaque personne dans le développement du jeu, et non de l'argent). Bien évidemment, cet accueil doit également permettre à l'utilisateur de se déconnecter, qui retournera sur la page de connexion.

Choix de la partie

Figure 6 - Vue : Choix de la Map

Lorsque l'utilisateur décide de lancer une nouvelle partie, une nouvelle fenêtre est affichée, dans laquelle le joueur pourra sélectionner la carte sur laquelle il souhaite jouer. Un affichage réduit de la carte sélectionnée est affiché à côté de la liste des cartes, ainsi que le nombre de joueurs connectés sur celle-ci. Une fois que l'utilisateur a fait son choix, il lance la partie. Dans le cas où il y a déjà des joueurs présents sur la carte sélectionnée, il rejoint la partie en cours, sinon, il doit attendre qu'un joueur supplémentaire se connecte à la carte.

Affichage du jeu (la carte)

Cette vue est la fonctionnalité principale de l'application puisqu'il s'agit du cœur du jeu (impossible de joueur sans carte). On y retrouvera les éléments suivants :

- L'affichage des serpents connectés (La couleur et le pseudo)
- L'affichage des bonus (sur la carte et possédés par le joueur)
- L'affichage des obstacles
- L'affichage du score du joueur dans la partie, de son expérience et de son niveau
- L'affichage Game over
- L'affichage de l'état des serpents (visibles, invisibles, brillants, etc…)

Elle récupère également les informations de jeu de l'utilisateur (ses changements de direction, l'utilisation de ses bonus) et en affiche les résultats en retour (déplacement, collisions, etc…).

Crédits

Dans la fenêtre des crédits, on y retrouvera tout simplement un affichage des informations sur le rôle de chaque intervenant dans la création, le développement et la mise en place du jeu.

3.2.4. Le contrôleur (côté serveur)

Nous allons ici parler d'un objet permettant de faire le lien entre nos vues et notre modèle lorsqu'une ou plusieurs actions utilisateur surviennent sur les vues ou sur la vue courante. De plus, c'est cet objet qui aura pour rôle de contrôler les données. En effet, il s'agit de notre contrôleur.

Nous ne décrirons pas de manière exhaustive dans cette partie notre contrôleur mais plutôt de manière générale. Nous décrirons quelques méthodes que contiennent notre objet mais pas toutes. Il s'agit ici de comprendre facilement et rapidement le rôle qu'à le contrôleur.

Globalement, les actions effectuées par les utilisateurs seront captées par le contrôleur qui vérifiera la cohérence des données et si nécessaire les transformer afin que le modèle les comprenne. De plus, c'est notre contrôleur également qui se chargera de demander à la vue ou aux vues de changer après un ou plusieurs changements d'états du modèle. Par ailleurs, il est tout à fait possible que le contrôleur demande à la vue de changer directement après une action utilisateur sans attendre le changement d'états du modèle. Par exemple, on désactive immédiatement un bouton quand un utilisateur clique sur celui-ci afin d'éviter une anomalie de fonctionnement quelconque du programme et en attendant que les traitements au sein du modèle soient effectués. Lors de l'implémentation de notre projet Snake, nous créerons une classe abstraite contrôleur afin de définir un super type de variable nous permettant par la suite si besoin d'utiliser différents contrôleurs de façon polymorphe. Concernant les méthodes de notre contrôleur, celui-ci en aura un grand nombre étant donné qu'il est seul et ces méthodes seront les opérations à effectuer après avoir reçu les actions des utilisateurs mais aussi pour faire changer d'état le modèle ou encore demander aux vues de se mettre à jour. Du boulot ici à faire.

3.2.5. Le modèle (côte serveur)

Enfin, la dernière partie de notre architecture MVC qui sera traitée ici est donc le modèle. Le modèle est dans notre cas composé d'un ensemble d'objets appelés aussi objets métiers. Celui-ci sera chargé de stocker toutes les données nécessaires à la réalisation d'un ou plusieurs traitements et d'avoir les résultats souhaités et tout ça en toute transparence pour l'utilisateur du programme. Nous sommes ici dans le cœur de notre logiciel.

Quelques remarques, il n'y a bien évidemment pas toutes les informations permettant d'implémenter totalement le modèle (absence de certaines méthodes et/ou d'attributs en fonction des classes). De plus, ils n'y sont pas ici mais la quasi-totalité de nos classes seront pourvues de getters/setters (dans le cas où cela serait nécessaire) ainsi que d'un constructeur initialisant l'ensemble des données également nécessaires au fonctionnement du jeu.

La classe Serpent

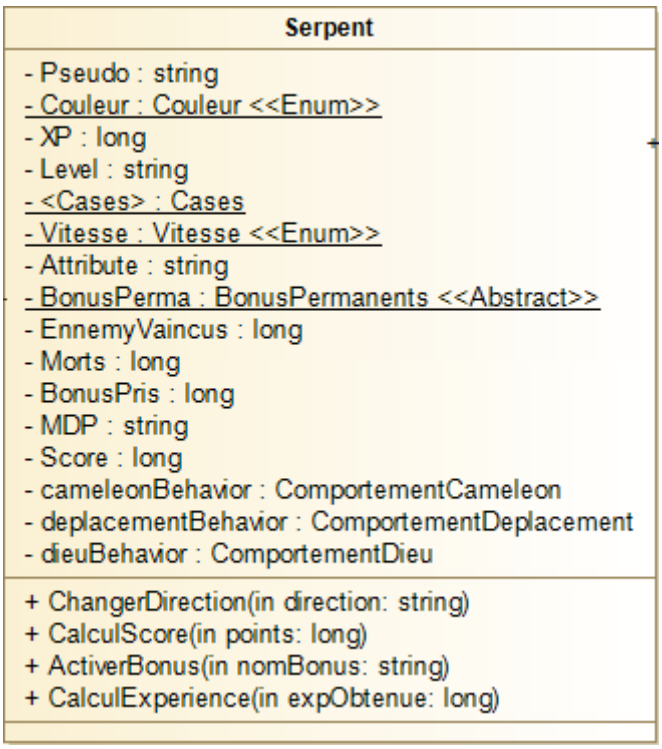

Figure 7 - Classe Snake

C'est à partir de cette classe que nous pourrons instancier des serpents. Il n'y a ici qu'un seul type de serpent. Les serpents seront différenciés entre eux en fonction de leur couleur. Nos serpents sont donc caractérisés par les attributs présents dans le diagramme et peuvent effectuer des opérations diverses grâce à leurs différentes méthodes.

Si nous faisons attention aux trois derniers attributs de notre classe serpent, nous constatons ici que notre serpent possède trois types de comportement, le cameleonBehavior, le deplacementBehavior et le dieuBehavior.

Ceci nous permet en fait d'aborder l'utilisation d'un autre design pattern qui est celui du pattern stratégie.

Le pattern Stratégie

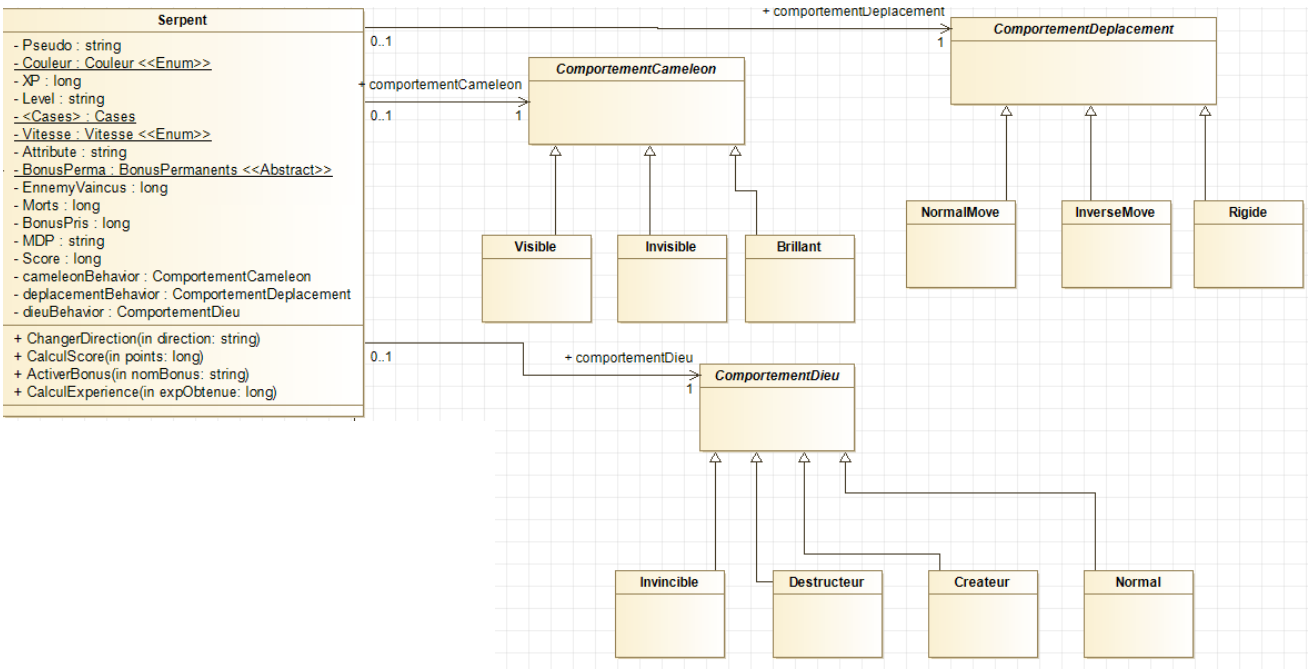

Figure 8 - Pattern Stratégie pour les comportements

Nous voici donc en face du pattern Stratégie. Pour comprendre son intérêt, il faut se situer dans le contexte. En fait, les comportements sont en lien avec les bonus qu'un serpent récupère. En effet, en fonction du bonus (que nous verrons plus loin), le serpent va changer de comportement et de cette manière il aura la capacité au niveau du comportement caméléon de devenir invisible ou de briller ou encore pour le comportement de déplacement de partir dans les directions inverses des actions de l'utilisateur (le serpent va en bas quand l'utilisateur appuie sur la flèche haut etc.). Etant donné que ce sont des codes qui varient mais aussi qu'on souhaite pouvoir changer dynamiquement durant l'exécution les différents comportements, il nous a semblé logique de nous tourner vers le pattern stratégie.

La classe Map

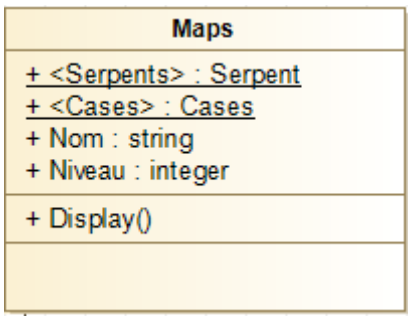

Figure 9 - Classe Map

Il s'agit ici de la classe Map qui représentera donc la carte du jeu. Celle-ci est composé de serpents et de cases. La méthode Display() permet son affichage.

La classe Cases

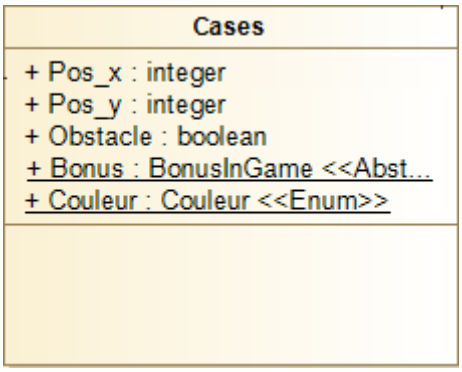

Figure 10 - classe Cases

Cette classe représente en fait les cases que vont contenir notre carte. Elles auront donc une position sur x et sur y. La case pourra également représenter soit un obstacle auquel cas la couleur de cette dernière sera grisée par exemple ou pourra contenir un bonus et dans ce cas, on y mettra une image ou une couleur représentant un bonus particulier.

Les Bonus

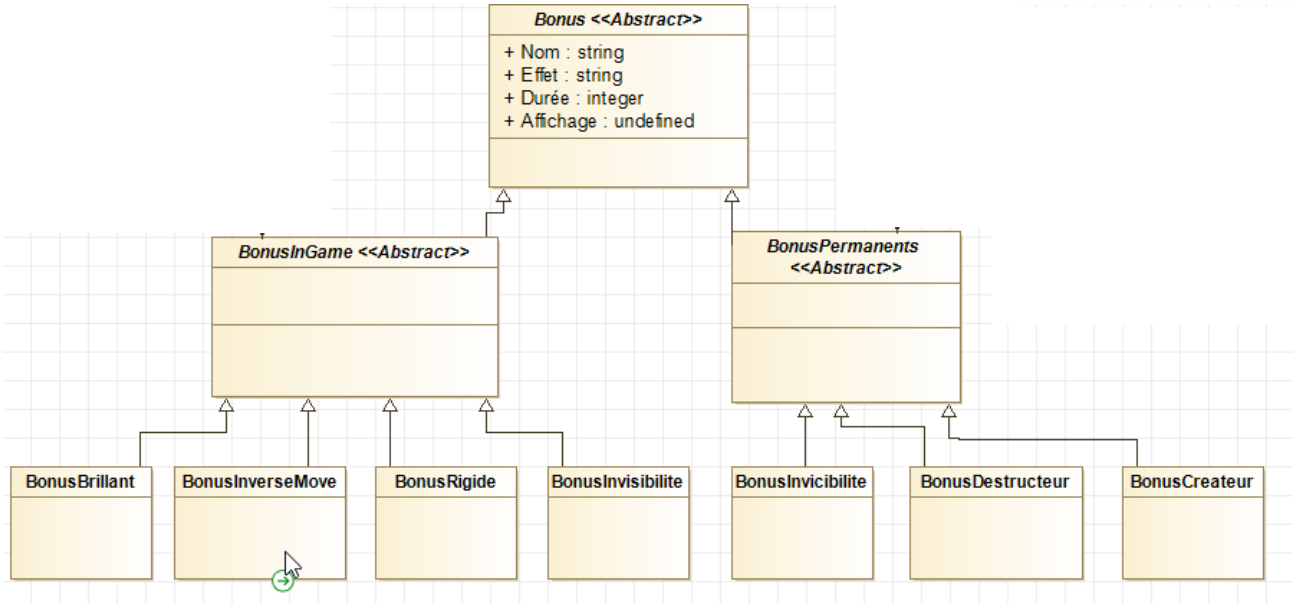

Figure 11 - Les bonus

Voici donc la hiérarchie de nos classes concernant les bonus. Ici, on peut remarquer qu'on aura des bonus qu'on obtiendra durant la partie, BonusInGame, mais aussi des bonus permanents qui seront obtenus après avoir atteint un certain niveau par exemple ou en ayant gagné un certain nombre de fois. Ces bonus permanents seront activables à l'aide d'une touche spécifique. Bien entendu, chaque bonus a donc un effet, par exemple BonusRigide empêchera le serpent de tourner. De plus, ces effets ne sont pas à durée illimitée et possèdent donc une durée de quelques secondes de fonctionnement.

NB : le bonus destructeur permet à un serpent de détruire un obstacle quelconque sur la carte quant au bonus créateur ce dernier créera un obstacle ou un bonus quelque part sur la carte.

Diagramme de classes du Modèle

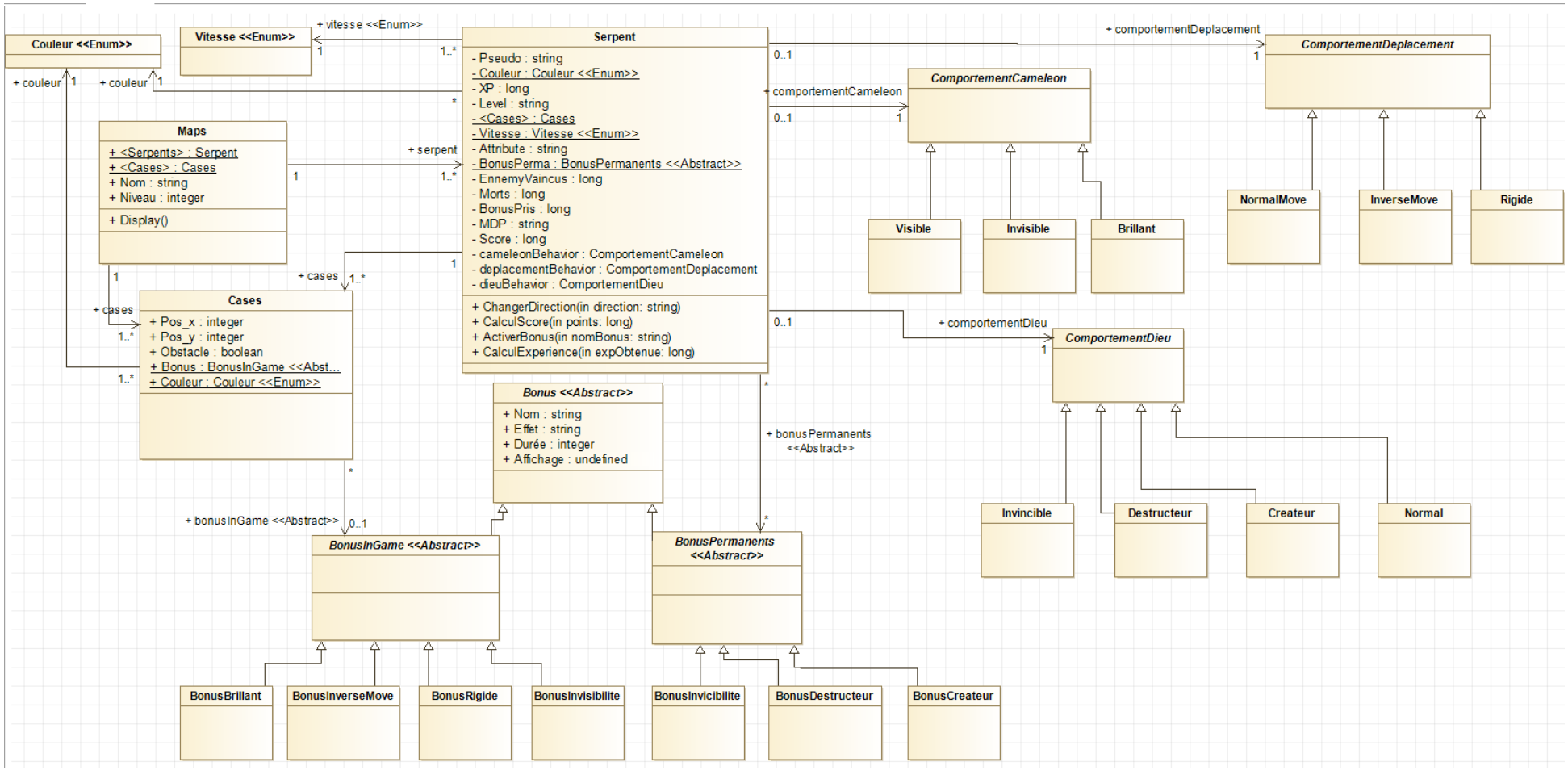

Figure 12 - Diagramme de classes du modèle

Quelques éléments sur la persistance des données

Pour sauvegarder les données de jeu, aussi bien d'une partie à l'autre que durant la partie (pour gérer le bonus de retour dans le passé notamment), une base de données relationnelles est nécessaire, considérant la grande quantité d'informations devant être sauvegardées. Plusieurs tables seront à prévoir dans cette base de données, afin de gérer différents types d'informations, mais également de limiter les temps de recherche au sein de la base. Ainsi, il faut prendre en compte les données des joueurs, de la carte, des bonus permanents, des bonus temporaires ainsi que les options de jeu qui doivent être sauvegardées d'une partie sur l'autre.

Pour déterminer quelles sont les informations à enregistrer dans la base de données, il faut se référer au diagramme de classe côté modèle, mais également veiller à prendre en compte les options de jeu (Volume de la musique, volume des bruitages, connexion automatique, lancement automatique d'une partie, le tout associé à un pseudo) sélectionnées du côté vue.

Toutefois, les données telles que présentées dans le diagramme de classe, ne permettent que la persistance des données d'une partie à une autre (on sauvegarde en effet les différentes map, les différents serpents, les bonus, etc…), mais il est impossible de gérer le bonus de retour en arrière avec seulement ces informations. Pour permettre le retour en arrière, il faut que chacune des informations concernant la map et les serpents soient enregistrées en permanence (le délai de sauvegarde des informations sera à déterminer durant la phase de tests du développement) dans la base de données afin de pouvoir par la suite effectuer une rétrospective de celles-ci. Bien évidemment, un bonus de retour dans le temps ne peut effectuer un retour en arrière que sur quelques secondes, une fois le délai passé, les informations plus anciennes doivent être effacés, tout en faisant attention à ce que les données des joueurs non connectés sur une map ne le soient pas.

En ce qui concerne notre choix technique, nous avons choisi d'utiliser PostgreSQL car il s'agit d'un SGBD (Système de Gestion de Bases de Données) libre, open-source, robuste et gratuit. Qui plus est, la réalisation de sauvegardes et Backups de la BDD (Base De Données) est relativement simple avec PostgreSQL.

4. Conclusion

Pour conclure rapidement, nous avons pu voir ici de quelle manière, pour la partie réseau, nous allons faire communiquer nos clients et notre serveur et pour la partie Design Pattern, comment modéliser un projet assez complexe, étant donné nos connaissances actuelles basiques des technologies à utiliser, en y intégrant certains design pattern. Ce projet est assez enrichissant aussi bien sur le plan technique que sur le plan gestion de projet. En effet, il n'est pas évident quand on est novice de bien s'organiser lorsque plusieurs personnes travaillent ensemble. Enfin, il est évident qu'il reste encore beaucoup de travail à effectuer et que bien entendu il nous faudra revenir sur des choses déjà faites mais c'est comme cela qu'on apprend et qu'on acquiert des connaissances et de l'expériences.