URPGG:DE

Ultimate RolePLaying Game Generator : Delux Edition

VANONI	Joachim	501b
HERAUD	Xavier	501a

Introduction :

Ce rapport est rédigé dans le cadre du projet d'objet et développement d'applications de la troisième année de licence. Une des volontés première de ce projet étais (en plus de répondre au attentes données par les spécifications de celui-ci) de pouvoir rendre un logiciel finis. Il faut comprendre en ce sens un logiciel qui correspondrait au attentes fixé par nous même et qui incorporerait toutes les fonctionnalités définit lors de la phase de conception. Ce but à été atteint de plusieurs manières, tout d'abord nos ambitions initiales on bel et bien été atteintes sur la partie "Génération" du programme, deuxièmement, le logiciel à été complexifié lors de la conception afin de pouvoir mieux satisfaire les contraintes communes à tout les projets de cette matière. Un mode "multijoueur" à donc été codé avec succès ce qui nous permet donc de dire dans dans ce rapport, que notre volonté de rendre un projet fini est accompli. Il faut cependant moduler le propos, le logiciel n'est pas terminer, il faut plutôt le voir comme en "bétâ" au lieu de n'être qu'une simple démonstration technique.

Quel est le but du programme ? :

Notre logiciel est à la base un générateur de fiches de personnages pour jeu de rôle (tout est donc dans le titre). Il faut comprendre par là que l'utilisateur du logiciel peut via le programme généré un document texte reprenant un personnage de jeu de rôle papier classique (type "Dungeons & Dragons" ou "Call of Cthulhu"). L'utilisateur a le choix entre créer un personnage (appelé PNJ dans le logiciel et la suite du rapport) entièrement personnalisé, de laisser la machine lui en créer un de manière aléatoire ou de faire un compromis entre les deux et de faire le choix aléatoire/personnaliser à chaque étape de la création du personnage. De plus, et ce afin de reproduire le rôle du Maître du Jeu (appeler MJ dans la suite du rapport), l'utilisateur peut créer de la même manière des monstre ou des "Boss" (versions améliorées des PNJs ou des monstres).

L'extension de ce logiciel est, comme dit précédemment, un mode "multijoueur". La première innovation par rapport à un simple générateur est la possibilité de jouer directement avec son ou ses PNJs créés par le programme. La seconde (d'où l'aspect "muli" de "multijoueur") est de pouvoir ce connecter en réseau local ou via IP (fonctionnalité non testé car le programme ne s'exécute à l'heure actuel que sur les ordinateurs de l'UFR science et technique). Le système de jeu est emprunt au jeu de rôle papier, il se présente donc sous la forme d'un chat entre plusieurs joueurs où chaque joueur à la possibilité outre de discuter, d'exécuter certaines actions spécifiques qui dépendent directement de son personnage.

Un des joueurs doit prendre le rôle du MJ et tout les autres seront alors joueurs, le MJ aura le droit de sélectionner plusieurs PNJ, monstres et "boss" qui lui serviront tout au long de la partie. Les joueurs eux se connectent à l'adresse du MJ et sélectionnent un et un seul PNJ qu'il utiliserons durant toute leurs session de jeu. Tout en dialogant avec leurs camarades, les joueurs pourrons faire des jets d'attaque ou de défense contre les personnages du MJ et étudier les résultats, le tout calculé par le programme. À tout moment, un joueur peut se déconnecter du salon pour revenir au début du programme. Il en est de

même pour le MJ mais si ce dernier ce déconnecte, les joueurs ce retrouveront alors automatiquement couper de la connexion au serveur et n'auront d'autre choix que de revenir au début du programme. Une fois revennu au début du programme, les utilisateur on le choix de résister à la tentation de rejoindre un autre salon ou de créer un nouveau personnage ou alors de fermer le programme et de s'ennuyer.

Un coup d'oeil sous le capot :

Dans cette section, nous passerons en revus le fonctionnement global du logiciel d'un point de vue de sa programmation. Le programme implémente trois design pattern comme spécifié dans les contraintes initiales. Chacune des deux fonctionnalités s'appuie sur un de ces patrons de conceptions, le générateur de personnage utilise le pattern factory method et le chat multijoueur le pattern observer, enfin ces deux fonctionalités sont mises en commun par un pattern state qui gère la liaison entre les deux principaux bloques du logiciel.

La création de personnages est la base de l'URPGG:DE, elle permet de construire des personnages qui ont tous la même base de statistique et de méthode : la force , qui influe sur les dommages, la constitution pour la résistance, la dexterité qui permet de toucher les ennemis, l'intelligence pour lancer des sorts (qui n'a pas été implémenté), la sagesse pour les test de sagesse en jeu et le charisme pour les interactions sociales. Les méthodes communes au trois types de personnages sont une méthode d'attaque et une méthode de défense. L'inventaire du personnage est un simple vecteur de std::string et à ce jours ils ne disposent pas "d'attaque spéciale. Le déroulement de la création seras revus en détail dans la section réserver au pattern Factrory method.

Le logiciel étant entièrement textuel puisque n'utilisant que la console, nous avons transformer cette faiblesse en force en implémentant un système de reconnaissance de commande. L'utilisateur et constamment invié à taper du texte dans le prompt et à chaque ligne lue la commande est interprété puis le programme à une réaction en fonction de si cette commande est valide ou non. De nombreuses commandes sont acceptées à différants moments de l'exécution et nous ne passerons pas en revue sur toutes.

A tout moment, quand on veut interpréter les commande rentrées par l'utilisateur (c'est souvent le cas lors de la génération non aléatoire des personnages et systématique aux autres moments) on fait appel au CommandeManager. Cette classe possède la fonction analyse(std::string message) qui reçoit la ligne tapé par l'utilisateur et tente de l'interpréter. Le reste de son fonctionnement seras revus plus en détail dans le pattern state, pour le moment il est juste important de savoir que cette classe est la classe pivot du logiciel.

Certain groupes de commandes sont identifiables :

- Les commandes "neutres" qui sont pour la plupart utilisable à tout moment. On y trouve entre autre la commande "help" qui affiche les commandes qui serons reconnu et la commande "exit" qui permet de remonter à l'étape précédente du programme.

- Les commandes de créations, sont des commande qui serons utile à l'utilisateur que lors de la création de son ou ses personnage(s). Ces commande retourne en générale un résultat pour la factory method afin d'identifier les choix de l'utilisateur. On retrouve dans

cette catégorie des commande qui parle d'elles même tel que : "femme, "personnaliser" ou bien encore "orc'".

- Les commandes de chat, se sont toutes des commandes appartenant à la première ou la seconde catégorie avec une légère différence. Pour assuré le bon fonctionnement du chat, tout les message ne doivent pas être traité comme des commandes utilisateur, au risque de ce retrouver avec le message "commande invalide, désoler" à chaque fois qu'il désire parler à ses amis. Pour éviter cela, l'utilisateur devras donc taper "/" devant la commande désiré, elle seras alors interpréter comme un désire de l'utilisateur de parler au programme et non au autres joueurs. Un exemple utile est la commande "/help"

Enfin, toujours en survolant les patterns, le chat est géré par un bloc "a part" appartenant globalement au fonctionnement du pattern observer. Le chat est géré par un système de socket implémenté de base dans les bibliothèques C et par le système de thread du c++11. Le besoin d'implémenter des threads ici s'explique simplement de la manière qui suit : pour pouvoir envoyer des message à ses camarades, un utilisateur doit faire appel à la fonction send(char* mess) qui accepte des appels ponctuels, en revanche pour en recevoir, la fonction receive() doit elle être appelé en permanence. Or le défaut de cette fonction est qu'elle fait à une fonction de socket listen(…) qui met le thread courant en pause tant qu'aucun message n'es reçu. Il à donc fallu déplacé cette fonction de réception dans un second thread pour ne pas bloquer l'utilisation de la suite du logiciel. De plus, pour rendre l'utilisation plus fluide, le joueur qui joue MJ et donc hote (serveur) de la partie implémente un second type thread. En effet, à la différence des joueurs, le MJ devras écouter plusieurs joueurs mais égaelment pouvoir accepter qu'un joueur ce connecte à lui à tout moment, on ce doute assez facilement à ce moment là de l'argumentation que pour recevoir un utilisateur, la fonction accept(…) qui est appeler met également le programme en pause. Un thread est donc créé pour l'acceptation de nouveaux utilisateurs et un thread d'écoute est créé à chaque fois qu'un utilisateur ce connecte.

La fabrique à ~~clampins~~ héros :

La fabrique permet à partir d'un objet de type "Character" de créer un objet de type soit "PNJ", "Monster" ou "Boss", la classe "Factory" a plusieurs methodes pour créer de différentes manière les nouveau personnages.

Toutes les classes filles de "Factory" on en commun la méthode "createAllRandom" qui, dans la classe "PNJFactory", créer complètement aléatoirement le personnage. Pour le nom il est choisi parmi une liste de nom en fonction de sa race et pour les points d'attribut, réparties aléatoirement parmi les cinq attributs qui sont la force, la constitution, la dextérité, l'intelligence, la sagesse et le charisme. La répartition des points d'attribut se fait par rapport a des intervalles propres à chaque race, ainsi l'affectation aléatoire n'est pas entièrement aléatoire et ainsi ne permet pas de cas ou tous les points se retrouvent dans un seul attribut. Dans la classe "MonsterFactory" cette méthode choisis aléatoirement un monstre dans un bestiaire, ce bestiaire contient une liste de monstres classique des jeux de rôles avec des statistiques pré faites, classés par catégories de monstres.

Comme autre méthode en commun on trouve "createPersonnalize" , Dans "PNJFactory" cette méthode demande à l'utilisateur à chaque étape s'il veut définir lui même

les statistiques de son personnage ou s'il veut qu'a cette étape la statistique soit attribué aléatoirement. Ainsi l'utilisateur peut soit créer entièrement lui même un personnage ou partiellement avec une part d'aléatoire. Pour la classe "MonsterFactory" c'est différent l'utilisateur peut soit créer entièrement lui même un personnage ou sélectionner un monstre pré fait du bestiaire. Dans chacune des classe cette méthode utilise la classe "CommandManager" pour analyser les réponses de l'utilisateur, sauf pour les cas ou les mots demandé ne sont par prit en charge par le "CommandManager".

La troisième méthode en commun est "createCharacterSaved" qui créer un objet de type "Monster", "PNJ" ou "Boss". Dans les trois factory la méthode marche de la même manière.

Enfin la dernière méthode est "createCharacter" qui est simplement l'interface principale qui demande à l'utilisateur s'il veut créer un personnage aléatoire ou un personnage personnalisé. Cette méthode fonctionne de la même manière dans la classe "PNJ" et "Monster".

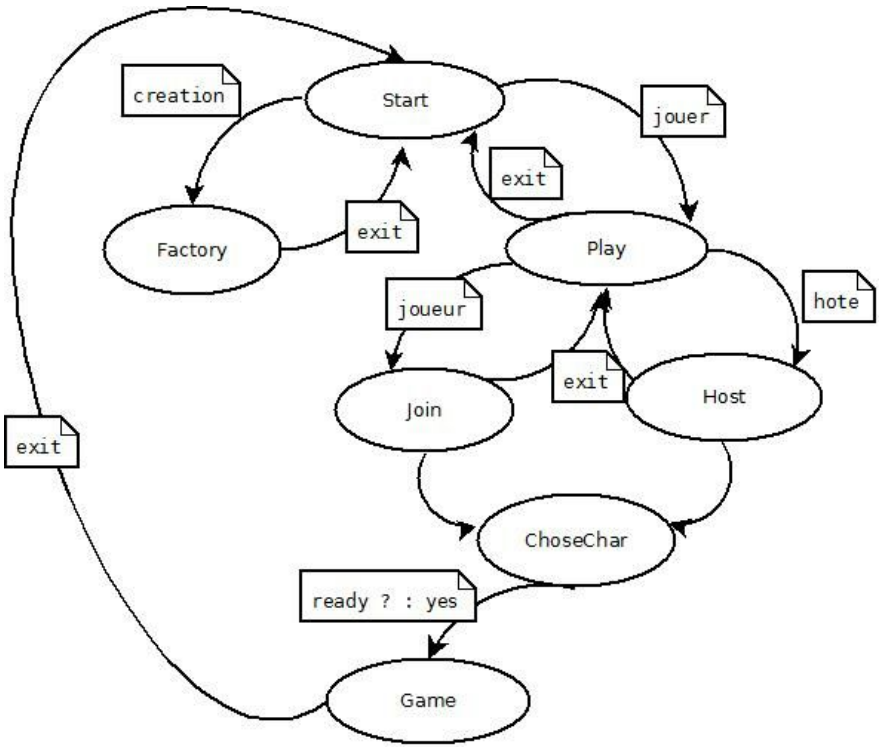
Vous l'avez sans doute remarqué je n'ai pas beaucoup parlé de la classe "Boss" et pour cause elle fonctionne assez différemment des autres classes. La classe "Boss" permet de créer soit un boss qui fonctionne comme un "PNJ" ou un boss qui fonctionne comme un "Monster", ainsi la classe "MonsterFactory" utilise les méthodes de "PNJFactory" et de "MonsterFactory" pour créer ses personnages et monstres, en leur appliquant un bonus pour les rendre plus fort. Donc pour chaque méthodes vu précédemment il sera demandé à l'utilisateur s'il veut un monstre ou un pnj et lancera la méthode de la classe associé.

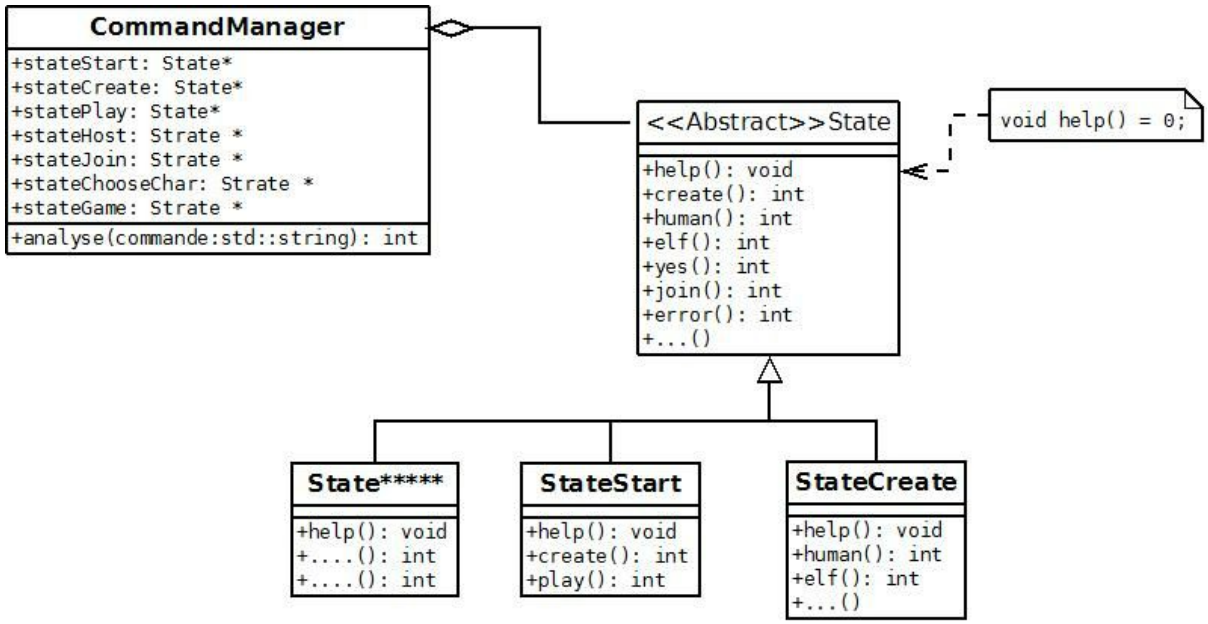
«abstract»
Character
-name: string
#sexe: string
#race: Race
#classe: Classes
#currentLifePoints: int
#lifePoints: int
#currentManaPoints: int
#strength: int
#constitution: int
#dexterity: int
#intelligence: int
#wisdom: int
#charisma: int
#attack: int
#defense: int
#protection: int
#level: int
#bagCapacity: int
#bag: Inventory*
#performAttack(): string
#defend(): string
+applyMinStat(): void
+applyModifications(): string
PNJ
-lastName: string
+performAttack(): string
+defend(): string
Monster
-dommagesDe: int
-dommagesAdditionnels: int
+performAttack(): string
+defend(): string
Boss
-lastName: string
-dommagesDe: int
-dommagesAdditionnels: int
+performAttack(): string
+defend(): string
«abstract»
Factory
#chara: Character*
#commandManager: CommandManager
#config: Config
+createAllRandom(chara:Character*): void
+createPersonnalize(chara:Character*): void
+createCharacterSaved(): Character*
+createCharacter(): Character*
PNJFactory
-randomSex(chara:Character*): void
-randomSkillPoints(chara:Character*): void
+createAllRandom(chara:Character*): void
+createPersonnalize(chara:Character*): void
+createCharacterSaved(): Character*
+createCharacter(): Character*
+applySkillPoints(chara:Character*): void
+save(chara:Character*,path:string,boss:bool): void
BossFactory
-pnjFact: PNJFactory*
-monsterFact: MonsterFactory*
+createAllRandom(chara:Character*): void
+createPersonnalize(chara:Character*): void
+createCharacterSaved(): Character*
+createCharacter(): Character*
MonsterFactory
-createPersonnalizeMonster(chara:Character*): void
-createBestiaryMonster(chara:Character*): void
+createAllRandom(chara:Character*): void
+createPersonnalize(chara:Character*): void
+createCharacterSaved(): Character*
+createCharacter(): Character*
+save(chara:Character*): void

L’état, leader suprême du programme :

Comme dit précédemment, l’utilisateur a, à tout moment, la possibilité de taper des commandes pour naviguer dans le programme ou pour configurer la création de son personnage. Ces commandes sont passées au CommandManager qui les interprètes, c’est en fait là ou le pattern State entre en piste. En effet le logiciel est continuellement dans un état précis, au début l’état “start” puis au fur et à mesure de ses choix dans les autres états du logiciel.

Bien que le commandManager interprète les commande utilisateur, ce n’est pas lui qui agit en fonction, il délègue l’ensemble de ces fonctionnalités à son état courant. Tout les états ayant bien sur été au préalable initialisés à la création de la classe CommandeManager qui se fait dés le lancement de l’URPGG:DE. Chaque état à des fonctions pour toutes les commandes valides durant cet états (par exemple “homme” est valide durant la création de personnage, surtout quand on demande si le personnage crée seras un homme ou une femme mais est totalement inutile durant la connexion quand on veut savoir si le joueur seras hote ou simple joueur). Le pattern state est ici très classique par rapport à celui vu en cour, on trouvera ci-dessous le diagramme UML simplifié et le diagramme d’état pour une meilleur visualisation.



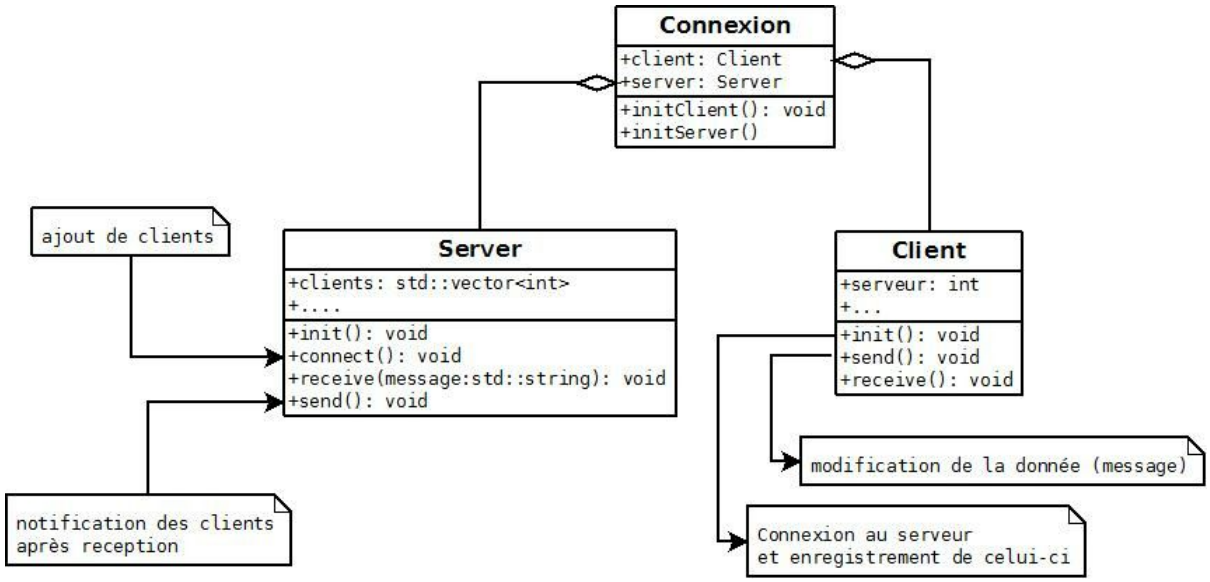


L'observateur, l'oeil qui ne voit pas tant que ça :

Dans notre projet, l'observer à un fonctionnement très particulier. La première particularité est que les observers (les clients) observent un sujet (le serveur) se trouvant sur une autre instance du programme en principe sur une autre machine. La deuxième particularité est que la donnée observer (les message envoyé) est modifié par un des client en envoyant un message, ce qui notifiera tout les autres clients. De plus comme le serveur est également un joueur, il peut lui même modifier sa donné en décidant de notifier tout les observers de son propre chef.

Les clients sont enregistrés auprès du server dans un vecteur d'INT et non pas un vecteur de Client. En effet chaque client étant connecter via un socket unique au serveur, c'est cette donnée que nous utilisons pour les identifier de manière encore plus certaine qu'en passant un objet. De plus cela facilite un certain nombre de traitements d'envois et réceptions de messages. De manière très similaire, le serveur est identifié par un int auprès du client pour les mêmes raisons. Enfin, il n'existe pas de "donnée" à proprement parlé, en tout cas pas de données observer.

Quand un client décide d'envoyer un message, celui-ci est transmit au serveur. Il est alors affiché pour le MJ puis retransmit à tout les autres utilisateurs sauf l'envoyeur initial. Quand le MJ (le serveur donc) envoie un message, il l'envoie alors simplement à tout les utilisateurs connectés. Quand un client se déconnecte, il est simplement retiré de la liste des clients. On retrouve alors toutes les fonctionnalités du patter observer (ajout, suppression et notification d'observer). On trouvera ci-suit le digramme UML simplifié des classes utilisant l'observer.



La conclusion :

Nous pouvons dire en conclusion de se rapport que ce projet nous à apporté beaucoup. D'abord bien sur dans la compréhension et la pratique des design pattern mais également sur la programmation en général. En effet on ne fait pas de gros projets sans en apprendre un peut plus à chaque fois. De plus le sentiment d'accomplissement qui accompagne ce projet (aussi modeste soit-il) est un véritable aspect positif pour nous. Enfin les technique utilisées (threading et socket) bien que pas totalement maitrisé dans ce projet on tout de même été apprise ce qui seras un plus conséquent pour notre future carrière de développeur.

Le programme possède de nombreuse ouverture potentiel pour en faire un logiciel plus concrète. Premièrement un support d'un autre système de jeu de rôle (voir même la possibilité de configurer dans le logiciel le système) pénétreras de rendre l'URPGG:DE bien plus versatile. Deuxièmement la fonctionnalités "multijoueur" peut être grandement améliorer et le support de la fiche de personnage plus important (par exemple en utilisant des compétence spéciale ou des technique spécifique) et avec plus de commande reconnue par le logiciel. Enfin une interface graphique ne ferait pas de mal.

Ceci conclus notre rapport sur ce projet d'objet et développement d'applications pour le premier semestre de L3 informatique.

Annexe 1, Diagramme UML générale :