Projet d'Objet et développement d'applications: "Dr. God or: How I Learned to Stop Worrying and Love the Plague"

Marie DELAVERGNE - Robin WIBAUX
Groupe 502C

2015–2016

Table des matières

Introduction

Le thème de ce projet fut l'élaboration d'un jeu-vidéo de survie RPG dans un monde médiéval. L'idée de base était de s'inspirer de différents RPG, bien sûr, mais aussi de Project Zomboid, Castle crashers pour les statistiques du personnage et

Dwarf Fortress. En rapport à ce dernier, il fallait faire un système poussé de localisation de blessures, voire des états. Il en résulte un jeu vu de dessus où on contrôle notre personnage en temps réel et affrontons quelques zombies dans un terrain unique pour le moment, mais interchangeable grâce au lecteur de carte de la classe Monde.

1 Mode de fonctionnement / Déroulement du jeu

Avant de jouer, le joueur doit commencer par choisir le nom et le sexe de son personnage. Ses statistiques, points de vie, force, etc, sont générés aléatoirement.

Le jeu démarre alors et le personnage se retrouve dans un sanatorium. Le scénario est le suivant : Dieu, dans sa toute bonté et amour de l'humanité -encore une fois-, a répandu la peste dans le monde. Mais pas n'importe quelle peste. Il ne veut pas des humains au paradis et les transforme donc en mort-vivant. Notre personnage y a survécu par chance et se retrouve dans le fameux sanatorium peuplé d'homme zombifiés par la pandémie. Un inconnu va lui faire une proposition qu'il ne pourra pas refuser (et cela sans tête de cheval sur un lit). Nous comptions donc faire un système de gestion de dommages liés à la peste et de statistiques que nous pourrions changer nous même, en fonction également de l'ancien métier de notre personnage.

Le monde est quadrillé : il est constitué de cases de 16*16 pixels. Une case peut être un sol, un mur ou un élément du décor. Le personnage se déplace donc de case en case, via des mouvements discrets (à l'image du jeu Pokemon). Certains éléments du décor tels que l'armoire ou la table prennent plusieurs cases d'espace. Un élément du décor peut être ou ne pas être accessible : un personnage peut aller sur une case sol, cadavre ou porte ouverte (si tant est qu'on peut ouvrir une porte) mais ne peut aller sur une case mur, armoire, table ou porte fermée.

Les actions possibles :

Déplacement Le personnage se déplace avec les touches directionnelles (touches fléchées ou Z,Q,S,D). Le déplacement se lance si la case visée est accessible et libre (aucun personnage ne s'y trouve) et dure un temps précis empêchant tout autre action.

Attaque On attaque avec les clics de la souris. Le personnage étant au centre de l'écran, il se tourne du côté où se trouve la souris (Nord, Sud, Est, Ouest). L'attaque s'exécute donc sur la ou les cases voisines vers la/lesquels il est tourné. L'attaque du clic gauche se fait en ligne droite depuis le personnage, sur autant que cases que le veut la portée de l'arme équipée (portée=1 : 1 case, portée=2 : 2 cases...). L'attaque du clic droite s'effectue sur les 3 cases, devant, devant à gauche et devant à droite du personnage. L'attaque, comme le déplacement, prend un temps précis durant lequel le personnage ne peut rien faire d'autre.

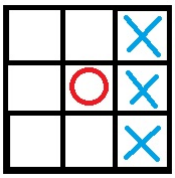
Soin

Inventaire On peut accéder à l'inventaire du personnage via la touche I. L'inventaire s'affiche alors dans le terminal, affichant les objets à s'équiper ou se déséquiper.

Interactions On peut interagir avec certains éléments du décors.

Concernant les PNJ : Concernant les PNJ : Les PNJ de ce jeu sont des zombies hostiles. Ils détectent le personnage lorsque celui-ci est à huit cases d'eux (à vol

FIGURE 1 – Bingo !



d'oiseau) et tentent alors de s'en approcher. Partant du principe que les capacités intellectuelles d'un zombie sont des plus limitées, nous avons restreint au mieux leur intelligence artificiel de manière à ce qu'il soit facile de les semer : ils ne contourneront aucun obstacle... ce sont des zombies, pas des dinosaures. Ils ne savent donc pas non plus ouvrir une porte. Lorsqu'il sont à portée d'attaque (à une case), ils attaquent. Tout comme le joueur, ils ont des statistiques, un équipement (des vêtements et leurs griffes) et peuvent être touchés aux différents endroits du corps. Lorsqu'ils meurent, ils sont remplacés par un cadavre, case accessible contenant l'équipement du zombie. (Vous pourrez looter des mains de zombies !)

2 Description des classes

2.1 Monde et Case

La classe Monde est la classe générant le terrain du jeu. Elle contient notamment un tableau de Case, classe unitaire du monde. Une Case contient plusieurs informations utiles : son accessibilité, l'éventuel identifiant du Personnage ou PNJ qui l'occupe, son éventuel Inventaire et son comportement quand on interagit avec. Elle contient aussi le sprite du décor. Le Monde possède la liste des textures utilisées par les Cases. Via ses méthodes, il sert d'interface entre les Cases et le Jeu. Notamment, sa méthode centrerSur(point) permet de centrer le décor sur le Personnage joueur et sa méthode draw(window) dessine toutes les Cases.

2.2 Jeu

La classe Jeu est la classe qui gèrent l'ensemble du jeu. C'est l'objet qui est déclaré et utilisé dans le fichier main.cpp. Elle contient en attributs le Monde, le Personnage jouable ainsi que la liste des PNJ. Elle récupère des données entrantes telles que le défilement du temps ou les boutons pressés par le joueur avec la méthode inputs(bool*in), gère les actions des Personnage et PNJs via la méthode gestion() et enfin, dessine tout avec draw(window) (qui lance le draw de Monde).

2.3 Personnage et PNJ

Personnage et PNJ sont les classes d'entités mouvantes du jeu. Elles héritent toutes deux de la classe Entite. Entite possède une texture, un sprite, des positions et un compteur de temps. Ses méthodes lui permettent de mettre un sprite en mouvement durant un temps précis ou d'effectuer une autre action à temps donné. Concernant les coordonnées, on y différencie la localisation pour la position entière

de l'entité sur la grille du monde et la position pour la position en pixel dans le terrain. Les classes Personnage et PNJ possèdent un Corps, un Equipement, des Stat et des méthodes d'attaque et de défense, soit les états de ces entités mouvantes. Le personnage a la particularité d'avoir quelques attributs en plus qui ne sont pas encore utilisé, ainsi qu'un inventaire qui peut contenir des Armes, Armures, et Objets. L'inventaire n'est pas limité en place. Quant à PNJ, il a les méthodes lui permettant de réagir : voitCase et detecteCase.

2.4 Inventaire, Arme, Armure et Objet

Un Inventaire contient une liste d'Arme, d'Armure et d'Objet. Les objets Inventaire permettent le stockage et le transport des items (employons l'anglais pour parler aussi bien de la classe Arme, Armure, et Objet). Les classes Arme et Armure ont les informations liées aux performances respectifs des objets les représentants : vitesse, puissance, résistance, durabilité,... Ces informations sont ainsi utilisés par les méthodes d'attaque et de défense de Personnage et de PNJ. Tout n'est pas encore utile : par exemple, la contondance devrait à terme servir à avoir une chance d'assomer un ennemi. L'Objet est la classe abstraite dont héritent Drap et Bandage. Ces classes ont pour elles une description et une méthode d'utilisation sur le Personnage l'utilisant (alors mis en paramètre). On peut déchirer un drap en bandage et utiliser un bandage sur son corps.

2.5 Stat

La classe Stat contient les informations liées aux performances du personnage, jouable ou non : les hitPoints, la mana, la force, etc. On y trouve aussi deux méthodes permettant d'augmenter ou de baisser la vitesse, méthodes appelées selon l'évolution de l'état du personnage (ex : jambe cassée).

2.6 Corps et Membre

Les classes Corps et Membre gèrent l'état du personnage. Corps contient une listes de Membre contenant eux même une liste de Membre. L'idée est ici de recréer le corps humain : tête, corps, bras, jambes, et leurs sous-membres : doigts, orteils, organes... On se sert d'un système de jet de dés assez poussé, utilisé en jeu de rôle, pour déterminer la localisation et les dommages. Les membres sont crées dans le corps avant d'être reliés un par un en une sorte d'arbre de membres. Chaque membre a ses fils, jusqu'aux extrémités et organes qui possèdent un vecteur de fils vide. Ces Membres possèdent des pv (points de vie) indicatifs de leur état, le tout géré par le Corps. Un membre qui tombe à zéro point de vie est sectionné, et tous les membres qu'il porte avec. On peut néanmoins soigner intégralement le corps avec un soin ou un bandage. Il est à bien noter que certes le corps, et donc les méthodes qui ont besoin de lui ne sont pas vraiment flexibles (défendre, soin, etc.), mais ce système étant fixé à un système déjà testé et bien rôdé, il n'y a théoriquement aucune raison que l'on ait besoin de le changer.

3 Patterns

Nous avons utilisé ici trois patterns, dont deux qui sont reliés entre eux. Ce sont malheureusement trois patterns de comportement. Nous avions prévu d'en avoir un quatrième qui consistait en une Object Pool, mais cela a impliqué beaucoup trop de complications pour ne rien gagner au final (l'Object Pool s'occupait des bandages et des draps). Nous détaillerons plus cela dans la section 4, mais nous n'avons donc pas réussi à remplir l'objectif de deux types différents, ne voulant pas forcer des patterns qui n'auraient pas été pertinents dans notre code.

3.1 Observer - Chain of responsibility

Nous avions besoin pour ce projet d'un Observer qui pourrait remonter les informations liées aux dommages sur les Membres. Prévoyant l'éventualité que les notifications changeraient selon les classes et feraient éventuellement évoluer l'information transmise, nous avons opté pour une chaîne d'Observer. Elle est décrite dans le *Game Programming Patterns* sous le nom de "Linked Observers", mais nous l'avons à peine modifiée de sorte que nous avons au final une chaîne de responsabilité d'observers. Notre classe "sujet" est le Membre, qui comporte un pointeur vers un Observer "tête" de la chaîne, en l'occurence le Corps. Quand nous créeons un corps, on donne directement la valeur de ce pointeur, puisque les membres sont créés à l'intérieur même de ce corps.

L'interface Observer en elle-même a un attribut protégé suiv qui est un pointeur vers l'Observer suivant. Elle possède également une méthode virtuelle pure traiter(Membre& m, int pv), et la méthode qui permet de passer l'information à l'observer suivant.

```
class Observer{
protected:
  Observer* suiv;

public:
  virtual void traiter(Membre& m, int pv)=0;
  virtual void passer(Membre& m);
}

void Observer::traiter(Membre& m, int pv){}
void Observer::passer(Membre& m){
  if (suiv)
    suiv->traiter(m, m.getPv());
}
```

Le membre a donc une méthode pour fixer son Observer et une méthode checkPv() qui est appelée dans défendre à chaque fois qu'un membre prend un coup. On commence donc à entrer dans la chaîne, prévenant le corps qu'un membre est touché.

Le corps traite donc le membre qu'il reçoit, ainsi que ses points de vie. Il vérifie si un membre est à zéro point de vie, ses sous-membres doivent aussi changer. A quoi rimerait une main attachée à un corps sans bras ? (coucou Michel Ancel !) Il possède également une méthode spécifique à lui updateObs() qui permet de s'assurer lors

d'une copie ou un assignement de corps que ses membres ont le bon pointeur en tant qu'observer. Mais l'on en reparlera dans la section 4. Il envoie également par la méthode passer à l'Observer suivant, c'est à dire le personnage si c'est le corps d'un personnage, ou pnj sinon.

Que ce soit l'un ou l'autre, on a un getter et un setter pour le pointeur vers l'Observer suiv, les définitions de l'interface dont ils héritent. Néanmoins, une nouvelle méthode apparaît dans chacune de ces classes, personnageMort() et pnjMort(PNJ& p). Cela permet de transmettre au jeu -qui est l'observer suivant, spoil !- si le membre qui était baladé depuis le début de la chaîne était un membre vital (tête et coeur pour le personnage, tête pour le zombie) et qu'il n'avait plus de points de vie.

L'observer suivant, Jeu, donc, se sert également de ces deux méthodes (plus celles de l'observer, bien entendu, puisqu'elles servent dans *traiter*). Selon que ce soit un personnage ou un zombie qui est mort, il va faire passer un message à l'interface pour dire que le jeu est fini ou il va retirer le zombie du jeu, en le transformant en cadavre, comme expliqué précédemment.

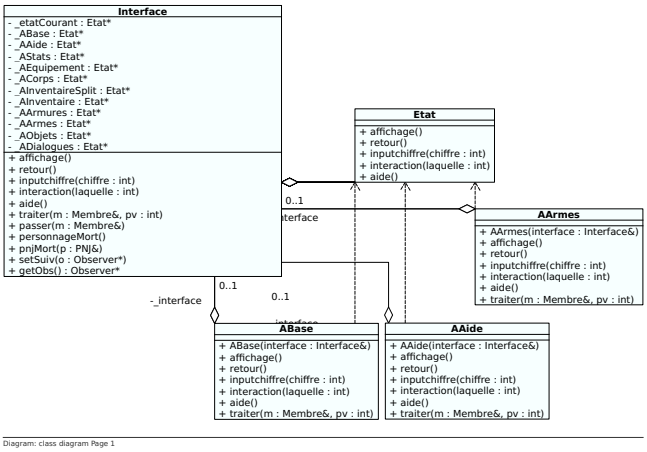
Enfin, l'information est transmise à l'interface puis à ses différents états pour qu'ils puissent afficher et réagir en conséquence. A noter que bien entendu, les Etat concrets ne se servent pas de suivant puisqu'ils sont en bout de file, et donc pas de passer non plus.

La magnifique (ça marche mieux dans *Game Programming Patterns*) figure en page 10 détaille le pattern plus en détails, ainsi que son mode de fonctionnement. Pour des raisons de lisibilité seul un Etat concret et seules les méthodes liées aux patterns Observer et Chain of Reponsibility ont été représentés.

3.2 State

Comme cela sera expliqué dans la section 4, nous voulions à la base utiliser TGUI pour créer l'interface de notre jeu. Plusieurs raisons nous ont poussés à changer de voie : tout d'abord, le manque de temps, qui, il faut bien l'avouer, est la raison principale, ensuite, quelques problèmes liés aux tutoriels - parce que oui, nous avions tout de même passé du temps à essayer de l'implémenter - et finalement l'envie de garder un côté un peu "retro" - les graphismes confirmeront -. Nous avons donc choisi de revenir sur un affichage de l'interface sur la bonne vieille console, estimant que l'on peut tout de même y faire quelque chose de fonctionnel et "sexy". La figure suivante explique le fonctionnement du pattern, qui est au final le pattern state "de base", comme vu en cours. Nous n'avons bien sûr pas représenté tous les Etats concrets (¡ spoiler alert ! Il y en a actuellement 11, mais ce chiffre n'est pas contractuel tant que le projet n'est pas rendu).

FIGURE 2 – Pattern State



Pour implémenter ce pattern, nous avons mis le client (interface) en référence dans les sujets concrets et un pointeur vers chaque état dans l'interface. Il est possible que ces classes changent encore après impression de ce rapport, comme un pointeur unique dans Interface, nous n'avons pas encore eu le temps de nous y pencher autant qu'il aurait été nécessaire.

MISE A JOUR : Même sans avoir eu le temps de refaire le diagramme, le pattern a plutôt pas mal changé, nous avons en attributs privés d'interface un pointeur vers l'état courant, une référence vers le personnage du jeu, un pointeur vers la dernière case avec laquelle on a interagit et un entier qui permet de déterminer si on veut un inventaire de personnage ou pnj. En attributs publiques, ses Etats statiques, ce qui nous permet en théorie d'y accéder de partout, mais également de ne pas avoir à créer de nouveaux états et effacer les anciens au fur et à mesure des changements d'états.

Une grosse amélioration aurait certainement pû être effectuée au niveau de l'état dialogue, avec carrément un autre pattern state qui aurait évité ce paquet difficilement lisible de switches, surtout avec le manque de temps pour commenter.

4 Difficultés rencontrées et améliorations possibles

4.1 Le terrain

Changement d'orientation à mis-parcours : Le jeu devait initialement se jouer sur un terrain continu et non quadrillé. Nous découvrions tous les deux la SFML et la création vidéo-ludique. Des recherches avaient été lancées, par exemple, concernant la gestion des collisions de différentes formes : point-cercle, cercle-cercle, cercle-rectangle, rectangle-rectangle ou autres formes convexes. Devant la complexité et la quantité finalement assez conséquents du travail à fournir, nous avons décidé comme pour tout bon projet qui se respecte de réduire nos ambitions. Donc un terrain quadrillé et non continu, pas d'armes à distances et un grand nombre de "ça viendra dans un DLC payant" (un season pass est prévu... ou pas). Cette décision fut prise durant les vacances d'Octobre, soit en milieu de période de projet, au vu de l'avancée pour le travail déjà fourni. Et nous revenons également à cette idée d'avoir

un jeu un peu rétro à la Pokemon, avec un personnage en SD (oui, la texture a été changée) qui tient sur une case au lieu de deux. Deux cases impliquaient trop de décisions que nous n'avions pas envie de prendre.

4.2 TGUI

Il a été envisagé comme déjà dit d'utiliser la bibliothèque TGUI. Ou SFGUI, à la base, également. Nous avons finalement décidé que les informations s'afficheraient sur le terminal, alors disposé à côté de la fenêtre par les soins du joueur. Dans le but OFFICIEL d'innover. Officieusement, les raisons ont déjà été données.

4.3 Le C++

Force est de reconnaître que c'est un langage où tout est fait pour l'optimisation de la vitesse d'exécution. Mais c'est également un langage assez difficile à maîtriser, et clairement, nous en sommes loin. Pour donner notre niveau, à la base, nous avions du mal à savoir pourquoi/comment faire un .hpp et un .cpp.

Première surprise, même si cela avait été expliqué en TD, il a fallu s'y heurter pour s'en souvenir : les passages qui ne se font pas par référence. Au bout d'un moment, ça rentre, mais il a fallu comprendre pourquoi un Personnage sans son esperluette en tant que paramètre se comportait... et bien, certainement pas comme il le devait. Faire la différence entre le & et le * a été vite requis, également.
Puis vint le moment où la construction d'une classe, même avec un passage par référence d'une autre classe, s'écroula. Et Marie et Robin découvrirent l'initialisation des membres dans le constructeur. Pour notre défense, nous n'avions pas du tout idée que les membres étaient initialisés avant même l'accolade.
Il y eut aussi l'incident du vecteur polymorphique, qui nous força à utiliser la librairie boost, une certaine personne de ce binôme -qui ne sera pas désignée- refusant de céder ce magnifique vecteur. Le ptr_vector de Boost s'occupe également lui-même des objets qui sont crées ou effacés. Cela fut par contre l'une des raisons de l'échec de la grande aventure de la PooldObjets, un problème avec la méthode transfer nous empêchant de continuer plus loin, et nous mettant face à son inutilité relative en premier lieu. Les cases étant définies au début du jeu, leur inventaire également, on ne gagnait pas de temps pour la création des objets bandages et draps, qui eux même ne contiennent que deux strings.
Et puisque nous en sommes aux inventaires, nous avons eu également une nouvelle surprise avec prendreArmure/Arme/Objet de cette classe. En effet, si nous avions correctement lu la magnifique documentation du vector, nous aurions gagné un temps non négligeable à comprendre pourquoi l'objet qui était retiré n'était pas celui mis dans l'équipement, comme voulu. Je cite : *Removes from the vector either a single element (position) or a range of elements ([first,last)). This effectively reduces the container size by the number of elements removed, WHICH ARE DESTROYED*. Pour notre défense, ce n'était pas écrit en capslock, dans la documentation. Voici donc un extrait du patch qui a donc été appliqué, et c'est bien le seul endroit où l'on renvoie une copie et non une référence.

```
1 Armure Inventaire::prendreArmure(int loca){
2   Armure arm = _armures.at(loca);
3   _armures.erase(_armures.begin() + loca);
```

```
4     return arm;
5 }
```

4.4 Les copies et l'opérateur =

En liaison avec notre chaîne d'Observer, nous avons été finalement obligés de les redéfinir pour les classes concernées qui perdaient leur pointeur suiv à chaque passage, causant un nombre incalculable de SEGFAULT, qui ne sont pas toujours faciles à comprendre au premier abord. Et puis, au final, on en vient à savoir que c'est le dernier observer qui a été mis en place qui la cause. Jusqu'à ce qu'on comprenne exactement à l'avance comment les éviter.

4.5 Le temps

Oui, ce fut un problème, même en s'y prenant autant que possible à l'avance. Pour une fois, en plus, c'en est vexant. Les problèmes s'accumulaient et pour une erreur de fixée, dix apparaissaient, que l'on a essayé de réparer souvent plus par patchs qu'en profondeur, ce qui a forcément créé d'autres surprises. Nous avions beaucoup d'idées en début de projet et ça n'a pas été toujours facile de les abandonner parfois... Et d'autres tellement évident : Pourquoi on trouverait un arc dans un monastère ? On retire les armes à distance. (Et oui, il y a des hallebardes dans les monastères, mais je n'en dirais pas plus pour ne pas spoiler.) La nourriture, le moral, les armures en parties réalistes (mais un plastron en tissu qui couvre aussi les mains, c'est bien aussi)... Il a fallu apprendre à mettre de côté des idées qui auraient consommé beaucoup trop de temps, et quelque part ce n'était pas plus mal. Espérons seulement maintenant que notre code soit suffisamment flexible pour étendre tout cela.

Conclusion

La création d'un jeu-vidéo fut une expérience intéressante quoiqu'épuisante. Les possibilités d'applications de design pattern sont diverses et laissent facilement place aux surprises, et donc à du temps supplémentaire pour résoudre les problèmes. Surtout en C++.

Pour finir sur une note plus positive, le projet sera rendu, mais nous espérons avoir gardé l'envie de continuer à le travailler. Et puis nous sommes passés d'un skill noob en c++ à ... débutant/intermédiaire ? Cela a été très formateur sur bien des points, pour le c++, les design patterns et l'avancement global des projets gérés de A à Z par nous-même. Même si l'on imagine que c'est le but de ce genre de projet, c'est important de le reconnaître après avoir sué.

Citation récursive (d'un auteur que j'ai oublié) : « La règle veut que toute équipe de développement prend inévitablement du retard dans la réalisation de leur projet. Y compris s'ils prennent en compte la-dite règle. » Si vous connaissez l'auteur de cette citation, dites-le moi et vous me rendrez heureux !

Annexes

FIGURE 3 – Observer - Chain of Responsibilty

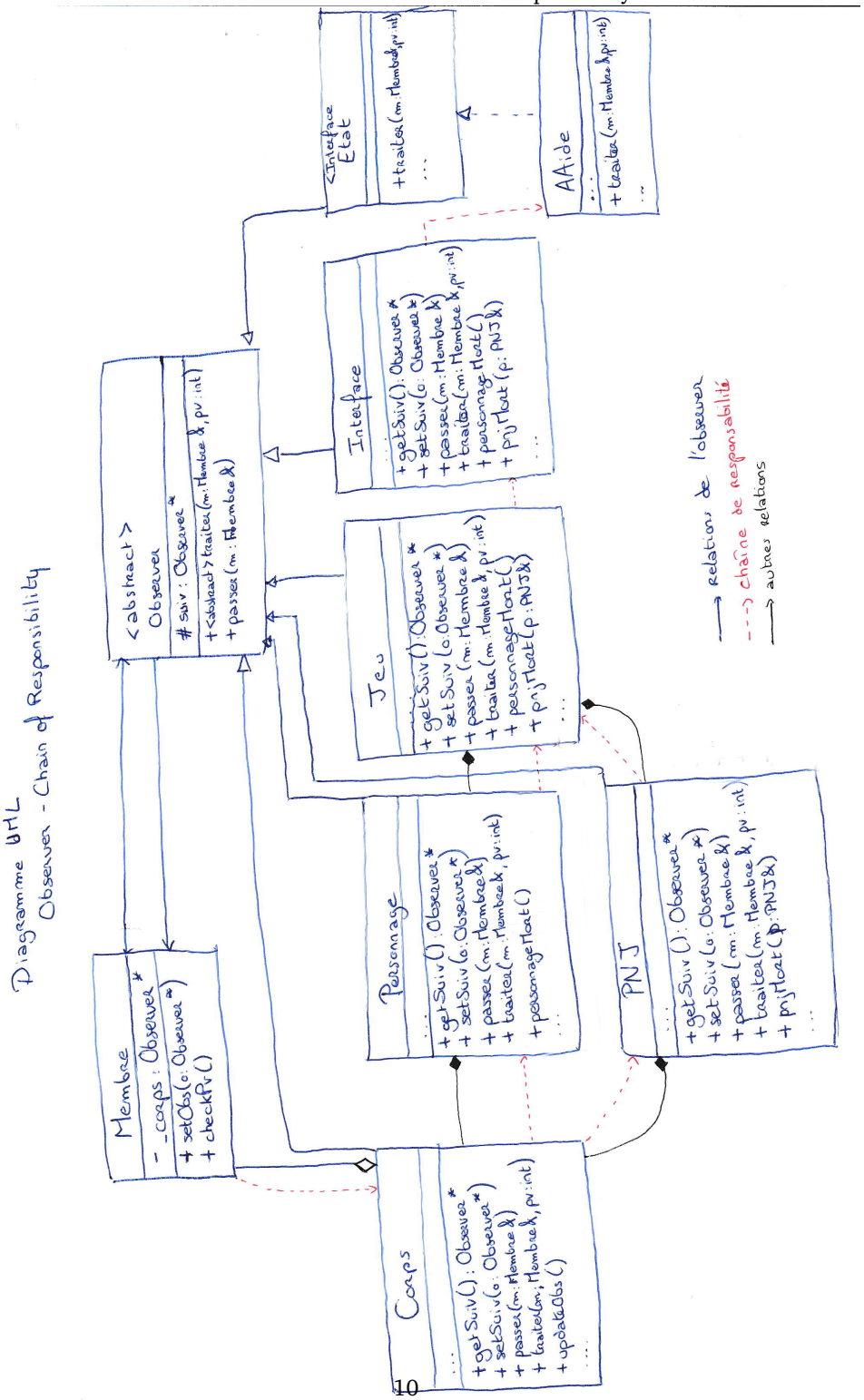


FIGURE 4 – UML complet

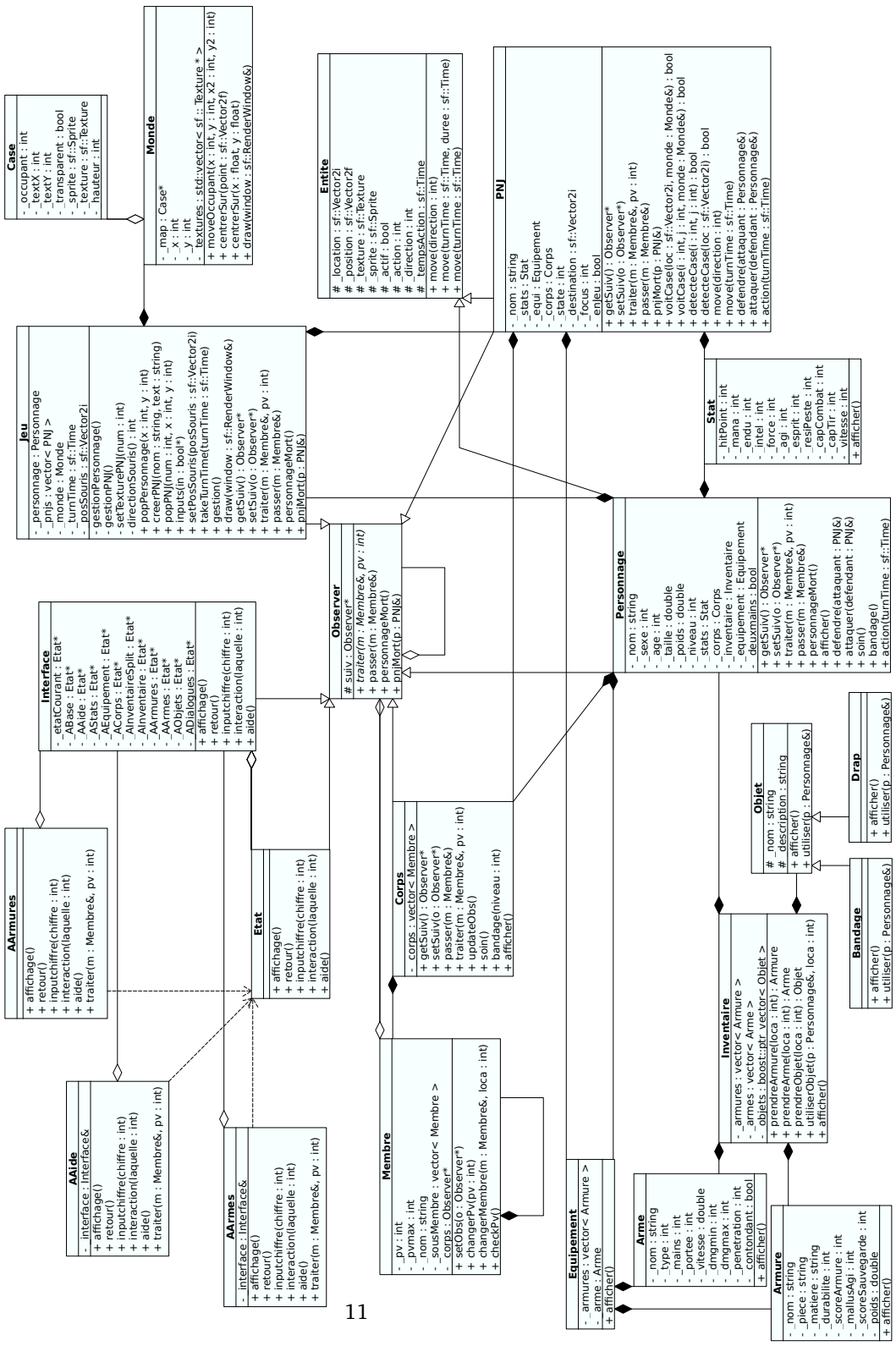


Diagram: UML Complet Page 1