ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Informatique orientée objet
Rapport projet Donjon

LANNOY Christophe
LOUVET Louis

Année académique
2016 - 2017

1 Introduction

En deuxième année de bachelier, les étudiants de l'école polytechnique de Bruxelles sont invités, dans le cadre du cours d'informatique sur la programmation orientée objet, à réaliser un programme en langage java. Cette année, le programme est un jeu de type Donjon. Ce rapport contient donc les différentes fonctions ayant été implémentées accompagnées d'une brève description de celles-ci, ainsi que deux différents types de diagrammes UML (Unified Modeling Language). Le premier est un diagramme de classe du package "Model" implémenté dans le jeu. Le second est un diagramme de séquence d'une séquence choisie arbitrairement et permettant d'illustrer la communication entre les différentes classes dans un cas précis.

2 Choix de carte et type d'inventaire

Carte Il a été décidé d'opter pour une carte de type statique. Une fois l'ensemble des monstres du niveau tués, une case "exitBlock" apparaît, et lorsque le joueur passe dessus il est "téléporté" sur un autre plateau. Le *game* génère les *map* de jeu à partir de fichier texte (au format .txt). La taille devant être paramétrable, le premier niveau est de taille 10x10, le deuxième de taille 20x20 et enfin le dernier de taille 40x40 block. Evidemment, les réglages ont été adaptés afin que, quel que soit le nombre de blocks sur la fenêtre de jeu, celle-ci adapte la taille des blocks afin que la place utilisée dans la fenêtre soit la même.

Inventaire L'inventaire est affiché à coté de la partie de la fenêtre associée au plateau de jeu. Vide au départ, le joueur peut ramasser les armes sur la map pour le remplir. Il peut aussi évidemment s'équiper de celles-ci (changement d'équipement courant). Pour cela, il doit appuyer sur 1 ou 2 selon qu'il souhaite s'équiper de la première ou deuxième arme disponible dans son inventaire. Lorsqu'il est équipé d'une arme de l'inventaire, celle-ci est entourée de rouge.

3 Fonctions implémentées

- **Potion vie et malus "ivresse"** : Sur chaque map sont générées des potions. Le joueur peut alors se déplacer jusqu'à celles-ci et une fois dessus, en cliquant sur la touche "D" (pour Drink), il boit alors cette potion. Celle-ci a un effet instantané, et rajoute au joueur le nombre de vies qu'elle a en argument, ou alors complète les vies du joueur jusque MaxLifes (paramètre du player) si ce dernier a déjà un bon nombre de vies. Cependant les potions ont

aussi un effet malus, et place le joueur en état d'ébriété pendant 5 secondes. Durant ce laps de temps, les touches sont inversées. Ces potions sont représentées par des cases rouges.

- **armes** :
  - **épée** : Première arme disponible pour le player. Lorsque que ce dernier est généré sur la première map, il ne sait pas encore attaquer les ennemis. Pour ce faire, il doit d'abord se déplacer jusqu'à la case épée, et lorsqu'il appuie sur la touche "T" (pour Take), il place celle-ci dans son inventaire. Ensuite il peut évidemment s'en équiper en appuyant sur 1 (qui équipe le joueur de la première arme disponible dans son inventaire), et peut alors, en se plaçant en face des ennemis, attaquer ceux-ci au "corps-à-corps". Une épée est représentée par une case noire.
  - **bâton de mage** : Seconde arme disponible. Le player doit à nouveau se déplacer sur la case, ramasser l'objet pour pouvoir ensuite s'en équiper. Celle-ci inflige un damage-area (dans un rayon de deux cases) aux ennemis autour du joueur. Un bâton de mage est représenté par une case verte.
- **Armure** : Une case armure peut aussi être générée. Elle observe les déplacements du player lorsqu'elle est crée. Si le joueur passe sur cette case, il est automatiquement équipé de celle-ci. Les ennemis l'attaquent toujours avec la même valeur, mais le joueur peut encaisser plus d'attaques avant de perdre une vie. En plus de ceci, lorsque qu'il passe par la case, le joueur sait qu'il est équipé de l'armure car un petit graphisme au couleur de l'armure (bleu clair) est présent autour de player.
- **Téléportation** : Les cases téléportations sont générées par 2 sur chaque map, et permettent au joueur, une fois sur la case, d'appuyer sur "P" et le player est alors transporté instantanément jusqu'à l'autre case téléportation. Il est le seul a pouvoir emprunter ces portes, et peut donc éventuellement se mettre à l'abri, ou accéder à des parties de la map auquel il n'aurait pas eu accès sans ces cases. Les cases téléportations sont blanches.
- **Parchemin** : Le principe est le même que pour l'armure. Le joueur n'a qu'a passer par la case parchemin pour le lire et en apprendre d'avantage sur le maniement des armes. Cette compréhension plus poussée de ses armes lui permet de doubler les dégats qu'il inflige aux ennemis à chaque fois qu'il attaque ceux-ci. Un parchemin est représenté par une case brune.
- **Intelligence artificielle** : Tout les ennemis générés disposent d'une "IA", c'est à dire qu'il vont tenter de se rapprocher du joueur à tout moment, et si ils sont à proximité directe de

celui-ci, ils lui infligent alors une attaque.

- **ExitBlock** : Lorsque le player tue le dernier ennemi, une case "exitBlock" apparaît et lorsque le joueur passe dessus, il est automatiquement transporté dans le niveau suivant. Ces cases sont noires et représentent un "trou" dans la map.

- **GameOver** : Lorsque le player perd sa dernière vie, une fenêtre est affichée avec une peinture de Mondrian (à l'image du jeu "Guerre des Blocks") sur laquelle il est écrit " GAME OVER".

En résumé : Toutes les fonctions suivantes sont implémentées dans le jeu : Potion de vie instantanée, armes (épée et bâton de mage), armure, parchemin permettant d'apprendre au player des compétences, modification de l'équipement courant, graphismes accompagnant certaines actions (s'équiper d'une arme, ramasser l'armure) et aussi permettant de connaître la direction du joueur, état d'ébriété durant un laps de temps, area-dammage, génération de nouvelle case permettant de changer de niveau, génération fenêtre lors de la mort du player, ennemi attaquant le player, disposant même d'une intelligence artificielle, téléportation,...

4 Diagrammes UML

4.1 Diagramme des classes

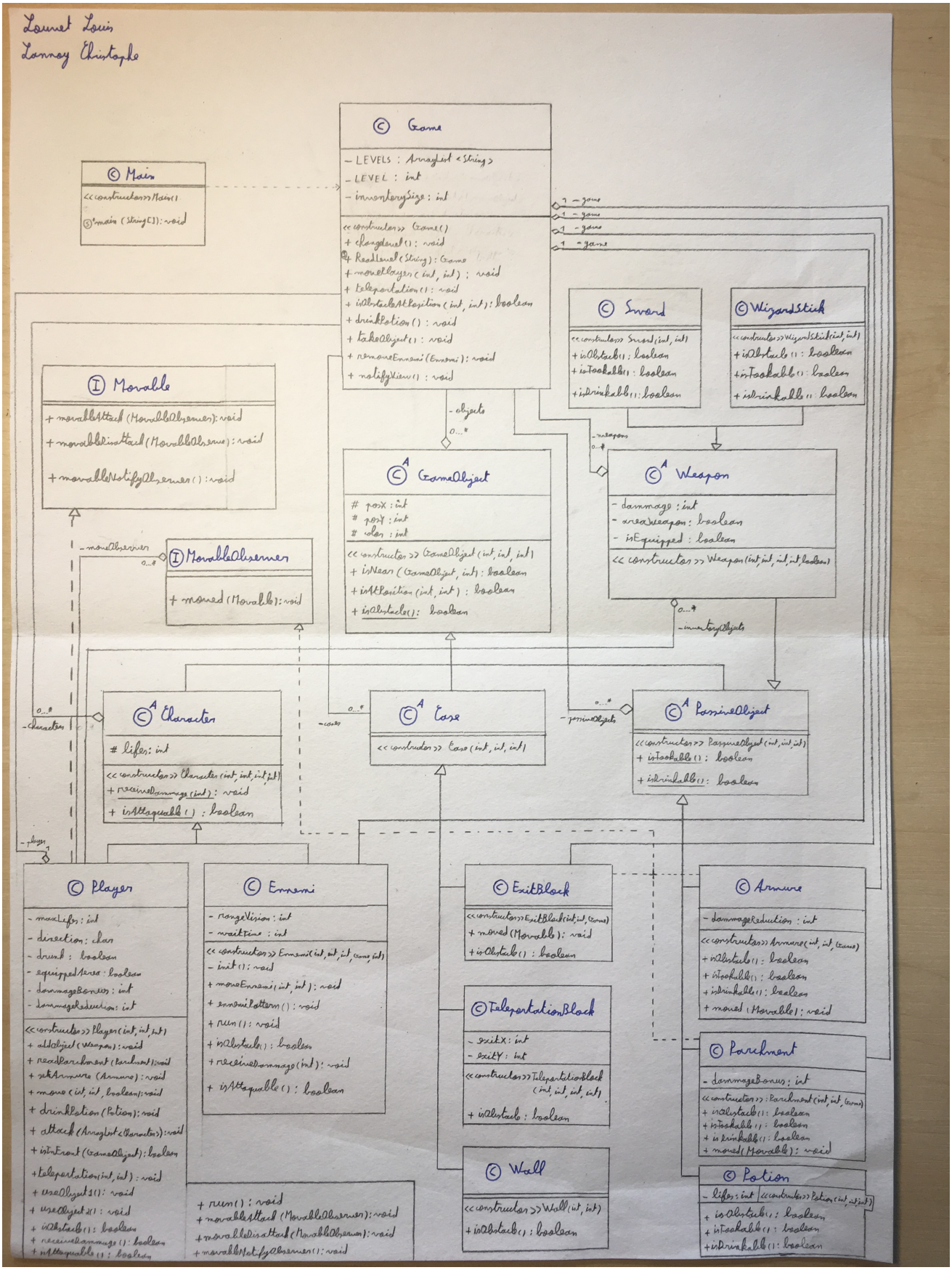


FIGURE 1 – Diagramme de classes (voir image jointe pour une meilleure résolution)

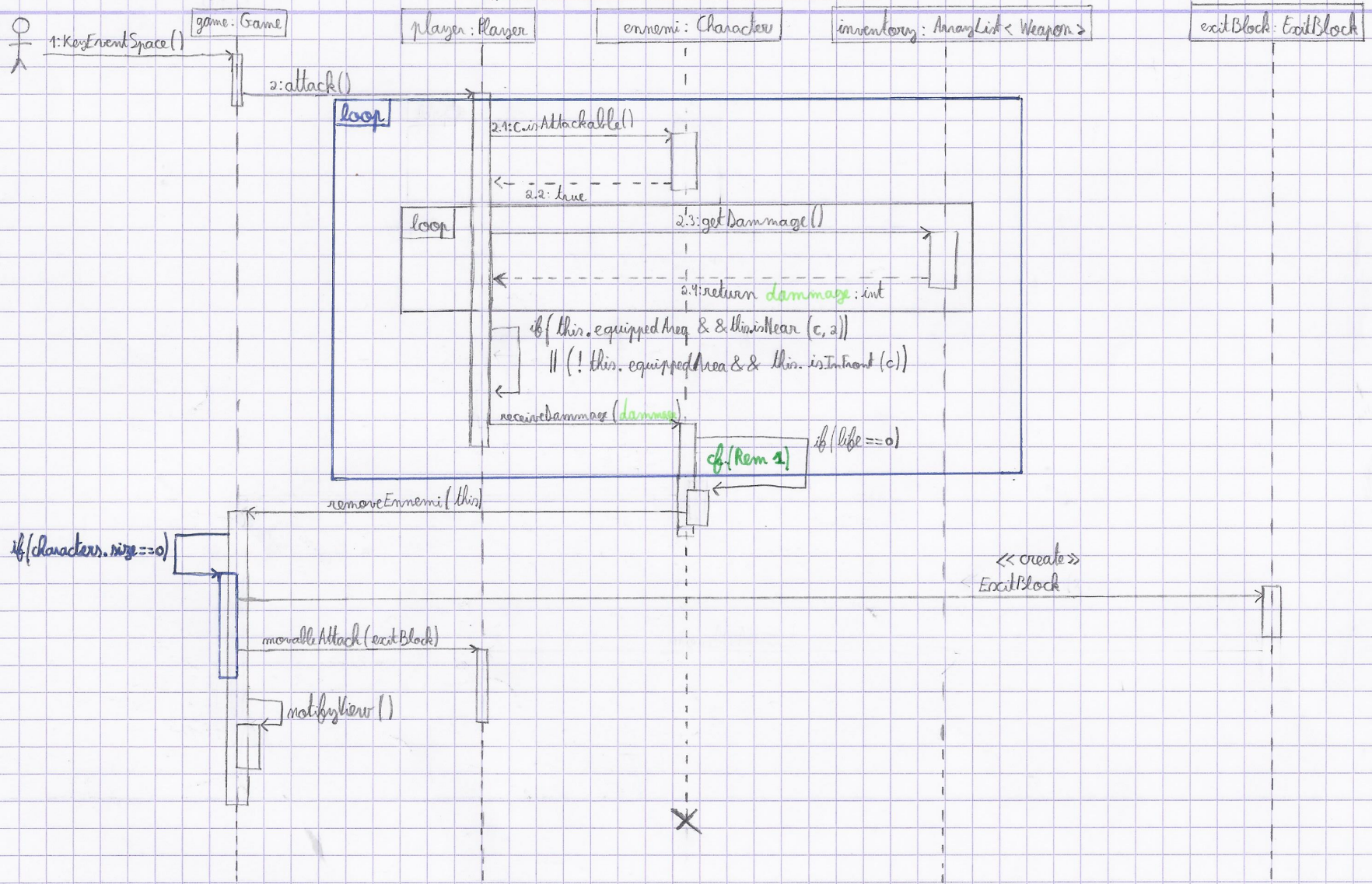
Le diagramme de classes tend à représenter l'architecture du programme le plus fidèlement possible tout en évitant de le surcharger. Les classes Window, Map et Keyboard ont donc été volontairement omises de ce diagramme afin de le rendre plus lisible. En effet, le programme suivant un patron de conception model-view-controller, ces classes sont quasiment identiques à celle utilisées dans le jeu Bomberman. Les méthodes de type "getter" et "setter" ne sont pas non plus représentées sur ce diagramme et les méthodes abstraites ont été soulignées. Des interfaces "Movable" et "MovableObserver" ont été utilisées afin de permettre aux objets "ExitBlock", "Armure" et "Parchment" d'être notifiés des déplacements du joueur et de pouvoir réagir si ce dernier se trouve au même endroit de la map. Nous avons donc utilisé ici les patrons de conceptions *modèle-vue-contrôleur* et *observateur*. Le patron de conceptions *observeur* permet donc ici à certains objets de réagir automatiquement à la présence du joueur sur certaines cases du jeu. Il aurait également été plus judicieux d'utiliser un deuxième patron de conceptions *observateur* (muni des interfaces "Demisable" et "DemisableObserver") pour supprimer les objets de type "Armure", "Parchment" et "Ennemi" de la map (au lieu de le faire via une référence du "game" donnée aux objets lors de leur création) mais comme, nous l'expliquons dans la conclusion, il été décidé de garder cette première architecture. Une image de ce diagramme a été jointe au dossier afin de profiter d'une meilleure résolution.

4.2 Diagramme de séquence

Le diagramme de séquence a pour but de représenter en détails une séquence bien précise du jeu, et ainsi de mettre en évidence les différentes interactions qui peuvent exister entre les différentes instances du jeu. Nous avions le choix de la séquence, nous avons donc décidé d'illustrer la séquence où le joueur fait "apparaître" l'exitBlock. Pour cela nous supposons qu'il ne reste plus qu'un seul ennemi, qui perdra ses derniers points de vie lors de la prochaine attaque du player. La séquence commence donc quand le player appuie sur la touche espace (message d'origine) pour lancer une attaque, qui va enlever les points de vie restants au dernier ennemi, qui va donc "mourir". Le game va alors le supprimer de la map et créer un block de la classe ExitBlock. Remarque : Dans ce diagramme de séquence, les condition sont supposées vraies pour que la séquence désirée ait bien lieu. Pour plus de clarté et de compréhension de la part du lecteur, il a été décidé de préciser ces conditions sur ce diagramme, même si ce ne sont pas des fonctions. Une image de ce diagramme a également été jointe au dossier afin de profiter d'une meilleure résolution.

Diagramme de séquence : apparition de l'ExitBlock

(dernière attaque sur le dernier ennemi)



Explications et remarques

- La grande boucle loop bleue est effectuée sur chaque instance de la classe abstraite "Character". Ainsi la méthode "attack()" permet d'attaquer plusieurs ennemis à la fois, même si le jeu est étendu et que l'on y rajoute d'autres types d'ennemis, ceux-ci n'auront qu'à hériter de character pour être attaquable ou pas. Le player étant aussi une instance de la classe Character, on pourrait également imaginer un mode multijoueur où les players pourraient s'attaquer entre eux. Le petit "c" représente donc à chaque fois une instance différente d'un Character.

- equippedArea est un argument booléen du player, et permet de savoir si l'arme qui équipe le player est une arme qui effectue des attaques d'aires ou pas. Ainsi quand on appuie sur 1 ou 2, et que le player change d'équipement, il va aller demander à l'arme en question si elle est du type area, et en fonction changer l'argument du player. De cette manière, on pourrait instancier toutes sortes d'armes. Il suffirait à chaque fois de préciser les dégâts de celle-ci et de définir si c'est une arme d'aire ou pas.

- "this.isNear (c,area)" et "this.isInFront(c)" permettent de vérifier si le character passé en argument est respectivement aux alentours ou en face du player.

- Rem 1 : La classe ennemi implémente l'interface Runnable, et sa fonction run() est lancée dès que l'ennemi est créé, et tourne tant que la vie de ce dernier est positive (while life >0). Ainsi la condition représentée sur le schéma est la condition de sortie de boucle while du run().

- movableAttach va placer l'exitBlock juste créé dans la liste des observeurs des déplacements du joueur.

5 Conclusion et critique personnelle

A l'issue de ce projet, il est important de porter un regard critique vis-à-vis du travail réalisé durant de ces dernières semaines. En effet, le concept d'orientation objet est un concept difficile à appréhender, et certaines erreurs de logique et de bonnes pratiques ont pu se glisser dans le code. Au fur et à mesure de l'avancée du projet, certains concepts ont été mieux compris et on a pu remarquer certains défauts que pouvait contenir notre code. Il a été décidé de ne pas toujours tout recommencer à chaque fois, si ces parties étaient fonctionnelles, pour ne pas recommencer à zéro,

mais plutôt de porter un regard critique et d'envisager une meilleure solution pour chaque problématique. Nous pensons que le but d'un apprentissage par projet est aussi de prendre connaissance de ses erreurs pour ne plus les commettre à l'avenir, et de cerner plus pratiquement des concepts délicats tel que l'orienté objet ou certains patrons de conceptions. Ce projet a aussi permis d'apprendre à utiliser et à maîtriser les diagrammes UML, permettant une communication efficace du code, peu importe le langage informatique utilisé. En conclusion, on peut affirmer que les apports intellectuels de ce projet sont nombreux et non négligeables, et justifient complètement la présence de ce dernier dans le cursus des étudiants en ingénierie civil.