BARKER Eliott
OUALI Chakib

Groupe C

2022 - 2023

UVSQ université PARIS-SACLAY
IUT de Vélizy-Rambouillet
CAMPUS DE VÉLIZY-VILLACOUBLAY

SAé-2

Simulation RPG

R2.01 - Développement orienté objet (D. Auger)
R2.02 - Développement d'application avec IHM (I. Robba)
R2.03 - Qualité de développement (T. Duffaud)

Table des matières

I. Introduction

Pour cette SAé, il nous était demandé de faire une simulation d'un RPG. Pour ce projet, nous avons déjà dû faire une évaluation du projet avec un planning et une étude de rentabilité. Nous devions aussi programmer en Java des classes pour la simulation. Ensuite, nous avons développé une interface graphique pour cette simulation. Durant toute cette SAé, nous avons également versionné notre code avec Git et nous avons aussi fait des tests unitaires sur les méthodes qu'on a programmé.

II. Nos outils

Lors de la création de notre programme, nous avons utilisé comme IDE, environnement de développement intégré, le logiciel **Intelli J** avec les extensions *Maven*, *JUnit* et *JavaFX*. Pour le début de notre projet, nous avons aussi utilisé l'extension *Code With Me* pour qu'on puisse programmer à deux sur le même fichier. Pour versionner notre code, nous avons utilisé **Git** ainsi que **GitHub**. Ensuite, nous avons aussi utilisé le logiciel **StarUML** pour le diagramme de classe.

III. Développement

1. Parsing des fichiers

Pour développer la classe Parsing, nous avons utilisé la classe *Scanner* de Java. Notre classe se divise en deux méthodes, la première méthode lit un fichier et appelle la deuxième méthode pour créer une quête et la rajouter dans le scénario créé par la première méthode.

2. Scénario et quêtes

Nous avons développé deux classes, Scenario et Quest. Le scénario contient des quêtes ainsi qu'un fichier associé. Les quêtes contiennent un identifiant, une position, des préconditions, une durée, de l'expérience et un titre.

3. Map

Nous avons développé une classe Map pour représenter une carte dans une matrice 2D grâce aux positions données dans les quêtes. Une map est associée à un seul scénario.

4. Solutions

Pour les solutions, nous avons décidé de faire une classe Solution qui sera une classe mère des autres classes solutions. Cette classe implémente les méthodes *doableScenario()* et *caracteristiques()* qui permettent respectivement de :

- renvoyer le scénario possible à partir des quêtes déjà effectuées et de l'expérience accumulées.
- renvoyer les caractéristiques d'une solution.

a. Gloutonne

Tout d'abord, nous avons créé une classe *Greedy*, classe fille de la classe *Solution*. Cette classe correspond à l'algorithme glouton de la simulation.
Pour vérifier les quêtes disponibles, on utilise la méthode *doableScenario()*.
Il y a deux types disponibles :

- Efficace : Solution qui effectue toujours les quêtes les plus proches (méthode *nearestQuest*) et qui effectue la quête finale lorsqu'elle est disponible.

- Exhaustive : Solution qui effectue toujours la quête la plus proche, mais qui n'effectue pas la quête finale tant que toutes les autres ne sont pas réalisées.

b. Speed Run

Dans un second temps, nous avons réalisé une solution *Speedrun*. Cette classe permet d'obtenir les solutions les plus optimisées. L'objectif étant de récupérer, avec un algorithme récursif, toutes les solutions possibles pour un scénario X. Cet algorithme effectue à la méthode d'un parcours en profondeur d'un graphe.

Toutes ces solutions sont ensuite entrées dans une méthode *BestSolution()* qui renvoie la meilleure solution parmi toutes celles en entrée par rapport à la durée de la simulation.

Cette solution est déclinée en 3 types distincts :

- Speedrun efficace : qui renvoie la solution optimale en fonction de la durée de simulation.

- Speedrun 100% : qui renvoie à la solution la plus optimale mais qui termine toutes les quêtes.

- Speedrun déplacement minimaux : à la manière du speedrun efficace, on renvoie la solution la plus optimale en fonction du nombre de déplacements effectués.

IV. Tests

Pour les tests nous avons fait face à un problème dès le début. En effet, depuis le début de notre apprentissage sur les tests, on avait appris à faire des tests pour une classe et on était seul dans le projet. Mais cette fois-ci nous avions plusieurs classes à tester et le responsable de ses tests changeait souvent, nous avons donc séparé le dossier de test par rapport aux classes et donc fait un sommaire pour que le dossier soit plus propre et plus ordonné.

Lors de l'écriture du programme, on s'est rendu compte aussi qu'on avait oublié certaines classes à tester lors de la gestion de projet.

Le dossier de test est disponible dans l'archive.

V. IHM

Nous avons décidé de commencer plus tôt l'IHM puisqu'on n'avait pas assez bien optimiser notre temps. Pour l'IHM, on a essayé de faire une interface claire et simple. Notre interface se divise en 3 parties :

- La première est une barre de menu où l'utilisateur peut quitter l'application, ouvrir ses dernières sauvegardes ou avoir les informations de l'application.
- La deuxième est un formulaire où l'utilisateur choisi ce qu'il veut faire comme simulations
- Et la troisième est un tableau où l'utilisateur peut retrouver tous les détails des simulations et les résultats. Il peut aussi enregistrer les résultats.

On a ajouté la possibilité de sauvegarder ses résultats pour les regarder plus tard à nouveau. On a aussi mis un bouton pour récupérer les informations de l'application pour qu'elle soit plus réaliste et que l'utilisateur puisse toujours retrouver le GitHub ou encore les auteurs de l'application.

Les éléments du tableau de gauche, qui permet de choisir les simulations à réaliser sont composées d'objets d'une sous-classe *Simulation* qui comporte le scénario et le type. C'est ensuite cet objet qui est envoyé au *controller* pour effectuer l'algorithme concerné.

VI. Points à améliorer

Pour le prochain projet, on développera plus la gestion de projet du côté de l'IHM.

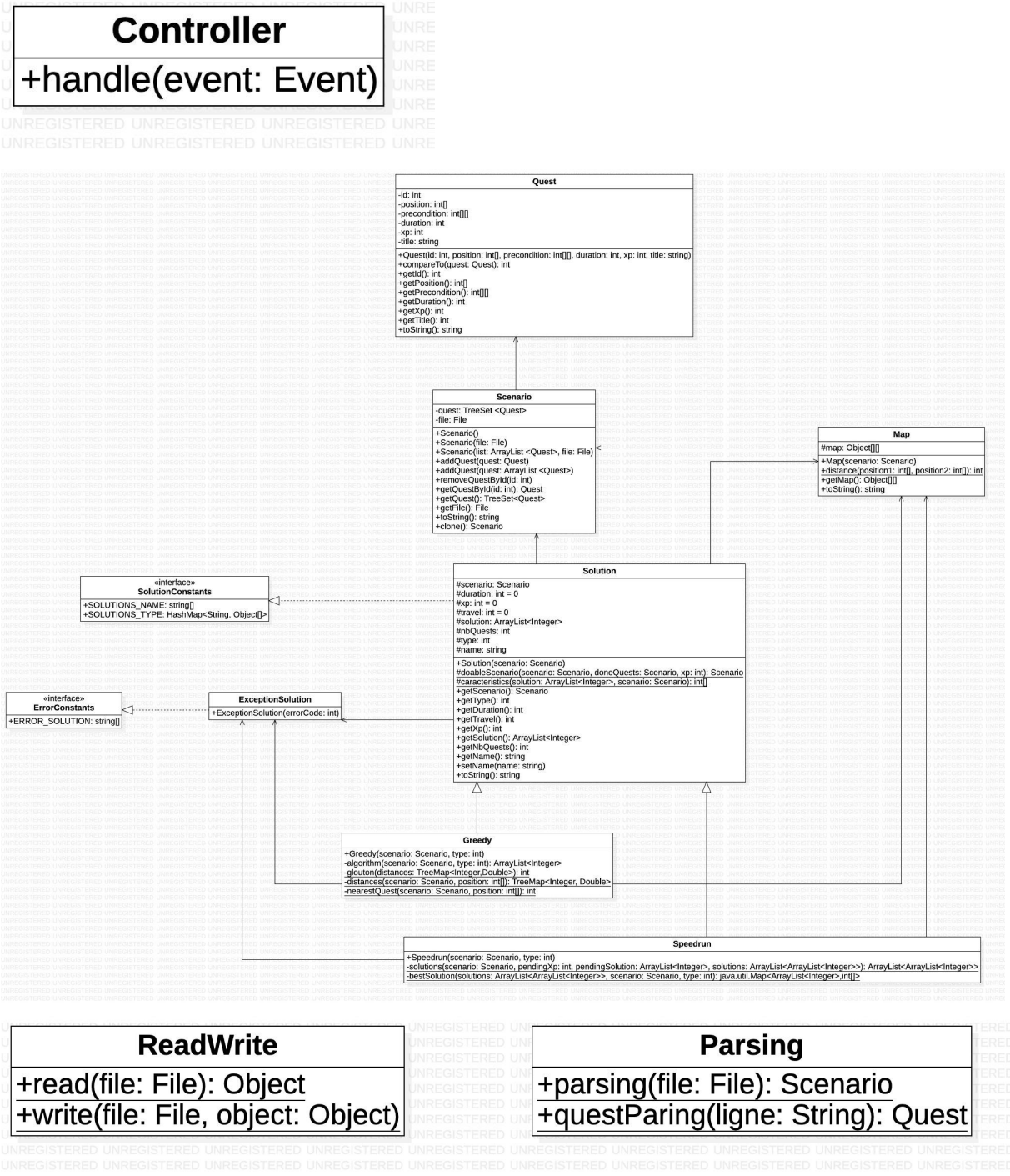
Aussi pour améliorer notre productivité et éviter d'avoir des problèmes avec Git, nous utiliserons les outils de 'Pull Request' et les 'Issues' de GitHub. L'outil 'Pull Request' est un outil qui nous permet de protéger notre branche 'main' et donc de pouvoir merge nos branches qu'après avoir consulté les différences entre les deux branches sur GitHub. L'outil 'Issue' nous permet d'écrire les bugs rencontrés et les fonctionnalités qu'on veut mettre, chaque bugs et fonctionnalités sont reliés à une branche. Donc grâce aux 'Issues', on peut avoir une liste des tâches à faire sur le projet.

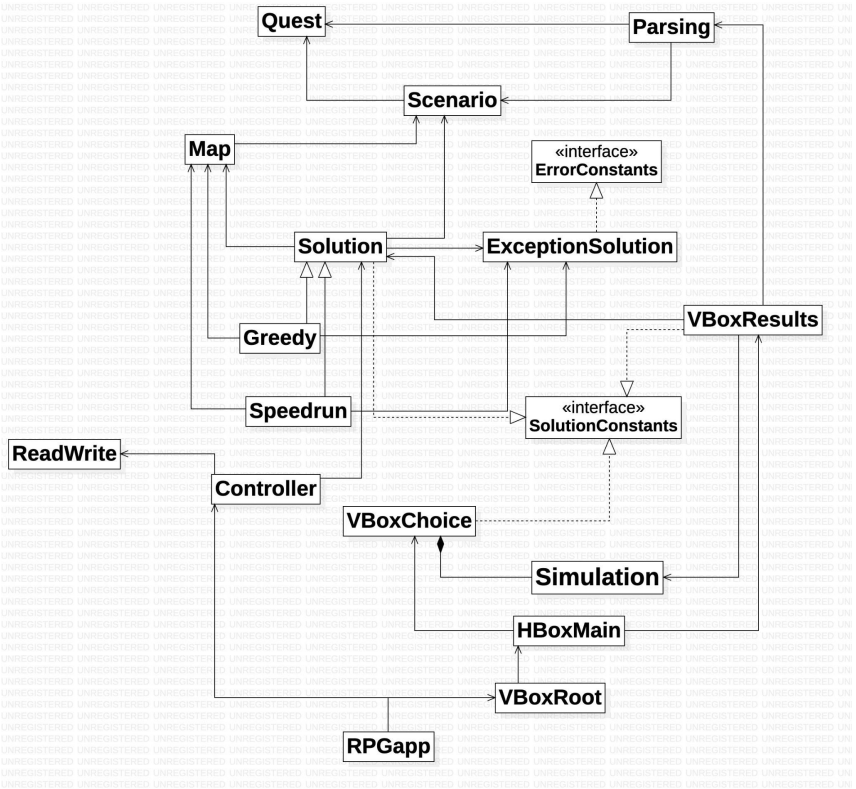
On peut aussi améliorer la création de notre projet sur IntelliJ. En effet, il aurait été mieux de faire directement un projet JavaFX avec JUnit dedans.

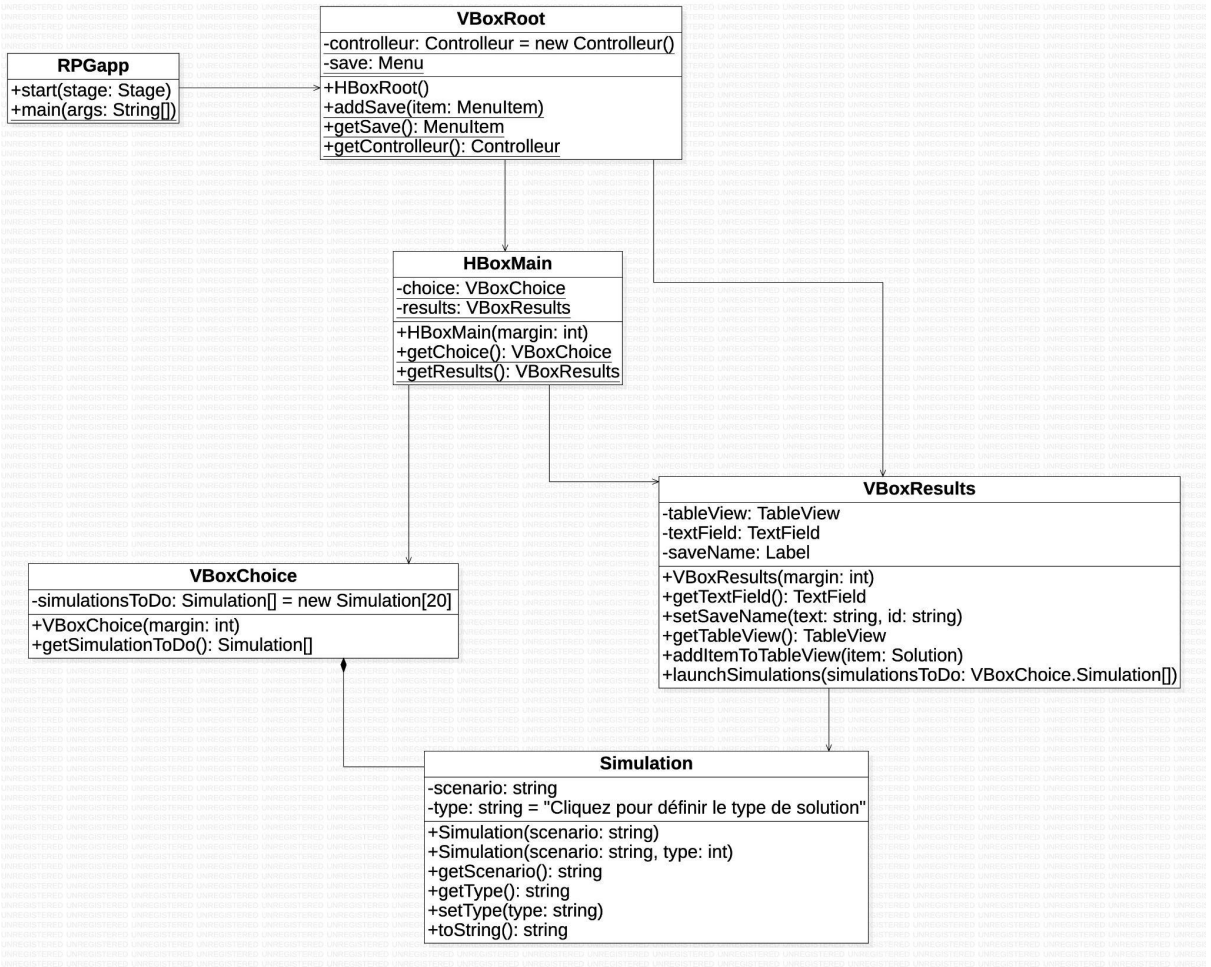
Pour l'algorithme de speedrun, il aurait été préférable lors de l'algorithme récursif de comparer directement les solutions entre elles, au lieu de toutes les garder en mémoire.
C'est notamment problématique lors de l'exécution de ces solutions à partir du scénario 8 qui sont relativement longues, et pour le scénario 10 qui demande trop de mémoire vive à la machine.

VII. Diagramme de classe

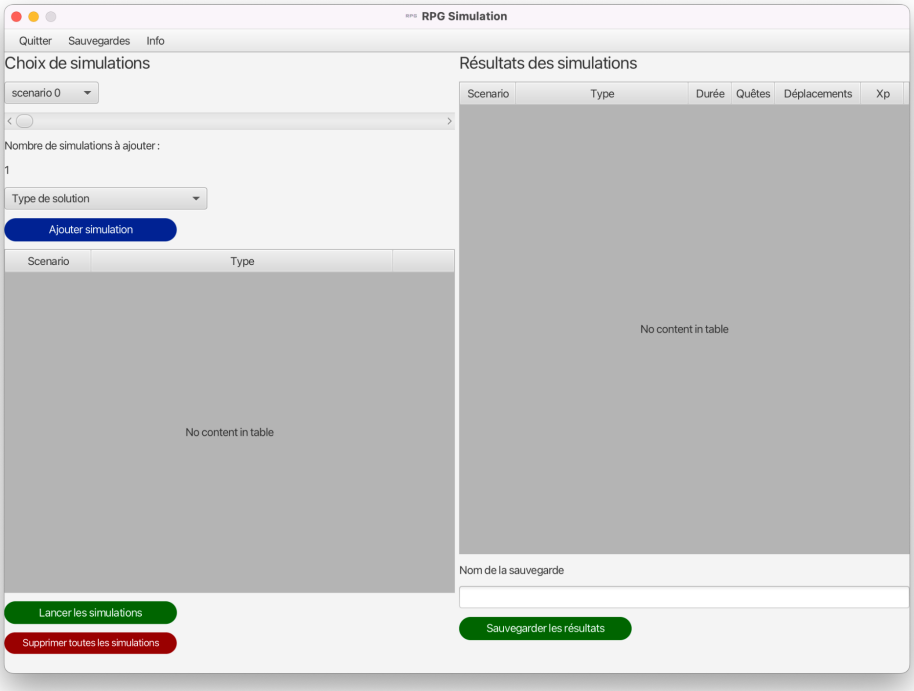
Le diagramme de classe est disponible dans notre archive en format HTML. Le voici aussi ci-dessous au format JPG :

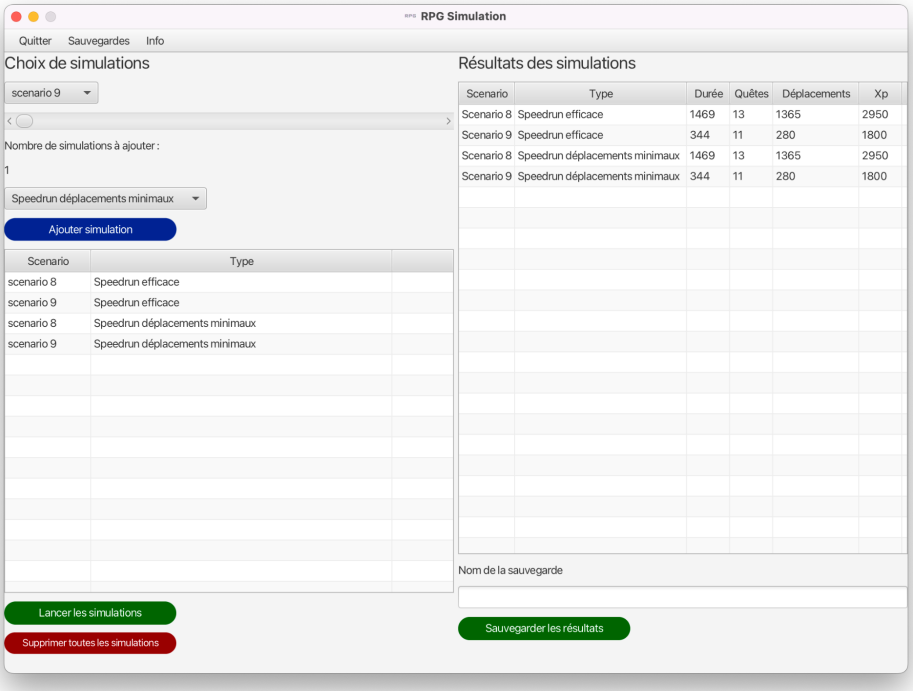



Quest
Parsing
Scenario
Map
«interface»
ErrorConstants
Solution
ExceptionSolution
VBoxResults
Greedy
Speedrun
«interface»
SolutionConstants
ReadWrite
Controller
VBoxChoice
Simulation
HBoxMain
VBoxRoot
RPGapp


RPGapp
+start(stage: Stage)
+main(args: String[])
VBoxRoot
-controlleur: Controlleur = new Controlleur()
-save: Menu
+HBoxRoot()
+addSave(item: MenuItem)
+getSave(): MenuItem
+getControlleur(): Controlleur
HBoxMain
-choice: VBoxChoice
-results: VBoxResults
+HBoxMain(margin: int)
+getChoice(): VBoxChoice
+getResults(): VBoxResults
VBoxResults
-tableView: TableView
-textField: TextField
-saveName: Label
+VBoxResults(margin: int)
+getTextField(): TextField
+setSaveName(text: string, id: string)
+getTableView(): TableView
+addItemToTableView(item: Solution)
+launchSimulations(simulationsToDo: VBoxChoice.Simulation[])
VBoxChoice
-simulationsToDo: Simulation[] = new Simulation[20]
+VBoxChoice(margin: int)
+getSimulationToDo(): Simulation[]
Simulation
-scenario: string
-type: string = "Cliquez pour définir le type de solution"
+Simulation(scenario: string)
+Simulation(scenario: string, type: int)
+getScenario(): string
+getType(): string
+setType(type: string)
+toString(): string

VIII. Captures de l’application





IX. Annexe Git

Notre dépôt Git est disponible à cette adresse : https://github.com/Chakib-Eliott/SAe-2/