Compte rendu SAE 2.02

Introduction

Ce projet vise à explorer algorithmiquement un problème lié à la durée de vie d'un jeu vidéo de type jeu de rôle (RPG) en monde ouvert. Pour cela, un modèle simplifié du jeu a été créé, représenté par une carte rectangulaire en deux dimensions avec des lieux identifiés par des coordonnées (x, y).

Le fonctionnement du jeu simplifié implique des déplacements horizontaux et verticaux sur la carte, avec chaque déplacement prenant une unité de temps. Les déplacements entre deux lieux se font par le chemin le plus court. Le jeu comprend plusieurs types de quêtes : explorations, combats et dialogues. Dans la modélisation, ces types de quêtes sont équivalents et regroupés sous l'appellation "quête".

Chaque quête est définie par un numéro, une position sur la carte, une précondition, une durée et une expérience. Les préconditions indiquent les quêtes préalables à accomplir pour débloquer la quête en question. La quête finale possède une interprétation légèrement différente, nécessitant à la fois de satisfaire sa précondition et d'avoir l'expérience nécessaire.

L'objectif du jeu est d'accumuler de l'expérience en réalisant des quêtes, puis de compléter la quête finale une fois qu'elle est disponible et que le personnage a l'expérience requise. Une solution du jeu est une séquence de quêtes respectant les préconditions et aboutissant à la quête finale. La durée d'une solution est la somme des durées des quêtes et des déplacements effectués.

Pour ce projet, nous devons réaliser des algorithmes capables de générer des solutions efficaces, qui font la quête finale dès qu'elle est disponible, ou exhaustives, qui font toutes les quêtes en terminant par la quête finale. Ils doivent aussi choisir entre une solution gloutonne ou optimale, et si elle est optimale, choisir entre une optimisation en terme de durée, de nombre de quêtes effectuées et de déplacements. Enfin, chaque algorithme doit être capable de générer les meilleures ou les pires solutions.

En termes de qualité, le code doit être clair, simple et bien commenté. Les algorithmes doivent être expliqués en détail, ainsi que les éventuelles optimisations réalisées. Le programme devrait permettre d'afficher les solutions de manière lisible et de les visualiser, tout en offrant des fonctionnalités supplémentaires telles que l'optimisation du nombre de points d'expérience ou le calcul de la durée moyenne.

Qualité de développement

Organisation du travail

Pour ce projet, nous avons choisi un cycle de vie en V récursif. C'est à dire que, contrairement à un cycle en V classique où on écrit la totalité des tests, puis la totalité du code, avant de tout tester, nous y allons classe par classe en écrivant le test de la classe, en codant la classe, puis en testant la classe. On répète ensuite l'opération pour chaque classe et/ou méthode que l'on ajoute au code.

Pour le module intitulé "modèle", notre binôme a travaillé en coopération afin de définir une base stable qui nous permettrait de correctement coder les modules suivants. Une fois ce module et ses tests rédigés, nous avons codé un premier moteur de génération de solution utilisant la méthode gloutonne. Loan s'est ensuite chargé du reste du module core, qui contient tous les algorithmes générant des solutions, ainsi que de ses tests. Thomas, lui, s'est chargé de toute la partie IHM, codant les modules vue et contrôleur.

Outils utilisés

Pour ce projet, nous avons utilisé deux librairies. La librairie Junit nous a permis de tester le modèle et les moteurs de notre projet, et la librairie Java-FX nous a permis de coder une interface graphique pour le projet. Nous avons utilisé Github pour versionner le projet, dont le lien est disponible en annexe, et nous avons utilisé l'IDE Intellij pour travailler sur le projet. Enfin, nous avons utilisé Google Docs pour rédiger ce compte rendu et le dossier de tests, ainsi que StarUML pour construire l'UML du projet.

Conception générale

Diagramme de classes de haut niveau

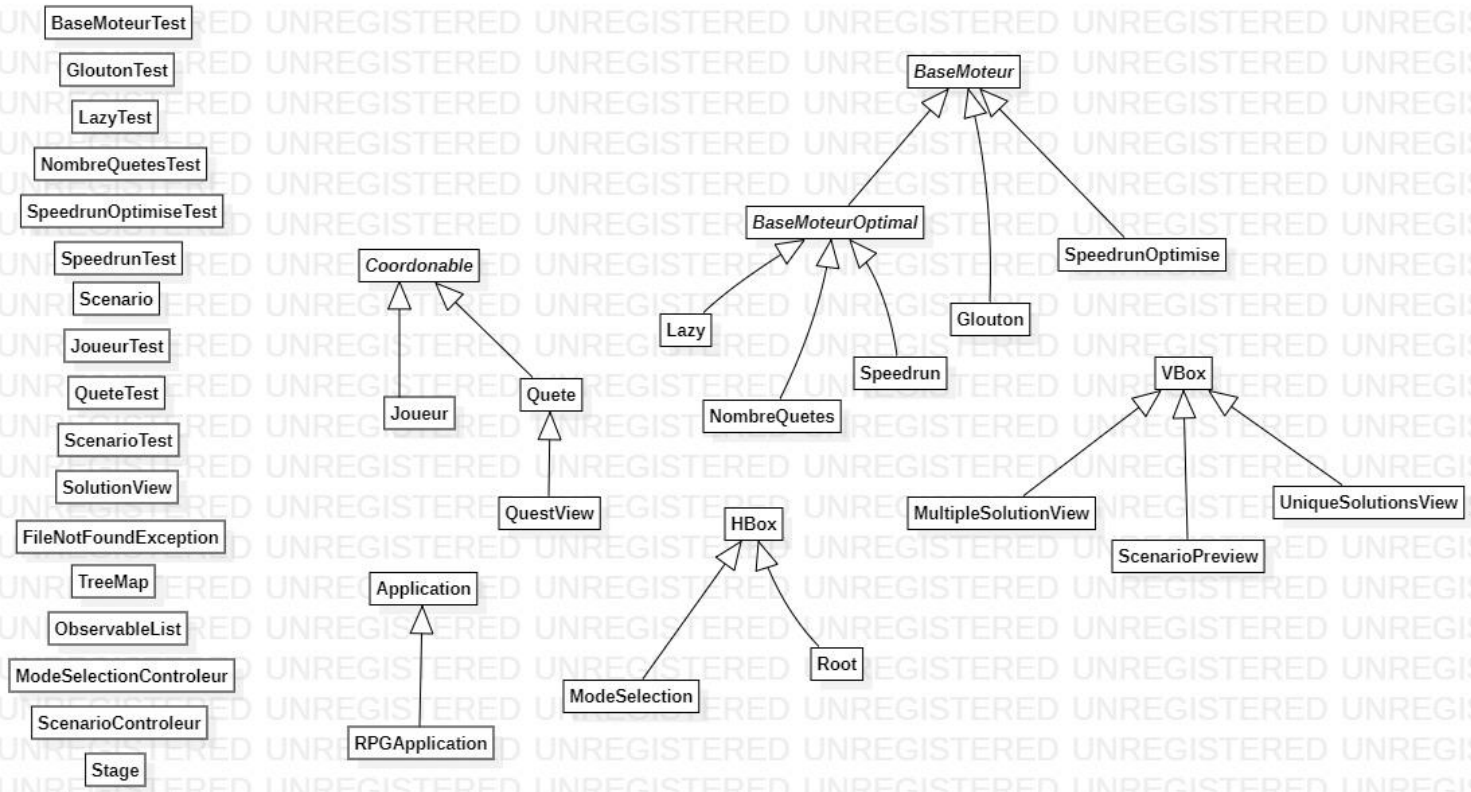
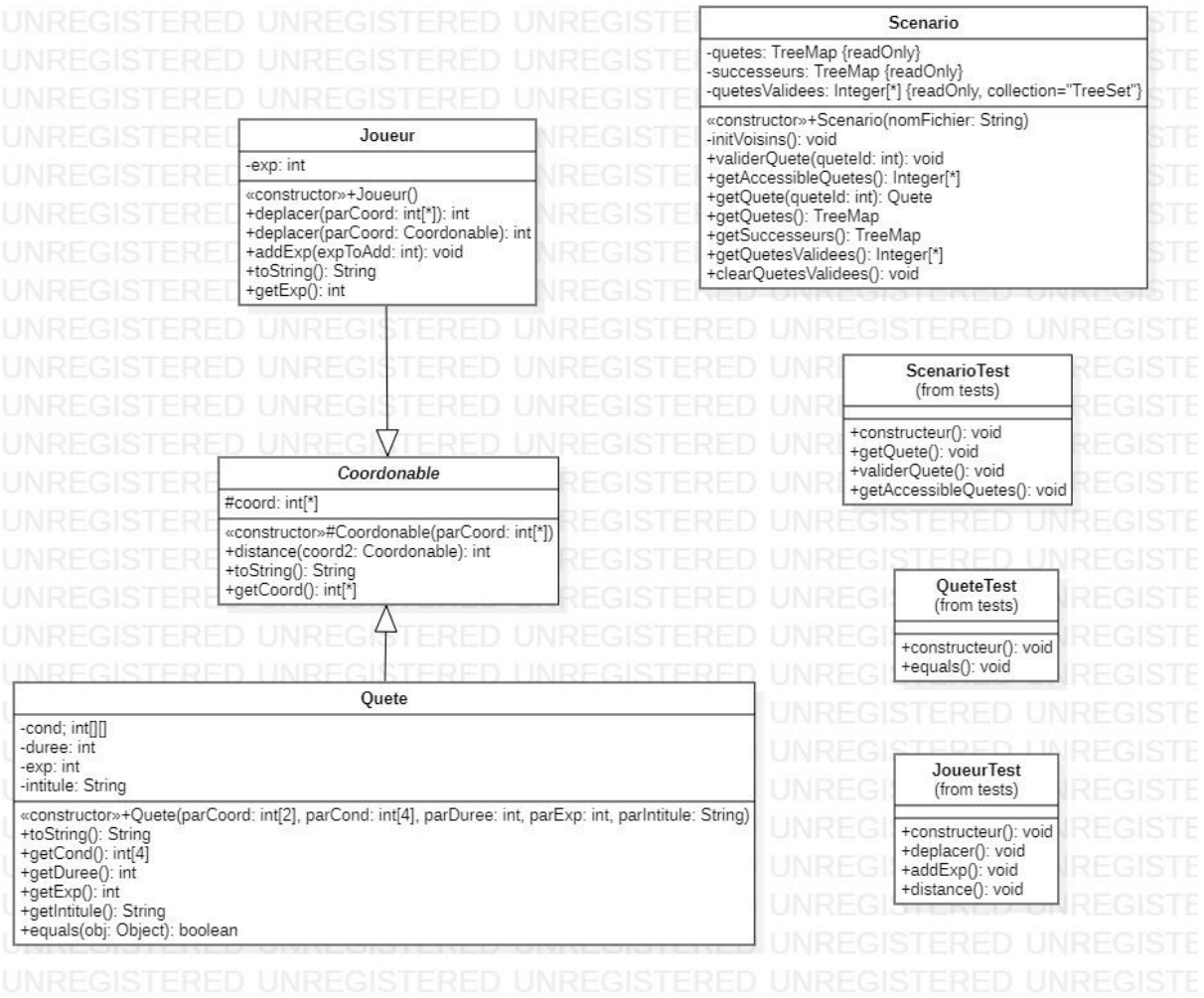


Diagramme des classes détaillé pour modèle et core



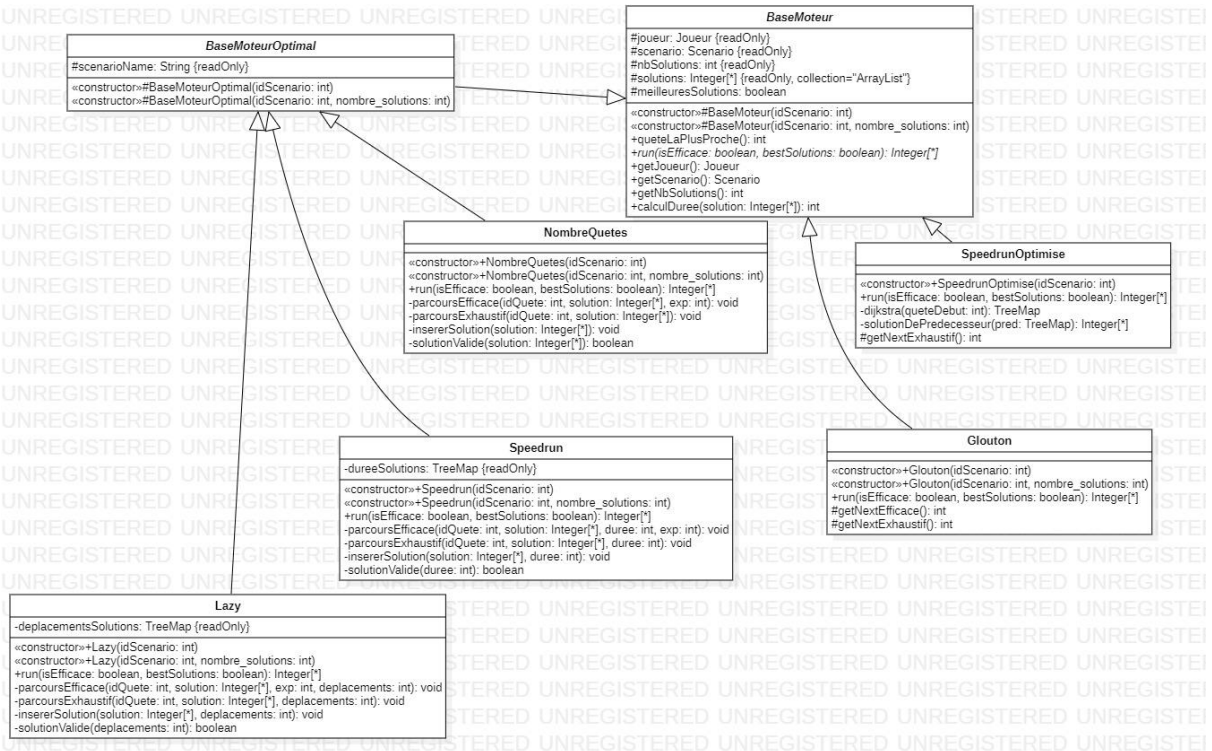

BaseMoteurOptimal
#scenarioName: String {readOnly}
«constructor»#BaseMoteurOptimal(idScenario: int)
«constructor»#BaseMoteurOptimal(idScenario: int, nombre_solutions: int)
BaseMoteur
#joueur: Joueur {readOnly}
#scenario: Scenario {readOnly}
#nbSolutions: int {readOnly}
#solutions: Integer[*] {readOnly, collection="ArrayList"}
#meilleuresSolutions: boolean
«constructor»#BaseMoteur(idScenario: int)
«constructor»#BaseMoteur(idScenario: int, nombre_solutions: int)
+queteLaPlusProche(): int
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
+getJoueur(): Joueur
+getScenario(): Scenario
+getNbSolutions(): int
+calculDuree(solution: Integer[*]): int
NombreQuetes
«constructor»+NombreQuetes(idScenario: int)
«constructor»+NombreQuetes(idScenario: int, nombre_solutions: int)
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
-parcoursEfficace(idQuete: int, solution: Integer[*], exp: int): void
-parcoursExhaustif(idQuete: int, solution: Integer[*]): void
-insererSolution(solution: Integer[*]): void
-solutionValide(solution: Integer[*]): boolean
SpeedrunOptimise
«constructor»+SpeedrunOptimise(idScenario: int)
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
-dijkstra(queteDebut: int): TreeMap
-solutionDePredecesseur(pred: TreeMap): Integer[*]
#getNextExhaustif(): int
Speedrun
-dureeSolutions: TreeMap {readOnly}
«constructor»+Speedrun(idScenario: int)
«constructor»+Speedrun(idScenario: int, nombre_solutions: int)
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
-parcoursEfficace(idQuete: int, solution: Integer[*], duree: int, exp: int): void
-parcoursExhaustif(idQuete: int, solution: Integer[*], duree: int): void
-insererSolution(solution: Integer[*], duree: int): void
-solutionValide(duree: int): boolean
Glouton
«constructor»+Glouton(idScenario: int)
«constructor»+Glouton(idScenario: int, nombre_solutions: int)
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
#getNextEfficace(): int
#getNextExhaustif(): int
Lazy
-deplacementsSolutions: TreeMap {readOnly}
«constructor»+Lazy(idScenario: int)
«constructor»+Lazy(idScenario: int, nombre_solutions: int)
+run(isEfficace: boolean, bestSolutions: boolean): Integer[*]
-parcoursEfficace(idQuete: int, solution: Integer[*], exp: int, deplacements: int): void
-parcoursExhaustif(idQuete: int, solution: Integer[*], deplacements: int): void
-insererSolution(solution: Integer[*], deplacements: int): void
-solutionValide(deplacements: int): boolean

Diagramme des classes détaillé pour vue

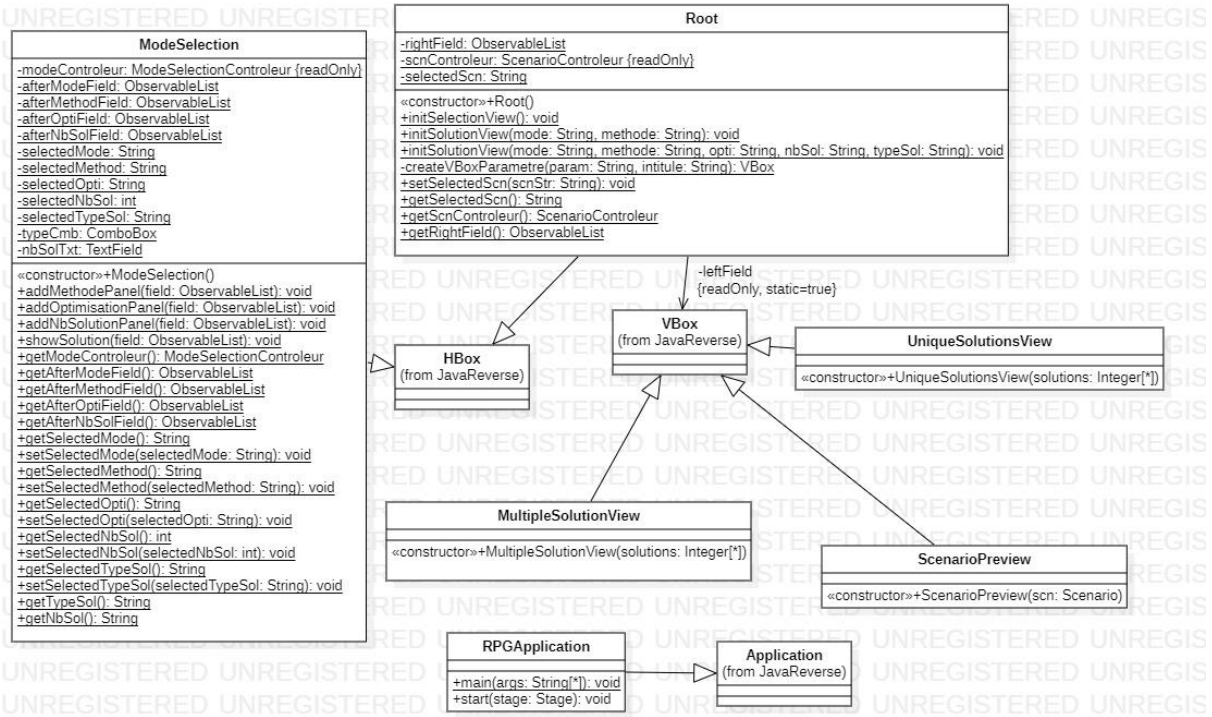
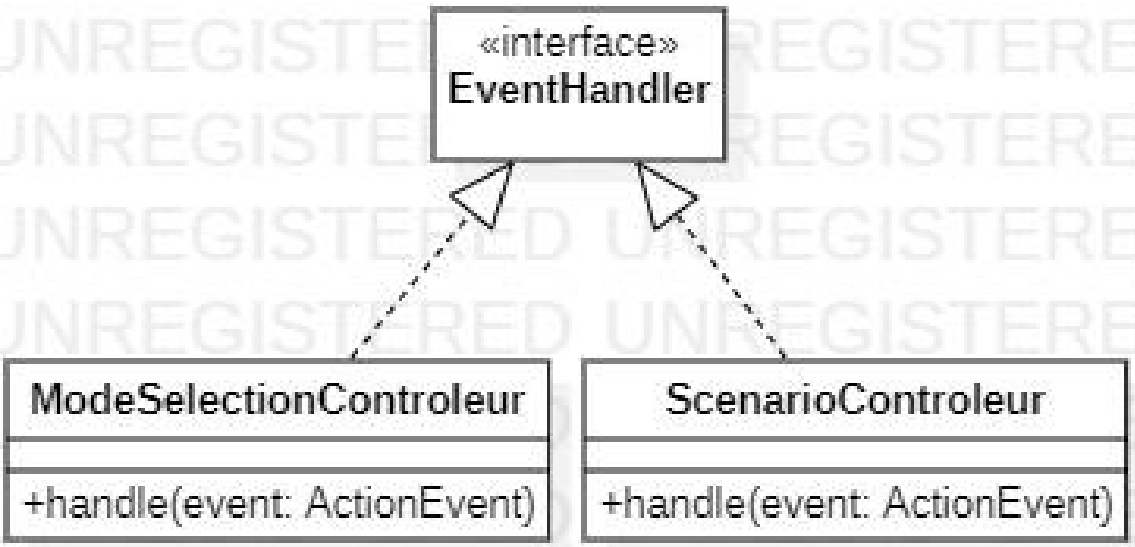


Diagramme des classes détaillé pour contrôleur



Présentation du programme

Notre projet contient quatres composants :

- **modèle** : Ce composant contient toutes les classes permettant de représenter le modèle simplifié du jeu. Le joueur, les quêtes, les scénarios etc sont contenues à l'intérieur. Un package tests est également contenu à l'intérieur, contenant tous les tests des classes du modèle.
- **core** : Ce composant contient tous les moteurs de génération de solutions. C'est eux qui utilisent des algorithmes pour générer des solutions. Ils utilisent les classes du composant modèle pour cela, et il contient également un package tests pour chacun des moteurs.
- **vue :** Ce composant contient toutes les classes permettant de générer l'interface graphique servant de visualisation du projet. Il ne contient pas de tests.
- **contrôleur** : Ce composant contient toutes les classes permettant de faire le lien entre modèle/core et vue. C'est ces classes qui permettent de faire la passerelle entre les intéractions de l'utilisateur et le code, et qui demande à la vue d'afficher les solutions générées.

Le moteur Glouton étant très simpliste, il ne permet de générer qu'une solution, et ne nécessite pas d'algorithme de recherche complexe. Les moteurs optimisés en revanche ont nécessité l'implémentation d'algorithmes. Ils fonctionnent tous de la même façon. En partant du point de départ du joueur, nous parcourons chacune des quêtes disponibles. Pour chacune d'entre elles, nous faisons un appel récursif où on boucle à nouveau sur chaque quête disponible après avoir validé celle-ci.

À chaque fois qu'une itération tombe sur la quête finale, elle l'ajoute à une liste de solutions. Cette liste est triée en fonction des meilleures ou des pires solutions selon un critère donné (ce critère change selon la classe utilisée). Lorsque la liste des solutions est pleine, nous terminons de parcourir le scénario, en remplaçant une des solutions existantes si une meilleure est trouvée.

Enfin, afin d'optimiser la charge mémoire et les temps de recherche, lorsque la liste de solutions est déjà pleine, si la solution que nous générons est déjà pire que la pire solution de la liste, nous abandonnons la recherche de cette solution entièrement.

Conclusion

Par faute de temps, nous n'avons pas pu implémenter d'algorithme plus efficace que celui décrit précédemment. Des restes d'une tentative d'implémenter un algorithme de dijkstra fonctionnel sont présents dans la classe SpeedrunOptimise, mais nous n'avons pas pu l'adapter afin de générer plusieurs solutions, il est uniquement capable de générer la meilleure (et encore, il génère parfois des solutions qui n'ont pas assez d'expérience pour faire la quête finale).

En dehors de cette défaite pour ce qui est d'optimiser les algorithmes, nous sommes néanmoins satisfaits du projet global. Les trois niveaux ont été réalisés avec succès, et l'interface graphique est claire, précise et agréable d'utilisation. Nous sommes également fiers de notre structure, et de notre première utilisation de Github. Nous aurions cependant pu mieux faire en ce qui concerne les tests, n'ayant pas toujours eu le temps de faire des partitions d'équivalence correctes pour chaque test.

Néanmoins, les attendus du sujet d'algorithme et d'IHM sont respectés, et nous pensons que ce projet est un succès à ce niveau. C'est ce que nous essayons d'accomplir.

Annexes

Lien du dépôt Git :
https://github.com/Blakeline-was-taken/SAE_2_02.git

Lien du dossier de tests unitaires : présent dans le dépôt Git