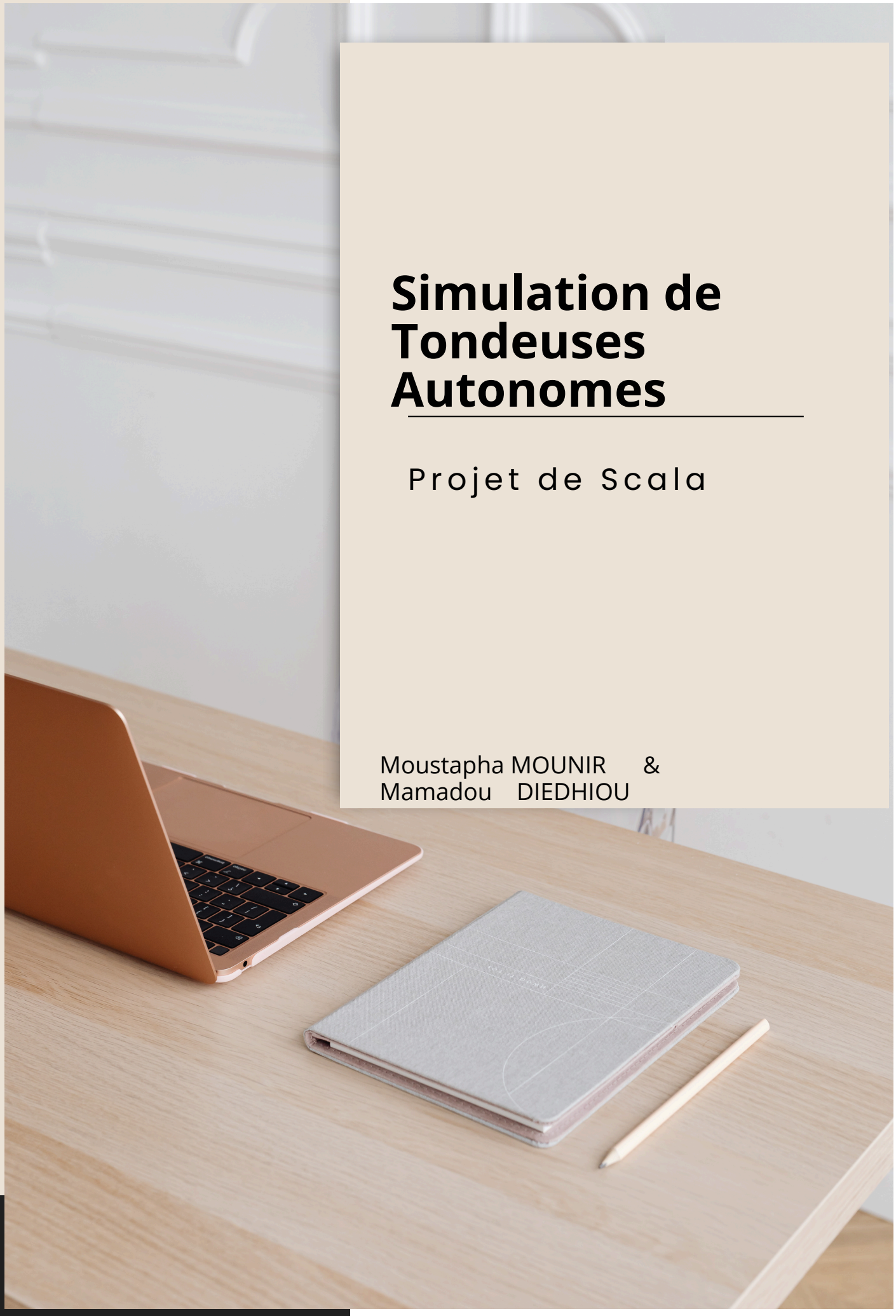
Simulation de Tondeuses Autonomes

Projet de Scala

Moustapha MOUNIR &
Mamadou DIEDHIOU



Introduction

Ce projet porte sur la simulation d'une tondeuse autonome capable de se déplacer sur une pelouse rectangulaire en suivant des instructions fournies via un fichier. L'objectif principal est de concevoir une solution algorithmique capable de lire les données d'entrée, de simuler les déplacements des tondeuses tout en respectant les contraintes (bordures de la pelouse), et de fournir leurs positions finales. Ce type de simulation illustre l'importance des algorithmes dans le développement de systèmes autonomes et programmables.

Dans ce rapport, nous détaillerons d'abord la conception de la solution, en expliquant la logique utilisée pour traiter les commandes et modéliser les déplacements. Ensuite, nous décrirons la structure du code, en mettant en avant les différentes classes et fonctions. Enfin, nous présenterons les choix techniques qui ont guidé le projet, comme l'utilisation de concepts avancés de Scala tels que le pattern matching et les structures immuables, ainsi que la gestion des erreurs.

Sommaire :

Dans cette présentation, vous trouverez les parties suivantes :

Conception de la solution

2.1. Description générale de la simulation

La simulation consiste à modéliser le déplacement de tondeuses autonomes sur une pelouse rectangulaire en **suivant des instructions prédéfinies**. Chaque tondeuse dispose d'une position **initiale** et **d'une orientation (Nord, Est, Sud, Ouest)**, ainsi que d'une série de commandes pour avancer ou pivoter. L'objectif est de **simuler** ces mouvements tout en respectant les limites de la pelouse, afin de déterminer la position finale et l'orientation de chaque tondeuse.

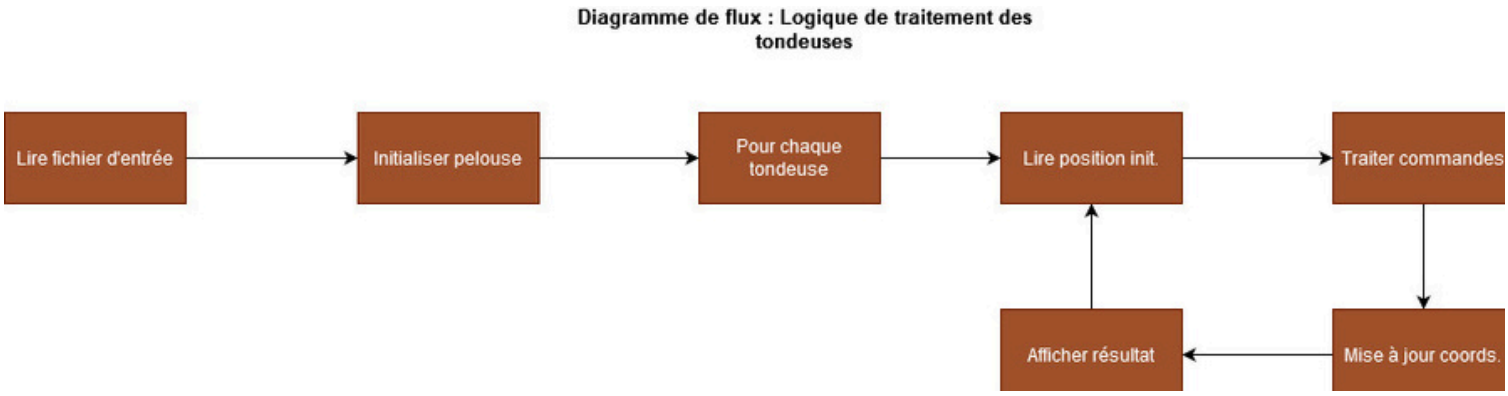
Notre programme fonctionne de manière séquentielle. Il lit un fichier d'entrée structuré contenant les dimensions de la pelouse, les positions initiales des tondeuses, et leurs instructions respectives. Chaque tondeuse est traitée indépendamment : ses commandes sont exécutées dans l'ordre, sans interférer avec les autres. Une fois les instructions terminées, la position finale et l'orientation de la tondeuse sont enregistrées et affichées

2.2. Description générale de la simulation

La pelouse est modélisée comme **une grille rectangulaire** définie par **deux coordonnées : (0, 0)** pour le coin inférieur gauche (implicite) et (maxX, maxY) pour le coin supérieur droit, qui fixent les limites au-delà desquelles les tondeuses ne peuvent pas se déplacer. Chaque tondeuse est caractérisée par ses coordonnées actuelles (x, y) et son **orientation (N, E, S, W)**. Elle peut effectuer trois actions principales : **avancer (A), pivoter à gauche (G) ou pivoter à droite (D)**, tout en respectant les limites de la pelouse. Ce modèle sépare clairement les rôles : la pelouse agit comme un objet passif définissant les contraintes, tandis que les tondeuses sont des objets actifs, capables de modifier leur état en fonction des instructions reçues

2.3. Algorithme pour le traitement des commandes et illustration de la logique.

L’algorithme débute par la lecture des dimensions de la pelouse et des données initiales de chaque tondeuse à partir du fichier d’entrée. Chaque tondeuse est initialisée avec ses coordonnées et son orientation initiale. Ensuite, le programme traite séquentiellement les instructions associées à chaque tondeuse. Ces commandes, représentées par les caractères D, G et A, sont interprétées via un pattern matching : D fait pivoter la tondeuse de 90° vers la droite, G de 90° vers la gauche, et A la fait avancer d’une case dans la direction actuelle, à condition de rester dans les limites de la pelouse. Les coordonnées et l’orientation sont mises à jour après chaque commande, et les résultats finaux sont affichés.



3. Structure du code

3.1. Organisation générale

Le projet est structuré autour de l'objet principal **MowerProject,** qui contient la méthode **main**, le point d'entrée de notre programme. Cet objet se charge de **lire le fichier d'entrée, d'initialiser les données, et de coordonner la simulation des tondeuses.** Nous avons également la **classe Mower,** qui représente chaque tondeuse avec ses coordonnées et son orientation, et qui encapsule ses actions possibles : **avancer, pivoter à gauche ou à droite.** Enfin, des méthodes utilitaires comme **processMower**, **turnLeft ou moveForward** permettent de gérer les commandes et de transformer l'état des tondeuses. Le projet suit une logique claire en quatre étapes : nous lisons d'abord les dimensions de la pelouse, puis nous initialisons les tondeuses avec leurs données de départ. Ensuite, nous traitons les commandes de chaque tondeuse de manière séquentielle, et enfin, nous affichons leurs positions finales. Cette organisation nous aide à maintenir un code lisible et modulaire.

3.2. Détails des classes et des fonctions

- **Classe Mower:**

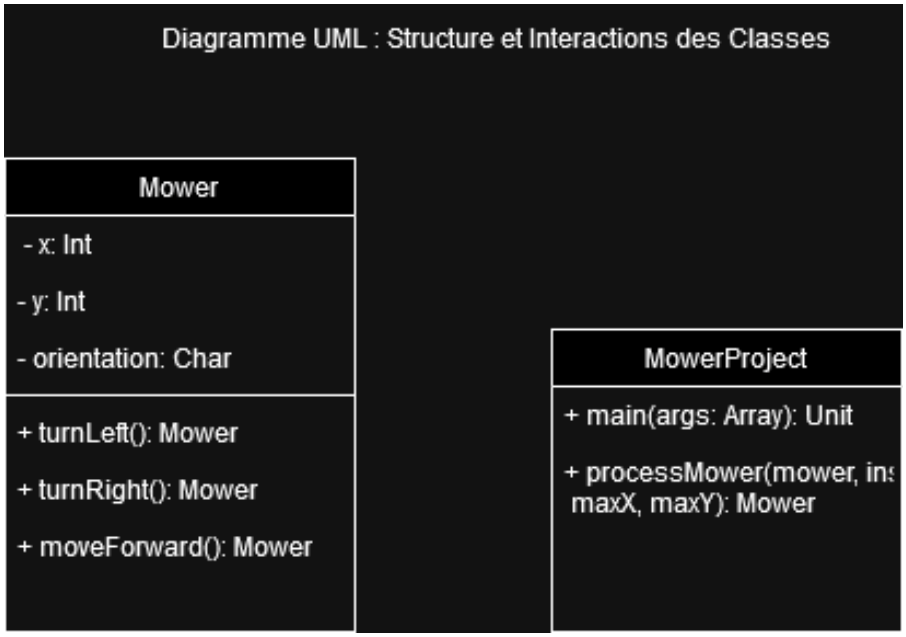
La classe Mower représente une tondeuse avec ses **coordonnées (x, y)** et son **orientation (N, E, S, W).** Elle inclut trois méthodes principales :

- **turnLeft** : Change l'orientation dans le sens anti-horaire (N → W → S → E → N).
- **turnRight** : Change l'orientation dans le sens horaire (N → E → S → W → N).
- **moveForward** : Avance d'une case si la nouvelle position reste dans les limites.

Méthodes principales

- **processMower** : Exécute les commandes (D, G, A) pour une tondeuse, en appliquant les actions correspondantes et en respectant les limites de la pelouse. Elle retourne l'état final de la tondeuse.
- **turnLeft et turnRight :** Modifient l'orientation.
- **moveForward** : Met à jour les coordonnées si possible, sinon la tondeuse reste immobile.

3.3. Schéma UML de notre projet



4. Choix techniques

4.1. Utilisation des fonctionnalités de Scala

Nous avons tiré parti de deux fonctionnalités clés du langage scala :

- **Pattern Matching :** Nous avons utilisé le pattern matching pour gérer les commandes des tondeuses (D, G, A). Cela nous a permis de simplifier la logique de traitement en associant chaque commande à une action précise, comme tourner ou avancer. Par exemple, la méthode processMower applique directement l'action appropriée grâce à ce mécanisme, rendant le code à la fois intuitif et maintenable.

- **Gestion des erreurs avec Option** : Pour éviter que notre programme plante face à des entrées invalides, nous avons adopté l'approche sécurisée de **Option**. Par exemple, la méthode **safeParseInt** nous a permis **de vérifier la validité des conversions numériques** avant de les utiliser, réduisant ainsi le risque d'erreurs inattendues. Cette gestion des erreurs a renforcé la **robustesse** de notre programme

4.2. Séparation des responsabilités et modularité

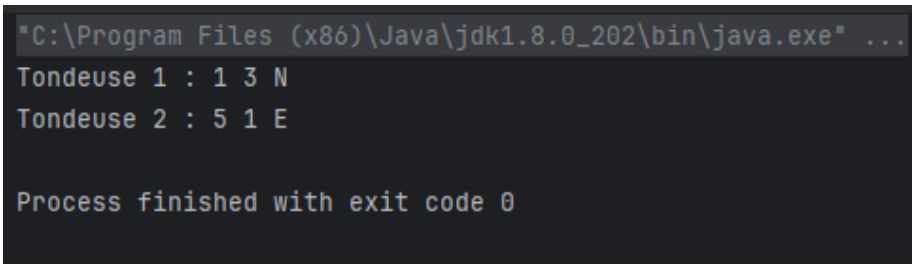
Nous avons structuré notre code pour garantir une séparation claire des responsabilités, ce qui facilite la maintenance et l'évolutivité. La classe Mower se concentre exclusivement sur les propriétés et comportements des tondeuses, encapsulant des actions comme turnLeft, turnRight, et moveForward. De son côté, l'objet MowerProject gère la logique globale, notamment la lecture des données, le traitement des commandes et l'affichage des résultats. En adoptant cette approche modulaire, nous avons pu isoler les comportements spécifiques des tondeuses de la gestion des données, rendant le programme plus intuitif et flexible. Cette organisation reflète notre volonté de produire un code propre et réutilisable

4.3. Améliorations et alternatives envisagées

Bien que notre solution réponde aux attentes du projet, nous avons identifié des pistes pour aller plus loin. Nous pourrions, par exemple, **remplacer Option par Either ou Try** pour fournir des messages d'erreur plus détaillés en cas d'entrées invalides, offrant ainsi une meilleure transparence. Nous avons également envisagé d'introduire des fonctionnalités supplémentaires, comme la gestion d'obstacles sur la pelouse ou des collisions entre tondeuses, ce qui enrichirait la simulation.
Enfin, pour rendre notre projet plus interactif, nous **pourrions intégrer une interface graphique** qui permettrait **de visualiser les mouvements des tondeuses en temps réel.** Une telle amélioration refléterait notre envie d'aller au-delà des exigences initiales et de relever de nouveaux défis techniques. Ces idées, bien qu'elles dépassent **le cadre actuel,** témoignent de notre engagement à exploiter pleinement les possibilités offertes par Scala et à concevoir un projet à la fois ambitieux et abouti.

4.4. Fichier de test

Afin de tester notre programme, nous avons utilisé le fichier d'entrée fourni dans l'énoncé du projet. En exécutant le programme avec le fichier d'entrée, nous avons le résultat ci-dessous qui correspond au résultat attendu.



Conclusion

Dans ce projet, nous avons **conçu** et **implémenté** une simulation robuste de tondeuses autonomes en utilisant Scala, tout en respectant les contraintes du problème. Grâce à une organisation claire et **modulaire**, nous avons pu **modéliser** efficacement les comportements des tondeuses et garantir un traitement fiable des commandes. Les choix techniques, comme **le pattern matching** et **la gestion des erreurs avec Option,** ont renforcé la lisibilité et la robustesse de notre solution. Ce projet nous a permis de développer nos compétences en programmation fonctionnelle et **orientée objet,** tout en ouvrant des perspectives d'amélioration et d'extension pour des versions futures plus ambitieuses.