Année 2024-2025

RAPPORT DE PROJET – Projet Qualité de programmation

Gestion des déplacements d'un robot dans un labyrinthe

Projet réalisé par :

BACHIRI Mehdi
FERHANI Youssef
HOFFSTETTER Anthony
MAKHLOUF Sanad

Projet encadré par :

Stéphane Rivière

Introduction

Contexte du projet

Dans le cadre de notre formation, nous avons entrepris un projet de Qualité de programmation visant à développer un programme permettant à un robot de sortir d'un labyrinthe. Ce projet repose sur l'implémentation de deux algorithmes différents pour résoudre ce problème : l'algorithme de la main droite et l'algorithme de Pledge.

L'objectif global est de mettre en pratique les compétences acquises tout au long du cours. Ces compétences incluent notamment l'utilisation de Git pour la gestion de version, la mise en œuvre de la programmation orientée objets en C++, l'écriture d'un code propre et évolutif, ainsi que l'adoption du développement dirigé par les tests. De plus, le projet avait pour but d'apprendre à réaliser des tests unitaires et à effectuer des remaniements de code pour garantir sa qualité et sa maintenabilité.

Chaque membre de l'équipe a participé activement à toutes les étapes du projet, réalisant ainsi un travail collaboratif et structurant.

Sommaire

1. Outils utilisés

Pour mener à bien ce projet, nous avons utilisé les outils suivants :

- **CodeBlocks** : CodeBlocks est un environnement de développement intégré (IDE) particulièrement adapté au langage C++. Il offre une interface conviviale et de nombreuses fonctionnalités, telles que la gestion des projets, l'intégration de compilateurs, et des outils de débogage. Grâce à son utilisation, nous avons pu développer et tester rapidement nos algorithmes tout en profitant d'une interface adaptée à la programmation orientée objets
  .
- **Git** : Git a été un outil essentiel pour la gestion des fichiers et la collaboration en équipe. En utilisant les bonnes commandes, chaque membre a pu contribuer efficacement au projet tout en gardant une trace des modifications apportées.

- **Doctest** : Doctest est une bibliothèque légère pour l'écriture de tests unitaires en C++. Elle nous a permis de valider le fonctionnement de nos classes en automatisant les tests. Cet outil nous a non seulement aidés à détecter les bugs à un stade précoce, mais il a également servi à garantir la robustesse de notre code au fur et à mesure de son évolution.

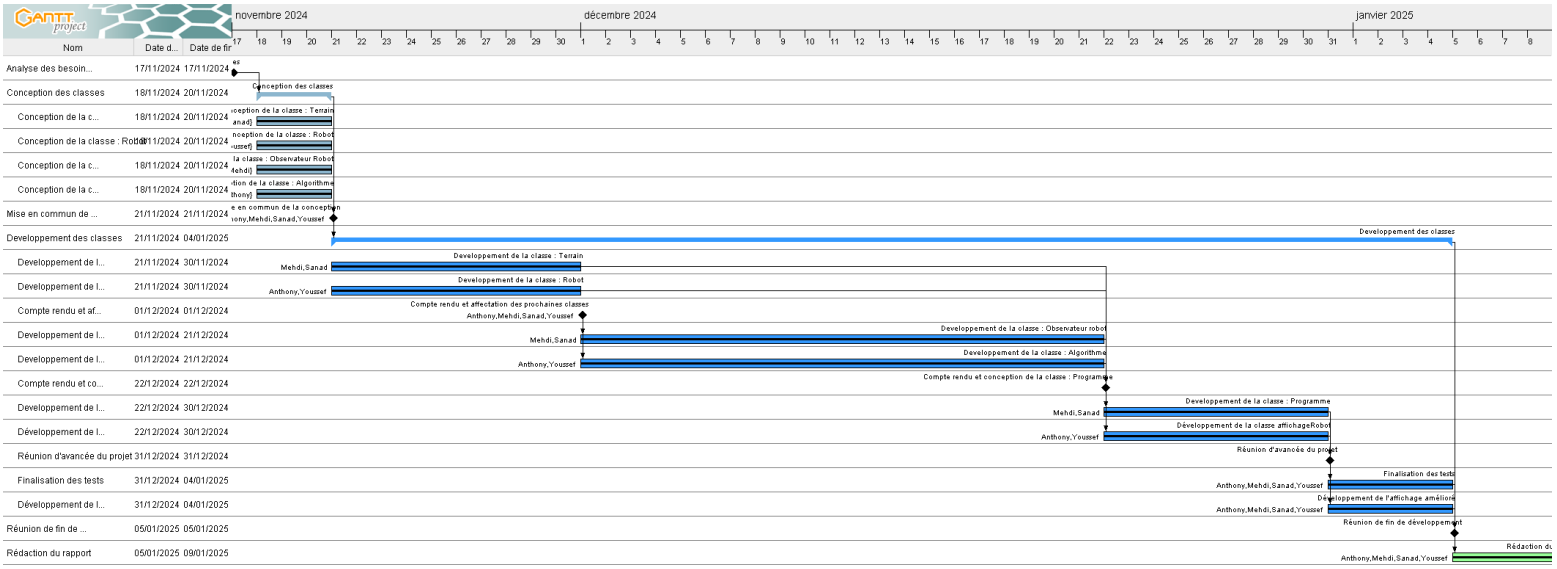
- **Discord** : En tant qu'outil de communication principal, Discord a été utilisé pour organiser des réunions d'équipe régulières. Grâce à ses fonctionnalités de partage d'écran, de canaux dédiés, et de messagerie instantanée, nous avons pu coordonner nos efforts et discuter des défis rencontrés en temps réel. Cet outil a joué un rôle clé dans la synchronisation de notre travail.

Chaque outil a été choisi pour répondre à des besoins spécifiques du projet, qu'il s'agisse de développement, de collaboration, ou de validation. Leur utilisation combinée nous a permis de respecter les exigences techniques et pédagogiques tout en améliorant la qualité de notre produit final.

2. Déroulement du projet

Calendrier

Le projet a été organisé selon un calendrier structuré pour respecter les délais impartis tout en garantissant une progression efficace. Voici notre diagramme de Gant :



Organisation de l'équipe

Le travail en équipe a été structuré de manière à exploiter les compétences et disponibilités de chaque membre. Voici comment nous avons collaboré :

- **Réunions hebdomadaires** : Tenues sur Discord pour faire le point sur les progrès et ajuster les tâches.
- **Répartition des tâches** : Chaque membre était responsable d'une partie spécifique du projet, nous avons souvent fait des groupes de deux pour avancer efficacement.
- **Code reviews** : Réalisées régulièrement lors des réunions pour garantir la qualité du code et apprendre des contributions des autres.

Organisation du travail

Tout au long du projet, nous avons suivi un diagramme de Gantt détaillé pour organiser notre travail de manière efficace et éviter tout retard. Nous avons débuté par une phase de conceptualisation rigoureuse, durant laquelle nous avons défini les objectifs, les classes nécessaires et leurs interactions. Cette étape préparatoire a ensuite été suivie par le développement des différentes parties du projet, accompagné de phases de tests régulières pour valider chaque étape et assurer une progression continue et maîtrisée. Voici comment s'est déroulé le projet :

- **Première réunion** : Cette réunion initiale nous a permis d'analyser les besoins du projet et de répartir les tâches. Nous avons défini les classes nécessaires : Robot, Terrain, Observateur, et Algorithme, ainsi que la classe Programme qui serait développée à la fin. Chaque membre a pris une semaine pour conceptualiser une classe, définissant ses méthodes et propriétés sans commencer le codage.

- **Mise en commun** : Après cette semaine, nous avons partagé nos conceptions et ajusté certains points. Cela a permis d'identifier les dépendances entre les classes et de standardiser les noms de paramètres et de fonctions.

- **Développement des classes** :

    - **Semaine 1 :** Mehdi et Sanad ont développé la classe Terrain, tandis qu'Anthony et Youssef se sont occupés de la classe Robot.

    - **Compte rendu et affectations des prochaines classes :** Après la première semaine de développement nous avons constaté l'avancement puis affecter les prochaines classes à développer.

    - **Semaine 2 et 3 :** Nous avons finalisé le développement des classes Terrain et Robot puis Mehdi et Sanad ont commencé à développer la classe Observateur, tandis qu'Anthony et Youssef se sont occupés de la classe Algorithme. Nous avons pris deux semaines car c'était une période où nous avions beaucoup d'examens, nous avons donc ralenti légèrement le travail sur le projet.

    - **Compte rendu et conception de la classe Programme :** Une fois le développement des classes principales fait, nous avons conceptualisé ensemble la classe Programme qui permet de finaliser le projet ainsi que l'observateur Afficheur Robot qui s'occupe de la mise à jour de l'affichage du robot dans le programme.

    - **Semaine 4 :** Mehdi et Sanad ont travaillé sur la classe Programme, tandis qu'Anthony et Youssef ont développé l'Afficheur Robot.

    - **Semaine 5 :** À la suite d'une réunion pour constater l'avancement du projet, nous avons lors de la dernière semaine de développement, développer l'affichage amélioré et nous avons également finalisé les tests unitaires.

- **Fin de développement et rapport** : une fois le développement terminé, nous avons réparti le travail pour la rédaction du rapport puis nous avons encore effectué du remaniement de code chacun de son côté avec le temps restant.

Cette organisation nous a permis de progresser efficacement tout en maintenant un haut niveau de collaboration et de coordination.

3. Description technique

Structure du Programme :

Le programme est structuré en plusieurs composants principaux, chacun jouant un rôle bien défini. Cette modularité permet une maintenance facile et une extensibilité du code.

Classe Terrain :

Cette classe représente le labyrinthe sous forme d'une grille de cases (murs, cases libres, départ, arrivée), elle stocke également les positions du point de départ et du point d'arrivée.

La classe Terrain dispose de méthodes classiques comme chargerDepuisFichier, qui permet de charger le labyrinthe depuis un fichier texte, et afficher, qui affiche la grille du terrain dans la console. Elle propose également des fonctionnalités spécifiques, comme estLibre, qui vérifie si une case donnée est accessible (non bloquée par un mur). Les méthodes getLargeur et getLongueur permettent d'obtenir les dimensions du labyrinthe, tandis que getCaseDepart et getCaseArrivee retournent respectivement les positions du point de départ et du point d'arrivée. Enfin, des méthodes comme transformerTexteAmeliore1 et afficherTexteAmeliore2 permettent de transformer et d'afficher une version améliorée du labyrinthe.

```
class terrain {
public:

    terrain();

    bool chargerDepuisFichier(const std::string& nomFichier);
    void afficher() const;
    void transformerTexteAmeliore1();
    void afficherTexteAmeliore2();
    int getLargeur() const;
    int getLongueur() const;

    position getCaseDepart() const;
    position getCaseArrivee() const;

    bool estLibre(const position& sp) const;

    private:
    std::vector<std::vector<char>> d_terrain;
    int d_largeur;
    int d_longueur;
    position d_depart;
    position d_arrivee;
};
```

Classe Position :

Cette classe a été créé pour faciliter la gestion des positions dans le labyrinthe, plutôt que d'utiliser les coordonnées x et y a chaque fois, on peut utiliser directement une position.

Cette classe position représente une position en deux dimensions à l'aide des coordonnées x et y. Elle dispose de constructeurs pour initialiser une position, ainsi que des méthodes comme getX et getY pour obtenir les coordonnées de la position. La méthode setPosition permet de modifier les coordonnées, tandis que estEgale compare deux positions pour vérifier leur égalité. Elle redéfinit également les opérateurs == et != pour comparer facilement deux objets position. Les attributs privés d_x et d_y stockent les coordonnées x et y de la position.

```
class position{
    public:
        position();
        position(int x,int y);
        int getX() const;
        int getY() const;
        void setPosition(int x,int y);
        bool estEgale(const position &p) const;
        bool operator==(const position &p1) const;
        bool operator!=(const position &p1) const;

    private:
        int d_x;
        int d_y;

};
```

Classe Robot :

La classe robot représente un robot capable de se déplacer et d'interagir avec son environnement tout en stockant une liste d'observateurs. Elle dispose d'un constructeur par défaut et d'un autre permettant de définir une position initiale et une direction. Les méthodes getPositionActuelle et getAnciennePosition permettent d'obtenir respectivement la position actuelle et la position précédente du robot. La direction est récupérable via getDirection et la position est modifiable avec setPosition.

Le robot peut détecter des obstacles avec les méthodes detecterObstacleDevant et detecterObstacleDroite, qui prennent en paramètre un objet terrain. Il est aussi possible d'ajouter des observateurs grâce à enregistrerObservateur et de les notifier via notifierObservateurs.

Les méthodes avancer, tournerGauche, et tournerDroite permettent de déplacer ou réorienter le robot. Enfin, afficherStatistiquesObservateurs fournit des informations sur les observateurs enregistrés.

Les attributs privés comprennent la position actuelle et ancienne du robot (d_position, d_anciennePosition), la direction actuelle (d_direction), ainsi qu'une liste d'observateurs gérée via des pointeurs.

```
class robot {
    private:
        position d_position;
        position d_anciennePosition;
        char d_direction;
        vector<unique_ptr<observateur>> d_observateurs;

    public:
        robot();
        robot(const position &pos, char direction);
        position getPositionActuelle() const;
        position getAnciennePosition() const;
        void setPosition(const position &p);
        char getDirection() const;
        int nombreObservateurs() const;
        bool detecterObstacleDevant(const terrain& terrain);
        bool detecterObstacleDroite(const terrain& terrain);
        void enregistrerObservateur(std::unique_ptr<observateur> obs);
        void notifierObservateurs();
        void avancer();
        void tournerGauche();
        void tournerDroite();
        void afficherStatistiquesObservateurs();

};
```

Classe Observateur :

La classe observateur est une classe abstraite qui sert de base pour créer différents observateurs, chacun spécialisé dans le calcul de statistiques ou d'autres fonctionnalités, comme l'afficheur du robot, un observateur essentiel. Lorsqu'un robot effectue une action, l'observateur est automatiquement notifié et met à jour ses données grâce à sa méthode virtuelle update, implémentée spécifiquement dans chaque sous-classe. Enfin, la méthode afficherStatistique permet d'afficher les résultats calculés par chaque observateur, offrant ainsi une vue claire et précise des statistiques ou des informations suivies.

```
#include "position.h"

class robot;


class observateur {
public:
    virtual ~observateur() = default;

    virtual void update(const robot &r) = 0;
    virtual void afficherStatistique()=0;
};

#endif // OBSERVATEUR_H
```

Classe programme :

La classe programme agit comme un orchestrateur, réunissant et coordonnant toutes les autres classes pour assurer le bon fonctionnement global. Chaque méthode de cette classe représente une étape spécifique du processus, contribuant à la méthode principale exécuter, qui gère l'exécution complète du programme. Cette méthode charge d'abord un labyrinthe, initialise le robot, puis invite l'utilisateur à choisir un mode d'affichage et un algorithme à appliquer pour contrôler le robot. Ainsi, la classe programme centralise et pilote l'ensemble des interactions entre les différentes composantes du système.

```
class programme {
public:

    programme();
    void chargerFichier(const string &nomFichier);
    bool isTerrainCharge() const { return terrainCharge; }
    void initialisationRobot(robot &r);
    void executer();
    void choisirAlgorithme(robot &r);
    void choisirAffichage();


private:

    bool terrainCharge;
    terrain ter;

};
```

Algorithmes Utilisés

Deux algorithmes de navigation ont été implémentés pour guider le robot à travers le labyrinthe. Chacun a ses spécificités et ses cas d'utilisation.

Algorithme de la Main Droite :

- Principe : Le robot suit toujours le mur à sa droite.

- Étapes :
    - Si aucun mur n'est à droite, tourner à droite et avancer.
    - Si un mur est à droite, avancer tout droit.
    - Si un obstacle est devant, tourner à gauche.

- Avantages : Simple à implémenter et efficace dans la plupart des labyrinthes simples.

- Limites : Peut échouer dans des labyrinthes avec des boucles.

Algorithme de Pledge :

- Principe : Combine des déplacements en ligne droite avec un suivi de mur pour éviter les boucles infinies.

- Étapes :
    - Avancer tout droit jusqu'à rencontrer un mur.
    - Longer le mur (par la droite ou la gauche) tout en comptant les changements de direction.
    - Lorsque le compteur de changements de direction atteint zéro, reprendre le déplacement en ligne droite.

- Avantages : Fonctionne dans des labyrinthes plus complexes et évite les boucles infinies.

- Limites : Plus complexe à implémenter que l'algorithme de la main droite.

Stratégies de Tests

Dans le cadre de ce projet, plusieurs stratégies de tests ont été mises en place pour assurer la robustesse et la fiabilité du programme. Voici une présentation détaillée des méthodes utilisées :

Utilisation de Doctest :

Nous avons utilisé la bibliothèque Doctest pour effectuer les tests unitaires. Doctest est une bibliothèque légère, simple à intégrer et performante pour écrire des tests. Elle permet de s'assurer de la validité du code en lançant des assertions et des sous-cas de tests, offrant ainsi une méthode efficace pour valider le fonctionnement du programme.

Tests Unitaires :

Les tests unitaires ont été appliqués à chaque fonction du programme pour vérifier son bon fonctionnement isolé. Ces tests se concentrent sur une seule unité de code, comme une fonction ou une classe, pour s'assurer que chaque composant fonctionne correctement.

Utilisation de fstream :

Pour tester certaines fonctionnalités, nous avons eu besoin de simuler des fichiers, comme lors du chargement d'un terrain ou d'une lecture de données. Nous avons utilisé des std::ofstream pour écrire des contenus dans des fichiers, puis lu ces fichiers pour vérifier que les données étaient correctement manipulées par le programme.

Voici un exemple de fonction auxiliaire utilisée pour créer des fichiers temporairement :

```
void creerFichierTest(const std::string& nomFichier, const std::string& contenu) {
    std::ofstream fichier(nomFichier);
    if (fichier.is_open()) {
        fichier << contenu;
        fichier.close();
    }
}
```

Utilisation de classes factices (Fake Classes) :

Pour certaines méthodes, notamment celles qui interagissaient avec des données complexes ou des modules externes, nous avons eu recours à des Fake Classes. Ces classes permettaient de simuler des objets ou des interfaces sans implémenter toute la logique, facilitant ainsi les tests unitaires.

Ces fake classes ont permis de tester des méthodes qui devaient gérer des interactions spécifiques, tout en assurant un contrôle strict sur les entrées et sorties des fonctions.

Exemple d'un fake observateur avec un booléen qui permet de savoir si la fonction update() a bien été appelée lorsqu'on notifie un observateur à partir d'un robot :

```
class fake_observateur : public observateur
{
    public:
        bool notifie = false;
        bool affiche = false;

        void update(const robot& r) override
        {
            notifie = true;
        }
        void afficherStatistique() override
        {
            affiche = true;
        }
};
```

Structure des Tests : TEST_CASE et SUBCASE :

Nous avons structuré nos tests avec des TEST_CASE pour regrouper les tests liés à une fonctionnalité spécifique, et SUBCASE pour décomposer chaque cas de test selon différents scénarios ou configurations. Cela nous a permis de tester de manière plus approfondie chaque aspect des fonctionnalités, des cas courants aux cas limites.

Utilisation des ASSERTIONS : REQUIRE :

Nous avons toujours utilisé REQUIRE pour les assertions, notamment lorsque nous devions garantir strictement que certaines conditions soient remplies, afin de détecter tout échec critique du test. Cette approche nous permet d'avoir une validation stricte des résultats, en assurant que chaque test passe sans compromis sur les exigences fondamentales du projet.

Résolution des bugs grâce aux tests :

Grâce aux tests rigoureux effectués, nous avons pu détecter divers bugs et problèmes au sein du code. Par exemple :

- Des erreurs dans les algorithmes
- Des problèmes d'indexation ou de gestion des positions dans les terrains chargés.
- Des situations où certaines méthodes n'étaient pas correctement adaptées à des cas particuliers.

Ces problèmes détectés ont permis une correction immédiate du code, suivie d'une reprogrammation ou d'une optimisation pour garantir que le programme fonctionne de manière stable.

4. Problèmes Rencontrés et Solutions Apportées

Durant le développement du projet, plusieurs défis techniques et problèmes de conception ont été rencontrés. Ces problèmes ont nécessité des réflexions approfondies et des ajustements pour aboutir à une solution optimale. Voici une description détaillée des principaux problèmes rencontrés et des solutions mises en œuvre.

Gestion des Mouvements du Robot

Problème :

L'un des premiers défis a été de déterminer comment gérer les mouvements du robot. La question était de savoir si les déplacements devaient être gérés directement dans la classe Robot ou si l'affichage des mouvements devait être délégué à un observateur.

- Option 1 : Gérer les mouvements directement dans la classe Robot.
  Avantage : Simple à implémenter.
  Inconvénient : Cela mélange la logique de déplacement avec l'affichage, ce qui réduit la modularité et la réutilisabilité du code.

- Option 2 : Utiliser le modèle observateur pour notifier les observateurs à chaque déplacement.

Avantage : Séparation des responsabilités (le robot se déplace, un autre composant affiche).
Inconvénient : Plus complexe à mettre en place.

Solution :

Nous avons opté pour la deuxième option en utilisant le modèle observateur. Cela permet de séparer la logique de déplacement du robot de son affichage, rendant le code plus modulaire et extensible.

Implémentation :

Une classe AfficheurRobot a été créée, qui hérite de la classe Observateur.
La méthode update() de AfficheurRobot a été modifiée pour afficher les déplacements du robot en utilisant la méthode goto (ou une fonction similaire) pour positionner le curseur dans la console.

À chaque déplacement du robot, la méthode notifierObservateurs() est appelée, ce qui déclenche l'affichage via AfficheurRobot.

Affichage en Temps Réel

Problème :

Un autre défi a été de réaliser un affichage en temps réel des déplacements du robot.

Solution :

Nous avons utilisé la méthode goto() pour repositionner le curseur dans la console à chaque déplacement du robot. Cela permet d'actualiser l'affichage sans avoir à redessiner tout le labyrinthe.

A chaque update de l'afficheurRobot, l'ancienne position est remplacée par une case libre (puisque le robot vient toujours d'une classe libre) et la position actuelle est remplacée par le robot (dans la direction dans laquelle il est).

```
void afficheurRobot::update(const robot &r)
{
    goto_xy(r.getAnciennePosition().getX(),r.getAnciennePosition().getY());
    std::cout<<".";
    goto_xy(r.getPositionActuelle().getX(),r.getPositionActuelle().getY());
    std::cout<<r.getDirection();
    Sleep(500);
}
```

Avantages : Affichage fluide et en temps réel et réduction de la complexité de l'affichage.

Inconvénients : Nécessite une gestion spécifique du curseur, qui peut varier selon le système d'exploitation.

5. Remaniements du code

Dans le cadre de l'amélioration continue de notre projet, plusieurs remaniements du code ont été réalisés pour optimiser la structure, améliorer la lisibilité et corriger des comportements inattendus. Dans cette section, nous allons présenter quelques exemples significatifs, tout en soulignant que ces changements ne représentent qu'une partie des optimisations apportées tout au long du développement.

Exemple d'améliorations apportées pour optimiser ou clarifier le code :

Division de la méthode exécuter en sous-méthodes :

La méthode principale exécuter a été divisé en plusieurs sous-méthodes afin d'améliorer la lisibilité et l'efficacité du code. Cette refactorisation permet une meilleure modularité, rendant chaque étape du processus plus concise et compréhensible.

Avant remaniement :

```
void programme::executerChoixAlgorithme() {
    initialiser();
    if (!terrainCharge) {
        cerr << "Erreur : Le terrain n'a pas ete charge. Initialisez d'abord le programme." << endl;
        return;
    }


    compteur_deplacement* compteur = new compteur_deplacement();
    afficheur = std::make_unique<afficheurRobot>();
    calcul_temps* timer = new calcul_temps();
    compteur_rotation* compteurRot = new compteur_rotation();
    position depart = ter.getCaseDepart();
    position arrivee = ter.getCaseArrivee();
    robot r(depart, 'v');
    r.enregistrerObservateur(std::unique_ptr<compteur_deplacement>(compteur));
    r.enregistrerObservateur(std::move(afficheur));
    r.enregistrerObservateur(std::unique_ptr<calcul_temps>(timer));
    r.enregistrerObservateur(std::unique_ptr<compteur_rotation>(compteurRot));
    int choix;
    cout<< endl;
    cout<< endl;
    cout << "Choisissez un algorithme: 1 pour Algorithme Main Droite, 2 pour Algorithme Pledge" << endl;
    cout<< endl;
    cin >> choix;
    cout<< endl;
    choisirAlgorithme(choix, r, ter);
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout << "Position de depart : (" << depart.getX() << ", " << depart.getY() << ")" << endl;
    cout << "Position d'arrivee : (" << arrivee.getX() << ", " << arrivee.getY() << ")" << endl;
    cout << "Nombre de deplacements : " << compteur->getNombreDeplacements() << endl;
    std::cout << "Nombre de rotations : " << compteurRot->getNombreRotations() << std::endl;
    timer->afficherTempsTotal();
    cout<< endl;
    cout<< endl;
    }
```

Après remaniement :

```
void programme::executer()
{
    chargerFichier("./test.txt");
    if (!terrainCharge)
    {
        cerr << "Erreur : Le terrain n'a pas ete charge. Initialisez d'abord le programme." << endl;
        return;
    }
    choisirAffichage();
    robot r{};
    initialisationRobot(r);
    choisirAlgorithme(r);

    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout<< endl;
    cout << "Position de depart : (" << ter.getCaseDepart().getX() << ", " << ter.getCaseArrivee().getY() << ")" << endl;
    cout << "Position d'arrivee : (" << r.getPositionActuelle().getX() << ", " << r.getPositionActuelle().getY() << ")" << endl;
    r.afficherStatistiquesObservateurs();
}
```

Création d'une méthode virtuelle afficherStatistique dans les observateurs :

Chaque observateur peut désormais afficher ses propres statistiques grâce à l'utilisation de la méthode virtuelle afficherStatistique. Auparavant, cette fonctionnalité était réalisée manuellement pour chaque observateur et l'affichage était écrit dans le programme, rendant le code moins maintenable et plus sujet aux erreurs.

Avant remaniement :

Dans les observateurs : (exemple avec le compteur de rotation)

```
int compteur_rotation::getNombreRotations() const {
    return nombreRotations;
}
```

Dans le programme :

```
cout << "Position d'arrivee : (" << arrivee.getX() << ", " << arrivee.getY() << ")" << endl;
cout << "Nombre de déplacements : " << compteur->getNombreDeplacements() << endl;
std::cout << "Nombre de rotations : " << compteurRot->getNombreRotations() << std::endl;
timer->afficherTempsTotal();
```

Après remaniement :

Dans la classe abstraite observateur :

```
class observateur {
public:
    virtual ~observateur() = default;

    virtual void update(const robot &r) = 0;
    virtual void afficherStatistique()=0;
};
```

Dans l'observateur compteur de rotation :

```
void compteur_rotation::afficherStatistique()
{
    std::cout << "Nombre de rotations : " << nombreRotations << std::endl;
}
```

Création d'une méthode afficherStatistiquesObservateurs dans la classe robot :

Evidemment, pour donner suite aux remaniements des observateurs, nous avons également remanier le code pour l'affichage des statistiques. Plutôt que d'afficher manuellement pour chaque observateur sa statistique nous avons pu désormais grâce a la nouvelle méthode afficherStatistique des observateurs, afficher directement toutes les statistiques dans le programme en utilisant cette fonction :

```
void robot::afficherStatistiquesObservateurs()
{
    for(int i=0;i<nombreObservateurs();i++)
    {
        d_observateurs[i]->afficherStatistique();
    }
}
```

Voici les changements que ça a fait dans le programme :

Avant remaniement :

```
cout << "Position d'arrivee : (" << arrivee.getX() << ", " << arrivee.getY() << ")" << endl;
cout << "Nombre de déplacements : " << compteur->getNombreDeplacements() << endl;
std::cout << "Nombre de rotations : " << compteurRot->getNombreRotations() << std::endl;
timer->afficherTempsTotal();
```

Apres remaniement :

```
r.afficherStatistiquesObservateurs();
```

Reprogrammation de la fonction algorithmeMainDroite :

Lors de la réalisation des tests, il a été constaté qu'une instruction manquait au départ de l'algorithme algorithmeMainDroite, ce qui empêchait le robot d'atteindre un mur et le faisait tourner en boucle infinie. Cette instruction a été ajoutée au début de la fonction pour corriger ce comportement :

```
if(!r.detecterObstacleDroite(ter))
{
    while (!r.detecterObstacleDevant(ter) && r.getPositionActuelle() != ter.getCaseArri
    {
        r.avancer();
    }
    r.tournerGauche();
}
```

6. Pertinence des outils et méthodes enseignés

L'utilisation des outils et des méthodes enseignés, tels que Git, Doctest, et la Programmation Orientée Objet (POO), a eu un impact significatif sur notre projet et sur le produit final. Ces outils et concepts ont permis d'améliorer la collaboration, la qualité du code et la gestion des tests, tout en rendant le développement plus efficace.

Git s'est révélé être un outil essentiel pour la gestion des versions, la collaboration en équipe et le suivi des changements. Il a permis de garder une traçabilité précise des modifications apportées au code, facilitant ainsi la détection et la correction des erreurs. Grâce à Git, nous avons pu maintenir une base de code propre et bien organisée, tout en favorisant une meilleure gestion des contributions des membres de l'équipe.

Doctest a également été une méthode clé dans le processus de développement. Il nous a permis d'écrire des tests simples et lisibles directement dans le code source, garantissant une vérification efficace des comportements attendus. Cela a contribué à une meilleure couverture des cas de test, réduisant ainsi le nombre d'erreurs et d'ajustements nécessaires au fil du développement.

Enfin, la Programmation Orientée Objet (POO) nous a permis de structurer le code de manière claire et modulable. L'utilisation de concepts tels que les classes, les objets, et les interactions entre ces derniers a favorisé une meilleure organisation du code, tout en simplifiant les évolutions futures du système. La séparation des responsabilités et la gestion efficace des différents modules ont permis de concevoir un produit plus robuste et maintenable.

En somme, ces outils et méthodes ont eu un impact positif sur notre projet en améliorant à la fois la qualité du code et l'efficacité globale du développement, ce qui s'est reflété directement dans le produit final.

7. Conclusion

Ce projet de gestion des déplacements d'un robot dans un labyrinthe a été une expérience très enrichissante, où nous avons pu appliquer tout ce que nous avons appris en cours. Nous avons utilisé des concepts comme l'héritage (avec une classe abstraite Observateur et des classe dérivée), le remaniement du code pour le rendre plus clair et plus facile à maintenir, et les tests unitaires pour vérifier que tout fonctionnait correctement. C'était l'occasion de voir comment ces notions théoriques se traduisent en pratique.

Nous avons réussi à implémenter deux algorithmes de navigation (main droite et Pledge) pour guider le robot à travers le labyrinthe. Le modèle observateur nous a permis de séparer les responsabilités : le robot se déplace, et l'affichage est géré par une autre classe. Les tests unitaires, réalisés avec doctest, nous ont aidés à détecter et corriger les erreurs rapidement, ce qui a rendu le code plus fiable.

En résumé, ce projet nous a permis de concrétiser nos connaissances et de relever des défis techniques intéressants. Les solutions que nous avons mises en place fonctionnent bien, et elles laissent la porte ouverte à des améliorations, comme un affichage graphique ou l'ajout de nouveaux algorithmes. C'est une belle preuve qu'avec une bonne organisation et une approche rigoureuse, on peut transformer une idée en un système fonctionnel et évolutif.

8. Annexe

Interface :

Choix de l'affichage du labyrinthe :

```
Terrain charge avec succes
Quel type d'affichage voulez vous ? : 1 pour texte simple, 2 pour texte ameliore 1, 3 pour texte ameliore 2
```

Affichage du terrain en fonction du choix de l'utilisateur :

Affichage simple

```
XXXXXXXXXXXXXXXXXXX
XD.....XXXX........X
X..................X
X..X...............X
X..X........X......X
X..X........X......X
X..X........XA.....X
XXXXXXXXXXXXXXXXXXX
```

Affichage amélioré 1

```
+------+##+--------+
|D.....+--+........|
|..................|
|..|...............|
|..|........|......|
|..|........|......|
|..|........|A.....|
+--+--------+------+
```

Affichage amélioré 2

```
D.....  ........
.....  ........
..|    ........
..|    ........|......
..|    ........|......
..|    ........|A.....
```

Choix de l'algorithme :

```
Choisissez un algorithme: 1 pour Algorithme Main Droite, 2 pour Algorithme Pledge

```

Déplacement du robot selon l'algorithme choisi, jusqu'à l'arrivée puis affichage des statistiques :

```
......  ........
..................
..|    ...........
..|    ........|......
..|    ........|......
..|    ........|v.....


Choisissez un algorithme: 1 pour Algorithme Main Droite, 2 pour Algorithme Pledge

1

Position de depart : (1, 6)
Position d'arrivee : (13, 6)
Nombre de deplacements : 39
Temps total ecoule : 315835 ms
Nombre de rotations : 8

Process returned 0 (0x0)   execution time : 317.117 s
Press any key to continue.
```