Rendu du TP POO

Equipe Teide 22

1 Affichage de la Carte

Pour répondre à la première partie du projet, nous avons implémenté les classes essentielles suivantes : `Carte`, `Case`, `Incendie`, `Direction`, `NatureTerrain`, et `DonneesSimulation`. Ces classes modélisent la structure de la carte et les différents éléments qui la composent, comme les incendies et les types de terrain, en assurant une séparation claire des responsabilités.

Conformément aux spécifications du sujet, nous avons respecté les schémas UML fournis. Chaque classe et relation est organisée de manière à correspondre aux diagrammes de classes et aux relations d'association indiqués, garantissant ainsi une architecture cohérente et conforme aux exigences.

La classe `LecteurDonnees`, existante, a été modifiée pour respecter au mieux les spécifications et simplifier le chargement des données depuis des fichiers. Cette classe permet maintenant de lire et créer les informations de la carte, des incendies et des robots tout en vérifiant le format pour minimiser les erreurs de lecture.

Initialement, l'affichage était minimaliste, utilisant des rectangles pour représenter les cases. Cette approche a été choisie pour valider rapidement la logique sous-jacente sans complexité graphique. Une fois la logique confirmée, nous avons amélioré l'affichage en important des images pour les différents types de terrain (e.g., forêt, eau). Cette séparation entre les données et l'affichage a facilité les modifications et améliorations, tout en maintenant une architecture lisible et modulaire.

Ces choix de conception ont permis une implémentation progressive et une transition fluide vers un affichage plus sophistiqué. Le recours à des classes bien définies a également facilité le débogage et les tests initiaux.

2 Exécution de Scénarios et Gestion des Événements

À l'étape 2, nous avons introduit la possibilité de faire bouger les robots en utilisant des scénarios prédéfinis. Pour cela, nous avons décidé de séparer les fichiers de scénarios du code source principal. Ces fichiers sont placés dans un sous-dossier indépendant nommé `scenarios`, tout en restant dans le projet global.

Cette conception présente plusieurs avantages.Contrairement à une approche où les scénarios sont directement codés dans la classe `TestEtape2`, l'utilisateur n'a pas besoin de modifier le code source pour tester un nouveau scénario. Il suffit d'ajouter un fichier texte dans le sous-dossier `scenarios` contenant le scénario souhaité. Ce fichier peut ensuite être associé à une carte spécifique grâce à l'utilisation du `Makefile`.

Pour gérer les actions des robots dans le temps, nous avons ajouté une classe abstraite `Evenement`, définissant une structure de base pour les événements avec une méthode abstraite `execute()` et un attribut `date` pour spécifier le moment d'exécution. Des sous-classes concrètes comme `EvenementDeplacement`, `EvenementIntervention`, `EvenementRemplissage`, et `EvenementErreur` ont été implémentées pour représenter différents types d'actions des robots (déplacement, intervention, remplissage, et gestion d'erreurs).

Dans la classe `Simulateur`, une `PriorityQueue` est utilisée pour stocker les événements par ordre de date, assurant ainsi une exécution synchronisée. À chaque appel de la méthode `next()`, l'événement le plus proche dans le temps est exécuté en utilisant la méthode `poll()` de la file de priorité, ce qui permet de maintenir les actions synchronisées avec le déroulement de la simulation. La méthode `restart()` a également été implémentée pour réinitialiser la simulation, en rechargeant les données et en réinitialisant la file d'événements.

Cette approche modulaire rend le système plus flexible et améliore la maintenabilité, en séparant clairement la logique des événements de la logique principale du simulateur. De plus, des messages d'erreur sont affichés dans le terminal pour chaque événement incorrect ou impossible, garantissant une gestion robuste des erreurs dans chaque étape de la simulation.

3 Calcul du Plus Court Chemin

Pour la troisième étape, nous avons implémenté le calcul du plus court chemin dans la classe abstraite `Robot` en utilisant l'algorithme de Dijkstra. Cette méthode a été choisie pour sa capacité à gérer efficacement les contraintes de déplacement des robots, telles que les terrains accessibles ou la vitesse. L'algorithme calcule le chemin optimal entre la position actuelle d'un robot et une destination donnée, tout en respectant les spécifications énoncées dans le sujet.

L'implémentation est divisée en deux parties principales :

- `calculerPlusCourtChemin` : Cette méthode calcule le chemin optimal en tenant compte des distances et des contraintes de déplacement. Une structure de données `HashMap` est utilisée pour stocker les distances, et une file de priorité gère les explorations des nœuds.

- `programmerDeplacements` : Cette méthode planifie les déplacements du robot à chaque étape du chemin, en ajoutant des événements au simulateur.

Lors des tests (classe `TestEtape3`), nous avons rencontré un problème sur les cartes contenant plusieurs robots du même type. En effet, sans un identifiant unique, seul le premier robot du type était pris en compte pour les déplacements. Pour résoudre ce problème, nous avons ajouté des identifiants uniques aux robots (e.g., `DRONE0`, `DRONE1`, `PATTES0`). Cette modification a permis d'individualiser les robots et d'assurer leur gestion correcte.

Nous avons testé le système sur différents types de robots et cartes, en simulant des déplacements avec des contraintes variées. Les résultats montrent que l'algorithme fonctionne correctement, y compris dans les scénarios complexes avec plusieurs robots sur la même carte. Cette conception garantit une extensibilité et une précision accrues dans la gestion des déplacements des robots.

4 Stratégies d'Affectation des Tâches

Dans cette dernière étape, nous avons ajouté une classe `ChefPompier` pour centraliser les décisions d'affectation des robots aux incendies. Cette classe comporte deux stratégies principales :

- **Stratégie élémentaire** : Une méthode simple qui parcourt naïvement la liste des incendies et affecte un robot disponible et compatible à chaque incendie. Cette stratégie garantit une intervention sur chaque incendie, mais sans optimisation particulière.
- **Stratégie évoluée** : Une méthode optimisée qui affecte chaque robot disponible à l'incendie le plus proche accessible. Cette approche améliore l'efficacité globale en minimisant les temps de déplacement.

Bien que des améliorations supplémentaires soient possibles (e.g., prendre en compte l'intensité des incendies ou coordonner les recharges des robots), nous avons choisi de nous concentrer sur des implémentations fonctionnelles qui respectent les consignes du sujet. Ce dernier précise qu'il n'est pas essentiel de fournir une solution parfaitement robuste, mais plutôt de garantir que les stratégies implémentées fonctionnent correctement.

Deux classes de tests, `ChefPompierTestElementaire` et `ChefPompierTestEvoluee`, ont été développées pour valider le fonctionnement des deux stratégies. Les tests montrent que les robots sont correctement affectés aux incendies et que les déplacements respectent les contraintes de terrain et de disponibilité des robots. Grâce à cette conception, nous avons pu implémenter rapidement une gestion efficace des incendies tout en laissant la possibilité d'améliorer les stratégies à l'avenir.