SOKOBAN Compte rendu

Date de création	@27 avril 2025 12:20
Étiquettes	

Projet SOKOBAN

Jassem TAMOURGH

Yanis CHEBBAH

https://github.com/JassemMT/Sokoban

Plan du Développement du Projet

1. **Création des Classes et Structure du Jeu :**
   - Nous avons commencé par concevoir la structure de base du jeu, en définissant les principales classes comme `Jeu`, `Perso`, `Element`, `ListeElements`, et `Labyrinthe`.
   - La classe `Jeu` a été placée au cœur du projet, avec des attributs essentiels pour gérer le personnage (`perso`), les caisses (`caisses`), les dépôts (`depots`), et le labyrinthe (`laby`).
2. **Développement des Méthodes Principales :**
   - Une fois la structure en place, nous avons développé les méthodes clés pour assurer le bon fonctionnement du jeu, en particulier celles qui gèrent les déplacements du personnage et des objets :
     - La méthode `getSuivant(int x, int y, String action)` permet de calculer la prochaine position en fonction de l'action donnée (haut, bas, gauche, droite).
     - La méthode `deplacerPerso(String action)` utilise la méthode `getSuivant` pour déplacer le personnage et gérer les interactions avec les caisses et les murs.
3. **Ajout des Tests Unités :**
   - À mesure que nous développions les méthodes, nous avons écrit des tests unitaires pour valider le bon fonctionnement de chaque fonctionnalité. Cela nous a permis de détecter des problèmes, comme celui des déplacements inversés, que nous avons pu corriger efficacement.
4. **Résolution des Problèmes Techniques :**
   - L'un des défis majeurs que nous avons rencontrés était l'inversion des mouvements, même si la méthode `deplacerPerso` semblait correcte. Après avoir analysé le problème, nous avons découvert que les coordonnées étaient mal gérées dans certains endroits du code, notamment lors du calcul du déplacement des éléments.
   - Grâce aux tests unitaires, nous avons pu localiser et résoudre ce problème.
5. **Gestion des Packages et Structure du Répertoire :**
   - Vers la fin du projet, nous avons réalisé que les classes devaient être organisées dans des packages pour permettre un accès correct entre elles, notamment entre les packages `jeu`, `graphisme`, et `test`. Nous avons donc restructuré le projet pour intégrer cette organisation en utilisant un répertoire `src` avec les sous-dossiers correspondants.
   - Nous avons aussi rendu certaines méthodes publiques pour permettre l'accès entre les différentes parties du projet.
6. **Utilisation de GitHub :**

- Tout au long du projet, GitHub a été un outil précieux pour la collaboration, le suivi des versions, et la gestion des commits. Nous avons utilisé les commits pour garder une trace de notre progression, ce qui nous a permis de revenir facilement sur certaines étapes si nécessaire. En cas de bugs, GitHub a facilité les rollbacks pour revenir à une version stable du projet.

Variables statiques

L'utilisation de variables statiques, (étudiée en **R2.01b)** , pour les directions (`HAUT` , `BAS` , `DROITE` , `GAUCHE`) dans notre code présente plusieurs avantages. Elles permettent de centraliser les valeurs, réduisant ainsi les risques d'erreurs de typographie et facilitant la maintenance du code. En cas de modification des valeurs, il suffit de changer les constantes à un seul endroit. Cela garantit aussi la cohérence et l'uniformité dans l'ensemble du projet.

En tant que constantes `static final` , ces variables sont partagées entre toutes les instances, ce qui optimise l'utilisation de la mémoire et les performances.

Héritage

L'héritage (étudié en **R2.01b)** est utilisé dans notre projet pour créer une hiérarchie d'objets permettant de gérer efficacement les éléments du jeu. Par exemple, la classe `Element` sert de base pour différents types d'objets, comme les caisses et les dépôts, qui héritent de cette classe. Cela permet de centraliser des comportements communs, comme les coordonnées (`x` , `y`) et les déplacements, tout en facilitant l'extension du code.

La classe `ListeElements` , quant à elle, gère une collection d'objets `Element` . Grâce à cette structure, nous pouvons manipuler et parcourir facilement tous les éléments du jeu sans répéter de code.

Exceptions

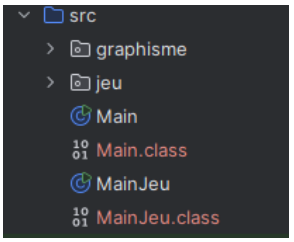
L'utilisation d'exceptions personnalisées, (étudié en **R2.03)** , comme dans la classe `FichierIncorrectException` , permet de distinguer les erreurs liées aux fichiers des autres types d'erreurs, facilitant ainsi le débogage et la gestion des exceptions.

```
public class FichierIncorrectException extends Exception {
   public FichierIncorrectException(String message) {
      super(message);
   }
```

Difficultés Rencontrées

En plus des développements techniques, plusieurs difficultés ont été rencontrées durant le projet, principalement autour de l'organisation du code et des tests des déplacements du personnage.

Problème de l'organisation des fichiers et des packages :
Une difficulté majeure a été la gestion des packages et des répertoires dans le projet. Nous avons découvert que les fichiers du répertoire `Graphisme` ne pouvaient accéder aux classes du jeu que si celles-ci étaient dans des packages, ce qui a entraîné des ajustements dans la structure des fichiers. Les classes ont été déplacées dans un dossier spécifique respectant l'arborescence `src/jeu` , `src/graphisme` , et `/test` , et les méthodes ont été rendues publiques pour garantir leur accessibilité à travers les différents modules.



Problème des mouvements inversés :
Lors du test des mouvements dans le jeu, nous avons observé que les déplacements du personnage étaient inversés (par exemple, déplacer vers le "haut" augmentait la coordonnée `y` au lieu de la diminuer).

Après avoir effectué des tests avec JUnit, nous avons identifié que le problème provenait de l'inversion des axes `x` et `y` dans le calcul des déplacements dans certaines parties du code, notamment dans la méthode `deplacement` du personnage.

Outils

Gestion avec Git :

L'utilisation de Git a été essentielle pour la gestion de version et la collaboration. Nous avons utilisé les commits pour suivre les modifications et les branches pour tester différentes fonctionnalités sans affecter la version principale. Cela nous a permis de revenir en arrière si nécessaire et de travailler sur des fonctionnalités en parallèle.

IntelliJ:

Excellent pour identifier les erreurs, déterminer les liens entre les classes et de manière globale, débugger, cet IDE nous a permis d'optimiser notre workflow.