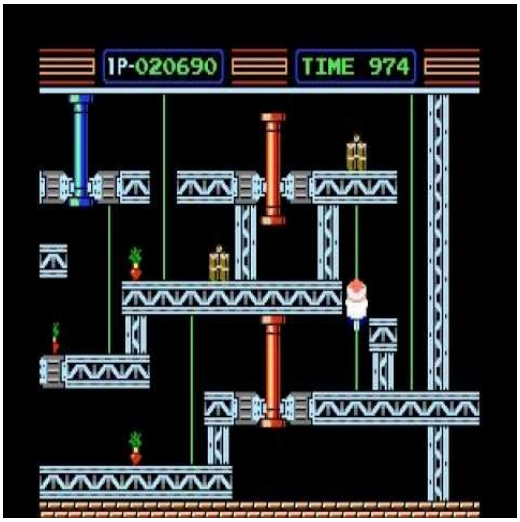
Projet - POO en Java

Gyromite

Critères d'évaluation :
- Qualité de l'analyse et du code associé
- Respect du Modèle Vue Contrôleur Strict
- Modularité
- Extensions proposées

1 Sujet

1.1 Gyromite



Description du jeu :

https://nintendo.fandom.com/wiki/Gyromite

Résumé :

Vous contrôlez le Prof. Hector qui se déplace pour ramasser les bombes dans son laboratoire, vu en 2D de côté. Le laboratoire est composé d'un ensemble de niveaux. Le scientifique peut se déplacer à gauche, à droite, et grimper ou descendre à l'aide de cordes. Un niveau peut contenir des piliers (bleus et rouges) qui peuvent monter ou descendre lorsque le joueur appuie sur une touche spécifique. Enfin, le professeur doit éviter les ennemis, les Smicks. L'objectif est donc de récupérer toutes les bombes, en évitant les ennemis, tout en manipulant les piliers pour créer des chemins si nécessaire.

Précisions concernant l'implémentation :

- Le plateau est représenté par une grille de cases du côté de la vue et une grille de cases du côté du modèle. Les murs et sols sont matérialisés par des cases pleines, les couloirs par des cases vides.

- Les mouvements sont discrétisés : on avance case par case, pas d'autres positions intermédiaires (pixel, etc.). Le temps est discrétisé : pour chaque entité, l'entité se déplace d'une seule case par pas de temps.

- Les déplacements des Entités sont gérés au tour-par-tour par le biais des classes RéalisateurDeDéplacement et Ordonnanceur. L'Ordonnanceur déclenche les RéalisateurDeDéplacements de manière séquentielle.

- Une instance de RéalisateurDeDéplacement représente le déplacement d'une ou plusieurs Entités, par le biais d'une méthode `realiserDeplacement()`. Cette méthode (à implémenter pour les classes filles de Deplacement) appelle la méthode `avancerDirectionChoisie(Direction)` sur les instances d'EntiteDynamique qui sont gérées par ce déplacement (par exemple, le Prof. ou toutes les cases d'une Colonne). En interne, `avancerDirectionChoisie(Direction)` utilise la méthode `deplacerEntite(Entite, Direction)` dans Jeu. Nous vous invitons à regarder le diagramme UML sur Claroline pour mieux comprendre les relations entre ces différentes classes ;

- Les Colonnes sont composées de N cases (N > 1) ; lorsqu'une Colonne se déplace, toutes ses cases se déplacent N-1 fois (à raison de 1 déplacement / tick de temps). Les Colonnes sont uniquement verticales. Illustration (vue horizontale) du déplacement (état initial à t → temps t+1 → temps t+2) :
_ x x x _ _ _ → _ _ x x x _ _ → _ _ _ x x x _

- Concernant les Smicks (ennemis), leur déplacement est principalement horizontal mais ils peuvent se déplacer à l'aide des cordes. En revanche, les Smicks ne peuvent pas tomber dans le vide : ils font demi-tour à l'extrémité d'une plateforme.

1.2 Travail en binôme

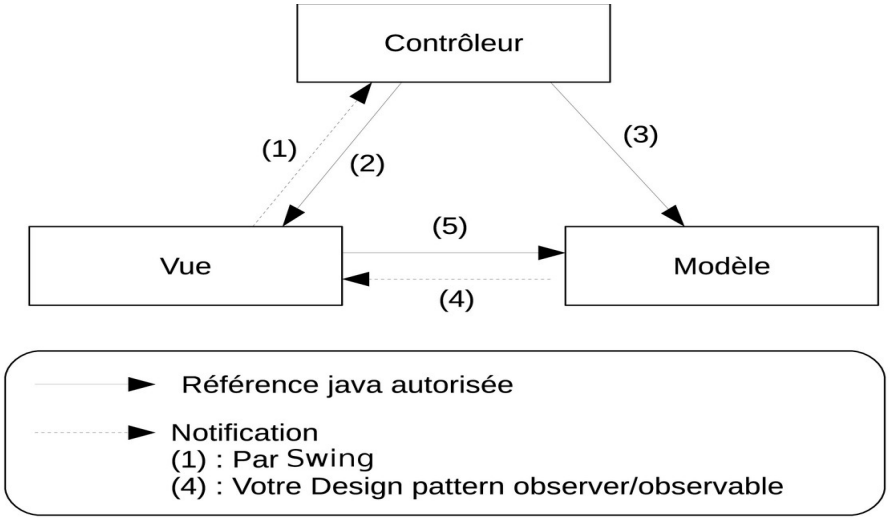
- Travail personnel entre les séances ;
- Évaluations individuelles ;
- Rapport par binôme : 8 pages au maximum, listes de fonctionnalités et extensions (indiquer la proportion de temps associée à chacune d'elles), documentation UML, justification de l'analyse UML, copies d'écran ;
- Démonstration lors de la dernière séance encadrée (présence des deux binômes obligatoire).

1.3 Travail à réaliser

Développer une application graphique Java la plus aboutie possible (utilisant Swing) de l'application, en respectant le modèle MVC Strict. Vous êtes responsables de votre analyse, et pouvez proposer des fonctionnalités supplémentaires. Veillez à proposer ces fonctionnalités incrémentalement, afin d'avoir une démonstration opérationnelle le jour de la soutenance. Il est recommandé de suivre les consignes de développement données dans la partie « Étapes de l'implémentation ». **Le code doit être le plus objet possible. Privilégiez donc une programmation objet plutôt qu'un algorithme central long et complexe.**

2 Rappel Modélisation MVC Strict

2.1 MVC Strict



MVC Strict :
(1) Récupération de l'événement Swing par le controleur
(2) Répercutions locale directe sur la vue sans exploitation du modèle
(3) Déclenchement d'un traitement pour le modèle
(4) Notification du modèle pour déclencher une mise à jour graphique
(5) Consultation de la vue pour réaliser la mise à jour Graphique

Application Calculette :
(1) récupération clic sur bouton de calculette
(2) construction de l'expression dans la vue (1,2...9, (,))
(3) déclenchement calcul (=)
(4) Calcul terminé, notification de la vue
(5) La vue consulte le résultat et l'affiche

Remarque : le code associé au contrôleur et à la vue peut être réalisé dans le même fichier .java tel que dans l'application *Calculette* (utilisation de classes anonymes ou de classes internes).

2.2 Modèle

Données :

Grille, Cases (états), Entités, Bombes, Bonus ramassables, Piliers, etc.

Processus :

- Initialiser : création des entités telles que les murs, les ennemis, le professeur etc.

- Réaliser les actions : déplacements des entités, gestion des collisions etc.
- Tests de fin de partie : plus de vies disponibles, toutes bombes ramassées etc.
- Etc.

2.3 Vue Plateau

Pour commencer la vue de votre projet, inspirez-vous du code fourni (disponible sur Claroline), qui respecte le MVC Strict, puis faites évoluer ce code.

Nous vous conseillons d'utiliser le code fourni (disponible sur Claroline) et de le faire évoluer pour ce projet en rajoutant les fonctionnalités, en utilisant Swing comme bibliothèque graphique.

Vous pouvez toutefois remplacer Swing par JavaFX et/ou reprendre le code depuis zéro si vous êtes suffisamment autonomes en Java et souhaitez approfondir vos connaissances.
Si vous utilisez JavaFX, évitez FXML (qui implique un MVC non Strict).

3 Étapes suggérées pour l'analyse et l'implémentation

3.1 Analyse sur papier : Modélisation Objet du problème (classes, périmètres des fonctionnalités, principaux traitements) : étudié en CM

3.2 Connexion Vue/Contrôleur – Modèle presque vide

Utilisez le code fourni comme base (MVC + Swing), et faites-le évoluer (changer le type de cases graphiques suivant vos besoins, changer le modèle, etc.).

Commencez par lire et comprendre le code fourni, les relations entre les différentes classes, la gestion des événements (MVC). Vérifiez également que le code fonctionne sur votre machine (Build+Run puis Package).

Remarque : il est normal d'avoir une structure de grille côté modèle et une structure de grille côté vue (gérées par Swing), cela est nécessaire pour garantir l'indépendance du modèle et de la vue. Les rôles des grilles sont différents, il ne s'agit pas d'une redondance.

3.3 Écrire les traitements du modèle

Commencez par ajouter l'environnement : rajouter les classes nécessaires dans le *modèle*, afin de représenter les Murs/Sols, les Ramassables, les Piliers et les Smicks (ennemis). Ensuite, ajoutez l'affichage de cet environnement dans la *vue*, en affichant la bonne image selon l'état du modèle.

Ajoutez incrémentalement les processus métiers afin que la partie de Gyromite soit jouable :
- Gravité (le Prof. tombe si pas de sol sous ses pieds)
- Gestion des collisions (Prof. / Murs, Prof. / Smicks, Smicks / Murs, Smicks / Smicks)
- Capacité de monter/descendre aux cordes lorsque le Prof. est sur une corde
- Capacité de ramasser les objets (Bombes, Bonus)
- Capacité de faire se déplacer les Piliers + collisions Piliers (une entité écrasée par un pilier est tuée)
  Bonus : le Prof. doit être déplacé en même temps s'il se situe au sommet d'un Pilier
- Système de points
- Règles du jeu (mort, fin du niveau…)

3.4 Ajouter une ou plusieurs extensions suivant votre avancement

Au moins **deux** extensions parmi la liste suivante (une seule si l'extension est *conséquente*, par ex. le multijoueur en réseau) :

- Une extension que vous proposez vous-même ;

- Plusieurs niveaux : faites attention, une méthode de génération à base de `if (niveau == 1) else if (niveau == 2)

else if (…)` n’est pas très propre ;

- Éditeur/Générateur de Niveau ;

- Meilleurs scores ;

- Capacité de ramasser et déposer des Radis ; un Radis distrait un Smick en cas de collision pendant quelques secondes ;

- Niveau « scrollables » (plus grands que l’écran) ;

- Jeu collaboratif sur un même plateau ou en réseau : vous êtes libres d’avoir deux professeurs Hector, d’avoir un PC qui contrôle les déplacements du professeur et un autre PC qui contrôle le déplacement des colonnes, amusez-vous ;

- Ajouter des Colonnes avec déplacement horizontal : faites attention de bien gérer tous les déplacements et toutes les collisions associés ;

- etc.

4 Évaluation

4.1 Présentation / Démo

Vous devrez présenter, en binôme, votre projet lors d’une soutenance :

- Quelques minutes de présentation (montrer le fonctionnement, préciser les choix de conception) ;

- Choissisez une partie de votre analyse pour l’expliquer / justifier (1-2 min)

- Quelques minutes de questions **individuelles** (les deux binômes doivent avoir compris **l’ensemble** du code, et pouvoir répondre aux questions) ;

Vous devrez montrer (si votre bande passante le permet) le rendu final du jeu et votre code ; pas de diaporama.

4.2 Rendu

Vous devrez rendre, sur Tomuss, une archive ***au format Zip*** contenant :
- le code source de votre projet : code Java, ressources (images, sons…) + librairies éventuelles
- un Jar compilé de votre projet (vérifier qu’il fonctionne depuis n’importe quelle machine et charge correctement les ressources)
- un rapport ***au format PDF***
Merci de rendre une archive « propre » (rapport à la racine, un dossier pour le projet, pas d’éléments inutiles tels que les fichiers objets). Merci de ne pas rendre de .rar, ni de .tar.gz, ni de générer un fichier avec une de ces extensions puis de le renommer en .zip.

Le rapport (8 pages maximum) doit contenir :
- la liste des fonctionnalités et extensions (proportion de temps associée à chacune d'elles)
- une documentation UML (au minimum un diagramme de classes)
- la justification de votre analyse (choix de conception)
- des copies d'écran d’une partie en cours (en particulier les fonctionnalités que vous souhaitez mettre en valeur)