Projet d'Algorithmique et de Développement Logiciel

Les Aventuriers du Rail

Incrément 2

Séances 10 et 11

Sommaire

1 Préambule

L'incrément 2 de PADL s'appuie sur la version produite pendant l'incrément 1 du jeu des Aventuriers du Rail (ADR).

L'objectif principal du second incrément est d'implémenter le changement de joueur, c'est-à-dire, pouvoir identifier le joueur courant et passer au joueur suivant.

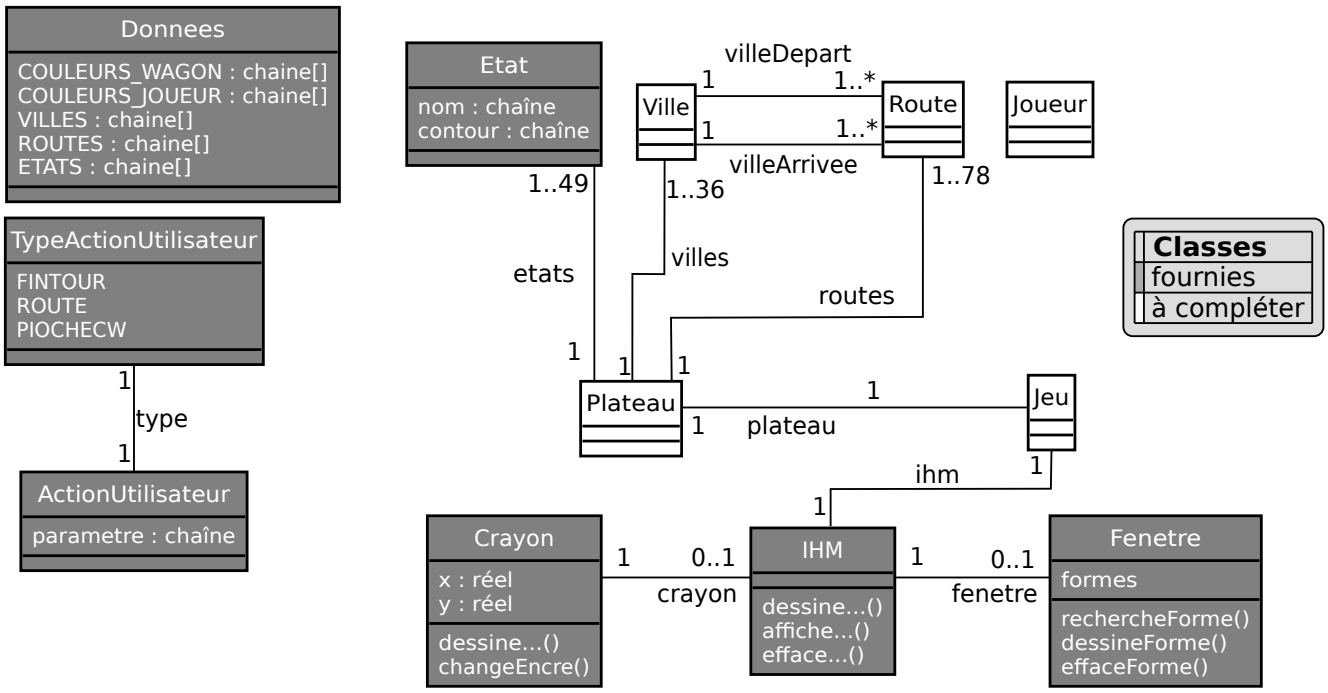

Figure 1: : Classes identifiées dans le jeu des ADR - Incrément 2

L'implémentation de l'incrément 2 se concentre sur la notion de **Joueur** et de **Jeu** (voir le diagramme de classes UML simplifié et incomplet en Illustration 1) :

- un **Joueur** qui permet de stocker les informations des joueurs dans le jeu (e.g. score, nombre de wagons)
- un **Jeu** qui définit et initialise les joueurs d'une partie et implémente un tour de jeu.

De nouveaux « utilitaires » (voir Illustration 1) sont nécessaires pour modéliser le comportement du jeu :

- les actions des utilisateurs (**ActionUtilisateur** *et* **TypeActionUtilisateur**)

2 Évaluation du travail

Le second incrément mesure vos compétences à choisir un mode de représentation pour stocker une information, et à utiliser cette information dans des algorithmes.

3 Travail demandé

Le changement de joueur dans une partie des ADR implique la définition d'un objet **Joueur**. Cette **classe Joueur** est utilisée dans le programme pour représenter chacun des joueurs d'une partie d'ADR.

On utilise l'objet **Joueur** et des informations supplémentaires pour identifier le joueur qui joue et « passer au joueur suivant ».

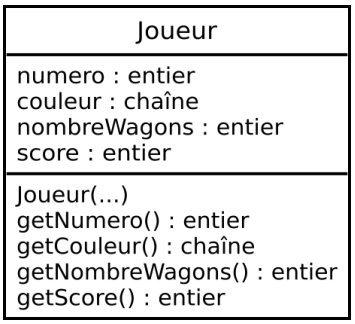

Figure 2: Diagramme de classe de la classe Joueur

3.1 Préparation de l'environnement

Importez l'incrément 2 dans votre projet existant (incrément 1) en suivant la procédure décrite dans la documentation du logiciel DevCube.

3.2 La classe Joueur

La classe **Joueur** représente un joueur du jeu des ADR. Le joueur est l'élément central de la mécanique de jeu des ADR : chaque joueur peut réaliser une et une seule action pendant son tour de jeu et la main est ensuite donnée au joueur suivant.

On définit un joueur au travers de son numéro, de sa couleur, de son nombre de wagons restants et de son score.

Ces informations sont stockées dans les attributs de la classe et interviennent dans les méthodes présentées dans la figure 2.

3.2.1 Définir la classe Joueur

Travail à faire

Définir la classe `Joueur` telle qu'elle est représentée dans la figure 2.

3.2.2 Accesseurs

Travail à faire

Implémenter les accesseurs de la classe **Joueur** pour chacun des attributs (voir figure 2).

3.2.3 Création d'un joueur et initialisation

Pour chaque joueur, il est nécessaire de créer un nouvel objet de type **Joueur**.

Dans le cas de la classe **Joueur**, on souhaite créer et initialiser un joueur à partir d'un numéro et d'une couleur.

Le nombre de wagons du joueur et son score sont initialisés dans le constructeur, respectivement aux valeurs 45 (voir Règles du jeu) et 0.

Travail à faire

Écrire le constructeur `Joueur()` de la classe **Joueur**.

3.2.4 Vérifier l'implémentation de la classe Joueur

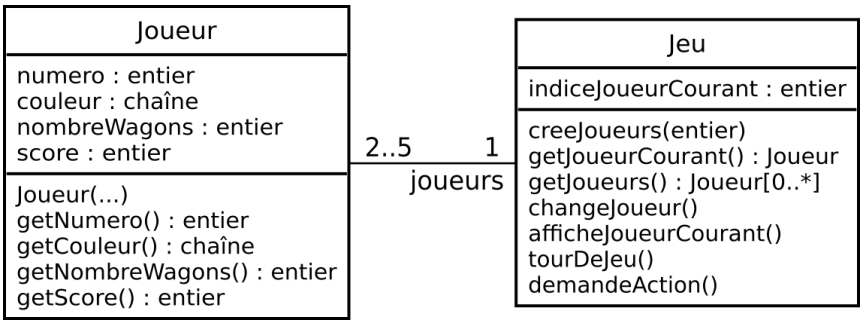

Figure 3: Diagramme de classe de la classe Jeu

Travail à faire

Exécuter la classe **TestJoueurInitialisation** pour valider l'implémentation de votre classe **Joueur**.
Si vous rencontrez des erreurs pendant la compilation ou l'exécution, corrigez-les avant de poursuivre.

3.3 La classe Jeu

Dans cet incrément, la classe **Jeu** se dote de deux nouveaux attributs Java pour, d'une part, stocker les joueurs du jeu, et d'autre part stocker l'indice du joueur courant.

Il est nécessaire de définir de nouvelles méthodes pour 1) créer et initialiser les joueurs du jeu, 2) changer de joueur, et 3) réagir aux évènements du jeu : on ajoute également 4) une méthode pour accéder au joueur courant, 5) une méthode pour afficher le joueur courant et 6) une méthode qui contient l'implémentation de l'algorithme du tour de jeu.

3.3.1 Modifier la classe Jeu

Travail à faire

Déclarer les nouveaux attributs et implémenter les accesseurs de la classe **Jeu**, présentée dans la figure 3.

3.3.2 Accesseurs

L'attribut `indiceJoueurCourant` sert à stocker une valeur entière qui correspond à l'indice du joueur dans le tableau `joueurs`.

Dans le jeu des ADR, l'information utile est le joueur courant et non l'indice du tableau qui lui correspond. L'accès à l'indice du joueur courant au travers de sa méthode « accesseur » n'est donc pas souhaité, on lui préfère l'accès direct au joueur courant au travers de la méthode `getJoueurCourant()` dont le fonctionnement est similaire à un accesseur.

Travail à faire

Implémenter l'accesseur de la classe **Jeu** pour le tableau `joueurs`.

Implémenter la méthode `getJoueurCourant()` dans la classe **Jeu** qui renvoie le joueur dont l'indice correspond à la valeur de l'attribut `indiceJoueurCourant`.

3.3.3 Méthode creeJoueurs

La méthode `creeJoueurs()` a pour objectif de créer un nombre de joueurs pour le jeu des ADR. Pour chaque joueur de la partie, on attribue un numéro et une couleur sélectionnée dans la liste des couleurs des joueurs de la classe **Donnees**.

La méthode `creeJoueurs()` définit, de plus, le premier joueur à jouer : par défaut, le joueur initialisé en premier est donc le joueur courant.

Travail à faire

Écrire la méthode `creeJoueurs()` qui crée et initialise un nombre de joueurs donné en paramètre.

La méthode initialise le tableau `joueurs`, crée autant d'objets **Joueur** que le nombre de joueurs requis et stocke les objets dans le tableau `joueurs`.

Lors de la création de l'objet **Joueur**, on lui affecte une couleur sélectionnée dans le tableau **COULEURS_JOUEUR** de la classe **Données**.

La méthode `creeJoueurs()` se termine par l'affectation de l'indice du joueur courant à la valeur 0 (premier joueur par défaut).

3.3.4 Vérifier l'implémentation de la méthode `creeJoueurs()`

Travail à faire

Exécuter la classe **TestJeuCreerJoueur** pour valider l'implémentation de la méthode `creeJoueurs()`.
Si vous rencontrez des erreurs pendant la compilation ou l'exécution, corrigez-les avant de poursuivre.

3.3.5 Vérifier l'implémentation de la méthode `getJoueurCourant()`

Travail à faire

Exécuter la classe **TestJeuJoueurCourant** pour valider l'implémentation de la méthode `getJoueurCourant()`.
Si vous rencontrez des erreurs pendant la compilation ou l'exécution, corrigez-les avant de poursuivre.

3.3.6 Méthode changeJoueur

La méthode `changeJoueur()` a la charge de modifier le joueur courant de l'application.

A chaque exécution, la méthode agit sur l'indice du joueur courant, stocké dans l'attribut `indiceJoueurCourant` de la classe **Jeu**.

En fonction du nombre de joueurs déclaré, le jeu passe du joueur 0 (par défaut) au joueur 1, puis au joueur 2 puis au joueur 3 et ainsi de suite jusqu'à revenir au joueur 0.

Travail à faire

Écrire la méthode `changeJoueur()` qui permet de modifier le joueur courant.

3.3.7 Vérifier l'implémentation de la méthode `changeJoueur()`

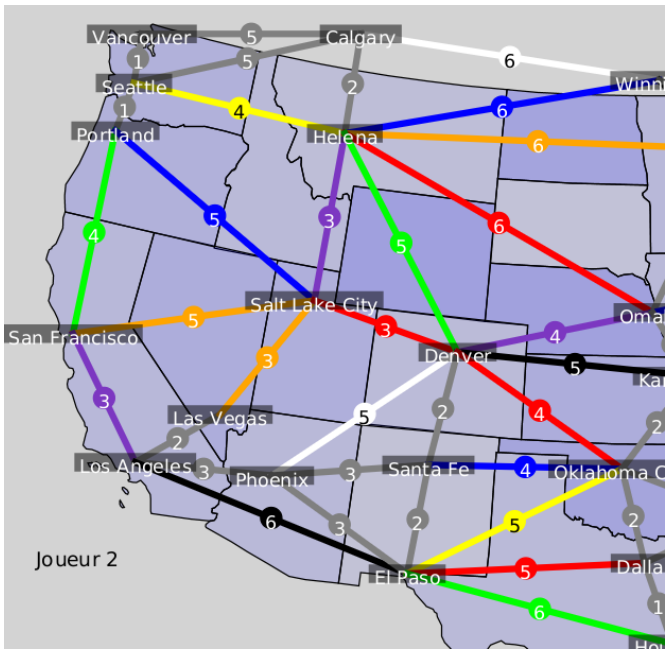

Figure 4: L'indice du joueur courant est affiché sur l'interface graphique

Travail à faire

S'inspirer des classes de test fournies (e.g. **TestJeu**, **TestPlateau**) et compléter la classe **TestJeuChangeJoueur.**

Valider l'implémentation de la méthode `changeJoueur()`.
Si vous rencontrez des erreurs pendant la compilation ou l'exécution, corrigez-les avant de poursuivre.

3.3.8 Méthode afficheJoueurCourant

La méthode `afficheJoueurCourant()` affiche les informations du joueur sur l'interface graphique. Dans cette version, on s'applique à afficher le numéro du joueur courant pour valider (visuellement) le changement de joueur.

Travail à faire

Écrire la méthode `afficheJoueurCourant()` qui permet d'afficher, sur le plateau de jeu, l'indice du joueur courant tel que présenté en illustration 4.

Pour afficher les informations du joueur, on utilise la méthode `afficheInformation()` de la classe **IHM**.

Se référer à la documentation pour plus d'informations.

3.3.9 Changer de joueur

Dans la version du jeu des ADR que l'on produit, c'est le joueur courant qui décide de terminer son tour et de passer la main au joueur suivant. Lorsque le joueur le décide, il utilise le bouton « joueur suivant »(voir illustration 5) fourni par l'interface graphique pour signaler la fin de son tour.

Le bouton « joueur suivant » est utilisé pour appeler la méthode `demandeAction()` qui va réagir à la demande de fin du tour du joueur et ainsi provoquer le changement de joueur dans le programme.

Réagir à une demande de l'utilisateur

Commençons par expliquer le principe général :

Figure 5: Bouton « joueur suivant » fourni par l'interface graphique.

1. Le programme affiche une interface graphique
2. L'utilisateur réalise une action sur l'interface graphique (clic sur la fenêtre du programme)
3. Le programme détecte l'action (clic sur un élément de l'interface graphique)
4. Le programme analyse l'action et vérifie qu'elle correspond aux actions autorisées
5. Le programme identifie l'action autorisée
6. Le programme effectue le traitement correspondant à l'action demandée
7. Le programme affiche un résultat

Sur ces 7 étapes, résumées ici de façon très simple, les étapes 1 à 5 sont réalisées par la classe **IHM** fournie dans le projet. Le travail restant à faire consiste à implémenter les étapes 6 et 7 pour « obtenir » un résultat et le transmettre à l'utilisateur.

La méthode `demandeAction()`, dont le code est présenté dans le listing 1, illustre les étapes 1 à 4.

```
1  void demandeAction() { // FOURNI
2    this.ihm.effaceBoutons();
3    this.ihm.dessineBoutonSuivant(false);
4    ActionUtilisateur reponse = this.ihm.attenteActionJoueur();
5    switch(reponse.getType()){
6      case FINTOUR : // A COMPLETER
7        break;
8      default : break;
9  }
10 }
```

Listing 1: Methode demandeAction() de la classe Jeu

Les lignes 2 et 3 affichent sur l'interface graphique le bouton « joueur suivant » qui peut détecter un clic « souris » de la part de l'utilisateur.

Le programme attend ensuite que l'utilisateur clique sur l'interface graphique et analyse l'action de l'utilisateur (ligne 4).

Si l'utilisateur a cliqué sur le bouton « joueur suivant » (action « FINTOUR »), le programme doit réagir en exécutant un bloc de code qui produit un résultat (ligne 5 et 6).

Dans l'incrément 2, la méthode `demandeAction()` ne réagit qu'au changement de joueur. Elle sera modifiée dans les incréments suivants pour réagir aux autres évènements du jeu (e.g. sélection d'une route, pioche de carte) en suivant le même principe que pour la fin du tour.

Travail à faire

Compléter la méthode `demandeAction()` en appelant le changement de joueur lorsque l'évènement « Fin du tour » est détecté.

Pour ceux qui veulent comprendre comment le programme détecte un évènement dans le jeu des ADR (optionnel)

L'implémentation de la méthode `demandeAction()` dépend de la manière dont les actions du joueur sont capturées par le programme.

*Dans l'interface graphique qui vous est fournie par la classe **IHM**, tous les éléments graphiques affichés (e.g. routes, villes, boutons) possèdent un identifiant.*

*Lorsque l'utilisateur du programme clique sur un de ces éléments avec la souris, l'interface graphique renvoie l'identifiant de l'élément « sélectionné ». On analyse alors l'identifiant pour détecter si l'élément « cliqué » correspond à un type d'action permise dans le jeu (e.g. fin du tour, sélection d'une route, pioche d'une carte). Les types d'actions sont listés dans la classe **TypeActionUtilisateur**.*

*Chaque clic « souris » crée également un objet de la classe **ActionUtilisateur** (ligne 4 - listing 1) qui contient le type d'action demandée par l'utilisateur. Le type de l'action est stocké dans l'attribut `type` de la classe **ActionUtilisateur**.*

On filtre alors les actions effectuées en fonction de leur type (ligne 5 - listing 1) et on peut « réagir »en fonction du type d'action détecté (ligne 6 - listing 1).

3.3.10 Tour de jeu

Le tour de jeu d'un joueur est implémenté dans une méthode spécifique, nommée `tourDeJeu()`. Cette méthode demande l'affichage du joueur courant (méthode `afficheJoueurCourant()`) et autorise le joueur courant à effectuer une action pendant son tour (méthode `demandeAction()`). On veille également à effacer les informations du joueur avant la fin du tour (méthode `effaceInformation()` de la classe **IHM**).

Travail à faire

Utiliser les méthodes déclarées dans les sections précédentes pour implémenter la méthode `tourDeJeu()`.

3.3.11 Initialisation du jeu

La méthode `main()` de la classe Jeu doit être modifiée pour y inclure l'initialisation des joueurs et le tour de jeu.

Le principe d'une partie des ADR est de faire jouer les joueurs, chacun leur tour, jusqu'au dernier tour de jeu (voir Règles du jeu). La détection de ce dernier tour de jeu fera l'objet d'une implémentation spécifique dans l'incrément 4. L'incrément 2 propose donc un programme qui initialise et démarre une partie sans que celle-ci ne puisse être terminée.

Pour anticiper les futurs développements du jeu, il est recommandé de mettre en place certains mécanismes nécessaires à la « bonne » gestion du dernier tour. Pour ce faire, nous allons utiliser un nouvel attribut booléen, initialisé à « faux », qui servira de **condition d'itération**.

On modélise le déroulement d'une partie dans l'algorithme suivant :

```
Creer 5 joueurs (methode creeJoueurs())
Afficher le plateau de jeu (increment 1)
Tantque le dernier tour n'est pas detecte faire
  un tour de jeu (methode tourDeJeu())
Fin tantque
```

Travail à faire

Déclarer l'attribut `dernierTour` de type booléen dans la classe **Jeu**.

Modifier la méthode `main()` pour initialiser, démarrer, puis jouer une partie.

3.3.12 Exécution du jeu

Travail à faire

Exécuter la classe **Jeu** et observer le résultat de votre implémentation.
En cas de problème, vérifier les implémentations du joueur et du jeu pour effectuer des corrections si nécessaire.

4 Analyse du code ADR

Pour concrétiser et mieux comprendre le fonctionnement des actions, il est demandé de fournir une analyse et une conception préliminaire des classes **ActionUtilisateur** et **TypeActionUtilisateur**.

Travail à faire

Proposer un diagramme de classes, similaire à la figure 3, qui liste les attributs et les associations entre ces classes.

Pour la classe Java **TypeActionUtilisateur**, trouver la représentation UML que l'on utilise pour modéliser des énumérations.

5 Fin de l'incrément 2

Vous avez atteint le principal objectif qui est de pouvoir changer de joueur pendant la partie.

En fonction du temps qu'il vous reste avant l'incrément suivant, vous pouvez :

- Réaliser les extensions de l'incrément 2 (optionnel - section 6)
- Travailler sur l'implémentation de la classe **CarteWagon** (en auto-évaluation)

6 Extensions

6.1 Sélection aléatoire du joueur courant en début de partie

Travail à faire

Modifier la méthode **creeJoueur()** pour choisir aléatoirement quel joueur commence à jouer lors d'une partie.

L'implémentation de cette extension modifie le comportement du jeu et provoque des erreurs dans les tests associés. Après vérification manuelle du comportement de la méthode **creeJoueur()**, ne pas tenir compte de ces erreurs.

6.2 Couleur aléatoire des joueurs

Travail à faire

Modifier la méthode **creeJoueur()** pour que la couleur d'un joueur soit choisie aléatoirement.

Veiller à ce que le programme ne puisse attribuer la même couleur à deux joueurs différents.

L'implémentation de cette extension modifie le comportement du jeu et provoque des erreurs dans les tests associés. Après vérification manuelle du comportement de la méthode **creeJoueur()**, ne pas tenir compte de ces erreurs.