IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP11-15 – Trains et circuits

Fondements théoriques du développement des logiciels concurrents

Objectifs

À la fin de l'activité, les élèves devront être capables de :

- modifier un diagramme de classes (ajouter des méthodes et/ou attributs à une partie des classes existantes) de petite taille (moins de 10 classes) pour satisfaire de nouvelles spécifications ;
- appliquer une méthode systématique de réalisation de la synchronisation de *threads* basée sur l'invariant de sûreté. La méthode implique la modification de plusieurs classes ;
- utiliser les moniteurs Java pour mettre en œuvre le blocage/déblocage de *threads* dans un programme concurrent.

Le contexte

Le contexte dans lequel nous allons travailler est celui des trains. Dans ce TP, nous allons nous intéresser au déplacement de ces trains (de manière sûre) sur une ligne.

Une ligne de chemin de fer est composée d'éléments qui peuvent être des gares ou des sections de ligne. Une ligne commence et se termine par une gare. Nous allons nous limiter à des lignes qui sont composées uniquement de sections de ligne sur la partie intermédiaire. Une gare peut contenir plusieurs quais.

Par ailleurs, un train est à tout instant sur un élément de la ligne. À l'instant d'après, il ne peut que se situer sur un élément adjacent (celui à gauche ou celui à droite selon le sens dans lequel il se déplace). Les trains ne peuvent partir initialement que d'une gare. Une section de ligne ne peut recevoir qu'un seul train.

1 Le modèle objet de départ

Nous partirons de la solution suivante qui modélise la notion de ligne de chemin de fer et de train tels que décrits précédemment. Une ligne (`Railway`) est composée d'une suite d'éléments (`Element`). Chaque élément appartient à une unique ligne. Les éléments peuvent être des sections

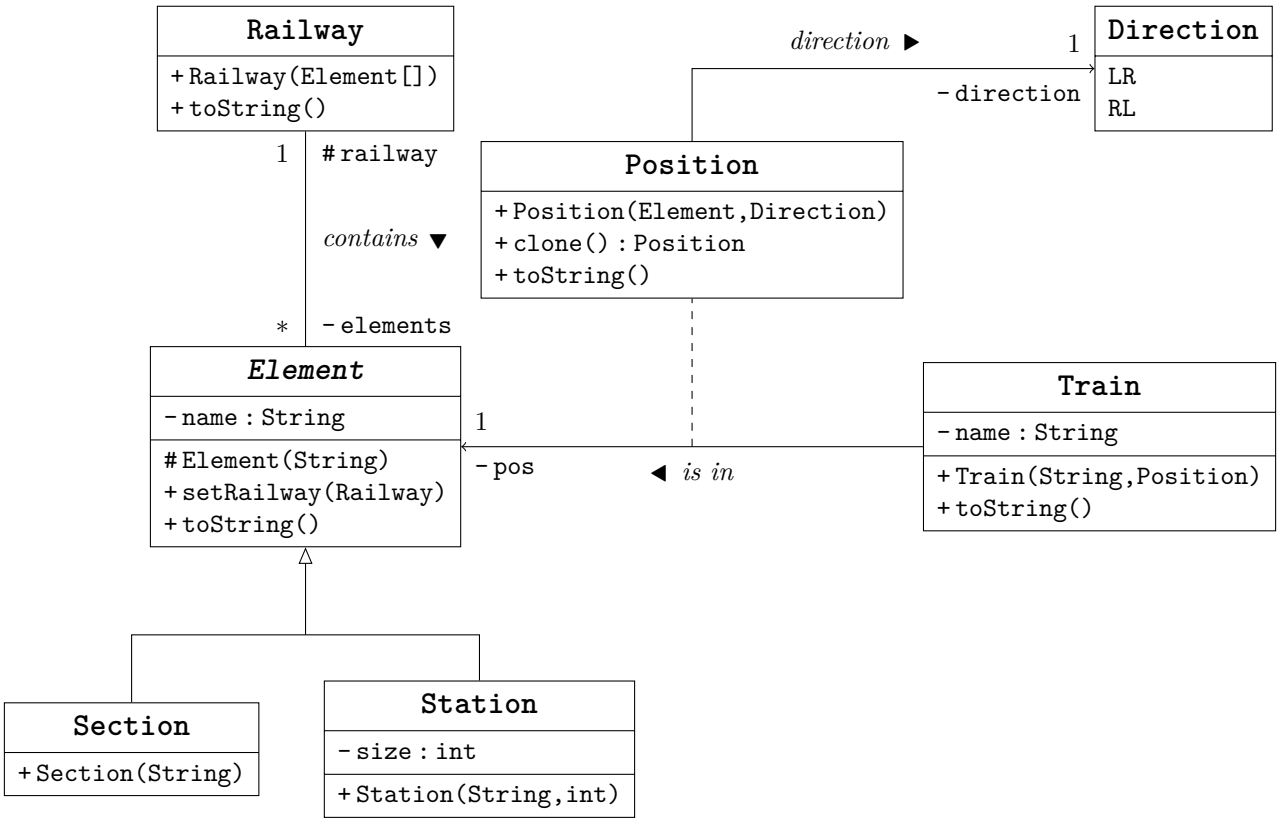

FIGURE 1 – Diagramme de classes pour des lignes de train et la présence de trains

de ligne (`Section`) ou des gares (`Station`). Les gares peuvent contenir plusieurs quais pouvant recevoir des trains (codés sous la forme d'un attribut `size`). Un train (`Train`) peut être positionné sur un seul élément de la ligne et un élément ne peut avoir qu'un seul train. De plus, la position d'un train sur un élément de la ligne est modélisée par une classe d'association (`Position`) qui connaît le sens dans lequel le train avance (`Direction`).

Le diagramme de classes de ce modèle est donné figure 1. Le code des différentes classes se trouve sur Moodle.

La classe `Main` suivante illustre la façon de construire une ligne :

```
public class Main {
  public static void main(String[] args) {
    Station A = new Station("GareA", 3);
    Station D = new Station("GareD", 3);
    Section AB = new Section("AB");
    Section BC = new Section("BC");
    Section CD = new Section("CD");
    Railway r = new Railway(new Element[] { A, AB, BC, CD, D });
    System.out.println("The railway is:");
    System.out.println("\t" + r);
    Position p = new Position(A, Direction.LR);
    try {
      Train t1 = new Train("1", p);
      Train t2 = new Train("2", p);
      Train t3 = new Train("3", p);
      System.out.println(t1);
```

```
      System.out.println(t2);
      System.out.println(t3);
    } catch (BadPositionForTrainException e) {
      System.out.println("Le train " + e.getMessage());
    }
  }
}
```

L'affichage à l'exécution du `Main` est :

```
The railway is:
 GareA--AB--BC--CD--GareD
```

2 Le déplacement des trains

Pour programmer le déplacement des trains sur la ligne, nous allons procéder en deux étapes. Dans un premier temps, nous allons programmer le comportement d'**un seul** train. Dans un second temps, nous permettrons à plusieurs trains de circuler sur une même ligne.

Exercice 1 (*Le comportement d'un train*)

Pour décrire le comportement du train, nous allons contraindre son déplacement sur la ligne. Pour le *main* précédemment montré, on a par exemple :

1. le train est en GareA,
2. le train sort de la gare et entre dans la section AB,
3. le train sort de AB et entre dans la section BC,
4. le train sort de BC et entre dans la section CD,
5. le train sort de CD et entre dans la gare GareD,
6. le train se retourne,
7. ...

▷ **Question 1.1 :**
Dans le diagramme de classes précédent, quel sera le rôle de chaque classe dans la réalisation du déplacement d'un train ?

▷ **Question 1.2 :**
Modifiez le diagramme de classes initial en ajoutant les méthodes et/ou attributs nécessaires à la réalisation du déplacement d'un train.

▷ **Question 1.3 :**
Donnez le code des méthodes identifiées. Pour valider le bon fonctionnement de vos méthodes, vous pouvez afficher l'état du train chaque fois qu'il change de position.

Exercice 2 (*Plusieurs trains sur la ligne*)

▷ **Question 2.1 :**
Modifiez votre programme pour qu'il puisse y avoir plusieurs trains actifs (en déplacement) sur la ligne.

Lorsque plusieurs trains sont en déplacement sur la ligne, il faut s'assurer qu'aucun accident ne puisse arriver. Pour cela il faut garantir que :

- les trains qui circulent dans le même sens ne se doublent pas : il faut garantir qu'il y a au maximum un train dans une section ;
- si un train est en déplacement dans un sens, aucun autre train n'est en déplacement dans le sens contraire.

Ces deux conditions forment l'*invariant de sûreté* du programme. Il doit être satisfait à tout instant de l'exécution de celui-ci. Les *threads* qui exécutent la tâche *train* doivent se synchroniser pour l'assurer.

Pour réaliser la synchronisation de *threads*, il faut :

- Identifier les variables qui permettent d'exprimer l'invariant de sûreté : l'état du système qui doit être garanti.
- Identifier les actions *critiques* (qui modifient l'invariant de sûreté) : ces actions sont implantées comme des méthodes dans le cas d'une solution de synchronisation avec des moniteurs.
- Les actions identifiées doivent être appelées en exclusion mutuelle puisqu'elles modifient l'invariant et donc les variables qui le représentent : les méthodes correspondant aux actions seront `synchronized` dans une solution de synchronisation avec des moniteurs.
- Identifier, pour chaque action critique, la condition d'attente pour la synchronisation (pour assurer que l'invariant de sûreté est satisfait) : chaque condition est implantée comme une méthode dont la valeur de retour est un `boolean`.

C'est dans l'identification des conditions d'attente pour la synchronisation que l'invariant de sûreté peut être utilisé. En particulier, le code de la méthode qui implante la condition d'attente suit la structure suivante :

- on *simule* l'exécution de l'action (les valeurs de variables correspondantes sont modifiées) ;
- on évalue si l'invariant de sûreté reste satisfait avec ces nouvelles valeurs ;
- si l'invariant n'est plus satisfait, on restaure l'ancienne valeur des variables (on annule l'action) et on bloque le *thread*.

Dans un premier temps, nous allons simplifier l'invariant de sûreté de manière à garantir que :

- le nombre de trains maximum dans une gare est égal au nombre de quais de la gare ;
- dans une section il y a au maximum un train.

▷ **Question 2.2 :**
Identifiez les variables qui permettent d'exprimer l'invariant de sûreté pour la ligne de trains.

▷ **Question 2.3 :**
À l'aide des variables identifiées, exprimez l'invariant de sûreté.

▷ **Question 2.4 :**
Quelles sont les actions « critiques » que peut effectuer un train ?

▷ **Question 2.5 :**
Dans quelles classes ces actions doivent être ajoutées ?

▷ **Question 2.6 :**
Selon la méthode de construction d'une solution de synchronisation donnée plus haut, quelles autres méthodes faut-il ajouter et dans quelle classe ?

▷ **Question 2.7 :**
Ajoutez les méthodes identifiées dans les classes correspondantes.

▷ **Question 2.8 :**
Modifiez maintenant le comportement d'un train pour qu'il utilise les méthodes ajoutées. Testez le bon fonctionnement de votre solution en démarrant l'exécution de un, puis deux, puis trois trains.

Exercice 3 (*Éviter les interblocages…*)

Au terme de l'exercice précédent, si vous avez plus d'une section entre les gares vous pouvez arriver dans une situation d'interblocage (*deadlock*). Si un train A est entré dans une section dans un sens (par exemple de gauche à droite) et un autre train B s'engage dans la section qui sera la destination de A (celle à sa droite) dans le sens inverse (de droite à gauche), on arrive à bloquer le système. En effet, A attend que B libère sa section alors B attend que A libère la sienne.

Une solution pour résoudre ce problème est d'empêcher les sorties de gare si des trains dans l'autre sens sont sur la ligne.

▷ **Question 3.1 :**
Identifiez les variables qui permettent d'exprimer la nouvelle condition.

▷ **Question 3.2 :**
À l'aide des nouvelles variables, identifiez la nouvelle condition pour l'invariant de sûreté.

▷ **Question 3.3 :**
Quelle est la classe responsable de la gestion de ces variables ?

▷ **Question 3.4 :**
Utilisez la méthode de construction d'une solution de synchronisation présentée dans l'exercice précédent pour tenir compte de cette nouvelle condition.

▷ **Question 3.5 :**
Modifiez les méthodes `leave` et `enter` de la classe `Section` pour tenir compte de la nouvelle condition. Testez votre solution.

3 Pour aller plus loin

Exercice 4 (*Gare intermédiaire*)

Le but de cet ultime exercice est d'ajouter une gare intermédiaire au milieu d'une suite de sections. Une telle gare permet à des trains de se croiser.

▷ **Question 4.1 :**
Modifiez votre code pour permettre d'ajouter des gares intermédiaires.

▷ **Question 4.2 :**
Constatez que vous devez ajouter un nouvel invariant de sûreté pour éviter un interblocage si la gare intermédiaire a n places et qu'il y a $n+2$ trains. Déterminer ce

nouvel invariant.

▷ **Question 4.3 :**
Modifiez votre code pour l'assurer.

Annexes

Code Java des classes

```
public abstract class Element {
  private final String name;
  protected Railway railway;
  protected Element(String name) {
    if(name == null)
      throw new NullPointerException();
    this.name = name;
  }
  public void setRailway(Railway r) {
    if(r == null)
      throw new NullPointerException();
    this.railway = r;
  }
  @Override
  public String toString() {
    return this.name;
  }
}
```

```
public class Railway {
  private final Element[] elements;
  public Railway(Element[] elements) {
    if(elements == null)
      throw new NullPointerException();
    this.elements = elements;
    for (Element e : elements)
      e.setRailway(this);
  }
  @Override
  public String toString() {
    StringBuilder result = new StringBuilder();
    boolean first = true;
    for (Element e : this.elements) {
      if (first)
        first = false;
```

```
      else
        result.append("--");
      result.append(e);
    }
    return result.toString();
  }
}
```

```
public class Section extends Element {
  public Section(String name) {
    super(name);
  }
}
```

```
public class Station extends Element {
  private final int size;
  public Station(String name, int size) {
    super(name);
    if(name == null || size <=0)
      throw new NullPointerException();
    this.size = size;
  }
}
```

```
public class Train {
  private final String name;
  private final Position pos;
  public Train(String name, Position p) throws BadPositionForTrainException {
    if (name == null || p == null)
      throw new NullPointerException();
    // A train should be first be in a station
    if (!(p.getPos() instanceof Station))
      throw new BadPositionForTrainException(name);
    this.name = name;
    this.pos = p.clone();
  }
  @Override
  public String toString() {
    StringBuilder result = new StringBuilder("Train[");
    result.append(this.name);
    result.append("]");
    result.append(" is on ");
    result.append(this.pos);
    return result.toString();
  }
}
```

```
public class Position implements Cloneable {
  private final Direction direction;
  private final Element pos;
```

```
  public Position(Element elt, Direction d) {
    if (elt == null || d == null)
      throw new NullPointerException();
    this.pos = elt;
    this.direction = d;
  }
  @Override
  public Position clone() {
    try {
      return (Position) super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
      return null;
    }
  }
  public Element getPos() {
    return pos;
  }
  @Override
  public String toString() {
    StringBuilder result = new StringBuilder(this.pos.toString());
    result.append(" going ");
    result.append(this.direction);
    return result.toString();
  }
}
```

```
public enum Direction {
  LR {
    @Override
    public String toString() {
      return "from left to right";
    }
  },
  RL {
    @Override
    public String toString() {
      return "from right to left";
    }
  };
}
```