SOMMAIRE

Exercice1

1. **le rôle de chaque classe dans la réalisation du déplacement d'un train**

- La classe Railway représente le support de déplacement du train.
- La classe Element représente les différents constituants de la ligne de train. Elle représente les différents points de déplacement de notre train. Elle permet notamment de connaître la trajectoire du train.
- La classe Section hérite de la classe Element et représente une partie de la ligne de train: Notre train se déplace donc d'une section à une autre
- La classe Station hérite également de la classe Element et permet de définir les deux extrémités de notre ligne de train.
- La classe Position permet de savoir où le train se situe à chaque étape de son déplacement.
- La classe Direction permet de connaître le sens de déplacement de notre train
- La classe Train représente notre train en lui-même

2. **Modélisation du diagramme de classe**
   (voir annexe)

3. **Code des méthodes identifiées**
   - **la méthode getNext() de la classe Railway**

Cette méthode permet de récupérer l'élément suivant à parcourir pour un train en fonction de sa position courante et de sa direction de circulation.

```
public Element getNext(Position pos) {

    Element current = pos.getPos();
    Direction direction = pos.getDirection();

    for (int i = 0; i < elements.length; i++) {
        if (elements[i] == current) {
            if (direction == Direction.LR && i + 1 < elements.length) return elements[i + 1];
            if (direction == Direction.RL && i - 1 >= 0) return elements[i - 1];
        }
    }
    return null;

}
```

- **la methode updatePosition() dans la classe Position**

```
/** La fonction updatePosition permet de modifier d'obtenir la position
 * actuelle pour pouvoir conserver la position initiale du train */
public void updatePosition(Element elt, Direction direction) {
    this.pos = elt;
    this.direction = direction;
}
```

- **les méthodes enter() et leave() de la classe Section**

Un train peut entrer ou sortir d'une section. Un attribut train permet de déterminer si un train est présent dans une section.

```
public synchronized void enter(Train train) throws Exception {
        if (this.train != null) {

        }
        this.train = train;
        System.out.println("le train "+train.getName()+" entre dans la section "+ this.toString());
    }

    public synchronized void leave(Train train) {
        if (this.train == train) {
            System.out.println("le train "+train.getName()+" sort de la section "+ this.toString());
            this.train = null;
        }
    }
```

- **les méthodes enter() et leave() de la classe Station**

Un train peut entrer dans une station si il y'a encore des quais libres dans celle-ci. On utilise donc une variable *occupied* qui compte le nombre de trains présents dans la gare afin d'empêcher d'autres trains d'entrer dans une gare saturée. Un train peut sortir d'une station si il y était présent.

```
public synchronized void enter(Train train) throws Exception {
    if (occupied >= size) {
        throw new Exception("Station is full.");
    }
    occupied++;
    System.out.println("le train "+train.getName()+ " entre dans la gare "+ this.toString());

}

public synchronized void leave(Train train) {
    if (occupied > 0) {
        System.out.println("le train "+  train.getName()+" sort dans la gare "+ this.toString());
        occupied--;

    }
}
```

- **La méthode move() de la classe Train**

Cette méthode permet de traduire le déplacement d'un élément à un autre. Pour avancer un train, on récupère la position et la direction du train et on cherche tant que celà est possible l'élément suivant qui devient la nouvelle position du train.

```
public void move() throws Exception {
        Railway railway = pos.getPos().railway;
        Element nextElement = railway.getNext(pos);

        if (nextElement != null) {
            pos.getPos().leave(this);  // Quitter l'élément actuel
            nextElement.enter(this);       // Entrer dans le nouvel élément
            pos.updatePosition(nextElement, pos.getDirection());
        } else {

        }
    }
```

- **La méthode run() de la classe Train**

Plusieurs trains doivent pouvoir fonctionner en même temps. La classe Train réalise donc l'interface Runnable et implémente la méthode run().

```
@Override
    public void run() {
        while (true) {
            try {
                move();
                Thread.sleep(2000); // Pause pour simuler le temps de déplacement
            } catch (Exception e) {
                System.out.println(name + " rencontre une erreur dans son déplacement: " + e.getMessage());
                break;
            }
        }
    }
```

<u>Test de bon fonctionnement</u>

En reprenant l'exemple fourni dans le main, lorsqu'un train est créé et assigné à une position et une ligne de train, on obtient le résultat suivant:

```
The railway is:
        GareA--AB--BC--CD--GareD
Train[1] is on GareA going from left to right
le train 1 entre dans la section AB
le train 1 sort de la section AB
le train 1 entre dans la section BC
le train 1 sort de la section BC
le train 1 entre dans la section CD
le train 1 sort de la section CD
le train 1 entre dans la gare GareD
```

Un train arrive à circuler d'une gare à une autre et on récupère sa trace.

Exercice 2

1. **Les variables qui permettent d'exprimer l'invariant de sûreté pour la ligne de trains.**

    - **train** (dans la classe *Section*): cet attribut permet de référencer un train occupant une section. Il est null lorsqu'il n'y a pas de train dans la section.
    - **occupied** (dans la classe *Station*) : Cet attribut permet de compter le nombre de trains dans la gare.
    - **size** (dans la classe *Station)*: Il permet de connaître la capacité maximale d'une gare.
    - **directionOccupied** (dans la classe Section): Cet attribut indique si un train est en déplacement dans une section afin de garantir qu'un train venant en sens inverse ne puisse pas accéder à une section déjà occupée par un train navigant dans l'autre sens.

2. **Expression de l'invariant de sûreté**
   L'invariant de sûreté doit permettre de garantir que :
    - Le nombre de trains maximum dans une gare est égal au nombre de quais de la gare .
      Expression de l'invariant de sûreté: station.occupied < station.size
    - Dans une section il y a au maximum un train:

Pour cette partie, il faut d'abord vérifier que deux trains circulant dans la même direction, ne peuvent pas avoir accès à une section simultanément (dépassement) , mais aussi que deux trains circulant dans des directions opposées ne peuvent pas partager la même section (collision)

- empêcher le dépassement: Le train ne peut accéder à une section que lorsque celle - ci ne contient pas de trains

  section.train == null
- empêcher les collisions: Un train ne peut accéder à une section que s'il n'y a aucun train dans cette section circulant en sens inverse dans cette section ou lorsque la section est occupée par un train allant dans la même direction.

  section.directionOccupied == null || directionOccupied == train.getDirection()

Expression de l'invariant de sûreté:

section.train == null && (section.directionOccupied == null || directionOccupied == train.getDirection())

3. **Les actions critiques que peut effectuer un train**
   Ce sont les actions *enter()* et *leave()* qui traduisent l'entrée et la sortie d'un train dans un élément. Ces actions vont pouvoir modifier les variables qui servent à exprimer l'invariant de sûreté. Elles seront donc réalisées comme des méthodes synchronized.

4. **Dans quelles classes ces actions doivent-être ajoutées**
   Ces actions sont propres aux classes **Section** et **Station.**

5. **Quelles autres méthodes faut-il ajouter et dans quelle classe**
   - Un méthode booléenne qui vérifie à tout instant qu'un autre train circule en sens inverse (dans la classe Section): Cette méthode permet de vérifier si un train en sens inverse attend dans la section et ainsi empêcher un autre train d'accéder à cette section.

6. **Les méthodes écrites et/ou modifiées**
   - **la méthode enter() de la classe Section**

```
/** La méthode enter() doit pouvoir vérifier qu'un seul train ne peut rentrer dans la section
 * Cette méthode doit aussi prendre en compte le fait qu'un autre train peut vouloi entrer par le sens opposé*/
    public synchronized void enter(Train train) throws InterruptedException {
        while(this.train != null || (directionOccupied != null && directionOccupied != train.getPosition().getDirection())) {
            System.out.println("le train "+ train.getName()+" ne peut entrer dans la section "+ this.toString()+
            " car elle est occupée par le train "+ this.train.getName());
            wait();
        }

        this.train= train;
        directionOccupied = train.getPosition().getDirection();
        System.out.println("le train "+train.getName()+" entre dans la section "+ this.toString());

    }
```

   - **La méthode leave() et la méthode OppositeMove() de la classe section**

La méthode leave permet à un train de quitter une section et notifier les autres trains qui attendent que celle-ci soit libre.
La méthode OppositeMove() qui permet d'identifier un train qui arrive dans un autre sens.

```
public synchronized void leave(Train train) {
    if(this.train==train) {
        System.out.println("le train "+train.getName()+" sort de la section "+ this.toString());
        this.train= null;
        directionOccupied = null;
        notifyAll();
    }


}
 public synchronized boolean OppositeMove(Direction trainDirection) {
        return (this.train != null && directionOccupied != trainDirection);
    }
```

- **La méthode enter() de la classe Station**

```
@Override
/** Un train peut entrer dans une station tant qu'il y'a un quai disponible */
public synchronized void enter(Train train) throws InterruptedException {

    while(occupied>=size) {
        System.out.println("le train "+ train.getName()+" ne peut entrer dans la gare "+ this.toString()+" car tous les quais sont occupés");
        wait();
    }
    occupied ++;
    System.out.println("le train "+train.getName()+ " entre dans la gare "+ this.toString());

}
```

- **Le méthode leave() de la classe Station**

```
@Override
/** Un train peut quitter une station à tout moment*/
public synchronized void leave(Train train) {
    if (occupied> 0) {
        System.out.println("le train "+  train.getName()+" sort dans la gare "+ this.toString());
        notifyAll();
    }


}
```

- **La méthode move() de la classe Train()**

Cette méthode reprend la méthode move() de l'exercice 1 et permet à un ou plusieurs trains de se déplacer en respectant les invariants de sûreté.

```
public void move() throws Exception {
    Element currentElement = pos.getPos();
    Element nextElement = railway.getNext(pos);
    if (nextElement != null) {
        synchronized(nextElement) {
            // Vérifier si un train en sens inverse attend dans la section
            while (nextElement instanceof Section && ((Section) nextElement).OppositeMove(pos.getDirection())) {
                System.out.println("Le train " + name + " attend car la section " + nextElement + " est occupée par un train en sens inverse.");
                nextElement.wait(); // Attendre que la section soit libérée
            }
            currentElement.leave(this);
            nextElement.enter(this);
            pos.updatePosition(nextElement, pos.getDirection());

        }
```

Test du bon fonctionnement

On crée trois trains , 2 circulant dans la même direction (le train 1 et 2) et le train 3 circulant en sens inverse

```
The railway is:
        GareA--AB--BC--CD--GareD
le train 2 entre dans la section AB
le train 3 entre dans la section CD
le train 1 ne peut entrer dans la section AB car elle est occup□e par le train 2
le train 3 sort de la section CD
le train 3 entre dans la section BC
Le train 2 attend car la section BC est occup□e par un train en sens inverse.
Le train 3 attend car la section AB est occup□e par un train en sens inverse.
```

Les trains 2 et 3 entament leur mouvement mais restent bloqués respectivement en AB et en BC car ils ne peuvent pas se dépasser au risque de rentrer en collision. On obtient une situation d'interblocage. Le train1 ne peut pas quitter la gareA car le train 2 occupe toujours la section AB

```
The railway is:
        GareA--AB--BC--CD--GareD
le train 1 entre dans la section AB
le train 3 entre dans la section CD
le train 2 ne peut entrer dans la section AB car elle est occup□e par le train 1
le train 1 sort de la section AB
le train 1 entre dans la section BC
le train 2 entre dans la section AB
Le train 3 attend car la section BC est occup□e par un train en sens inverse.
Le train 1 attend car la section CD est occup□e par un train en sens inverse.
le train 2 sort de la section AB
le train 2 ne peut entrer dans la section BC car elle est occup□e par le train 1
```

Dans ce 2ème cas, le train 1 et 2 arrivent à avancer sans partager la même section et se bloquent respectivement en BC et en AB. Le train 3 quant à lui avance jusqu'à la section CD ou il reste bloqué car la section BC est occupée par le train 1.

On remarque donc que tant qu'un train circule en sens inverse on aboutit à une situation d'interblocage

Exercice 3: Éviter les interblocages

Pour résoudre le problème d'interblocage, on peut empêcher les sorties de gare si des trains dans l'autre sens sont sur la ligne.

,

1. **Les variables qui permettent d'exprimer la nouvelle condition**

Pour empêcher les sorties de gare, il faudra connaître si un train circule déjà dans une des deux directions. Les variables clés sont:

- isDirectionOccupiedLR: qui indique si un train circule de gauche à droite
- isDirectionOccupiedRL: qui indique si un train circule de droite à gauche
- activeTrainsRL : qui compte le nombre de trains circulant dans la direction RL et pouvoir garantir que tous les trains circulant dans la direction RL arrivent en gare avant qu'un train en sens inverse ne puisse démarrer.
- activeTrainsLR: qui compte le nombre de trains circulant dans la direction LR et pouvoir garantir que tous les trains circulant dans la direction LR arrivent en gare avant qu'un train en sens inverse ne puisse démarrer.

2. **La nouvelle condition pour l'invariant de sûreté.**

Un train peut partir seulement si aucun train ne s'est déjà engagé dans l'autre sens.
Expression de l'invariant de sûreté:

((direction == Direction.LR && (isDirectionOccupiedRL || activeTrainsRL>0)) || (direction == Direction.RL && (isDirectionOccupiedLR||activeTrainsLR>0)))

3. Classe responsable de la gestion de ces variables

La classe responsable de ces variables est la classe Railway car c'est elle qui voit l'ensemble de la ligne et peut imposer une règle globale.

4. Utilisation d'une solution de synchronisation

On peut créer deux méthodes askEntry() et releaseEntry(). Elles vont pouvoir modifier les variables qui servent à exprimer l'invariant de sûreté. Elles seront donc réalisées comme des méthodes synchronized.

- **askEntry():**

Cette méthode bloque un train tant qu'un autre circule dans le sens opposé. Elle permet aussi d'accorder une priorité à un train si un autre train circule déjà dans le même sens.

```
/** askEntry() bloque un train tant qu'un autre circule dans le sens opposé */
 public synchronized void askEntry(Direction dir) throws InterruptedException {
        // Attendre tant qu'un train circule déjà dans l'autre sens
        while ((dir == Direction.LR && (isDirectionOccupiedRL || activeTrainsRL>0)) ||
               (dir == Direction.RL && (isDirectionOccupiedLR||activeTrainsLR>0))) {

            wait();
        }
        // Bloquer la direction pour empêcher d'autres trains de démarrer en sens inverse
        if (dir == Direction.LR) {
            isDirectionOccupiedLR = true;
            activeTrainsLR++;
        } else {
            isDirectionOccupiedRL = true;
            activeTrainsRL++;

        }
    }
```

- **releaseEntry():**

Cette méthode permet de libérer la ligne à l'arrivée de tous les trains circulant dans un même sens pour permettre à d'autres trains de circuler en sens inverse .

```
/** releaseEntry() bloque un train tant qu'un autre circule dans le sens opposé */
public synchronized void releaseEntry(Direction direction) {
       if (direction == Direction.LR) {
           activeTrainsLR--;
           if (activeTrainsLR == 0) {
           //  Si il n'y a plus de trains sur cette ligne circulant dans la direction LR, libérer la ligne
               isDirectionOccupiedLR = false;
               notifyAll(); //  Réveiller les trains RL en attente
           }
       } else {
           activeTrainsRL--;
           if (activeTrainsRL == 0) {
               //  Si il n'y a plus de trains sur cette ligne circulant dans la direction RL, libérer la ligne
               isDirectionOccupiedRL = false;
               notifyAll(); //  Réveiller les trains LR en attente
           }
       }
   }
```

5. Modification des méthodes appropriées.

Selon notre conception, les méthodes enter() et leave() de la classe Section, ne prennent en compte que l'état local de chaque section, et non l'état global de la ligne.

L'objectif de **askEntry()** est d'empêcher un train en sens inverse de démarrer tant qu'un autre train est en circulation. Si cette méthode était placée dans **enter()**, elle ne s'appliquerait qu'à une seule section à la fois. Un train pourrait alors entrer dans une première section, puis libérer cette section dans **leave()**, permettant ainsi à un train en sens inverse de commencer son trajet avant que le premier train ne soit arrivé à destination. Cela créerait un risque d'interblocage, où deux trains partiraient simultanément et se retrouveraient face à face sur la ligne, incapables d'avancer.

À l'inverse, en plaçant **askEntry()** au début du trajet, dans **run()**, nous verrouillons la ligne entière avant même que le train ne commence à bouger. Cela garantit qu'aucun train en sens inverse ne pourra partir tant que la ligne n'est pas complètement libre. De même, **releaseEntry()**, placé dans **move()**, ne libère la ligne que lorsque le train arrive à destination, et non à chaque section traversée.

- **la méthode run()**

Au début de son exécution, la méthode run() demande l'accès exclusif à la ligne via requestEntry(), empêchant ainsi un train en sens inverse de démarrer simultanément. Ensuite, tant que le train n'a pas atteint sa destination, il avance section par section en appelant move(), avec une pause de 2 secondes entre chaque déplacement pour simuler le trajet. Lorsqu'il arrive à la gare finale, il libère la ligne avec releaseEntry(), permettant à d'autres trains de circuler. Enfin, la boucle s'arrête automatiquement (hasArrived = true), garantissant que le train ne continue pas à s'exécuter inutilement.

```
@Override
public void run() {

        try {
            railway.askEntry(pos.getDirection());//lorsqu'un train veut circuler il demande d'abord à verrouiller
            while(!isArrived) {
            move();
            Thread.sleep(2000);}
        }catch(Exception e) {
            System.out.println(name +"rencontre une erreur dans son déplacement:" + e.getMessage());

        }
}
```

- **la méthode move()**

La méthode move() permet au train d'avancer section par section jusqu'à sa destination. D'abord, elle vérifie si le train est déjà arrivé afin d'éviter tout déplacement inutile. Ensuite, elle récupère la section suivante et s'assure qu'elle est libre avant d'y entrer, en attendant si nécessaire. Une fois la section accessible, le train quitte son emplacement actuel, entre dans la nouvelle section et met à jour sa position. Si aucune section suivante n'existe, cela signifie que le train est arrivé à destination : il libère alors la ligne en appelant releaseEntry() et marque son arrivée avec hasArrived = true, arrêtant ainsi tout déplacement.

```
public void move() throws Exception {
    if (isArrived) return;
    Element currentElement = pos.getPos();
    Element nextElement = railway.getNext(pos);
    if (nextElement != null) {
        synchronized(nextElement) {

            currentElement.leave(this);
            nextElement.enter(this);
            pos.updatePosition(nextElement, pos.getDirection());

        }

    }else {
        System.out.println("Le train " + name + " est arrivé à destination et libère la ligne.");
        railway.releaseEntry(pos.getDirection()); // le train libère la ligne à l'arrivée
        isArrived = true;
    }

}
```

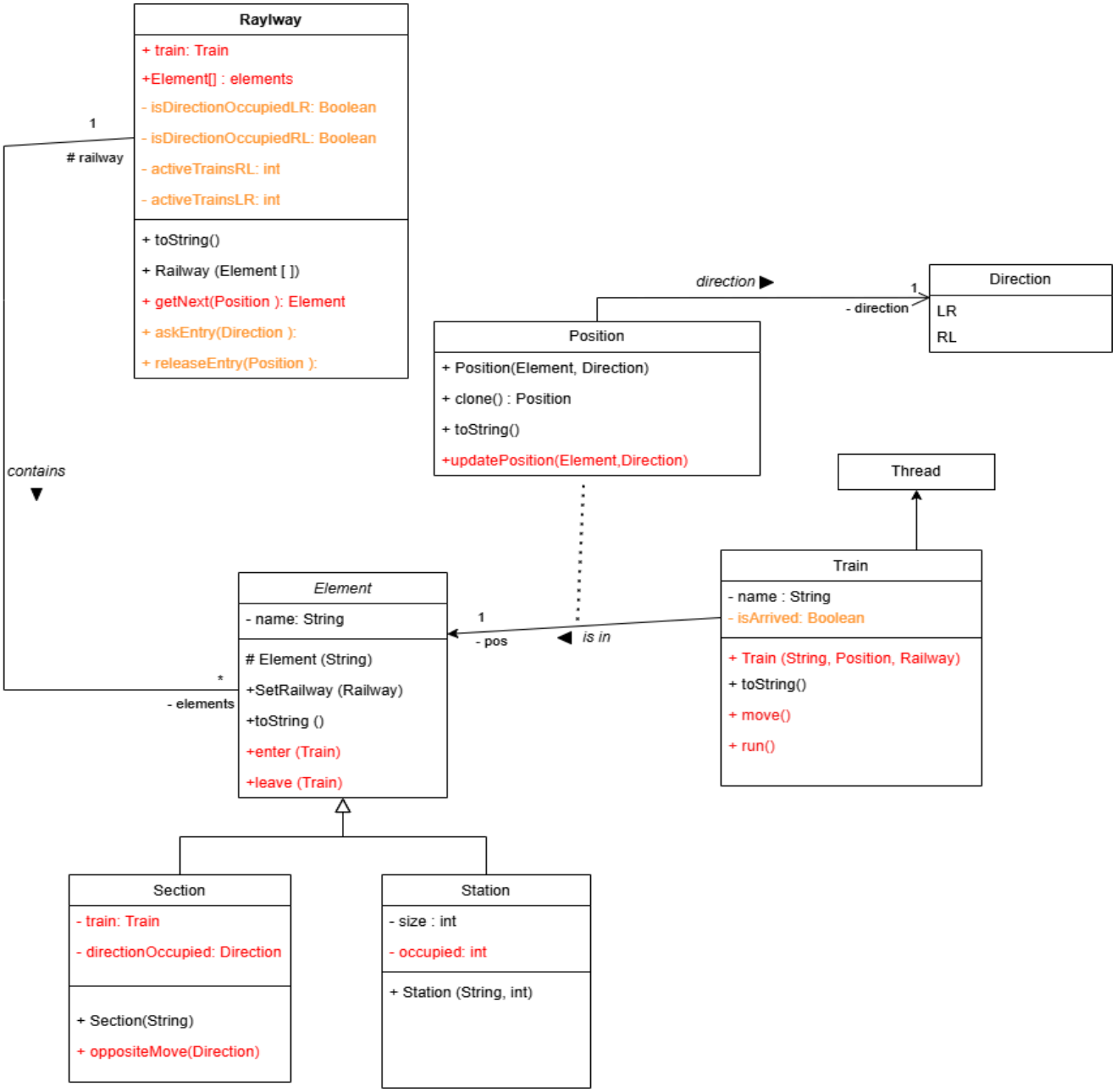
Test du bon fonctionnement

On crée trois trains , 2 circulant dans la même direction LR (le train 1 et 2) et le train 3 circulant en sens inverse RL.

Les trains 1 et 2 circulent section par section sans se dépasser et le train 3 quant à lui ne démarre pas jusqu'à ce que les trains 1 et 2 libèrent la ligne.

```
The railway is:
        GareA--AB--BC--CD--GareD
Train[1] is on GareA going from left to right
Train[2] is on GareA going from left to right
Train[3] is on GareD going from right to left
le train 1 entre dans la section AB
le train 2 ne peut entrer dans la section AB car elle est occup□e par le train 1
le train 1 sort de la section AB
le train 1 entre dans la section BC
le train 2 entre dans la section AB
le train 2 sort de la section AB
le train 2 ne peut entrer dans la section BC car elle est occup□e par le train 1
le train 1 sort de la section BC
le train 1 entre dans la section CD
le train 2 entre dans la section BC
le train 1 sort de la section CD
le train 1 entre dans la gare GareD
le train 2 sort de la section BC
le train 2 entre dans la section CD
le train 2 sort de la section CD
le train 2 entre dans la gare GareD
Le train 1 est arriv□ □ destination et lib□re la ligne.
Le train 2 est arriv□ □ destination et lib□re la ligne.
le train 3 sort dans la gare GareD
le train 3 entre dans la section CD
le train 3 sort de la section CD
le train 3 entre dans la section BC
le train 3 sort de la section BC
le train 3 entre dans la section AB
le train 3 sort de la section AB
le train 3 entre dans la gare GareA
Le train 3 est arriv□ □ destination et lib□re la ligne.
```

ANNEXE:



Le diagramme de classe ci-dessus est le diagramme de classe final du projet. Ce diagramme a évolué tout au long du TP.
Les méthodes et attributs ajoutés en rouge représentent les ajouts faits pour l'exercice 1 et 2.
Les méthodes/attributs en orange représentent les ajouts faits pour répondre au problème de l'exercice 3.
NB: Afin de ne pas surcharger les tableaux, nous supposons que dans chaque classe, des getters ont été créés afin de pouvoir récupérer les attributs privés de ces classes.
De même, les méthodes ajoutées ont été expliquées plus haut et commentées dans le programme.