Projet Trains et Circuits – Nicolas Sempéré

**Remarque : les fichiers LTS (situés dans le dossier « Fichier LTS ») contiennent des commentaires et des informations pertinentes pour comprendre le fonctionnement de l’implémentation en Java.**

Table des matières

[Exercice 1 - Le comportement d’un train 1](#_Toc125826987)

[Exercice 2 (Partie A) - Plusieurs trains sur la ligne 5](#_Toc125826991)

[Exercice 2 (Partie B) - Contrôleur 6](#_Toc125827000)

[Exercice 3 - Éviter les interblocages 8](#_Toc125827001)

[Exercice 4 - Gare intermédiaire 9](#_Toc125827007)

# Exercice 1 - Le comportement d’un train

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

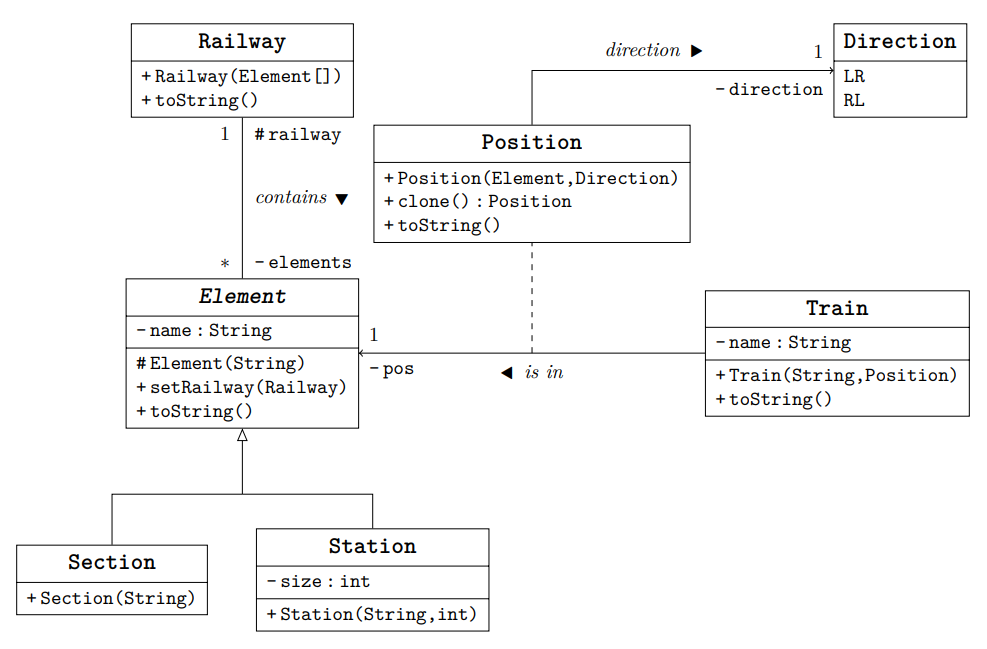


Diagramme de classe initial (fourni dans le sujet)

La **classe Train** va avoir un comportement actif : on lui ajoute des méthodes lui permettant de se déplacer.

La **classe Position** permet de commander le déplacement du train : en fonction de sa position et de sa direction.

La **classe Railway** permet de définir la ligne de chemin de fer (composée d’éléments), sur lesquels le train se déplace.

Le chemin de fer (**classe Railway**) est de la forme suivante : Gare A – Section 2 – Section 3 – Gare B. Les trains (**classe Train**) sont initialement en Gare A. Le déplacement (**classe Position**) d’un train est le suivant : il part de la Gare A, arrive en Section 2, puis en Section 3, et enfin en Gare B ; il fait ensuite demi-tour, arrive en Section 3, puis en Section 2, et enfin en Gare A ; fait demi-tour et répète ce mouvement.

## ⊳ 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.**

Les méthodes et attributs ajoutées sont écrits en bleu.

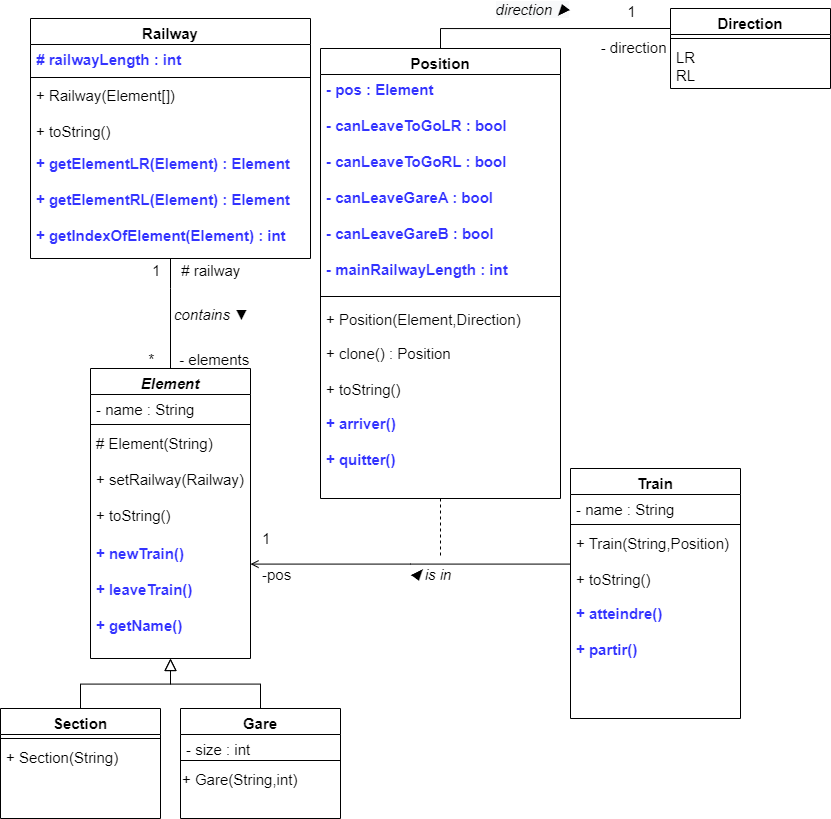


Diagramme de classe – Déplacement d’un unique train

**Le déplacement d'un train est divisé en deux temps :**

* **Etape 0 : Le train se trouve sur un élément.**
* **Etape 1 : Le train quitte cet élément,**
* **Etape 2 : Le train arrive à l'élément suivant.**

**Lorsqu'un train quitte un élément, il est "en route" vers l'élément suivant. Durant ce temps de déplacement (nul dans cette modélisation), le train se trouve sur une "liaison" entre deux éléments.  
Par exemple, si la ligne de chemin de fer est de la forme : Gare A – Section 2 – Section 3 – Gare B, il y a quatre éléments, et trois liaisons (représentées par des tirets).**

Pour réaliser le déplacement d'un train :

Dans la classe "**train**", la méthode "**atteindre**()" est appelée :

- la méthode "**arriver**()" de la classe **Position** vérifie où le train peut se rendre (en fonction de sa position courante et de sa direction) ; on utilise notamment la méthode « **getIndexOfElement**() » de la classe **Railway** pour réaliser cette vérification.

Dans la classe "**train**", la méthode "**partir**()" est appelée :

- la méthode "**quitter**()" de la classe **Position** réalise concrètement le déplacement (la position du train est changée via cette méthode).

- les méthodes "**getElementLR**()" et "**getElementRL**()" de la classe **Railway** permettent d’obtenir la nouvelle position du train.

**Note importante :**

Un train est initialement en position 0 (il se trouve en GareAvantDeploiement). Cette Gare ne fait pas partie "réellement" de la ligne de chemin de fer. Elle sert uniquement à initialiser le déplacement d'un train. On pourrait initialiser la position des trains en Gare A, mais cela poserait problème dans l'appel de la méthode "arriver" de la classe Position.

En effet, la méthode « arriver » libère la liaison entre le précédent élément et l'élément où le train vient d'arriver : initialement, un train n'a pas "d'élément précédent". On ajoute donc une gare d'initialisation, nommée "GareAvantDeploiement". Lorsqu'un train se trouve dans cette gare, la méthode "arriver" envoie le train directement dans la gare A, en vérifiant uniquement le nombre de quais disponibles.

Au terme de l’exécution de la méthode "position", le train est prêt à circuler sur la ligne de chemin de fer. Il exécute répétitivement les méthodes « partir » et « atteindre ». Une fois en gare A il va suivre le comportement suivant :

* Il vérifie qu’il peut quitter la gare A (vérifications de plus en plus nombreuses d’un exercice à l’autre).
* S’il le peut, il quitte la gare A. Sinon, il attend que la ligne soit disponible.
* Il se retrouve en route vers la Section 2, c’est-à-dire sur la liaison entre la Gare A et la Section2 : le trajet est d’une durée nulle dans ce modèle.
* S’il le peut, il arrive en Section 2. Sinon, il attend (cas où il y a un autre train en Section 2).
* Il répète ce processus (de la Section 2 vers la Section 3, etc).

## ⊳ 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.**

Les méthodes ajoutées sont toutes dans le dossier src/train, elles sont répertoriées ci-dessous.

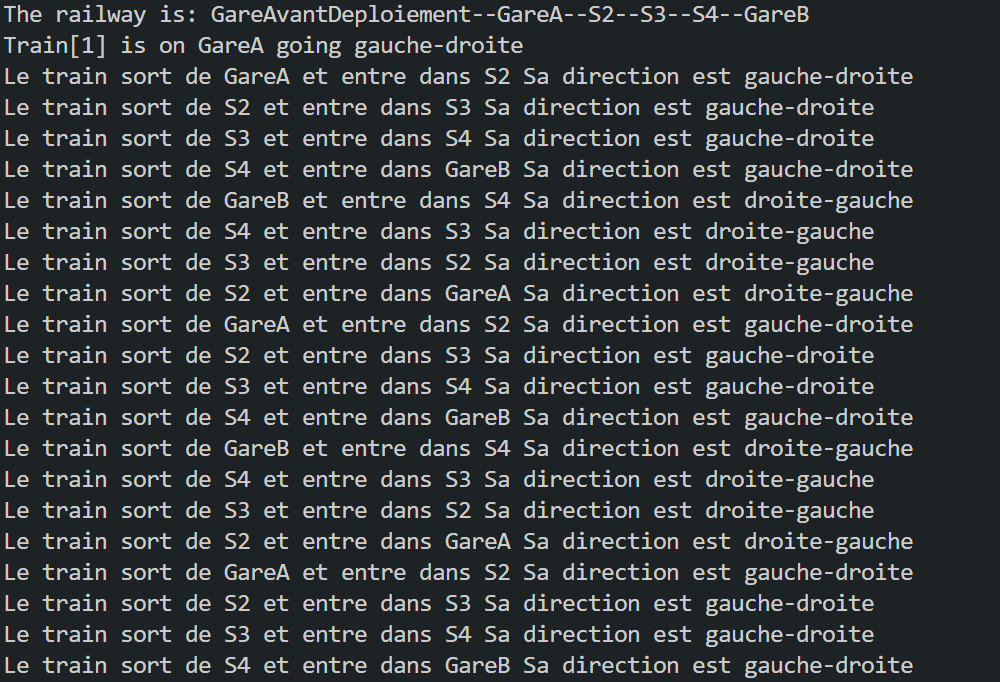
|  |  |  |  |
| --- | --- | --- | --- |
| Train | Position | Element | Railway |
| atteindre() | **arriver()** | **newTrain()** | **getElementLR()** |
| partir() | **quitter()** | **leaveTrain()** | **getElementRL()** |
|  |  | **getName()** | **getIndexOfElement()** |

Méthodes ajoutées pour réaliser le déplacement d’un train

On ajoute la méthode "**getName**()" à la classe **Element** afin d'afficher la position du train à tout instant.

On appelle les méthodes **atteindre**() et **partir**() 20 fois dans le **Main**. La position et la direction du train sont affichées dès que la position est modifiée.

**Le comportement du train est celui attendu : il part de la gare A, rejoint la gare B, fait demi-tour, rejoint la gare A, puis recommence ce parcours (cf figure ci-dessous).**

****

**Exemple d’exécution du Main de l’exercice 1.**

# Exercice 2 (Partie A) - 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.**

La **classe Train** implémente à présent **l’interface** **Runnable**. On ajoute dans la **classe** **Main** plusieurs instanciations de threads de la classe Train. On complète la **méthode** « **run**() » de la classe Train pour qu’il se déplace : les méthodes « atteindre() » et « partir() » sont appelées 15 fois successivement.

Cependant, les trains peuvent :

- être plus nombreux dans une gare que son nombre de quais (**Résolution en** **Partie A de l’exercice 2**),

- se retrouver à plusieurs sur la même section de rails (**Résolution en** **Partie A de l’exercice 2**),

- se doubler et se croiser (**Résolution en** **Partie B de l’exercice 2**).

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

Il y a deux invariants de sûreté à assurer (cf le sujet du projet) :

* 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.

Le premier est assuré dans la **classe Gare**: grâce aux **attributs** **size** et **quaisDispos** (des entiers).

Le second est assuré dans la **classe Section**: grâce à **l’attribut** **sectionDispo** (booléen).

## ⊳ Question 2.3 : **À l’aide des variables identifiées, exprimez l’invariant de sûreté.**

L’expression du premier invariant est la suivante : **0 <= quaisDispos <= size**.

Le deuxième invariant n’est pas vérifié dans le code Java : on utilise uniquement une condition d’attente qui est réalisée grâce à **l’attribut sectionDispo**.

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

Les deux actions critiques que peut effectuer un train sont « **newTrain**() » et « **leaveTrain**() » : le train arrive dans un élément (gare ou section), ou le quitte. Ces deux méthodes sont appelées dans la **classe Position**, dans les méthodes **quitter()** et **arriver()**.

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

Ces deux méthodes sont ajoutées dans la **classe** **abstraite** **Element**, puis redéfinies respectivement dans les **classes** **Gare** et **Section**.

## ⊳ 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 ?**

On ajoute les trois méthodes suivantes à la **classe Gare**:

* canNewTrain () : boolean
* canLeaveTrain() : boolean
* invariant() : boolean

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

Voir les **méthodes** des **classes** **Position, Element, Section, Gare.**

# Exercice 2 (Partie B) - Contrôleur

Cependant, les trains peuvent toujours se doubler et se croiser.

Exemple illustrant la possibilité de se croiser :

On considère la situation suivante (la ligne est : « Gare A – Section 2 – Section 3 – Gare B ») :

* Un Train 1 se positionne en Section 3, en direction de la Section 2.
* Un Train 2 se positionne en Section 2, en direction de la Section 3.
* Le Train 1 quitte la Section 3.
* Le Train 2 quitte la Section 2.
* *Les deux sections (2 et 3) se retrouvent alors vides (des trains peuvent y arriver).*
* Le Train 2 arrive en Section 3.
* Le Train 1 arrive en Section 2.
* **Les deux trains se sont croisés.**

Exemple illustrant la possibilité de se doubler :

On considère la situation suivante (la ligne est : « Gare A – Section 2 – Section 3 – Gare B ») :

* Un Train 1 se positionne en Section 2, en direction de la Section 3.
* Un Train 2 se positionne en Gare A, en direction de la Section 2.
* Le Train 1 quitte la Section 2.
* *La section 2 se retrouve alors vide (un train peut y arriver).*
* Le Train 2 quitte la Gare A.
* Le Train 2 arrive en Section 2.
* Le Train 2 quitte la Section 2.
* Le Train 2 arrive en Section 3.
* **Le Train 2 a doublé le Train 1.**

**Solution : processus « Controller » en FSP et attribut « Controller » dans la classe Railway.**

Un **contrôleur** (*cf le fichier LTS de l’exercice 2*) est ajouté. Il est implémenté en Java sous la forme d’une classe nommée « **controller** ». Le contrôleur consiste en un tableau d’entiers (valant 0 ou 1). A chaque élément de ce tableau correspond une **liaison** (représentée par un tiret « - ») entre deux éléments de la ligne de chemin de fer. Chaque élément de ce tableau empêche deux trains de se retrouver sur la même liaison.

Par exemple : si la ligne est : « Gare A – Section 2 – Section 3 – Gare B », alors **controller** est initialement [1,1,1]. Lorsqu’un train quitte la Section 2, vers la Section 3, il modifie le **controller** (**méthode inUse()**) qui vaut alors [1,0,1], empêchant tout autre train de quitter la Section 3 vers la Section 2.   
Lorsque le train arrive en Section 3, il modifie à nouveau le **controller** (**méthode free()**), qui vaut enfin [1,1,1].

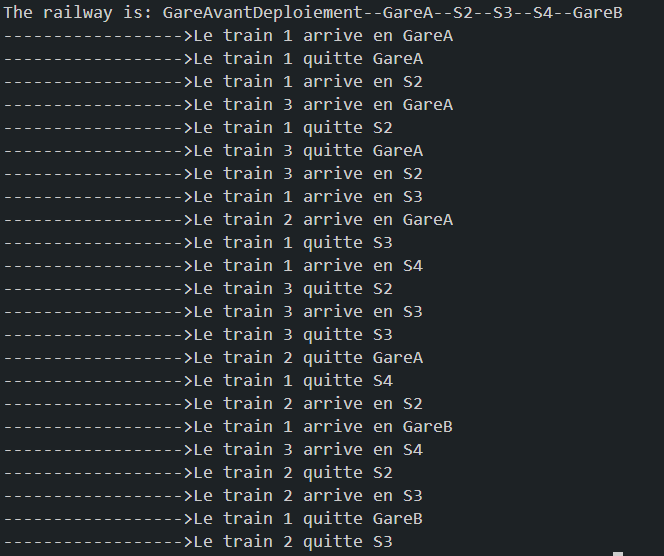
Les méthodes **inUse**() et **free**() sont **synchronisées**, et ont une condition d’attente similaire à celles vues précédemment dans la **classe Section**.

Ce contrôleur permet donc d’atteindre un interblocage (souhaité), empêchant des trains de se croiser ou se doubler. En reprenant l’exemple des trains qui se croisent : lorsque le Train 1 quitte la Section 3, le système atteint un interblocage : comportement souhaité étant donné que deux trains ne doivent pas se croiser (le Train 3 ne peut pas arriver en Section 2 car le Train 2 s’y trouve, et le Train 2 ne peut pas quitter la Section 2 car le contrôleur l’en empêche).

## ⊳ 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 d’un, puis de deux, puis de trois trains.**

**Pour observer le comportement du code Java a l’exercice 2, il suffit de lancer le Main.**

**Le comportement du programme est celui attendu : les trains ne se doublent pas, ne se croisent pas, et respectent le nombre maximal de trains qu’il peut y avoir sur chaque élément. Un interblocage finit par être atteint.**



**Exemple d’exécution du Main de l’exercice 2.**

# Exercice 3 - Éviter les interblocages

## ⊳ Question 3.1 : **Identifiez les variables qui permettent d’exprimer la nouvelle condition.**

Pour éviter l’interblocage qui survient en fin d’exercice 2, un processus **CONTROLLER\_AB** est créé en **FSP**, et une **classe** **ControllerAB** est créée en **Java**. Ce contrôleur empêche l’ensemble des trains de s’engager sur la ligne dans des sens opposés.

On ajoute deux **attributs** à la **classe** **ControllerAB** : **nbrTrainsLR** et **nbrTrainsRL**. Ces attributs (des entiers), correspondent au nombre de trains engagés entre les gares A et B. Le premier correspond au nombre de trains engagés dans le sens gauche-droite, le second correspond au sens droite-gauche.

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

**L’invariant de sûreté** est le suivant :

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

La classe responsable de la gestion de ces variables est la **classe** **ControllerAB**.

## ⊳ 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.**

On ajoute **sept** **méthodes** dans la **classe** **ControllerAB**:

* **newTrainLR**() : un train s’engage sur la ligne, dans le sens gauche-droite
* **newTrainRL**() : un train s’engage sur la ligne, dans le sens droite-gauche
* **arrivedTrainLR**() : un train arrive en gare B
* **arrivedTrainRL**() : un train arrive en gare A
* **canNewTrainLR**() : condition d’attente pour un train s’engageant dans le sens gauche-droite
* **canNewTrainRL**() : condition d’attente pour un train s’engageant dans le sens droite-gauche
* **invariant**() : vérifications quant à l’invariant

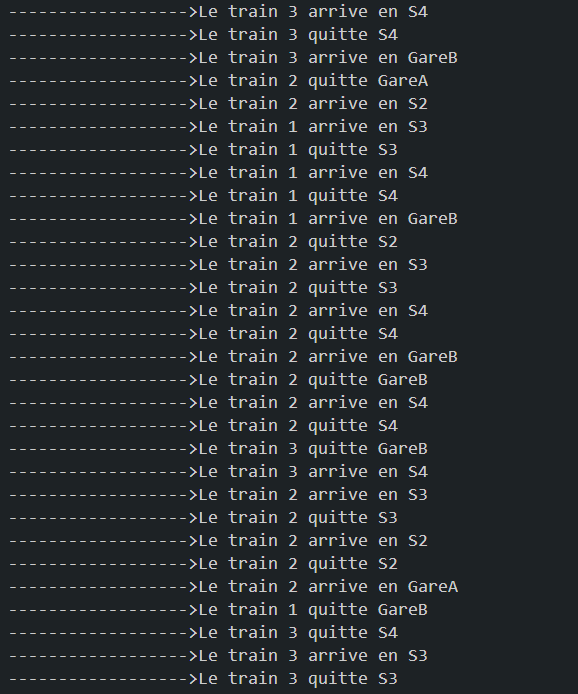
Les quatre premières méthodes : **newTrainLR**(), **newTrainRL**(), **arrivedTrainLR**(), **arrivedTrainRL**() sont **publiques** et **synchronisées**. Elles sont **appelées** depuis la **classe** **Position**.

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

Dans l’implémentation faite, la **classe** **Section** n’est pas modifiée. On utilise uniquement la **liaison** **dynamique** : la **classe** **Position** possède un **attribut** **ControllerAB**.

**Cette solution fonctionne comme attendu** : si les gares possèdent suffisamment de quais (plus que le nombre de train), le système ne possède plus d’interblocage.

**L’exécution du Main est « infinie », c’est-à-dire que les trains se déplacent jusqu’à ce qu’on arrête manuellement le programme.**



**Exemple d’exécution du Main de l’exercice 3.**

Cependant, s’il y a plus de trains que de quais par gare, le système se bloque : un train se retrouve en chemin vers une gare, qui se remplie durant son parcours. Ce problème est réglé en exercice 4.

# Exercice 4 - Gare intermédiaire

## ⊳ Question 4.1 : **Modifiez votre code pour permettre d’ajouter des gares intermédiaires**

On modifie le main (ajout de sections et de la gare M).

La ligne de chemin de fer est maintenant de la forme :

Gare A – Section 2 – Section 3 – Gare M – Section 5 – Section 6 – Gare B.

La Gare M possède QM quais.

## ⊳ 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.**

On arrive dans un interblocage lorsque :

* Un train est en déplacement gauche-droite vers la gare du milieu.
* Un train est en déplacement droite-gauche vers la gare du milieu.
* La gare du milieu est pleine (elle contient QM train à quai).

Pour éviter cet interblocage, on ajoute un troisième contrôleur « controller\_milieu » (cf le fichier LTS de l’exercice 4). Ce contrôleur permet d’avoir au maximum QM trains en déplacement entre les gares A et B.

On ajoute **quatre** **méthodes** à la **classe** **ControllerMilieu** :

* **newTrainToM**() : un train s’engage sur la ligne, depuis la gare A ou la gare B
* **arrivedTrainFromM**() : un train atteint la gare A ou la gare B.
* **canNewTrainToM**() : condition d’attente pour un train s’engageant sur la ligne
* **invariant**() : vérifications quant à l’invariant

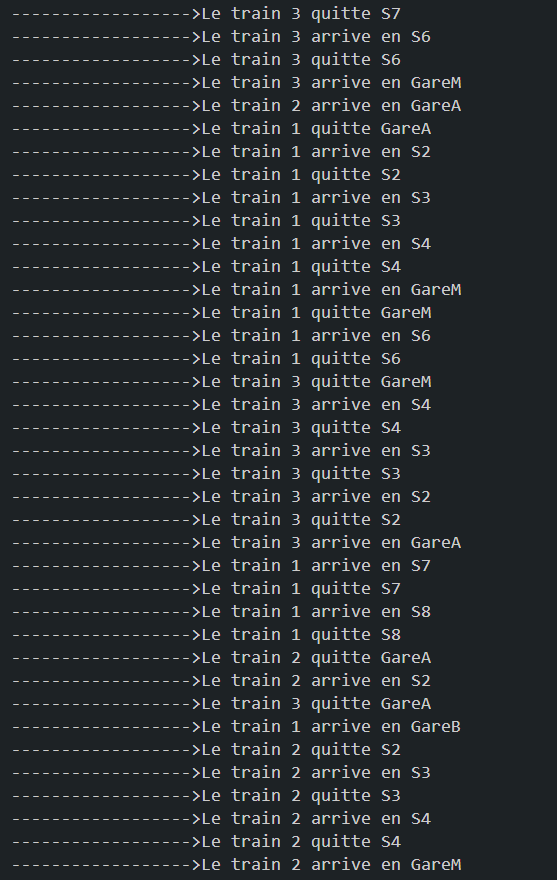
Les deux premières méthodes : **newTrainToM**(), **arrivedTrainFromM**(), sont **publiques** et **synchronisées**. Elles sont **appelées** depuis la **classe** **Position**.

On compte donc le nombre de trains qui s’engagent sur la ligne en direction de M (ceux qui sortent de la gare A, et ceux qui sortent de la gare B). Ce nombre est un **attribut** de la **classe** **ControllerMilieu** nommé **nbrTrainsToM.**

**L’invariant** est : **nbrTrainsToM <= QM.**  (Où QM est le nombre de quais de la gare M).

## ⊳ Question 4.3 : **Modifiez votre code pour l’assurer.**

**Le code fonctionne comme attendu. La gare du milieu ne provoque plus d’interblocage.**



**Exemple d’exécution du Main de l’exercice 4.**