Séance 8 - TP - Gestion des clients et ClassLoader

Ajout des classes permettant de gérer les clients lors de la simulation
Ajout d'un ClassLoader personnel pour gérer les multiples chargements de la librairie libTwisk.

L'énoncé de cette séance est en 2 parties.

Partie A - Gestion des clients

L'objectif est de préparer la visualisation des clients (processus C) sur l'interface graphique (laquelle est encore en construction dans l'UE d'Interfaces Graphiques). Pour ce faire, il faut mémoriser les positions des clients : on veut savoir, à chaque "snapshot" du monde dans quelle étape se trouve chaque client.

☆ Qui est responsable de la gestion des positions des clients ?

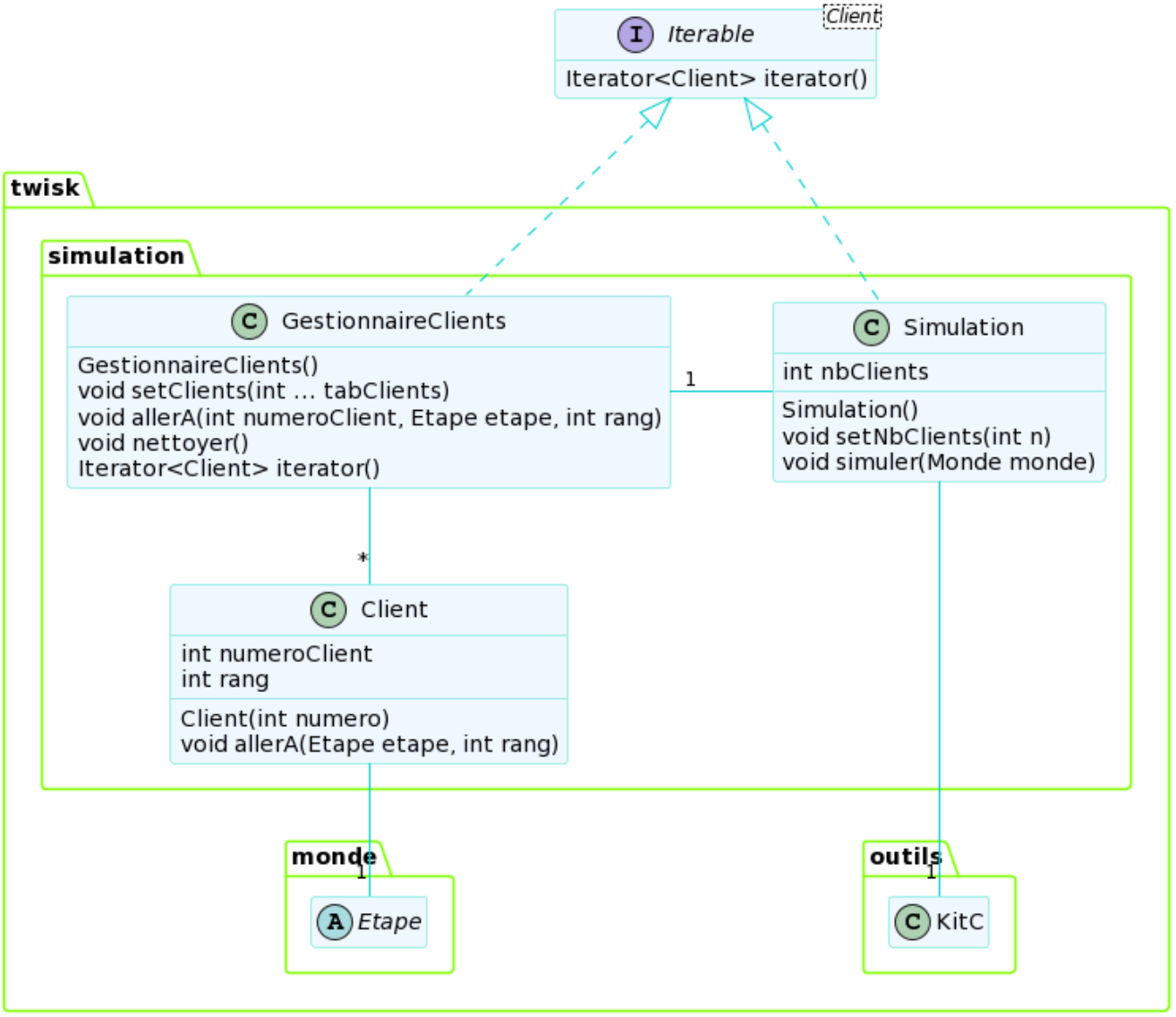
C'est une information dynamique, qui va évoluer au fur et à mesure de la simulation. Il n'est donc pas opportun de stocker cette information dans l'une des classes du monde (ces classes gèrent des informations statiques).

On va donc ajouter une classe **Client** dont les nombreuses instances sont gérées par la classe **GestionnaireClients**, ces deux classes sont placées dans le package gérant les données dynamiques de la simulation : **twisk.simulation**.

☆ Diagramme de classes incomplet de mémorisation des clients

Les fonctions de **GestionnaireClients** :

- **setClients** : instancie les clients identifiés par leur numéro de processus (numéro de client)
- **allerA** : met à jour les attributs **etape** et **rang** d'un client : **etape** est l'étape du **Monde** dans laquelle se trouve le client, le **rang** sera utile pour visualiser un ordre dans une file d'attente
- **nettoyer** : fait le ménage dans les clients, pour traiter une nouvelle simulation



- ☆ **Dans quelle classe est instancié GestionnaireClients ? Dans quelle fonction ?**
- ☆ **Dans quelle classe sont instanciés les clients ? Dans quelle fonction ?**
- ☆ **Dans quelle fonction est appelée la fonction allerA de GestionnaireClients ?**

Question 1 - Gestion des clients

Programmez et testez les deux nouvelles classes **GestionnaireClients** et **Client**.

Question 2 - Fonction simuler

Modifiez la classe **Simulation** pour instancier un **GestionnaireClients** et déplacer les clients dans les étapes au fur et à mesure de la simulation : c'est-à-dire lors de l'observation de l'exécution des processus C par la fonction **ou_sont_les_clients**.

Le résultat de la simulation est toujours sur la sortie standard.

Partie B - ClassLoader personnel

Dans le **main** de la classe **ClientTwisk** de votre application **twisk**, créez successivement deux mondes différents (choisissez un deuxième monde contenant plus d'étapes que le premier) et demandez-en les simulations. Regardez attentivement les affichages obtenus (prenez un petit nombre de clients).

Si le deuxième monde contient plus d'étapes que le premier, vous devez constater que les clients ne passent pas par toutes les étapes (même en augmentant les délais pour être sûr de pouvoir les observer).

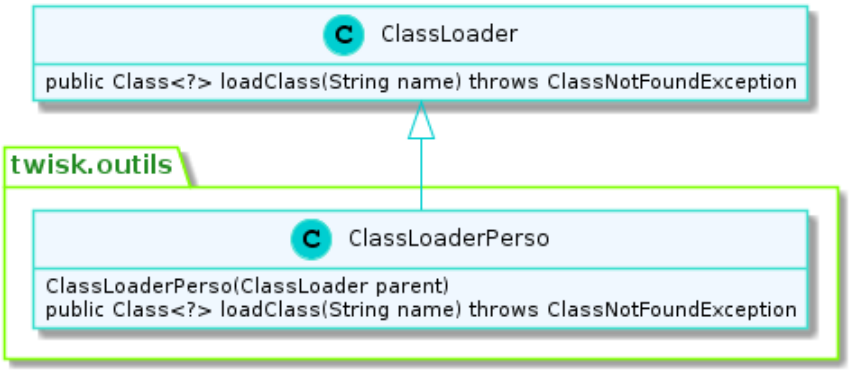
Pourtant, tout se passe bien pour ce deuxième monde :

- le code des clients est correctement généré,
- les compilations et éditions de liens sont également correctes,
- la librairie **libTwisk** est bien reconstruite et on en demande bien le chargement dans la fonction **simuler** de **Simulation**.

Puisqu'une librairie chargée reste accessible dans le **ClassLoader** courant, la solution consiste (comme vu en cours) à définir un **ClassLoader** personnel gérant les instances de la classe **Simulation** et à en instancier un nouveau à chaque chargement de la librairie **libTwisk**.

Question 1 - La classe ClassLoader personnelle

La librairie **libTwisk** est chargée par la fonction **simuler** de la classe **Simulation**. Il faut donc créer une classe **ClassLoaderPerso** personnelle, héritant de **ClassLoader** de java, pour gérer uniquement les instances de la classe **Simulation** :



Le code source de la classe **ClassLoaderPerso** se trouve sur arche, insérez-là en bonne place dans votre projet.

Question 2 - Instancier un monde et en faire la simulation

Modifiez la classe **ClientTwisk** de votre application de telle sorte qu'elle :

- crée un seul monde,
- instancie un **ClassLoaderPerso** ; lors de cette instanciation, il faut passer en argument le **ClassLoader** courant auquel on peut accéder par : this.getClass().getClassLoader()
- lui demande de charger la classe **Simulation**
- instancie la classe **Simulation** avec le constructeur par défaut (un constructeur par défaut est un constructeur sans paramètre). Attention, il faut que cette classe **Simulation** soit instanciée via votre **ClassLoaderPerso**, il faut donc faire de l'introspection… et appeler la fonction **newInstance** … retournez voir le cours de BPO.
- et appelle, toujours via l'introspection, les fonctions **setNbClients** et **simuler** sur cette instance de la classe **Simulation**

☆ **Que faire de toutes les exceptions potentiellement déclenchées par l'appel des fonctions ?**

Question 3 - Instancier plusieurs mondes et simuler

Quand votre code permet de lancer une simulation sur le monde créé, ajoutez le deuxième monde à la suite et simulez. Les deux simulations consécutives sur les deux mondes différents devraient être correctes.