Séance 2 - TD salle machine - Programmation du monde

Modèle : les classes Monde, Etape & co . Quelques tests JUnit
*Création de l'archive **twisk.jar** et utilisation de la classe cliente **atelier.ClientMonde***
Constitution des binômes et ouverture d'un compte sur gitlab.

Question 1 - Gestion de versions

Sous **IntelliJ**, l'un des deux étudiants du binôme crée un nouveau projet **Java**.

Suivez les indications du document "Gestion de versions" sur Arche pour versionner votre projet **intelliJ** afin de partager le projet créé avec votre binôme.

Question 2 - Création du monde et tests unitaires

Sur la base du diagramme de classes présenté ci-après, programmez l'application **twisk** limitée pour l'instant aux classes représentant le monde. Commencez par lire tout le texte de la question 2 puis développez la classe **twisk.monde.Etape** et ses sous-classes. Construisez au fur et à mesure différents cas de tests **JUnit**. Prévoyez une classe de tests par classe. Ne soyez pas avares de cas de tests.

Pour que votre application puisse être utilisée par d'autres classes clientes, il est indispensable de respecter les nom des packages, les noms des classes et les fonctions. Bien évidemment, vous avez toute latitude pour ajouter d'autres fonctions, dans la limite du raisonnable (par exemple ne pas exporter les structures internes des classes). Ceci peut être nécessaire, en particulier pour écrire des séquences de tests.

Rappels :

- La création des sas d'entrée et de sortie est du ressort du constructeur du **Monde** ; ces deux activités font aussi partie intégrante de la collection des étapes gérées par le monde (c'est redondant, mais c'est un choix ….).
- Les constructeurs de **Activité** et **Guichet** avec l'argument **String nom** seulement initialisent par défaut les autres attributs. Le choix des valeurs est laissé libre.
- Chaque classe doit aussi avoir une fonction **toString**. La forme de l'affichage sur la sortie standard est laissée libre mais doit être précise et compréhensible. L'ordre d'affichage des étapes d'un monde n'est pas imposé ; en voilà un exemple :

```
entrée : 1 successeur - balade au zoo
sortie : 0 successeur -
balade au zoo : 1 successeur - accès au toboggan
accès au toboggan : 1 successeur - toboggan
toboggan : 1 successeur - sortie
```

Quelques pistes pour les tests :

Il est inutile de tester du code simpliste (un *getter* de base, un constructeur qui ne fait rien, …), ou du code généré automatiquement par **IntelliJ** (**equals**, **toString**, …) et non modifié bien sûr.

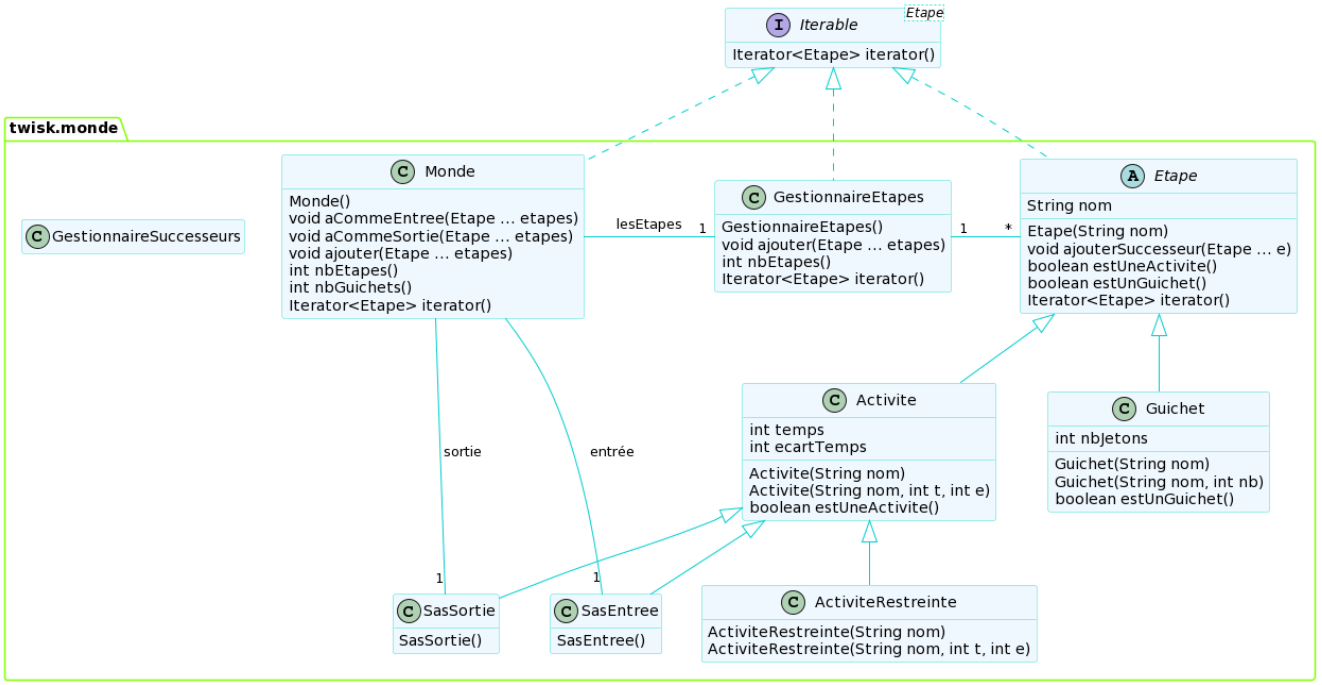
À l'inverse, il peut être pratique d'ajouter des fonctions pour écrire facilement des tests : par exemple, pour tester **ajouterSuccesseur** dans **Etape**, vous pouvez imaginer ajouter une fonction **nbSuccesseurs** dans **Etape** et tester **nbSuccesseurs** plutôt que **ajouterSuccesseur**.

Réfléchir aux tests permet de prendre du recul par rapport au code. Quand on travaille à deux, il peut être judicieux de se partager le travail : l'un écrit une classe, l'autre la teste … en inversant les rôles régulièrement.

Écrire des tests sans connaître l'implémentation d'une classe permet d'avoir un regard différent sur les cas de tests. Un exemple : le test de la fonction **iterator()** de la classe **Etape**. Si on teste sans savoir comment est implantée la collection des successeurs, le test doit s'assurer que l'itérateur fournit l'accès à tous les successeurs ajoutés, indépendamment de l'ordre dans lequel ils ont été ajoutés. Garantir cette indépendance est un peu fastidieux à programmer dans le test ; en contrepartie, on écrit un test qui est insensible à un changement de choix de structure.

En présence d'une hiérarchie (comme **Etape**/**Activite** …), on peut tester chaque sous-classe indépendamment des autres. Cela conduit probablement à la duplication de code de test. Pour éviter cette duplication, il est possible d'utiliser l'héritage entre classes de tests. Par exemple la classe **EtapeTest** regroupe les tests communs à toutes les étapes ; la classe **ActiviteTest**, sous-classe de **EtapeTest**, se contente de prévoir les cas de tests spécifiques aux activités. L'héritage sert ici à factoriser du code.

Pour finir, il n'y a pas vraiment de règle générale pour la construction des tests. Chacun peut imaginer sa méthode. L'idée est de couvrir le maximum de cas de tests, de faire des tests indépendants et de trouver une méthode qui minimise le temps qu'on y passe.



Question 3 - Construction d'une classe cliente

Écrivez la classe **twisk.simulation.Simulation** avec un constructeur sans argument et une fonction **simuler(Monde monde)** qui affiche simplement le monde sur la sortie standard. Ajoutez une classe cliente **twisk.ClientTwisk** proposant la fonction **main** pour créer un monde, instancier **Simulation** et demander l'exécution de **simuler**.

Testez les mondes dessinés lors de la séance précédente. Pour cela, écrivez dans la classe **twisk.ClientTwisk** différentes fonctions de construction d'un monde. Conservez tous les mondes créés.

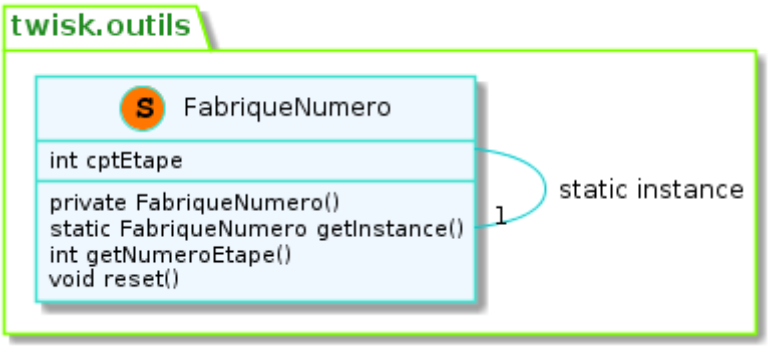
Question 4 - Numérotation des étapes (activités et guichets)

Pour identifier les étapes dans le code C, il faut attribuer un numéro unique à chacune d'elles, à partir de 0.

Numérotation des étapes

Les numéros des étapes seront utilisés dans le code C comme un indice dans un tableau. Ces étapes doivent donc être obligatoirement numérotées à partir de 0, inclus.

Ajoutez un champ de type **int** dans la classe **Etape** ; ce champ est fixé par le constructeur par le biais d'une fabrique de numéros. Cette fabrique **twisk.outils.FabriqueNumero** doit fournir un nombre entier positif ; tous les numéros fournis par la fabrique sont différents, par ordre croissant à partir de 0 inclus. Cette classe est programmée sous forme d'un **Singleton**.



Ajoutez également la fonction **reset** de réinitialisation de tous les compteurs, cela sera utile quand on demandera la création successive de plusieurs mondes.

Complétez les tests.

Question 5 - Numérotation des sémaphores des guichets

En plus de son numéro d'étape unique, chaque guichet possède un numéro unique correspondant à ce qu'on appelle un sémaphore et qui va permettre la synchronisation de tous les processus qui vont s'exécuter sur votre ordinateur lors du lancement de la simulation sur un monde. Ce numéro de sémaphore doit être attribué dans l'ordre croissant à partir de 1 inclus.

Numérotation des sémaphores

Les numéros des sémaphores associés aux guichets sont utilisés dans le code C comme un indice dans un tableau, mais le premier indice est déjà utilisé dans le code C donné. Ces sémaphores doivent donc être numérotés à partir de 1.

Ajoutez un champ de type **int** dans la classe **Guichet** ; ce champ est fixé par le constructeur par le biais de la fabrique de numéros, dans laquelle il faut ajouter un nouveau compteur **cptSemaphore** et une nouvelle fonction **int getNumeroSemaphore()**.

Complétez les tests.

Question 6 - Création d'une archive

Déposez une archive **twisk.jar** de votre projet dans l'atelier dédié sur arche en respectant les contraintes demandées :

- **chaque étudiant** dépose une archive **twisk.jar** (cette archive sera bien évidemment identique pour les étudiants appartenant au même binôme)
- aucun fichier source n'est déposé dans cette archive
- pour ce dépôt, privilégiez l'usage de la version 11 de java pour être sûr que les autres étudiants vont également pouvoir utiliser et tester votre archive.

Pour vous assurer que votre application respecte les contraintes données, vous pouvez demander l'exécution dans une fenêtre terminal de la classe **atelier.ClientMonde** disponible sur arche :

- créez un répertoire **atelier**, où vous voulez mais en dehors de votre projet **twisk**,
- copiez le fichier **ClientMonde.java** sous le répertoire **atelier**, compilez cette classe :

javac -classpath twisk.jar:. atelier/ClientMonde.java

- demandez l'exécution de la classe **atelier.ClientMonde** :

java -classpath twisk.jar:. atelier.ClientMonde

ajoutez les options de la VM si nécessaire

Surtout n'incluez pas la classe **ClientMonde** dans votre projet **IntelliJ**, elle doit rester externe à votre archive.