TP 4 – VLille

⚠ *Pour ce TP, les tests des classes à écrire sont fournis. Vous n'aurez donc pas à créer ces tests. Mais, vous devez consulter le code des méthodes de test pour les comprendre et vous familiariser avec leur écriture. Ces exemples pourront vous aider lorsque vous aurez plus tard à écrire vos propres tests. De plus, vous devrez les exécuter pour valider votre code.*

Q 1 . Récupérez sur le portail l'archive fournie et placez les fichiers qu'elle contient dans un dossier nommé `tp4/` de votre dépôt local. Parmi les fichiers extraits se trouve le fichier `junit-console.jar` qui permet l'exécution des tests.

⚠ *Vous remarquerez que l'on prolonge la bonne pratique d'organisation des fichiers d'un projet avec la présence d'un dossier* `test/` *qui contiendra les fichiers de tests.*

1 Paquetages

En accompagnement de la vidéo fournie dans le semainier.

Q 2 . Effectuez les modifications nécessaires pour définir la classe `BikeModel` comme appartenant au paquetage `vlille.util`.

⚠ *À partir de maintenant, le* «nom complet» *de la classe est* `vlille.util.BikeModel`. *Par raccourci de langage, dans le discours on continue néanmoins le plus souvent à utiliser son* «nom court» *`BikeModel`, sauf en cas d'ambiguïté. L'appartenance à un paquetage reste implicite.*

Q 3 . L'existence des paquetages a un impact sur la compilation puisqu'il faut tenir compte de la présence du dossier qui correspond à ce paquetage. Pour compiler la classe `BikeModel`, la commande à exécuter est donc.

```
.../poo/tp4/$ javac -sourcepath src src/vlille/util/BikeModel.java -d classes
```

Q 4 . Effectuez les modifications nécessaires pour définir la classe `Bike` comme appartenant au paquetage `vlille`.

Q 5 . Compilez la classe `vlille.Bike` en vous inspirant de la commande précédente.

Vous devez constater des erreurs lors de cette compilation. Elle sont toutes dues à la classe `ville.util.BikeModel` qui ne semble pas être trouvée.

Le problème vient du fait que les deux classes n'appartenant pas au même paquetage, il est nécessaire d'importer explicitement la classe `BikeModel` dans la classe `Bike`.

Q 6 . Ajoutez l'instruction d'importation dans le fichier `Bike.java` puis compilez, sans erreur normalement cette fois.

Q 7 . On s'occupe de la classe `BikeMain`.

Q 7.1. Faites le nécessaire pour la déclarer comme appartenant au paquetage `vlille`.

Q 7.2. Compilez cette classe, en corrigeant les éventuels problèmes rencontrés.

Q 7.3. L'existence des paquetages a également un effet sur la commande d'exécution, puisqu'il est nécessaire d'utiliser le nom complet de la classe ciblée dans la commande d'exécution.

Ainsi l'exécution de la classe `vlille.BikeMain` se réalise à l'aide de la commande :

```
.../poo/tp4/$ java -classpath classes vlille.BikeMain
```

Pour générer la documentation d'un paquetage et de ses sous-paquetages, la commande à utiliser est[1] :

```
.../poo/tp4/$ javadoc -sourcepath src -subpackages vlille -d docs
```

⚠ Bonne pratique ⚠
Il faut toujours définir les classes au sein d'un paquetage. L'espace de nommage défini par le paquetage facilite leur réutilisation en réduisant les risques de confusion.

[1] En cas de plusieurs paquetages, il suffit de les citer les uns à la suite des autres : « `javadoc -sourcepath src -subpackages pack1 pack2 pack3 -d docs` »

Q 8 . Faites le nécessaire pour définir les classes `BikeStation` et `BikeNotAvailableException` dans le paquetage `vlille`.

Compilez ces classes, sans erreur.

2 Premiers tests unitaires.

⚠ *Pour réaliser ces tests unitaires nous utilisons la bibliothèque `JUnit5` qui est très largement utilisée et fait référence pour les tests unitaires en java.*

Q 9 . Consultez le contenu du fichier de test `test/vlille/BikeTest.java`.

Il définit la classe `vlille.BikeTest` qui constitue une classe de test pour la classe `vlille.Bike`. C'est-à-dire une classe qui contient plusieurs méthodes de test pour les méthodes de cette classe. Sans que cela soit véritablement obligatoire, les noms des classes de test commenceront ou se termineront par « `Test` ».

Vous pouvez noter les `import` du paquetage `org.junit.jupiter.api` réalisés en début de ce fichier. Ils permettent la mise en place des tests JUnit5. Ces imports seront nécessaires pour les classes de test que vous aurez à écrire par la suite.

Les méthodes de test sont identifiables par l'annotation `@Test` qui les précède. Dans `BikeTest` on trouve 4 méthodes de test :

- une méthode pour vérifier que la construction de l'objet `Bike` se passe correctement, et que les accesseurs fournissent les bonnes valeurs ;
- trois méthodes pour tester les différents cas de la méthode `equals`.
  Les noms de ces trois méthode permettent d'identifier facilement les situations testées à chaque fois.

> ⚠ **Bonne pratique** ⚠
> Il faut choisir des noms de méthode de tests qui décrivent explicitement le test réalisé. Cela dispense d'ajouter une documentation ou un commentaire à la méthode et, en cas d'échec de la méthode, d'identifier plus rapidement la situation problématique.

Vous constatez que les méthodes de tests contiennent une ou plusieurs assertions de test identifiées par les méthodes dont le nom commence par « `assert` » : `assertEquals`, `assertTrue` et `assertFalse`. On comprend facilement que la première est vérifiée quand ses deux arguments sont égaux (au sens `equals`) et que les deux suivantes sont vérifiées quand l'expression fournie en paramètre vaut `true` ou `false`. Ces assertions sont rendues disponibles par l'`import static` réalisé en début de classe.

Un test (c'est-à-dire une méthode de test) est réussi, et donc passé avec succès par le code qu'il évalue, quand toutes les assertions qu'il contient sont vérifiées.

⚠ *D'autres méthodes d'assertion existent. Vous devez consulter le document sur le portail, zone* Documents *pour les découvrir.*

Une méthode de test est une méthode « normale ». En plus de ces assertions, elle peut donc contenir n'importe quel code Java valide. De même qu'une classe de test est pleinement une classe. Elle peut avoir des attributs, des méthodes autres que de tests, etc.

> ⚠ **Bonne pratique** ⚠
> Il est recommandé de reprendre l'organisation de paquetages des classes testées au niveau des classes de test. C'est le cas ici avec la classe `BikeTest` qui appartient aussi au paquetage `vlille`.

Q 10 . Avant de pouvoir exécuter les tests définis dans `BikeTest` et ainsi vérifier si l'implémentation de la méthode `equals` de `Bike` est correcte il faut avoir compilé la classe `Bike` (fait ci-dessus) et il faut également avoir compilé le fichier `BikeTest.java` en exécutant la commande :

```
.../poo/tp4$ javac -classpath junit-console.jar:classes test/vlille/BikeTest.java
```

⚠ *Le fichier* `junit-console.jar` *est l'archive qui permet l'utilisation de JUnit5 en ligne de commande. Elle est rendue disponible par son ajout au paramètre* `-classpath`.
Elle a été renommée ici, le nom du fichier disponible sur les sites de téléchargements est `junit-platform-console-standalone-1.9.3.jar`.

⚠ *Comme déjà vu, sous Windows, il faut modifier le paramètre de l'option* `-classpath` *en «* `"junit-console.jar;classes"` *».*

⚠ *En complément du document présent sur le portail, il est possible d'ajouter le chemin* `test` *et l'archive* `junit-console.jar` *à la variable d'environnement* `CLASSPATH` *pour alléger les commandes de compilation et exécution, avec la définition :* `export CLASSPATH="src:classes:test:junit-console.jar"`

Q 11 . On peut alors exécuter les tests définis dans `BikeTest` grâce à la commande[2] :

```
.../poo/tp4$ java -jar junit-console.jar
                        -classpath test:classes -select-class vlille.BikeTest
```

La classe dont on veut exécuter les tests est précisée grâce à l'option `-select-class`.
Comme déjà vu, l'option `-classpath` permet de préciser où trouver les classes nécessaires à l'exécution.

Dans la console, vous pouvez observer une trace d'exécution des tests. Elle présente, sous la forme d'un arbre, un résumé des tests où apparaissent le nom de la classe testée ainsi que le nom des méthodes de test toutes passées avec succès ici. À gauche sous linux, à droite sous windows. :

```
╷
├─ JUnit Jupiter ✓
│  └─ BikeTest ✓
│     ├─ testEqualsReturnsFalseWhenNotSameId() ✓
│     ├─ testEqualsReturnsTrueWhenSameId() ✓
│     ├─ modelAndIdAreCorrectAfterCreation() ✓
│     └─ testEqualsReturnsFalseWhenNotABikeOrNull() ✓
├─ JUnit Vintage ✓
└─ JUnit Platform Suite ✓
```

```
.
+-- JUnit Jupiter [OK]
| '-- BikeTest [OK]
|   +-- testEqualsReturnsFalseWhenNotSameId() [OK]
|   +-- testEqualsReturnsTrueWhenSameId() [OK]
|   +-- modelAndIdAreCorrectAfterCreation() [OK]
|   '-- testEqualsReturnsFalseWhenNotABikeOrNull() [OK]
+-- JUnit Vintage [OK]
'-- JUnit Platform Suite [OK]
```

Vient ensuite une synthèse de l'ensemble des tests exécutés où sont indiqués les nombres de méthodes de tests trouvées (ici 4), passées avec succès (ici 4) ou en échec (ici 0) :

```
Test run finished after 132 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         0 tests failed          ]
```

Q 12 . Etudions ce qu'il se passe lorsqu'un test échoue. Pour cela, reprenez les commandes précédentes pour compiler la classe de test `BikeSecondTest.java`, puis l'exécuter.

La trace que vous obtenez fait apparaitre une erreur dans l'une des méthodes de test. On identifie facilement le nom de la classe et de la méthode de test en échec et le problème rencontré est précisé : ici on s'attendait à obtenir la valeur « `"red"` » mais la valeur « `null` » a été obtenue.

```
╷
├─ JUnit Jupiter ✓
│  └─ BikeTest2 ✓
│     ├─ modelAndIdAreCorrectAfterCreation() ✓
│     └─ getColorReturnsDefaultColor() ✗ expected: <red> but was: <null>
├─ JUnit Vintage ✓
└─ JUnit Platform Suite ✓
```

```
.
+-- JUnit Jupiter [OK]
| '-- BikeTest2 [OK]
|   +-- modelAndIdAreCorrectAfterCreation() [OK]
|   '-- getColorReturnsDefaultColor() [X] expected: <red> but was: <null>
+-- JUnit Vintage [OK]
'-- JUnit Platform Suite [OK]
```

À la suite on trouve la trace d'éxécution où on retrouve les informations présentes dans le résumé précédent avec, en plus, l'information sur le numéro de la ligne de code où apparaît l'assertion en échec dans le fichier source (ici la ligne 20 du fichier `BikeSecondTest.java`).

```
Failures (1):
  JUnit Jupiter:BikeSecondTest:getColorReturnsDefaultColor()
    MethodSource [className = 'vlille.BikeSecondTest', methodName = 'getColorReturnsDefaultColor',
    => org.opentest4j.AssertionFailedError: expected: <red> but was: <null>
```

[2] La signification de l'option `-jar` sera présentée dans un TP ultérieur.

```
org.junit.jupiter.api.AssertionFailureBuilder.build(AssertionFailureBuilder.java:151)
org.junit.jupiter.api.AssertionFailureBuilder.buildAndThrow(AssertionFailureBuilder.java:132)
org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:197)
org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:182)
org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:177)
org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:1142)
vlille.BikeSecondTest.getColorReturnsDefaultColor(BikeSecondTest.java:20)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
[...]
```

Si plusieurs tests sont en échec, on retrouve une trace similaire à celle-ci pour chacun des échecs. En cas d'échecs nombreux on peut donc avoir une trace assez longue.

Enfin, après les détails de chacun des échecs, la synthèse, qui rappelle le nombre de tests en succès et en échec, est affichée.

```
Test run finished after 132 ms
[      (...)                ]
[      2 tests found       ]
[      0 tests skipped     ]
[      4 tests started     ]
[      0 tests aborted     ]
[      1 tests successful  ]
[      1 tests failed      ]
```

Q 13 . Dans le cas, le plus fréquent, où les tests sont écrits dans plusieurs classes, il est bien sûr possible d'utiliser le joker « `*.java` » pour compiler toutes les classes de test d'un dossier. Par exemple :

```
.../poo/tp4$ javac -classpath junit-console.jar:classes test/vlille/*.java
```

Il est également d'exécuter tous les tests des classes positionnées dans le *classpath*. Il faut pour cela utiliser l'option `-scan-classpath`[3] au lieu de l'option `-select-class`.

```
.../poo/tp4$ java -jar junit-console.jar -classpath test:classes -scan-classpath
```

⚠ *Tant que vous placez toujours vos classes compilées dans un dossier nommé `classes` et vos classes de test dans un dossier nommé `test`, cette commande reste valide. Vous pouvez donc définir un fichier de commande reprenant cette commande pour ne pas avoir à la réécrire à chaque fois.*

Le résumé fait alors apparaitre les différents tests, en succès ou en échec, regroupés par classe de test.

```
├─ JUnit Jupiter ✓
│  ├─ BikeTest ✓
│  │  ├─ testEqualsReturnsFalseWhenNotSameId() ✓
│  │  ├─ testEqualsReturnsTrueWhenSameId() ✓
│  │  ├─ modelAndIdAreCorrectAfterCreation() ✓
│  │  └─ testEqualsReturnsFalseWhenNotABikeOrNull() ✓
│  └─ BikeSecondTest ✓
│     ├─ modelAndIdAreCorrectAfterCreation() ✓
│     └─ getColorReturnsDefaultColor() ✗ expected: <red> but was: <null>
├─ JUnit Vintage ✓
└─ JUnit Platform Suite ✓
```

Les traces des différentes méthodes en erreur puis la synthèse sont ensuite affichées.

Q 14 . L'échec de la méthode de test `getColorReturnsDefaultColor` met en évidence un problème avec cette méthode. Il faut donc le corriger.

Faites le en faisant en sorte que la méthode `getColor` de la classe `vlille.Bike` renvoie la valeur de l'attribut `color`.

Recompilez la classe `vlille.bike` modifié. Il est inutile de recompiler la classe de test qui, elle, n'a pas été modifiée.

Exécutez à nouveau les tests pour vérifier que cette fois il n'y a pas plus de méthode en erreur.

[3] Sauf à devoir faire des paramétrages supplémentaires, cette option ne trouvera que les classes de test qui commenceront ou se termineront par « `Test` ».

3 La classe `BikeStation`

Q 15 . Commencez par copier le fichier `BikeStationTest.java` depuis le dossier `tmp/` vers le dossier `test/vlille/`, puis procédez comme précédemment pour compiler le fichier de test `test/vlille/BikeStationTest.java`.

Q 16 . Utilisez la commande vue précédemment pour exécuter l'ensemble des tests du dossier `test`.

Dans le résumé vous retrouvez les tests réussis de `vlille.BikeTest` et `vlille.BikeSecondTest`, mais aussi les 10 tests en échec de la classe `vlille.BikeStationTest`. En fait, tous les tests de cette classe sont en échec. On trouve donc à la suite la (longue) trace de ces 10 échecs.

Pour la suite du TP l'objectif est de passer tous les tests avec succès.

Ecriture des méthodes = objectif zéro test en échec.

Dans la suite du TP l'objectif est d'avoir de moins en moins de tests en erreur après chaque question et donc à la fin du TP que tous les tests aient été passés avec succès.
Pour l'évaluation de votre TP, ces tests seront exécutés sur votre code.

Dans les questions suivantes, pour chacune des méthodes à implémenter, vous

1. commencerez par écrire la documentation de cette méthode ;
2. puis vous étudierez la ou les méthodes de test qui lui sont associées dans `vlille.BikeStationTest` ;
3. vous procéderez **ensuite** à l'écriture du code de la méthode.

Vous validerez votre implémentation en vérifiant que le test correspondant est réussi.
Vous ne devez passer à la question suivante que si tous les tests liés à une question ont été passés avec succès.

⚠ *N'oubliez pas : il est nécessaire de recompiler le code source après chaque modification pour que celle-ci soit prise en compte.*

La classe `BikeStation`

Les vélos évoqués ci-dessus sont mis à disposition dans des *stations de location* modélisées par le type `BikeStation`. Une station de location comporte un certain nombre d'emplacements (*slots*) qui peuvent accueillir les vélos en location. Chacun de ces emplacements peut être libre ou occupé par un vélo disponible.
Chaque station a un nom et le nombre de vélos que peut contenir une station de location est fixe. Celui est précisé lors de la construction de l'objet `BikeStation`. Les emplacements pour vélos d'une station sont donc gérés dans un tableau (attribut `bikes`). L'absence d'un vélo à un emplacement se traduit par une valeur `null` dans ce tableau.

Q 17 . Définissez le constructeur, sachant qu'aucun vélo n'est présent dans une station à sa création.

Définissez ensuite les méthodes `getName()` et `getCapacity()`, ainsi que la méthode `getNumberOfBikes()` qui a pour résultat le nombre de vélos actuellement disponibles (présents) dans la station.

Q 18 . Compilez la classe `BikeStation` et exécutez les tests de `BikeStationTest`, vous devez avoir avec succès un test passé en plus.

Q 19 . Ecrivez la documentation puis donnez le **code** de la méthode `firstFreeSlot` de `BikeStation` qui a pour résultat le plus petit indice correspondant à un emplacement libre pour un vélo. Si aucun emplacement n'est libre le résultat de la méthode est -1.

⚠ *Les tests de cette méthode dépendent de la méthode `dropBike()` définie à la question suivante, vous ne pourrez donc en valider les tests qu'une fois la question suivante également réalisée.*

Q 20 . Ecrivez la javadoc et le code de la méthode `dropBike`.

Cette méthode permet de déposer un vélo dans une station. Le vélo est alors mis au premier emplacement libre dans la station s'il y en a un. Sinon rien ne se passe. La méthode a pour résultat un booléen qui vaut *vrai* s'il a été possible de déposer le vélo et *faux* sinon.

Q 21 . Ecrivez la documentation et le code de la méthode `takeBike` qui permet de prendre un vélo dans une station.

Cette méthode prend en paramètre un entier qui correspond à la position de l'emplacement où l'on essaie de prendre un vélo. S'il y a bien un vélo disponible à cet emplacement, ce vélo est renvoyé comme résultat de la méthode et l'emplacement est libéré dans la station. S'il n'y a pas de vélo à l'emplacement demandé ou si l'emplacement n'existe pas une exception `BikeNotAvailableException` est levée.

Q 22 . Définissez une classe `BikeStationMain` qui définit une méthode `main` qui :

- crée une station de location de vélos de nom `Timoleon` et qui peut accueillir 10 vélos
- crée et ajoute à la station deux vélos. Le premier est un vélo classique et a pour référence `"b001"`. Le second est électrique et a pour référence `"b002"`.
- prend un vélo à l'emplacement numéro *n* dont la valeur est définie par le premier argument passé en paramètre lors de l'exécution de cette classe (et donc de sa méthode `main`).

  Si il y avait bien un vélo à cet emplacement alors un affichage annonce l'identifiant du vélo pris.

  S'il n'y a pas de vélo disponible à l'emplacement *n* un affichage annonce l'absence de vélo.

  Vous devez gérer l'absence du paramètre sur la ligne de commande.

Rendre le travail sur gitlab (complété)

Dans le dossier `tp4/` de votre dépôt local,

- créez un fichier `readme.md`[4] où vous indiquerez de manière structurée (ce sera à faire pour chaque TP) :
  - les noms des membres du binômes,
  - un paragraphe présentant le TP et ses objectifs,
  - un paragraphe précisant comment générer et consulter la documentation,
  - un paragraphe précisant comment compiler les classes du projet,
  - un paragraphe précisant comment compiler et exécuter les tests,
  - un paragraphe précisant comment exécuter le programme (le ou les `main` inclus), en donnant des exemples. Vous pouvez aussi donner des exemples de trace d'exécution.

  Ce fichier `readme.md` doit être suffisamment précis pour que quelqu'un qui ne connaît pas le projet (par exemple n'a pas lu le sujet) puisse en comprendre les objectifs puis l'utiliser (documentation, tests et exécution). Vous devez donc à chaque fois fournir les commandes précises à exécuter. On doit pouvoir copier/coller ces commandes pour les exécuter.
- ajoutez à votre dépot `gitlab`
  - ce fichier `readme.md`,
  - les dossiers `src/` et `test/` et leurs fichiers `.java`,
  - le fichier `junit-console.jar`

    ⚠ *Il est possible qu'il soit nécessaire de forcer l'ajout de ce fichier `.jar` sur votre dépôt car votre `.ignore` est probablement paramétré pour rejeter les archives (et c'est normal). Pour **f**orcer l'ajout, vous devez ajouter l'option `-f` lors du `add` : « `git add -f junit-console.jar` »*

⚠ *Vous ne devez pas déposer sur Gitlab les contenus des dossiers `docs` et `classes`, car les fichiers `.class` et de documentation peuvent être générés. L'exclusion de ces fichiers doit être gérée par le fichier `.gitignore` de votre dépôt comme présenté dans le document d'introduction à GitLab.*

⚠ Bonne pratique ⚠

Une bonne pratique est d'ajouter à votre dépôt `gitlab` les fichiers de travail au fur et à mesure de leur création sans attendre la fin du TP. Au cours de votre travail, vous devez régulièrement procéder à des séquences `add`/`commit`/`push` pour valider votre travail, en indiquant un message pertinent lors du `commit`. Par exemple, lorsqu'une méthode a été ajoutée avec succès, ou qu'une erreur a été corrigée.

[4] voir les fiches de synthèse de la syntaxe markdown dans la zone document du portail