

Un Mud en TDD

Qualité de développement

BTS 2TSSI

2025-2026

Résumé

Cette feuille vous permet de vous entraîner au développement dirigé par les tests.

Dans ce TP, vous allez mettre en application ce que vous avez appris dans le précédent TP, et vous entraîner au développement dirigé par les tests sur une nouvelle petite bibliothèque.

1 Créez un nouveau projet

Dans votre home (par exemple), vous devez créer un nouveau projet pour ce TP :

```
$ git init tdd-backlog-mud
Initialized empty Git repository in /home/xxx/tdd-backlog-mud/.git/
$ cd tdd-backlog-mud
```

Une fois que vous avez créé votre dépôt en local, connectez-vous à votre compte github et créez un dépôt privé que vous associerez à votre dépôt local. Il faudra régulièrement synchroniser votre répertoire local avec votre dépôt distant.

Rappel

Depuis les machines du Lycée, il faut procéder à plusieurs vérifications afin de pouvoir associer votre dépôt distant sur Github à votre répertoire local :

- Il faut vérifier que vous avez bien renommé la branche principal en `main:git branch -M main`
- Il faut que vous ayez un token valide ou un moyen autre de connexion à votre compte Github (ou Gitlab).



2 Rappel de la méthodologie

Nous vous proposons un *backlog* d'items à réaliser.

Rappel



Un **backlog** est une liste de tâches à réaliser. On associe aux tâches un **niveau de priorité**.

À retenir

Vous **devez** suivre la méthodologie suivante :

- prendre un élément du backlog
- le traduire par un test
- faire tourner les tests pour vérifier que le nouveau test échoue par manque de fonctionnalité (et non pas à cause d'une erreur de syntaxe)
- commiter le test
- implémenter la fonctionnalité
- faire tourner les tests pour vérifier que le test passe
- commiter cette nouvelle fonctionnalité
- passer à l'item suivant du backlog



3 Une bibliothèque pourquoi faire ?

Ce TP commence à vous préparer à la mise en place d'un MUD (Multi-user dungeon https://fr.wikipedia.org/wiki/Multi-user_dungeon). Dans un MUD il y a des lieux connectés par des passages, il y a des portes qui peuvent être verrouillées, il y a des coffres qui peuvent contenir des objets, il peut même y avoir des ennemis contre lesquels il faut se battre, etc...

La question importante est : comment peut-on représenter toute ces choses ? Nous allons les représenter par des objets ayant des attributs et des méthodes.

Afin d'organiser un peu notre travail, nous allons nous fixer un certain nombre de **features**. Ce terme provient de **Git Workflow** qui est un ensemble de bonnes pratiques à suivre lorsqu'on utilise Git.

À retenir

Pour faire simple, une feature peut représenter différentes choses comme par exemple :

- résoudre un bug
- développer une nouvelle fonctionnalité
- ajouter de la documentation
- ...



Une feature doit rester quelque chose de précis et d'assez "petit". Une liste de features est appelée un backlog. Nous allons donc nous consacrer maintenant à la feature : *Bibliothèque Box*. Chaque feature doit être développée dans une branche séparée.

4 Bibliothèque Box : Premiers éléments du backlog

Voici les deux premiers éléments du backlog :

- on veut pouvoir créer des boîtes
- on veut pouvoir mettre des trucs dedans

et nous allons les réaliser ensemble.

Question 1 On veut pouvoir créer des boîtes! Traduisez d'abord cet item de backlog en un test.

Nous devons traduire cet item en un test. Pour cela, avec un éditeur de texte, nous créons le fichier `TestsBoxes.java` avec le contenu suivant :

```
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class TestsBoxes{
    @Test
    public void testBoxCreate() {
        Box b = new Box();
    }
};
```

On sauvegarde le fichier et on compile les tests JUnit :

```
javac -cp .:junit-4.13.2.jar TestsBoxes.java
```

Est-ce que ça compile?

NON : le test échoue : il n'y a pas d'erreur de syntaxe; l'échec est dû au fait que `Box` n'existe pas encore. On se prépare à commiter le test dans une branche `test-box-create` :

```
git branch test-box-create
git switch test-box-create
On branch test-box-create
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
TestsBoxes.class
TestsBoxes.java
nothing added to commit but untracked files present (use "git add" to track)
```

Question 2 Le fichier `TestsBoxes.java` n'est pas encore tracké. Nous allons donc devoir l'ajouter à l'index avant de valider les modifications avec un commit. Suivez les étapes décrites ci-dessous.

Question 3 Tout d'abord, nous souhaitons éviter de versionner les fichiers `.class`. Comment faire?

Astuce



On peut gérer les `.class` avec un `.gitignore` adapté. Faites-le et commitez le `.gitignore` correspondant dans une branche séparée!

Question 4 Commitez à présent le test. Attention, nous allons essayer de respecter une convention pour l'écriture des commits. Nous allons indiquer le nom de la feature sur laquelle nous travaillons.

```
$ git add TestsBoxes.java
$ git commit -m "Biblio box: ajout testBoxCreate"
[main (root-commit) c9f3264] Biblio Box: ajout testBoxCreate
1 file changed, 3 insertions(+)
create mode 100644 testBoxCreate
```

Question 5 On a maintenant un test qui échoue. On passe donc à l'implémentation de la fonctionnalité.

Avec notre éditeur, créons le fichier `Box.java` avec le contenu minimal suivant :

```
public class Box {
    public Box(){System.out.println("Box créée");}
}
```

Question 6 Relancez les tests pour vérifier que le test passe à présent.

On recompile les tests :

```
javac -cp .:junit-4.13.2.jar TestsBoxes.java
```

Cette fois ça compile!

On relance donc les tests pour vérifier que cette fois le test passe :

```
java -cp .:junit-4.13.2.jar:hamcrest-2.2.jar org.junit.runner.JUnitCore TestsBoxes
```

Le test passe. Commitez la nouvelle fonctionnalité offerte par le fichier `Box.java`.

On revient sur la branche `main` puis on push la branche `test-box-create` sur le dépôt distant :

```
git switch main
git push origin test-box-create
```

On crée le pull request correspondant sur Github.

Astuce

Une fois que le pull request est accepté, pensez à vous mettre tous à jour sur la branche `main` :



```
git pull
```

Vous avez réalisé votre première feature. Bravo!

À retenir

Pour chaque nouvelle feature à développer, vous procéderez à présent de la manière suivante :

- Créer une branche nommée `test-<nom de la feature>`.
- Basculer sur cette nouvelle branche avec la commande `git switch <nom de la branche>`.
- Implémenter un test qui échoue.
- Implémenter la fonctionnalité.
- Vérifier que le test passe.
- Re-basculer sur la branche `main`
- Faire un `git push` de la branche puis une pull request sur Github
- Mettre à jour la branche `main` avec un `git pull`



Question 7 On veut pouvoir mettre des trucs dans une Box! Ecrivez d'abord le test correspondant...

Dans le fichier de tests, `TestsBoxes.java` on ajoute un nouveau test :

```
import org.junit.*;
public class TestsBoxes {
    @Test
    public void testBoxCreate() {
        Box b = new Box();
    }
    /** on veut pouvoir mettre des trucs dedans */
    @Test
    public void testBoxAdd(){
        Box b = new Box()
        b.add("truc1")
        b.add("truc2")
    }
}
```

On recompile :

```
javac -cp .:junit-4.13.2.jar TestsBoxes.java
```

Cette compilation échoue comme prévu : il n'y a pas d'erreur de syntaxe; il manque simplement la nouvelle fonctionnalité. Nous commitons le nouveau test :

```
git commit -a -m "Biblio Box: ajout de test_box_add"
[main 8b10324] Biblio Box: ajout de testBoxAdd
1 file changed, 6 insertions(+)
```

Question 8 Comment implémenter la fonctionnalité demandée par le test?

Dans le fichier `Box.java`, nous implantons à présent cette fonctionnalité : il faut une méthode `add` pour ajouter un `String` à la boîte. Donc il faut aussi un endroit pour mettre ces `String` : par exemple, un attribut qui serait une liste de `String`. Nous modifions donc la définition de la classe `Box` comme suit :

```
import java.util.ArrayList;
class Box{

    ArrayList<String> contents = new ArrayList<String>();

    public void add(String truc) {
        this.contents.add(truc);
    }
}
```

On recompile les tests :

```
javac -cp .:junit-4.13.2.jar TestsBoxes.java
```

Ça compile! On fait tourner les tests :

```
java -cp .:junit-4.13.2.jar:hamcrest-2.2.jar org.junit.runner.JUnitCore TestsBoxes
```

Ça passe! on peut donc commiter la nouvelle fonctionnalité :

```
$ git commit -a -m "Biblio Box: attribut Box.contents et methode Box.add, testBoxAdd"
[main 7fef816] Biblio Box: attribut Box.contents et methode Box.add, testBoxAdd
 1 file changed, 6 insertions(+), 1 deletion(-)
$ git status
On branch main
nothing to commit, working directory clean
```

Question 9 On voudrait maintenant mettre des Thing dans la boîte. Comment modifier la classe Box pour que cela soit possible? Par exemple :

```
import java.util.ArrayList;

class Thing{
    String name;
    public Thing(String name){
        this.name = name;
    }
}

class Box{

    ArrayList<Thing> contents = new ArrayList<Thing>();

    public void add(Thing truc):
        this.contents.add(truc);
}
```

Testez! et commitez la nouvelle version de la classe Box.

5 Bibliothèque Box : Éléments suivants du backlog

- l'attribut contents d'une boîte devrait vraiment être privé. Rendez l'attribut contents privé.

- on veut pouvoir tester si un truc est dans une boîte. Utiliser la méthode `contains` de la classe `java.util.ArrayList` pour proposer une méthode pour cela dans la classe `Box`. Consultez la documentation : [https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html#contains\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html#contains(java.lang.Object)).
- on veut pouvoir retirer un truc d'une boîte : après qu'on l'ai retiré, il ne devrait plus être dedans. Si on essaie de retirer un truc qui n'est pas présent dans la boîte, il faudra déclencher/lever une exception.

Une exception est un signal qui se déclenche en cas de problème. Les exceptions permettent de gérer les cas d'erreur et de rétablir une situation stable.

Pour vous aider à utiliser cette nouvelle notion, vous pouvez consulter les liens suivants :

- https://fr.wikibooks.org/wiki/Programmation_Java/Exceptions
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

Pour écrire un test en tenant compte d'une éventuelle exception levée en Java , vous pouvez faire comme ceci :

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
    int i = 1/0;
}
```

Consulter éventuellement les liens suivants :

- <https://www.baeldung.com/junit-assert-exception>
- <https://www.digitalocean.com/community/tutorials/junit-assert-exception-expected>
- une boîte peut être ouverte ou fermée : on voudrait pouvoir tester avec `b.isOpen()`. On voudrait aussi pouvoir la fermer avec `b.close()`, après quoi elle devrait être fermée, et l'ouvrir avec `b.open()`, après quoi elle devrait être ouverte.
- on veut qu'une boîte permette `b.actionLook()` pour regarder dedans. Cette action retourne une `String` qui nous dit "la boîte contient: ceci, cela" si la boîte est ouverte et contient les strings "ceci" et "cela". Si la boîte est fermée, elle retourne le String "la boîte est fermée".

Astuce



Si on a une liste de type `ArrayList<String>` et qu'on veut produire la chaîne de ces strings séparées par des virgule-espace on fait :

```
String listString = String.join(", ", liste);
```

- on veut pouvoir créer des choses (à mettre dans des boîtes) qui prennent de la place. Une chose de volume 3 devrait s'obtenir par `Thing(3)`.
- on veut pouvoir connaître la place prise par une chose : `t.volume()`
- on veut pouvoir donner à une boîte une capacité : `b.setCapacity(5)`, et obtenir sa capacité : `b.capacity()`

- quand on n'a pas donné une capacité à une boîte, cette capacité est `-1`.
- on voudrait pouvoir tester si il reste de la place dans une boîte pour une nouvelle chose : `b.hasRoomFor(t)`. Quand on ajoute une chose à une boîte, elle consomme son volume de la capacité de la boîte; et il en reste d'autant moins pour les choses suivantes.
- Une boîte de capacité `-1` est considérée comme ayant une capacité illimitée.
- on voudrait qu'une boîte permette `b.addAction(t)` qui permette de mettre une chose dans une boîte mais seulement s'il y a suffisamment de place. Quand l'action échoue, elle lève une exception.
- `b.addAction(t)` devrait aussi échouer si la boîte est fermée.
- il faudrait pouvoir donner un nom à chaque chose pour qu'elle s'affiche de manière intelligible (elle affiche juste son nom) : `t.setName("bidule")`
On convertit un objet en string avec la fonction `toString(t)`. Pour qu'un objet sache effectuer cette conversion, il faut ajouter à sa classe cette méthode `toString` qui retourne un `String`.
- on voudrait pouvoir tester si une chose a un nom donné : `t.hasName("bidule")`
- on voudrait qu'une boîte permette `b.find("bidule")` pour trouver et retourner l'objet ayant ce nom et contenu dans la boîte. La méthode lève une exception si elle n'a rien trouvé.
- on voudrait que `find` lève aussi une exception si la boîte est fermée.
- on voudrait pouvoir optionnellement donner le nom d'une chose dans son constructeur (au lieu d'avoir à utiliser `setName` ensuite).
Pour cela il faut ajouter un constructeur à la classe `Thing` qui a un argument de type `String` et qui appelle le constructeur par défaut, éventuellement ajouter un autre constructeur avec les deux arguments.
- on voudrait de même pouvoir optionnellement préciser dans le constructeur si une boîte est ouverte ou fermée, et sa capacité :

6 JSON

À retenir



JSON est une notation simplifiée pour représenter des structures de données et échanger des données entre applications, notamment sur le web.

Voir <http://en.wikipedia.org/wiki/JSON> et <https://www.json.org/json-fr.html>.

Voici un exemple de JSON qui représente une liste de boîtes :

```
[
  {
    "name": "Boite 1",
```

```

        "capacity":3,
        "open":true
    },
    {
        "name":"Boite 2",
        "capacity":6,
        "open":false
    },
    {
        "name":"Boite 3",
        "capacity":-1,
        "open":true
    }
]

```

6.1 Méthode statique en java

On peut ajouter à la classe `Box` une méthode statique `Box.fromJSON()` qui permet de créer une liste de boîtes à partir d'un objet JSON du type ci-dessus. On pourra par exemple utiliser en java la librairie pour Gson parser du JSON; son jar est disponible sur Célène. Voir la documentation officielle sur <https://www.javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/module-summary.html>, et quelques exemples dans le manuel utilisateur <https://github.com/google/gson/blob/master/UserGuide.md#object-examples>. Pour utiliser la classe Gson, il faut inclure le jar de gson dans le *classpath*, et importer la classe Gson avec `import com.google.gson.Gson;`

Pour ouvrir un fichier en lecture en java et l'utiliser dans gson, il faut :

- i Créer une instance de `FileReader` avec :

```
FileReader fr = new FileReader("chemin/du/fichier.json");
```

- ii Passer l'objet `fr` à la méthode `fromJson` d'un objet de la classe `Gson` :

```
Gson gson = new Gson();
```

```
Box ma_boite = gson.fromJson(fr, Box.class);
```

6.2 Backlog utilisant JSON

- on voudrait pouvoir construire une liste de choses à partir d'une description en JSON. Il faudrait pour cela une méthode statique `Thing.fromJSON(data)` construisant une chose à partir des données la concernant.
- on voudrait pouvoir construire une liste de boîtes et de choses à partir d'une description en JSON. Pour cela, il faudrait que chaque élément de la liste JSON ait une clé `type` ayant pour valeur soit `box` soit `thing`. Il faudrait aussi une fonction globale `listFromJSON()` qui prenne en argument une liste de données obtenue de JSON et pour chacune construise soit une boîte soit une chose selon la valeur de la clé `type`.
- on voudrait pouvoir faire une sauvegarde JSON complète d'une liste de boîtes et de leur contenu.