Ce projet a été réalisé par **Johan GIRARD** et **Pierre ODIN** dans le cadre du cours de JML du Master 2 Pro. GI. Les tests ont été réalisés sous l'environnement Linux des machines de l'UFR IM²AG.

Rapport Projet JML

"Stockage de produits dangereux"

1 Lecture et test d'invariant

1.1 Invariant n° 1

Description de l'invariant : Il doit y avoir au maximum 50 incompatibilités renseignées dans le tableau `incomp`. La valeur de la variable `nb_inc` doit être positive et doit être inférieur à 50 car ce nombre est utilisé pour gérer les indices du tableau `incomp`, il doit donc être situé entre l'indice minimum (0) et l'indice maximum (49).

Test invalidant l'invariant : Il s'agit de `testFailInvariant1()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test fait trop d'appels à `add_incomp(...)` et `nb_inc` devient supérieur à 49.

1.2 Invariant n° 2

Description de l'invariant : Il doit y avoir au maximum 30 affectations renseignées dans le tableau `assign`. La valeur de la variable `nb_assign` doit être positive et doit être inférieur à 30 car ce nombre est utilisé pour gérer les indices du tableau `assign`, il doit donc être situé entre l'indice minimum (0) et l'indice maximum (29). Cet invariant indique également qu'il peut y avoir au maximum 30 bâtiments différents répertoriés dans `assign`.

Test invalidant l'invariant : Il s'agit de `testFailInvariant2()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test fait trop d'appels à `add_assign(...)` et `nb_assign` devient supérieur à 29.

1.3 Invariant n° 3

Description de l'invariant : Tous les produits doivent être référencés par un nom ayant pour préfixe `"Prod"` dans la liste des incompatibilités (⇔ toutes les valeurs définies dans le tableau `incomp` doivent commencer par `"Prod"`).

Test invalidant l'invariant : Il s'agit de `testFailInvariant3()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test fait un ajout d'une incompatibilité dont le nom des produits ne commence pas par `"Prod"` (un seul nom de produit ne commençant pas par `"Prod"` serait suffisant pour invalider l'invariant).

1.4 Invariant n° 4

Description de l'invariant : Tous les bâtiments doivent être référencés par un nom ayant pour préfixe `"Bat"` et tous les produits doivent être référencés par un nom ayant pour préfixe `"Prod"` dans la liste des affectations. Cela signifie que toutes les valeurs définies dans le tableau `assign` doivent commencer par `"Bat"` pour le premier élément et par `"Prod"` sur le second.

Test invalidant l'invariant : Il s'agit de `testFailInvariant4()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test fait un ajout d'une assignation dont le nom du bâtiment ne commence pas par `"Bat"` et le nom du produit ne commence pas par `"Prod"` (une seule de ces deux irrégularités serait suffisante pour invalider l'invariant).

1.5 Invariant n° 5

Description de l'invariant : Un produit ne doit pas être incompatible avec lui-même. Cela signifie que le tableau `incomp` ne doit pas avoir une entrée avec deux fois le même produit.

Test invalidant l'invariant : Il s'agit de `testFailInvariant5()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test fait un ajout d'une incompatibilité où la paire de produits contient deux fois le même produit.

1.6 Invariant n° 6

Description de l'invariant : Si un produit `X` est incompatible avec un produit `Y`, alors le produit `Y` est incompatible avec le produit `X`. Cela signifie que le tableau `incomp` doit contenir deux entrées pour chaque incompatibilité : la première de la forme `[X][Y]` et l'autre de la forme `[Y][X]`.

Test invalidant l'invariant : Il s'agit de `testFailInvariant6()` dans `TestExplosivesJUnit4Public.java`.

Description du test : La fonction `add_incomp(...)` ajoutent deux entrées dans le tableau `incomp` (`[X][Y]` et `[Y][X]`) à chaque appel. Il n'est donc pas possible d'invalider cette propriété en utilisant uniquement les méthodes de la classe. Le test modifie donc un attribut public de la classe pour invalider l'invariant : une incompatibilité est ajoutée, puis la deuxième entrée du tableau est retirée et enfin un appel à `skip()` est fait.

1.7 Invariant n° 7

Description de l'invariant : Deux produits assignés au même bâtiment ne doivent pas être incompatibles. Cela signifie que si le tableau `assign` contient une entrée `[B][P1]` et une entrée `[B][P2]`, alors les entrées `[P1][P2]` et `[P2][P1]` ne doivent pas se trouver dans le tableau `incomp`.

Test invalidant l'invariant : Il s'agit de `testFailInvariant7()` dans `TestExplosivesJUnit4.java`.

Description du test : Le test ajoute des incompatibilités entre des produits, puis il ajoute deux assignations dans le même bâtiment de deux produits déclarés incompatibles.

2 Calcul de préconditions

2.1 Méthode add_incomp(...)

La précondition réalisée pour la méthode `add_incomp(...)` est la suivante :

```
/*@ requires (0 <= nb_inc+2 && nb_inc+2 < 50) ;
  @ requires (prod1.startsWith("Prod") && prod2.startsWith("Prod")) ;
  @ requires (!prod1.equals(prod2)) ;
  @*/
```

Cette précondition vérifie trois éléments :
- La valeur de `nb_inc` en entrée puis modifiée (ajout de 2 comme dans l'implémentation de la méthode) est comprise en 0 et 49.
- Les arguments de la méthode sont préfixés par `"Prod"`.
- Les deux arguments ne sont pas égaux.

En ajoutant cette précondition, les tests suivants deviennent inconclusifs :
→ `testFailInvariant1();testFailInvariant3();testFailInvariant5()`

2.2 Méthode add_assign(...)

La précondition réalisée pour la méthode `add_assign(...)` est la suivante :

```
/*@ requires (0 <= nb_assign+1 && nb_assign+1 < 30) ;
  @ requires (bat.startsWith("Bat") && prod.startsWith("Prod")) ;
  @ requires (\forall int i; 0 <= i &&  i < nb_assign;
  @             (\forall int j; 0 <= j && j < nb_inc;
  @                ( (assign[i][0].equals(bat) && incomp[j][0].equals(assign[i][1]))
  @                ==> ( !incomp[j][1].equals(prod) )
  @          ))) ;
  @*/
```

Cette précondition vérifie trois éléments :
- La valeur de `nb_assign` en entrée puis modifiée (ajout de 1 comme dans l'implémentation de la méthode) est comprise en 0 et 29.
- Le premier argument de la méthode est préfixé par `"Bat"` et le second par `"Prod"`.
- Il n'y a pas de produits incompatibles avec le produit passé en argument dans le bâtiment passé en argument. Cela se traduit par une précondition qui vérifie que s'il existe un produit incompatible avec un produit du bâtiment passé en argument, alors ce produit n'est pas celui passé en argument.

En ajoutant cette précondition, les tests suivants deviennent inconclusifs :
→ `testFailInvariant2();testFailInvariant4();testFailInvariant7()`

3 Recherche d'un bâtiment

Précision sur l'énoncé : La méthode `findBat(...)` retourne un nouveau nom de bâtiment dans le cas où aucun des bâtiments référencés dans le tableau `assign` ne peut recevoir le nouveau produit. La solution *"trop simple"* qui consiste à retourner un nouveau bâtiment à chaque appel n'est donc pas autorisée avec cet énoncé. Cependant, nous fournissons tout de même des méthodes et un fichier de test permettant de mettre en évidence les différences entre la solution *"trop simple"* et notre solution pour `findBat(...)`.

3.1 Méthode compatible(...) et existe_bat(...)

```
/*@ requires (prod1.startsWith("Prod") && prod2.startsWith("Prod")) ;
  @ ensures (\result == true) ==>
  @     (\forall int i; 0 <= i && i < nb_inc;
  @         !(incomp[i][0].equals(prod1) && incomp[i][1].equals(prod2)) ) ;
  @ ensures (\result == false) ==>
  @     (\exists int i; 0 <= i && i < nb_inc;
  @         incomp[i][0].equals(prod1) && incomp[i][1].equals(prod2) ) ;
  @*/
public /*@ pure @*/ boolean compatible (String prod1, String prod2){ ... }
```

La méthode `compatible(...)` teste si deux produits sont déclarés comme compatibles. Sa spécification vérifie d'une part que les arguments sont valides et d'autre part que la méthode retourne `true` que si les produits sont compatibles et `false` que s'ils ne le sont pas.

```
/*@ requires (bat.startsWith("Bat")) ;
  @ ensures (\result == true) ==>
  @     (\forall int i; 0 <= i &&  i < nb_assign;
  @         !assign[i][0].equals(bat)) ;
  @ ensures (\result == false) ==>
  @     (\exists int i; 0 <= i && i < nb_assign;
  @         assign[i][0].equals(bat));
  @*/
public /*@ pure @*/ boolean existe_bat (String bat){ ... }
```

La méthode `existe_bat(...)` teste si un bâtiment stocke au moins un produit. Sa spécification vérifie d'une part que l'argument est valide et d'autre part que la méthode retourne `true` que si le bâtiment n'apparaît pas dans `assign` et `false` que s'il y a au moins une entrée dans `assign` pour le bâtiment.

3.2 Méthode findBat(...)

```
/*@ requires (prod.startsWith("Prod")) ;
  @ ensures (\result.startsWith("Bat")) ;
  @ ensures (\forall int i; 0 <= i &&  i < nb_assign;
  @     (assign[i][0].equals(\result)) ==> (compatible(assign[i][1],prod)) );
  @ ensures (existe_bat(\result)) ==>
  @     (\forall int i; 0 <= i && i < nb_assign;
  @       (\exists int k; 0 <= k && k < nb_assign;
  @         (assign[i][0].equals(assign[k][0]) && !compatible(assign[k][1],prod))));
  @*/
```

Cette spécification vérifie les éléments suivants :

- L'argument et le résultat sont syntaxiquement valides.
- Le produit est stocké dans un bâtiment où il est compatible avec tous les autres produits stockés dans ce bâtiment.
- Si le bâtiment retourné ne stockait aucun produit avant l'appel à `findBat(...)`, alors cela signifie que le produit ne peut être stocké dans aucun des bâtiments déclarés dans `assign`. Cette post-condition est utilisé pour interdire la solution *"trop simple"*.

Le principe de la méthode `findBat(...)` est le suivant :

```
public String findBat (String  prod){
    // On crée une liste de tous les bâtiments (L1)
    // On crée une liste de bâtiments qui ne peuvent pas recevoir le produit (L2)
    // On parcourt le tableau "assign" pour remplir ces deux listes
            // On test la compatibilité
            // Si le bâtiment n'est pas dans (L1), on l'ajoute dans (L1)
            // Si le bâtiment contient un produit incompatible, on l'ajoute dans (L2)
    // On cherche le premier bâtiment de (L1) qui n'est pas dans (L2)
    // S'il n'y en a pas, on retourne un nouveau bâtiment
}
```

La méthode est testée dans le fichier `TestExplosivesJUnit4FindBat.java`. Le test le plus intéressant est implémenté dans le fichier `testFindBat7()`. Il constitue une liste de 17 produits, il crée ensuite plusieurs incompatibilités entre certains de ces produits et enfin il cherche un bâtiment pour chaque produit. La trace des assignations permet de vérifier que les incompatibilités ne sont pas violées.

⚠ L'implémentation de `findBat(...)` retourne le premier bâtiment qui peut stocker le produit. La répartition proposée après un test de ce type est donc valide mais elle n'est pas forcément optimale (cela dépend de l'ordre d'insertion dans le tableau `assign`). Une amélioration possible pourrait être de ré-organiser les assignations dans `findBat(...)` pour toujours assurer que la répartition utilise le moins de bâtiments possibles.

3.3 Comparaison des méthodes findBatSimple(...), findBatSimpleInterdit(...) et findBat(...)

Le fichier de `TestExplosivesJUnit4FindBatVSFindBatSimple.java` permet de mettre en évidence le rôle de la post-condition que nous avons ajoutée pour éviter la solution *"trop simple"*. Le test réalisé est celui décrit dans la section précédente avec à chaque fois une méthode différente. Le tableau suivant résume cette comparaison :

Méthode utilisée	**Post-condition anti-*"trop simple"***	**Implémentation *simple***	**Résultat du test**
`findBatSimple(...)`	✗	✓	`Validé` : 1 produit = 1 bâtiment
`findBatSimpleInterdit(...)`	✓	✓	`FAILURE`
`findBat(...)`	✓	✗	`Validé` : 17 produits → 3 bâtiments