Question 1 - lecture et test d'invariant

Propriété 1 :

Le variable incomp est un tableau 2 dimensions [50][2], donc le nombre de paires de produits incompatibles ne peut pas dépasser à 50. Il faut contraindre nb_inc entre 0 et 50.

Propriété 2 :

Le variable assign est un tableau 2 dimensions [30][2], donc le nombre de paires d'entités ne peut pas dépasser à 30. Il faut contraindre nb_assign entre 0 et 30.

Propriété 3 :

Tous les paires d'éléments dans le tableau incomp, il faut commencer par "Prod".

Propriété 4 :

Pour tous les paires d'éléments dans le tableau assign, il faut verifier que tous sont bien typées comme dessus:
Le premier instance dans le couple est un bâtiment, alors il faut commencer par "Bat".
Deuxième instance dans le couple est un produit, alors il faut commencer par "Prod".

Propriété 5 :

Il faut vérifier que les deux instances au paires sont différents dans le tableau incomp.
CTD on ne peut pas avoir un produit qui est incompatible avec soi-même.

Propriété 6 :

Pour chaque paire d'instances dans le tableau incomp, les deux instances au paire, ils ont un ordre. S'il existe un couple (A , B) dans le tableau incomp, il faut vérifier qu'il existe un entité à l'inverse(B, A).

Propriété 7 :

A ==> B égale avec !A||B,
CTD, si A est vrai, alors B est obligatoirement vrai;
sinon A est faux, on ne vérifie pas B.
Donc pour ce cas où, soit les produits ne sont pas dans le même bâtiment, soit si ils déposent dans le même endroit, ils doivent être compatibles.
Cette invariant est pour assurer qu'il n'y pas des produits incompatibles qui sont déposés dans le même bâtiment.

Question 2 - Calcul de préconditions

Fonction add_incomp :

```
/*@requires prod1 != null;
  @requires prod2 != null;
  @requires nb_inc < 48;
  @requires !prod1.equal(prod2);
  @requires prod1.startsWith("Prod") && prod2.startsWith("Prod")
  @*/
```

- Pour chaque paramètre, il faut vérifier qu'il n'est pas null.
- En même temps, il faut assurer que le variable nb_inc est inférieur à 50, alors nb_inc

+ 2 < 50, donc comme une précondition, on doit assurer nb_inc < 48.

- Chaque paire de produits, ils doivent être différents.
- Il faut vérifier que chaque paire paramètres sont bien typés. Ils commencent par « Prod ».

Fonction add_assign

```
/*@requires bat != null;
  @requires prod != null;
  @requires nb_assign < 29;
  @requires bat.startsWith("Bat") && prod.startsWith("Prod");
  @requires
  @       (/forall int i; 0 <= i && i < nb_assign;
  @             (assign[i][0].equals(bat)) ==>
  @             !(/exists int j; 0 <= j && j < nb_inc;
  @                   (assign[i][1].equals(incomp[j][0]) && incomp[j][1].equals(prod))));
  @*/
```

- Pour chaque paramètre, il faut vérifier qu'il n'est pas null.
- En même temps, il faut assurer que le variable nb_assign est inférieur à 30, alors nb_inc + 1 < 30, donc comme une précondition, on doit assurer nb_assign < 29.
- Assurer chaque paramètre est bien typé. Pour un bâtiment, il faut commence par « Bat », et alors pour un produit, il faut commence par « Prod ».
- Pour chaque paire de paramètre, on doit assurer que le produit qu'on veut stocker est compatible avec tous les autres produits qui sont déposés dans ce même bâtiment.

Fonction findBat

```
//@requires prod != null;
//@requires prod.startsWith("Prod");
```

- Assurer que prod n'est pas null .
- Assurer que le paramètre est bien typé, il commence par « Prod » .

Fonction compatible

```
//@requires !prod1.equals(prod2);
//@requires prod1 != null;
//@requires prod2 != null;
/*@ensures \result == true <==>
  @             (/forall int i; 0<=i && i<nb_inc;
  @                   !(incomp[i][0].equals(prod1) && incomp[i][1].equals(prod2)));
  @*/
```

- Assurer que ces deux produits sont pas le même.
- Vérifier que ces deux produits sont pas null.
- Après d'exécution de cette fonction, on doit vérifier le résultat retourné est correct. Il on donne vrai, si et seulement si, dans le tableau incomp, il n'existe pas un entité qui corresponds au le paire de paramètre.

Question 3 - Recherche d'un bâtiment

```
/*@requires //specifier en JML
```

```
@(/exists int i; 0<=i && i<nb_assign;
@       (/forall int j; 0<=j && j<nb_assign;
@             (assign[i][0].equals(assign[j][0]) ==>
@                   (!compatible(assign[j][1], prod))));
@*/
```

Comme une précondition, on va tester s'il existe un bâtiment, tous les produits dedans sont compatibles avec le paramètre.

```
public /*@ pure @*/ String findBat(String prod){
        boolean trueFlag = false;
        for(int i = 0; i < nb_assign; i++){
            for(int k = 0; k < nb_assign; k++){
                if(!assign[i][0].equals(assign[k][0]))
                    continue;
                if(!compatible(assign[k][1], prod)){
                    trueFlag = false;
                    break;
                }
                else
                    trueFlag = true;
            }
        if(trueFlag == true)
            return assign[i][0];
        }
        return null;
    }


public /*@ pure @*/ boolean compatible(String prod1, String prod2){
        for( int i = 0; i < nb_inc; i++){
            if(incomp[i][0].equals(prod1) && incomp[i][1].equals(prod2)){
                return false;
            }
        }
        return true;
    }
```