# Question 1 - lecture et  test d'invariant

## Propriété 1 :

Le variable incomp est un tableau 2 dimensions [50][2], nb_inc est le nombre d'incompatibilites stockees dans le tableau incomp. Ce nombre est contraint par la taille du tableau incomp (donc inferieur a 50), et comme il represente un `compteur`ne peut etre inferieur a 0.

## Propriété 2 :

Le variable assign est un tableau 2 dimensions [30][2], nb_assign est le nombre d'assignation stockees dans le tableau assign. Ce nombre est contraint par la taille du tableau assign (donc inferieur a 30), et comme il represente un `compteur`ne peut etre inferieur a 0.

## Propriété 3 :

Contraint les elements contenus dans le tableau nb_inc. Dans ce tableau il s'agit de produit, et tous les produits sont identifies par le prefixe `Prod`.

## Propriété 4 :

Contraint les elements contenus dans le tableau nb_assign. Dans ce tableau, nous associons des batiments a des produits. Tous les produits sont identifies par le prefixe `Prod`. Tous les batiments sont identifies par le prefixe `Bat`.

## Propriété 5 :

Verifie qu'un produit n'est pas marque incompatible avec lui meme.

## Propriété 6 :

Verifie que si un produit A est marque incompatible qvec le produit B, le produit B est marque incompatible avec le produit A.

## Propriété 7 :

Verifie que pour tous les produits stockes, 2 produits incompatibles ne sont pas stockes dans le meme batiment.

# Question 2 - Calcul de préconditions

## Fonction add_incomp :

```
/*@requires prod1 != null;
  @requires prod2 != null;
  @requires nb_inc +2 < 50;
  @requires !prod1.equals(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 +2 < 50.
- Chaque paire de produits, ils doivent être différents.
- Il faut vérifier que chaque paire de paramètres sont bien typés en commencant par « Prod ».

## Fonction add_assign

```
/*@requires bat != null;
  @requires prod != null;
  @requires nb_assign +1 < 30;
  @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 +1 < 30.
- Assurer chaque paramètre est bien typé. Pour un bâtiment, il faut commence par « Bat », et 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)));
      @*/
```

- Vérifier que ces deux produits sont pas null et que les deux produits commencent par Prod. Le méthode est désignée comme pure et on peut les utiliser dans le autre spécifications de JML.
- 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)))));
  @*/

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

On spécifie cette question en JML comme une précondition, on va tester s'il existe un bâtiment, tous les produits dedans sont compatibles avec le paramètre « prod ».  S'il n'existe pas un batiment compatible, il va enlever une JML exception.

Pour la fonction findBat, il y a deux  façon à faire:

1ere moyenne

```
public /*@ pure @*/ String findBat(String prod) {
        /* code -2 means not compatable.
           -1 means compatible but product isnt already in the building.
           >=0 is the index of the building where the product is already found.
           */
        //un map qui contient notre list de batiment availible.
        HashMap batList = new HashMap();
        for (int i = 0; i < nb_assign; i++) {
            String bat = assign[i][0];
            String product = assign[i][1];
            if (batList.containsKey(bat)) {
            //our map already contains the building  we only
            // change the marker of a building if it becomes incompatible or to mark it as
            // already containing the product.
                if (!compatible(prod, product)) {
                  //mark the building as incompatible
                   batList.put(bat, Integer.valueOf(-2));
                }
                else if (((Integer)batList.get(bat)).intValue() ==-1 && prod.equals(product))
                {
                    //mark the building already containing the prod.
                    batList.put(bat, Integer.valueOf(i));
                }
            }
            else {  //building doesnt exist added and set intial building marker code.
                System.out.println("bulding doesnt exsit adding it to the map");
                if (compatible(prod, product)) {
                    if (prod.equals(product)) {
                        //add the batiment to the array with index of building containing
                        // the prod already
                        batList.put(bat, Integer.valueOf(i));
                        System.out.println(" added at index " + i);
                    } else {
                        //add the name of the building with a marker that it is compatible for
                        // the product.
                        batList.put(bat, Integer.valueOf(-1));
                        System.out.println("compatible but isnt the same");
                    }
                }
                else {
```

```
                //add building with marker that the building is not compatable
                System.out.println("not comp new building gets a -2");
                batList.put(bat, Integer.valueOf(-2));
            }
        }
    }
    String bestBuildingMatch= null; //compatible and has the product
    String secondBuildingMatch= null; //compatible but doesnt have the product
    System.out.println("new map size "+batList.size());
    //find the best matching building
    for (Iterator it = batList.entrySet().iterator(); it.hasNext();) {
        Map.Entry pairs = (Map.Entry) it.next();
        System.out.println(pairs.getKey() + " = " + (Integer)pairs.getValue());
        int comp = ((Integer) pairs.getValue()).intValue();
        String name = pairs.getKey().toString();
        if(comp >=0){
            bestBuildingMatch = name;
        }
        else if(comp == -1){
            secondBuildingMatch = name;
        }
    }
    //return the best matching building
    System.out.println("batlist? "+batList.values());
    if(bestBuildingMatch !=null){
        return bestBuildingMatch;
    }
    else if (secondBuildingMatch !=null){
        return secondBuildingMatch;
    }
    else{   //no compatible building found so we create a new one
        int count = 0;
        while(true) { //loop until we find a name that doesnt exsist.
            String name = "Bat_" + count;
            if (! batList.containsKey(name)) {
                return name;
            } else {
                count++;
            }
        }
    }
}
```

## 2eme moyenne

```
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;
    }
```

## Fonction compatible

```
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;
    }
```