Vérification et Preuves Formelles

Projet 1 : Logique de Séparation avec Verifast

M1 ALMA

■ ■ ■

Ce projet devra être réalisé en binôme (ou exceptionnellement seul). Une archive contenant un répertoire nommé **Projet_Nom1_Nom2** disposant :

* d'un fichier **Usine1.java** contenant le code Java et les annotations Verifast du projet pour les questions 1 à 12 incluses ;
* d'un fichier **Usine2.java** contenant le code Java et les annotations Verifast du projet pour les questions 13 à 18 incluses ;
* d'un rapport (au format pdf) **rapport.pdf**

devra être rendu au plus tard le **jeudi 20 décembre à 23h59** à l'adresse guilhem.jaber@univ-nantes.fr, avec comme sujet de l'email **[Projet VPF] Nom1 Nom2**. Tout retard ou non-respect de ces consignes entraînera un malus sur la note.
Toute questions sur ce projet pourra être adressée à guilhem.jaber@univ-nantes.fr avec comme préfixe du sujet de l'email **[Projet VPF]**. Les questions devront être précises, et l'on s'assurera avant d'envoyer une question que sa réponse n'est pas incluse dans les explications de Verifast ci-dessous.
Vous pourrez indiquer factuellement dans le rapport les difficultés que vous avez rencontré lorsque vous n'êtes pas arrivés à réaliser une des questions.

On désactivera la vérification de dépassements d'entier de Verifast en décochant l'option "Check arithmetic overflow" dans le menu "Verify".
Lorsque l'on demande par la suite de spécifier une méthode ou un constructeur, il s'agit de fournir une pre et une post-condition en utilisant les annotations //@ *requires* ...; (pour la pre-condition) et //@ *ensures* ...; (pour la post-condition). Ces annotations doivent être placées entre la déclaration de la méthode ou du constructeur, et l'accolade ouvrante précédant le code de la méthode ou du constructeur. Elles se terminent par un point-virgule.
On utilisera toujours l'opérateur == (et jamais =) pour spécifier l'égalité entre deux variables ou valeurs dans une spécification ou un prédicat. Pour spécifier le champ f d'un objet o, on utilisera l'opérateur **o.f |-> v** où v pourra être une variable (potentiellement une variable fantôme, qui sont introduites avec le préfixe ? devant le nom de la variable). La conjonction de séparation **&*&** (et non pas la conjonction "standard" **&&**) devra être utilisée dès lors que l'on spécifie les champs d'un objet. Pour définir un prédicat en fonction d'une condition booléenne, il est possible d'utiliser l'opérateur ternaire de Java **b ? p1 : p2** dans les spécifications.
Finalement, les prédicats, qui permettent de spécifier les objets d'une classe, sont définis via l'annotation //@ *predicate* ...; qui doit être placé **à l'extérieur de la classe**. On distinguera bien l'utilisation du point virgule et de la virgule dans la liste des arguments du prédicat, qui permet de contrôler la **précision** (le fait que Verifast se permet ou non d'ouvrir automatiquement un prédicat lorsqu'il est appliqué sur *this*).

L'objectif de ce projet est de concevoir et de vérifier avec **Verifast** un programme Java permettant de modéliser une **usine** qui fait effectuer par des **travailleurs** des **taches**, et ainsi accumule ou perd de l'argent.

1. Définir une classe *Tache* disposant de champs privés de type *int* :

* *temps_necessaire* pour indiquer le temps nécessaire pour réaliser la tache ;
* *gain* pour indiquer le gain financier obtenu après avoir réalisé la tache.

Définir de plus un constructeur *Tache*(*int temps_necessaire*, *int gain*) et deux accesseurs *get_temps_necessaire*() et *get_gain*() pour obtenir la valeur des deux champs de la classe.

2. Définir un prédicat *tache*(*Tache tache*; *int temps_necessaire*, *int gain*) permettant de spécifier les deux champs privés d'un objet *tache*. Il s'assurera que les valeurs de ces deux champs sont toujours strictement positives. L'utiliser pour spécifier le plus précisément possible le constructeur et les deux accesseurs définis précédemment.

3. Définir une classe *Travailleur* disposant de champs privés de type *int* :
- *temps_dispo* pour indiquer le temps de travail disponible du travailleur ;
- *salaire_horaire* pour indiquer le salaire que le travailleur gagne pour une heure de travail ;
- *salaire_percu* pour indiquer le salaire perçu jusqu'à présent par le travailleur.

Définir de plus :
- un constructeur *Travailleur*(*int temps_dispo*, *int salaire_horaire*) (qui initialise *salaire_percu* à 0) ;
- un accesseur *get_temps_dispo*() permettant d'obtenir la valeur du champ *temps_dispo* ;
- un accesseur *get_salaire_horaire*() permettant d'obtenir la valeur du champ *salaire_horaire* ;
- un accesseur *get_salaire_percu*() permettant d'obtenir la valeur du champ *salaire_percu*.

4. Définir un prédicat

travailleur(*Travailleur travailleur*; *int temps_dispo*, *int salaire_horaire*, *int salaire_percu*)

permettant de spécifier les trois champs privés d'un objet travailleur.
Utiliser ce prédicat pour spécifier le plus précisément possible le constructeur et les accesseurs de la classe *Travailleur*.

5. Définir une méthode *public int travailler*(*int t*) dans la classe *Travailleur* qui permet au travailleur courant (i.e. *this*) d'effectuer t heures de travail, qui seront déduites de son temps disponible, et de percevoir un salaire pour ces t heures de travail (calculé en utilisant *salaire_horaire*) qui est rajouté à son salaire perçu. Cette fonction retournera de plus le salaire perçu par le travailleur pour ces t heures.
La spécifier la plus précisément possible. Imposer de plus que cette méthode ne puisse être utilisée qu'avec un argument t positif.

6. Supposons que l'on souhaite modifier la classe *Travailleur* pour qu'il stocke un champ *temps_travaille*, représentant le temps de travail effectué auparavant, à la place de *salaire_percu*. Le salaire perçu pouvant ensuite être calculé en utilisant le salaire horaire. On veut de plus ne pas avoir à modifier les spécifications du constructeur et des méthodes de la classe *Travailleur*.
Expliquer brièvement dans le rapport les modifications à apporter à la définition du prédicat

travailleur(*Travailleur travailleur*; *int temps_dispo*, *int salaire_horaire*, *int salaire_percu*)

et au code de la classe *Travailleur* qu'il faudrait effectuer (Il n'est pas demandé d'intégrer ces modifications au code source que vous fournirez).

7. Définir une classe *Usine* disposant d'un champ privé *balance* de type *int* indiquant de combien d'argent elle dispose. Définir de plus :
- un constructeur *Usine*(*int depot_initial*) qui crée un objet dont le champ *balance* vaut *depot_initial* ;
- un accesseur *get_Balance*() permettant d'obtenir la valeur du champ *balance* ;
- une méthode *depose_argent*(*int argent*) permettant de déposer le montant *argent* (qui peut potentiellement être négatif) à la balance de l'objet courant.

8. Définir un prédicat *usine*(*Usine usine*; *int balance*) permettant de spécifier le champ *balance* d'un objet *usine* de la classe *Usine*. L'utiliser pour spécifier le plus précisément possible le constructeur, l'accesseur *getBalance* et la méthode *depose_argent* de la classe *Usine*.

9. Définir dans la classe *Usine* une méthode *effectueTache*(*Tache tache*, *Travailleur travailleur*) qui fait effectuer une tache par un travailleur. Le montant du salaire nécessaire par le travailleur est alors soustrait à la balance de l'objet courant, et le gain obtenu pour avoir réalisé la tache est ajouté à la balance de l'objet courant. Vous utiliserez la méthode *travailler* de la classe *Travailleur* pour cela, en vous assurant que le travailleur a suffisamment de temps disponible pour effectuer cette tache. Il pourra être utile de rajouter les annotations

```
//@ open tache(tache,_,_);
//@ open travailleur(travailleur,_,_,_);
```

dans le code de cette méthode pour que Verifast accepte la spécification.

10. Rajouter une classe *UsineTest* disposant d'une fonction

```
public static void main(String[] args)
//@requires true;
//@ensures true;
{
...
}
```

et compléter cette fonction *main* pour tester les fonctionnalités et les spécifications du programme. On pourra utiliser le mot-clé Java **assert** pour vérifier avec Verifast que les spécifications sont correctes.

11. On souhaite maintenant s'assurer que le champ *temps_dispo* d'un objet de la classe *Travailleur* soit toujours positif ou nul. Modifier votre programme et ses spécifications pour y parvenir. Expliquer dans le rapport les difficultés que vous avez pu rencontrer, et comment Verifast a pu vous aider pour déterminer les modifications à effectuer.

12. Rajouter une fonction

public static boolean est_rentable(*Tache tache*, *Travailleur travailleur*)

dans la classe Usine pour déterminer si le gain financier d'une tache est supérieur à son cout salarial. La spécifier le plus précisément possible.
Utiliser cette fonction pour s'assurer que l'on effectue une tache que lorsqu'elle est rentable dans la fonction *effectueTache*. Modifier sa spécification en conséquence.

13. On souhaite ne pas stocker directement la balance dans la classe *Usine*, mais plutôt les dépenses de salaire et les gains accumulées lors de la réalisation des taches. Modifier le code en conséquence, et expliquer votre démarche dans le rapport. Ces modifications, de même que celles demandées dans les questions suivantes, devront maintenant être effectuées dans un nouveau fichier **Usine2.java**.
Rajouter dans la fonction *main* de *TestUsine* des instructions pour tester ces modifications.

14. Vérifier dans la fonction *main* de la classe *TestUsine* que Verifast accepte d'effectuer plusieurs fois la même tache.
On souhaite interdire cela, en ne modifiant que les spécifications mais pas le code Java. Indiquer dans le rapport la modification la plus simple possible que vous pouvez réaliser pour le garantir, et l'implémenter.

15. On souhaite maintenant qu'un travailleur soit d'abord embauché par une usine avant de pouvoir effectuer un tache pour elle. On veut pouvoir garantir cette propriété en enrichissant les spécifications.

Pour cela, on rajoute le prédicat abstrait **non-précis** (qui ne sera donc pas ouvert ou fermé automatiquement par Verifast)

```
//@ predicate estEmbauche(Usine usine, Travailleur travailleur) = true;
```

On utilisera donc l'annotation

```
//@ open estEmbauche(usine, travailleur);
```

pour consommer ce prédicat, et l'annotation

```
//@ close estEmbauche(usine, travailleur);
```

pour le générer.
On rajoute de plus la fonction *embaucher* :

```
public void embaucher(Travailleur travailleur)
/*@ requires travailleur(travailleur, ?temps_dispo,
      ?salaire_horaire, ?salaire_percu);@*/
/*@ ensures travailleur(travailleur, temps_dispo, salaire_horaire
      , salaire_percu) &*& estEmbauche(this, travailleur); @*/
{
  //@close estEmbauche(this, travailleur);
}
```

Noter que le corps de cette fonction ne contient qu'une annotation mais aucune instruction.
Modifier la spécification de la méthode *effectueTache* pour s'assurer qu'un travailleur doit d'abord être embauché avant d'effectuer une tache
Rajouter dans la fonction *main* de *TestUsine* des instructions pour tester ces fonctions.
Expliquer dans le rapport comment fonctionne cette spécification, et s'il est possible de contourner cette garantie.

16. En suivant le même esprit que la fonction *embaucher*, définir et spécifier dans la classe *Usine* une fonction *public void licencier*(*Travailleur travailleur*) qui permet de licencier un travailleur, qui ne pourra alors plus effectuer de taches.
Rajouter dans la fonction *main* de *TestUsine* des instructions pour tester ces fonctions.
Expliquer dans le rapport les limitations de cette approche.

17. Modifier votre programme et ses spécifications pour que la fonction *effectueTache* ne puisse plus être appelée sur un travailleur à partir du moment où le temps de travail disponible du travailleur atteint zéro (i.e. Verifast doit produire une erreur lorsque cela arrive).
Expliquer votre approche dans le rapport et rajouter dans la fonction *main* de *TestUsine* des instructions pour tester cette possibilité.

18. Résumer brièvement les garanties que nous apporte Verifast lorsqu'il valide le code d'un programme utilisant les fonctionnalités de la classe *Usine*. Quelles difficultés aurait-on rencontrées pour obtenir les mêmes garanties si on avait du les imposer directement dans le code source, sans utiliser de spécification ?