Master 1 Informatique 2009/2010 P. Collet
UNS 8/12/2009

Génie logiciel
1ère session

Durée : 2 heures.
Seuls documents autorisés : notes de cours.

Remarques : Toute ambiguïté que vous pourriez rencontrer dans ce sujet devra être résolue en décrivant brièvement le choix que vous avez fait.

1. Questions rapides (10 pts)

a) Expliquez pourquoi il existe deux types de propriétés (PATH et CLASSPATH) pour manipuler les chemins dans l'outil ant.

b) Expliquez pourquoi, dans un outil comme ant, les valeurs des propriétés passées en paramètre de la commande masquent toujours les valeurs définies dans le fichier *build.xml.*

c) Dans quels cas est-il nécessaire d'utiliser la primitive *Class.forName(« nom de la classe »)* plutôt que le suffixe *.class* après le nom la classe à charger ? Donnez un exemple.

d) Expliquez à quoi servent les annotations @Before et @BeforeClass en JUnit 4. Expliquez notamment pourquoi il y a deux annotations différentes.

e) Expliquez les avantages qu'a l'architecture JUnit 4 par rapport à JUnit 3 en utilisant le mécanisme d'annotations (@Test, @Before,…) pour organiser les tests unitaires.

f) Expliquez pourquoi un compilateur Java5/6 refuse de compiler le code suivant en pointant une erreur de typage sur l'appel à la méthode *add* à la dernière ligne (BebeteEmergente est bien une sous-classe de Bebete) :

```
ChampDeBebetes champ = new ChampDeBebetes(640,480,200);
ArrayList<? extends Bebete> l = new ArrayList<BebeteEmergente>();
l.add(new BebeteEmergente(champ, 200, 200, 0, 10, Color.RED));
```

g) Expliquez quel est le surcoût engendré par l'implémentation en Java d'un héritage par mariage de convenance (simulation d'un héritage multiple par un héritage simple de la première classe, et l'utilisation d'un attribut vers la seconde classe avec délégation des méthodes).

2. Etude de cas (10 pts)

On se propose de développer une extension à JUnit pour automatiser certaines vérifications d'oracle à partir de propriétés logiques censées être vraies avant ou après l'appel de la méthode testée.
La première étape consiste à imposer une convention de nommage aux classes testées. Pour chaque méthode à tester, on peut ajouter des méthodes, avec les mêmes paramètres et rendant un booléen, dont le nom est celui de la méthode suivi de *_pre* pour la fonction précondition, *_post* pour la fonction postcondition. Ainsi pour la classe Money et sa méthode *add* utilisées dans le cours, on peut ajouter deux méthodes publiques *add_pre* et *add_post* qui représentent respectivement :

- La fonction booléenne censée être vraie avant l'appel de la méthode *add*
- La fonction booléenne censée être vraie après l'appel de la méthode *add*

La classe prend donc la forme suivante :

```
class Money {

    private int fAmount;
```

```
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }

    // ajoute m à Money (m doit être positif et de même currency que this)
    public void add(Money m) {
         fAmount=fAmount+m.amount();
    }

    // precondition de add
    public boolean add_pre(Money m) {
         return (m>=0) && (currency().equals(m.currency()));
    }

    // postcondition de add (true pour l'instant)
    public boolean add_post(Money m) {
         return true;
    }
}
```

A l'intérieur d'un test, on peut ensuite utiliser la méthode static *call* de la classe utilitaire TestPP que l'on veut créer ici :

```
  @Before public void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
}

  @Test public void testadd() {
    // Equivalent de m1.add(m2);
    TestPP.call(m1,"add",m2);
}
```

La méthode *call* a la signature suivante :

```
  void call(Object cible, String methodName, Object ... params)
```

Les paramètres sont :

- cible : un objet qui sera la cible de l'appel
- methodName, le nom de la méthode à appeler sur la cible
- params : un tableau de paramètres à passer à la méthode à appeler

Rappel : La déclaration du dernier paramètre « Object ... params » correspond à des arguments variables Java 5. Chaque appel de ***call*** *peut ainsi utiliser de 0 à un nombre quelconque de paramètres. Dans le corps de la méthode call, le* ***params*** *est accessible comme un tableau d'Object (Object[]).*

a) Donnez l'implémentation de la méthode *call* qui doit réaliser l'algorithme suivant :
   - Récupérer la méthode de la cible qui correspond au nom *methodName*, avec les paramètres *params*
   - Récupérer, si elles existent, les méthodes suffixées par *_pre* et *_post* à partir de la première méthode

- Appeler la méthode suffixée *_pre,* et en cas de retour « false », afficher un message d'erreur explicite (nom de la méthode, etc.) sur la sortie d'erreur et terminer la méthode normalement sans effectuer la suite (la précondition n'est pas respectée, le test n'a aucune valeur)
- Appeler la méthode à tester
- Appeler la méthode suffixée *_post,* et en cas de retour « false », faire échouer le test en appelant la méthode *fail(String message)* de la classe *org.junit.Assert* avec un message explicite.
- En cas d'autres erreurs, lors d'une manipulation par réflexivité par exemple, la méthode lèvera une Exception nommée *WrongTestPPException*.

b) Expliquez comment vous testeriez cette fonction, en donnant les détails sur les caractéristiques de ou des classes « remarquables » que vous créeriez pour effectuer les tests. Pensez bien aux cas limites et aux cas exceptionnels !

On souhaite maintenant étendre le système pour pouvoir sauvegarder certaines valeurs avant l'appel pour être utilisées dans les postconditions. On doit par exemple pouvoir « injecter » dans la postcondition la valeur de *amount()* avant l'appel, afin que l'on puisse écrire dans cette postcondition quelque chose qui ressemble à :

```
return amount() == (pre_amount+m.amount())
```

ce qui correspond à rendre le booléen qui teste que la valeur de *amount()* après l'appel est bien égale à la valeur avant cet appel (*pre_amount*) additionnée à *m.amount()*.

c) Expliquez les modifications à réaliser dans l'architecture et les modifications de la fonction *call* pour gérer cette fonctionnalité.

d) Expliquez comment vous testeriez cette nouvelle fonction, en reprenant et adaptant ce que vous avez défini dans la question b).