# IN413 (Design patterns) : Contrôle final

ESIEE, Jean-Michel DOUIN & Denis BUREAU, 5 novembre 2008

2 heures, AVEC UN SEUL DOCUMENT, SANS CALCULATRICE

## Question 1 (/ 4)

**1.1)** Soient les 3 classes suivantes :

```
public abstract class Transaction
{
  private Commande cmd;

  public Transaction( Commande cmd )
  {
    this.cmd= cmd;
  }

  public void execute()
  {
    try {
      begin();
      cmd.execute();
      commit();
    }
    catch ( Exception e ) {
      rollback();
    }
  }

  public abstract void begin();

  public abstract void commit();

  public abstract void rollback();
}
```

```
public class TransactionFictive
       extends Transaction
{
  public TransactionFictive( Commande cmd )
  {  super( cmd );  }

  public void begin()
  {
    System.out.println( "begin" );
  }

  public void commit()
  {
    System.out.println( "commit" );
  }

  public void rollback()
  {
    System.out.println( "rollback" );
  }
}


public interface Commande
{
  public void execute() throws Exception;
}
```

**Quel est le patron utilisé pour les classes Transaction et TransactionFictive ?**

**1.2)** Soient les 2 méthodes de test suivantes :

```
public void testQuestion1_2_a()
{
  Transaction t =
    new TransactionFictive(
      new Commande()
      {
        public void execute()
          throws Exception
          {
            System.out.println("execute...");
          }
      } );
  t.execute();
}
```

```
public void testQuestion1_2_b()
{
  Transaction t =
    new TransactionFictive(
      new Commande()
      {
        public void execute()
          throws Exception
          {
            throw new Exception();
          }
      } );
  t.execute();
}
```

**Quelle est la trace obtenue sur le terminal à la suite de l'appel de ces deux méthodes ?**

**1.3) Qu'appelle-t-on « injection de dépendance » ? Est-elle utilisée sur l'exemple précédent ?**

## Question 2 (/ 5)

Soient les 4 classes suivantes :

```
public interface ServiceI
{
  public String requête(
    String utilisateur, String url );
}
```

```
import java.util.List;

public abstract class Utilisateurs
{
  public abstract List<String> agréés();
}
```

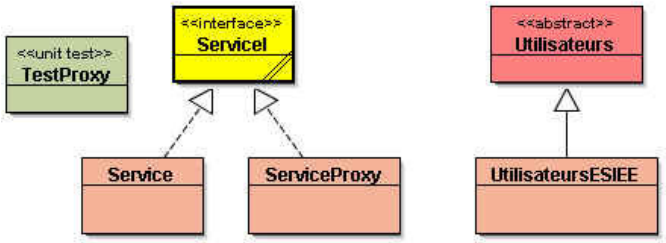
```
public class Service
       implements ServiceI
{
  public String requête(
    String utilisateur, String url )
  {
    String résultat =
      "requête effectuée pour : "
      + utilisateur + " en " + url;
    // requête effective ...
    return résultat;
  }
}
```

```
import java.util.List;
import java.util.ArrayList;

public class UtilisateursESIEE
       extends Utilisateurs
{
  private static final List<String>
    LISTE_ESIEE = new ArrayList<String>();

  static { LISTE_ESIEE.add( "java" ); }

  public List<String> agréés()
  {  return LISTE_ESIEE;  }
}
```

**On demande d'installer le patron Proxy/Procuration afin d'autoriser la requête aux seuls utilisateurs agréés.**

**2.1) Proposez la classe ServiceProxy.**

**2.2) Proposez un exemple d'utilisation de ce mandataire pour contrôler les accès.**




## Question 3 (/ 5)

Soient les 2 classes suivantes :

```
public interface PileI<E>
{
  public void empiler( E e ); // 1
  public E dépiler();         // 2
  public boolean estVide();   // 3
}
```

```
public interface StackI<E>
{
  public void push( E e );  // 1
  public E pop();           // 2
  public boolean isEmpty(); // 3
}
```

Nous ne disposons d'aucune implémentation de l'interface PileI.

Par contre, nous disposons de plusieurs implémentations de l'interface StackI.

L'utilisateur est francophone et souhaite vivement continuer d'appeler les méthodes déclarées dans l'interface PileI (qui correspondent chacune à la méthode de même numéro dans l'interface StackI).

**3.1) Quel est le patron permettant à cet utilisateur de respecter ses souhaits ?**

**3.2) Implémentez complètement la solution.**

**3.3) Proposez un scénario d'utilisation (dans une classe) pour notre client francophone.**

## Question 4 (/ 5)

Nous souhaitons décorer un texte de balises HTML.

Par exemple, ***ce texte*** correspond à cette séquence : `<b><i>ce texte</i></b>`
et doit pouvoir être obtenu avec cette instruction :

```
AbstractTexte texte = new B( new I( new Texte( "ce texte" ) ) );
```

Soient les 2 classes suivantes :

```
public abstract class AbstractTexte
{
  public abstract String enHTML();
}
```

```
public class Texte extends AbstractTexte
{
  private String texte;

  public Texte( String texte )
  {  this.texte= texte;  }

  public String enHTML()
  {  return texte;  }
}
```

**4.1) Proposez une implémentation du patron décorateur afin d'obtenir un texte décoré de balises HTML.**

**4.2) Proposez une méthode (pour essayer) affichant le code HTML pour la phrase :**
« Je peux afficher de l'*italique* et du **gras**, ou les ***deux*** ! ».

## Qualité générale de la programmation (/ 2)

- Très bonne ==> +2 (bonus !)
- Normale ==> +1
- Mauvaise ==> +0

## interface Collection<E>

boolean **add**(E e)

Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns.

**Parameters:** e - element whose presence in this collection is to be ensured
**Returns:** true if this collection changed as a result of the call
**Throws:**
UnsupportedOperationException - if the add operation is not supported by this collection
ClassCastException - if the class of the specified element prevents it from being added to this collection
NullPointerException - if the specified element is null and this collection does not permit null elements
IllegalArgumentException - if some property of the element prevents it from being added to this collection
IllegalStateException - if the element cannot be added at this time due to insertion restrictions

---

## class ArrayList<E>

public boolean **add**(E e)

Appends the specified element to the end of this list.
**Specified by:** add in interface Collection<E>
**Specified by:** add in interface List<E>
**Overrides:** add in class AbstractList<E>
**Parameters:** e - element to be appended to this list
**Returns:** true (as specified by Collection.add(E))

---

## interface Collection<E>

boolean **contains**(Object o)

Returns true if this collection contains the specified element. More formally, returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e)).
**Parameters:** o - element whose presence in this collection is to be tested
**Returns:** true if this collection contains the specified element
**Throws:**
ClassCastException - if the type of the specified element is incompatible with this collection (optional)
NullPointerException - if the specified element is null and this collection does not permit null elements (optional)

---

## class ArrayList<E>

public boolean **contains**(Object o)

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
**Specified by:** contains in interface Collection<E>
**Specified by:** contains in interface List<E>
**Overrides:** contains in class AbstractCollection<E>
**Parameters:** o - element whose presence in this list is to be tested
**Returns:** true if this list contains the specified element

---