# Examen de Java modèle 6 - Corrigé

**Session :** ING1 - Juin 2008 - session 1
**Durée :** 2h30

## Thème : les réseaux sociaux

L'actualité du Web est très prolixe en ce moment sur les thèmes des réseaux de personne. Nous utilisons ce prétexte conjoncturel pour vous faire réfléchir sur les nécessités "programmatiques" d'applications qui les utilisent.

Le but du devoir est de modéliser la façon dont une application de type Facebook peut organiser et gérer les mutiples petites applications qu'elle propose aux utilisateurs.

La programmation doit être claire.

La programmation doit faire usage des collections, et ne pas réimplémenter des algorithmes triviaux.

Vous devez mettre en oeuvre toutes vos connaissances Java pour proposer des solutions aux questions posées.

COMMENTEZ ETJUSTIFIEZ VOS INTENTIONS.

## Questions de cours POO (4 points)

Pourquoi une classe finale ET abstraite n'a aucun sens ?

Si elle est finale, elle ne peut être dérivé.

Si elle est abstraite, toutes ses méthodes ne sont pas définies et seule une dérivée de cette classe est utilisable.

Donc une classe finale abstraite ne pourra jamais servir à personne.

En quoi l'utilisation exclusive de membres publics ne tire aucun avantage de l'encapsulation ?

L'encapsulation permet de protéger l'état interne d'un objet en n'autorisant que des modifications "légales". C'est surtout vrai lorsque des membres portents des données interdépendantes. L'utilisation de membre publics permet l'utilisation de l'objet, mais laisse aux développeurs la liberté d'intervenir dans l'état de l'objet EN DEPIT DE TOUTES les règles de contrôle de conhérence de cet état.

Peut-on réaliser la même chose en programmation procédurale et en programmation objet ?

Oui, car les deux façons de programmer aboutissent quasiment au même résultat : la gestion dans le temps d'un ensemble de données sensées représenter un modèle du problème à traiter. A des détails près d'organisation des données et/ou du code du traitement, les deux façons de programmer, quand on parle de "langages complets" (à opposer à des scripts partiels ou spécifiques à un contexte) permettront de trouver une écriture qui produit le même effet. Il n'est pas certain, par contre que le nombre de lignes de code ou la performance pure de la solution soit équivalents.

Réexprimez les trois concepts de classe, d'objet et d'instance.

La classe est la description d'un "modèle d'objet". Elle exprime la façon de constituer une intance et son fonctionnement.

L'objet est principalement un concept qui réunit la classe et l'instance : la classe est la définition de l'objet, l'instance en est sa "matérialisation" dans la mémoire. Plus largement l'objet est un concept qui peut s'étendre au réel.

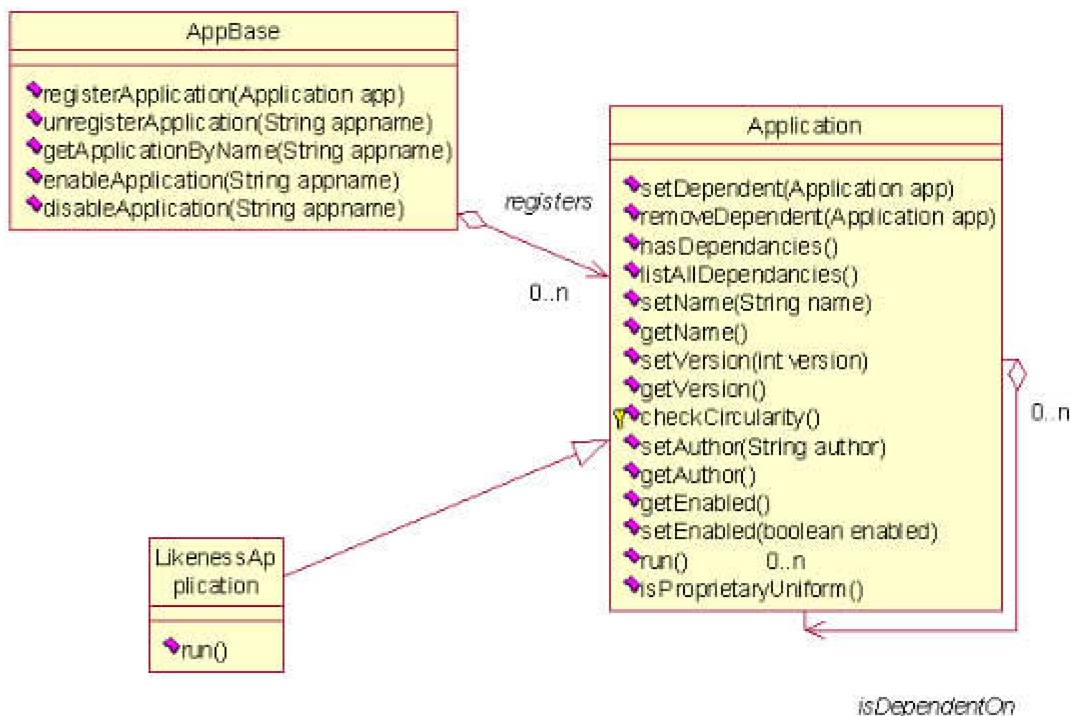
Une instance est la matérialisation d'un exemplaire d'objet dans la mémoire. L'instance est le résultat d'une "création", elle a une durée de vie et peut être détruite.

## Conception de classes (6 points)

Une application de type Facebook est une application qui utilise d'autres applications créées par les développeurs/utilisateurs/contributeurs. Il s'agit de déteminer un ensemble de classes qui permet la "gestion" de ces sous-applications.

1. Déterminez la classe Application représentant une application Facebook proposée par un utilisateur connu. Fournissez entre trois et cinq champs descriptifs. La classe Application doit être totalement mutable et comporte un numéro de version. **La classe Application est abstraite** car on ne connait pas son fonctionnement (de l'application, matérialisée par la méthode "run()"). Les applications peuvent être activées ou désactivées, installées ou simplement identifiées.
2. Déterminez une façon d'enregistrer les dépendances entre les différentes applications. Identifiez (la ou) les collections nécessaires pour que :
   - On puisse déclarer qu'une application "dépend" d'autres applications.
   - Il ne peut y avoir de boucles de dépendance : si A dépend de B et B dépend de A, directement ou indirectement, la maintenance de B et de A est indéterminable.
   - Il peut y avoir plusieurs fois la même application dans la chaîne de dépendance, mais toujours en versions décroissantes :
     B(2) => A(1) => B(1) est légitime, A(2) => A(1) aussi, C(1) => B(3) => C(4) est interdit.
3. Constituez les classes sous forme de carcasses SANS ECRIRE LE CODE DES METHODES. Votre objectif est de bien décider des membres et des méthodes utiles pour manipuler ce modèle et de savoir le proposer en JAVA.
4. La base d'applications est connue en tant que tel (classe explicite AppBase). Elle doit apparaître comme une identité explicite et implémente donc un comportement de container (Collection à organiser).
5. La base d'applications doit être manipulable, c'est-à-dire que toutes les méthodes qui permettent de la construire à partir d'une base vide doivent être présentes => réviser et compléter les signatures si nécessaire.
6. On désire pouvoir savoir si une application est uniformément propriétaire : c'est le cas si toutes les dépendances (à tous les niveaux de distance) appartiennent à la même personne. Trouvez les méthodes à définir pour que ce calcul soit possible, et expliquez la chaîne d'utilisation (le diagramme de séquence). Complétez les signatures de ces méthodes dans la/les classe(s).

Pour vous aider dans votre implémentation, on vous fournit le shéma UML suivant (pas nécessairement complet) :




On remarque que le problème énoncé dans le sujet ne demande pas nécessairement une grande quantité de classes.

Barème :

- Explications et analyse du problème
- carcasse de la classe abstraite Application

```
import java.util.ArrayList;


public abstract class Application {

 FaceBookAppContainer parent;
 protected String name;
 protected int version;
```

```
protected boolean enabled;
protected String author;
protected ArrayList<Application> deps;
protected static boolean installed;
/**
 * @param name
 * @param version
 * @param enabled
 */
public Application(String name, int version,
String author, boolean enabled) {
 this.name = name;
 this.version = version;
 this.enabled = enabled;
 this.author = author;
 deps = new ArrayList<Application>();
}

/**
 *
 * @return
 */
public FaceBookAppContainer getParent() {
 return parent;
}

/**
 *
 */
public void setParent(FaceBookAppContainer parent) {
 this.parent = parent;
}


/**
 *
 * @return
 */
public String getName() {
 return name;
}

/*
 *
 */
public void setName(String name) {
 this.name = name;
}

/**
 *
 * @return
 */
public int getVersion() {
 return version;
}

/**
 *
 * @param version
 */
public void setVersion(int version) {
 this.version = version;
}

/**
 *
 * @return
 */
public boolean isEnabled() {
 return enabled;
```

```
 }

 /**
  *
  * @param enabled
  */
 public void setEnabled(boolean enabled) {
  this.enabled = enabled;
 }

 /*
  *
  */
 public abstract boolean isInstalled();

 /**
  *
  */
 static public void install(){
  installed = true;
 }



 /**
  *
  */
 public void registerDependancy(Application app)
 throws ForbiddenDependancyException {
  if (app.name.equals(name) && version <= app.version)
 throw (new ForbiddenDependancyException());
  if (!deps.contains(app)){
   deps.add(app);
  }
 }

 /**
  * we can get an app from the FaceBookAppContainer by name
  */
 public void unregisterDependancy(Application app){
  if (deps.contains(app)){
   deps.remove(app);
  }
 }

 /**
  *
  */
 public abstract void run();


 }
```

- carcasse de la classe LikenessApplication réalisant l'abstraction "par défaut".

```
import facebookappexceptions.NotInstalledException;


public class LikenessApplication extends Application {

 /**
 * dans ce cas le membre statique convient mieux : une
 * application est installée ou pas, quelque soit le
 * nombre de ses instances.
 */
 protected static boolean installed = false;

 public LikenessApplication(int version)
 throws NotInstalledException {
  super("Likeness", version, "vf", false);
```

```
  // TODO Auto-generated constructor stub
  if (!installed){
   throw (new NotInstalledException());
  }
 }

 @Override
 /**
 * c'est la réalisation de l'abstraction qui permettra
 * d'instancier une classe
 */
 public void run() {
 }

 public static void install() {
  installed = true;
  // TODO make real install

 }

 @Override
 public boolean isInstalled() {
  return installed;
 }

}
```

- carcasse de la classe AppBase

Renommée pour l'occasion FacebookAppContainer

```
import java.util.ArrayList;
import java.util.Iterator;

import facebookappexceptions.NotInstalledException;

public class FaceBookAppContainer {

 protected ArrayList<Application> plugins;

 /**
  * Constructeur élémentaire
  */
 public FaceBookAppContainer(){
  plugins = new ArrayList<Application>();
 }

 /**
  * Enregistre une application dans le container.
 * Le container devient le "père" de l'application
  * @param app
  * @return
  */
 public boolean registerApplication(Application app){
  if (!plugins.contains(app)){
   plugins.add(app);
   app.setParent(this);
   return true;
  }
  return false;
 }

 /**
  * Désenregistre une application
  * @param app
  */
 public boolean unregisterApplication(String appname){
  Iterator<Application> it = plugins.iterator();
```

```
 while(it.hasNext()){
     Application app = it.next();
  if(app.getName().equals(appname)){
      it.remove();
      app.setParent(null);
      return true;
     }
 }
 return false;
}

/**
 * Connaissant le nom d'une application, la retrouve dans le
* container.
 * @param app
 */
public Application getApplicationByName(String appname)
throws NoSuchApplicationException {
 Iterator<Application> it = plugins.iterator();
 while(it.hasNext()){
  Application app = it.next();
     if(app.getName().equals(appname)){
      return app;
     }
 }
 throw (new NoSuchApplicationException() );
}

/**
 * @param args
 */
public static void main(String[] args) {
}

}
```

L'utilisation de génériques (collections typées) sera évidemment appréciée.

## Programmation de méthodes (5 points)

1. Ecrivez le code de la méthode citée en 6 et qui détermine si une application Facebook est uniformément propriétaire.

Dans le code de la classe Application

```
/**
 *
 */
public boolean isUniformelyPropertyOf(String author) {

 String appauthor;

 // used to initiate recursion with the first author = null
 if (author == null){
  appauthor = this.author;
 } else {
  // check myself
  System.out.println(this.author+"/" + author);
  if (!author.equals(this.author)) {
   return false;
  }
  appauthor = author;
 }
 // check all my dependancies
 for(Application child : deps){
  if (!child.isUniformelyPropertyOf(appauthor)){
   return false;
  }
 }
```

```
 return true;
}
```

1. Ecrivez le code d'une méthode qui fournit la liste de toutes les dépendances d'une application donnée.

Toujours dans le code de la classe Application

```
/**
 * very fast coding using Collection functions.
 */
public ArrayList<Application> getAllDependancies(Application app) {

 ArrayList<Application> local = (ArrayList<Application>)deps.clone();

 // add all my dependancies
 for(Application child : deps){
  local.addAll(getAllDependancies(child));
 }

 return local;
}
```

1. Ecrivez les méthodes toString() permettant d'exprimer une représentation lisible des instances.

Encore dans le code de la classe Application, c'est le principal endroit, mais on peut surcharger dans LikenessApplication car on rajouterait probablement des nouvelles propriétés.

```
public String toString(){
 String str = "";

 str += "Application[";
 str += "name : " + name + "\n";
 str += "version : " + version + "\n";
 str += "author : " + author + "\n";
 str += "enabled : " + enabled + "\n";
 str += "deps : " + deps.size() + "\n";
 str += "]";

 return str;
}
```

Barème :

- 2 point pour la première,
- 1 point pour la deuxième (réutiliser le principe de la première),
- 1 point pour les impressions.

## Programmation d'application (5 points)

1. Ecrivez les constructeurs des classes ci-dessus et

Donnés dans le code ci-dessus

1. un programme principal qui construit un jeu de test et imprime la construction.

Voici un exemple de scénario de démonstration. Les exceptions mentionnées sont à créer dans le projet sur la base d'une Exception simple.

```
/**
 * @param args
 */
public static void main(String[] args) {
 // TODO Auto-generated method stub
```

```
  LikenessApplication.install();
  LikenessApplication app1 = null;

  try {
   app1 = new LikenessApplication(1);
   LikenessApplication app2 = new LikenessApplication(2);
   app2.registerDependancy(app1);

   System.out.println(app1);
   System.out.println(app2);
   System.out.println("Test propriété:\n");
   System.out.println(app2.isUniformelyPropertyOf(null));

  } catch (NotInstalledException e) {
   // TODO Auto-generated catch block
   System.out.println("Installation !!");
   e.printStackTrace();
  } catch (ForbiddenDependancyException e) {
   // TODO Auto-generated catch block
   System.out.println("Dépendances !!");
   e.printStackTrace();
  }

 }
```

Résultat de la solution proposée :

```
Application[name : Likeness
version : 1
author : vf
enabled : false
deps : 0
]
Application[name : Likeness
version : 2
author : vf
enabled : false
deps : 1
]
Test propriété:

true
```

Barème :

- 1 point par constructeur
- 1 point pour la classe principale portant le main().