Université Abou Beckr Belkaid
Master 2 Génie Logiciel
Année Universitaire 2018/2019

Examen Final

—

Les technique de construction d'architectures logicielles avancées

—

Remarques :

- Les documents ne sont pas autorisés ainsi que les appareils électroniques (PC, Tablette, téléphone..).
- La lisibilité et la clarté de vos réponses et de votre code sont très importantes. Une réponse pas claire ne sera pas prise en compte lors de la correction.

Exercice 1 :

Nous souhaitons maitre en place un système de gestion d'adresse et de numéro de téléphone. Pour chaque adresse et numéro de téléphone il existe deux formats :

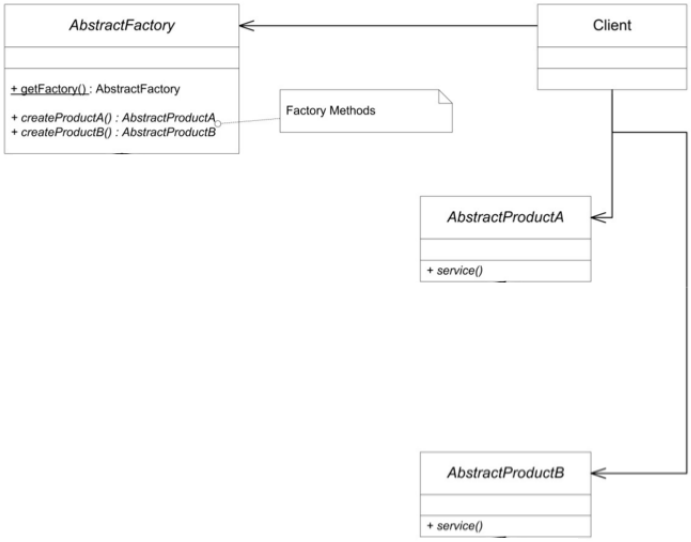
- Le format algérien :
  - Adresse : Rue, Code postal, Ville, Pays.
  - Téléphone : 213.XX.XX.XX.XX
- Le format américain :
  - Adresse : Rue, Ville, Région, Code postal, Pays.
  - Téléphone : (1).XXX.XXX.XXXX

Afin d'abstraire les différents types d'adresses et de numéros de téléphone aux clients, nous proposons d'utiliser le design pattern Abstract Factory.

Questions :

1 A quelle groupe de design pattern appartient Abstract Factory. Donner deux avantages de ce pattern.

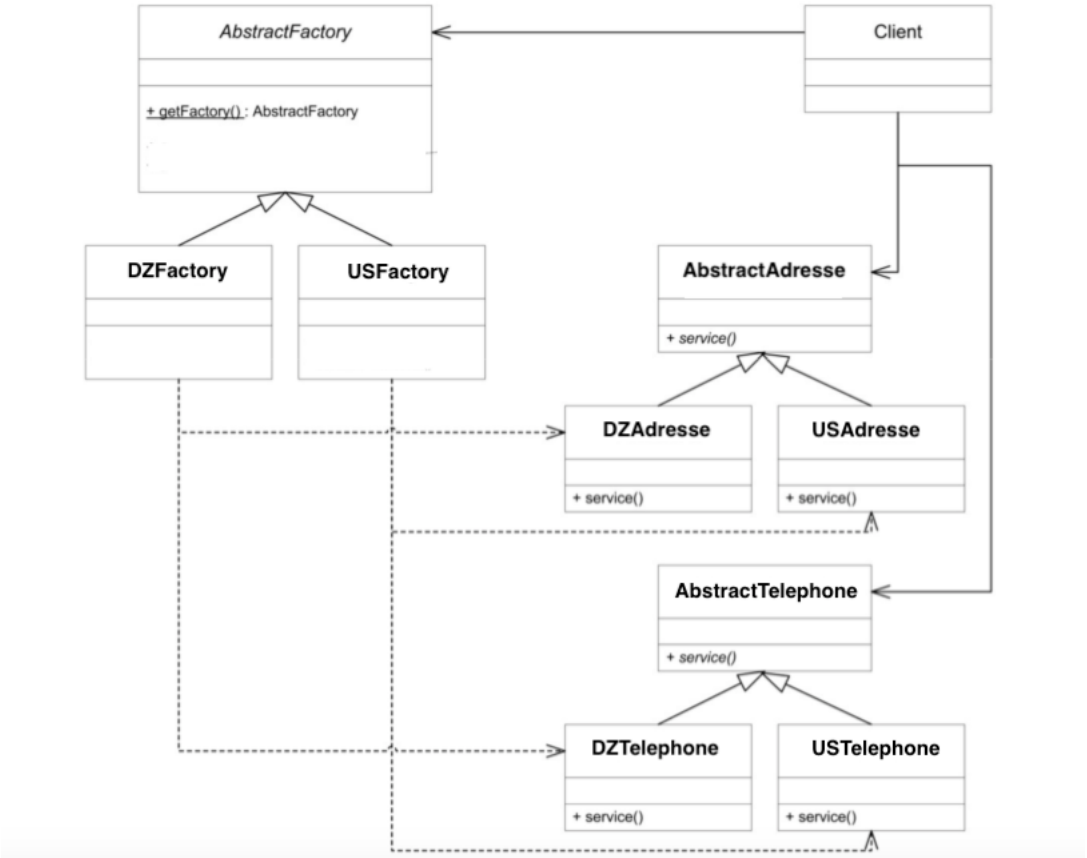
2 En utilisant Abstract Factory, proposer une conception à ce système en complétant la conception suivante et en remplaçant AbstractProductA et AbstractProductB par des noms de classes adéquates :



.

3 Proposer une implémentation de la classe *AbstractFactory* et une implémentation du main liée à la classe *Client*.

Solution :




```java
public abstract class AbstractFactory {

  // getFactory()
  public static AbstractFactory createFactory(String type) {
    AbstractFactory factory;
    if (type.equalsIgnoreCase("dz")) {
      factory = new DZFactory();
    } else {
      factory = new USFactory();
    }
    return factory;
  }

  // Factory Methods
  public abstract AbstractAdresse createAdresse();
  public abstract AbstractTelephone createTelephone();

}
public class Main {

  public static void main(String[] args) {
    String pays = "dz";

    AbstractFactory factory;
    factory = AbstractFactory.createFactory(pays);

    AbstractAdresse adresse = factory.createAdresse();
    adresse.setRue("Les cerisiers");
    adresse.setVille("Tlemcen");
    adresse.setRegion("");
    adresse.setCodePostal("13000");

    AbstractTelephone telephone = factory.createTelephone();
    telephone.setNumero("422345432");

    System.out.print(adresse.toString());
    System.out.print(telephone.toString());
  }

}
```

Exercice 2 :

Nous avons une classe Java simple nommée *Point*. Un objet de type *Point* est représenté par deux coordon-

nées *x* et *y*. Le code de la classe point est donné comme suite :

```
class Point {

  protected int x = 0;
  protected int y = 0;

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public void setRectangular(int newX, int newY) {
    setX(newX);
    setY(newY);
  }

  public void setX(int newX) {
    x = newX;
  }

  public void setY(int newY) {
    y = newY;
  }

  public void offset(int deltaX, int deltaY) {
    setRectangular(x + deltaX, y + deltaY);
  }

  public String toString() {
    return "(" + getX() + ", " + getY() + ")" ;
  }
}
```

.

Le but de l'exercice est de transformer la classe Point en une classe Java bean en utilisant la programmation orientée aspect.

Questions :

1 Donner la définition de la programmation orientée aspect.
2 Donner les caractéristiques d'une classe Java Bean.
3 Quelles sont les propriétés à ajouter à la classe Point pour la transformer en une classe Java Bean ?
4 En utilisant la programmation orientée aspect, proposer une solution à ce problème (en donnant le code associé).
3 En utilisant la programmation orienté aspect, transformer les attributs x et y en des propriétés liées (en donnant le code associé).

Solution :

1 L'AOP est un paradigme de programmation qui permet d'ajouter des fonctionnalités à un code existant sans toucher à ce dernier.
2 — La classe doit implémenté l'interface *Serializable*.
— La classe doit avoir un constructeur sans paramètre.
— La classe doit avoir des *getters* et *setters* pour des attributs *private*.
— La classe peut avoir des propriétés liées.
3 La classe doit implémenté *Serializable*.
4 Voici le code de l'aspect à ajouter :

```
aspect BoundPoint {
  private PropertyChangeSupport Point.support = new PropertyChangeSupport(this);

  public void Point.addPropertyChangeListener(PropertyChangeListener listener){
    support.addPropertyChangeListener(listener);
  }
  public void Point.addPropertyChangeListener(String propertyName,
                                              PropertyChangeListener listener){

    support.addPropertyChangeListener(propertyName, listener);
  }
  public void Point.removePropertyChangeListener(String propertyName,
                                                 PropertyChangeListener listener) {
    support.removePropertyChangeListener(propertyName, listener);
  }
  public void Point.removePropertyChangeListener(PropertyChangeListener listener) {
    support.removePropertyChangeListener(listener);
  }
  public void Point.hasListeners(String propertyName) {
    support.hasListeners(propertyName);
  }

  declare parents: Point implements Serializable;

  pointcut setter(Point p): call(void Point.set*(*)) && target(p);

  void around(Point p): setter(p) {
        String propertyName =
      thisJoinPointStaticPart.getSignature().getName().substring("set".length());
        int oldX = p.getX();
        int oldY = p.getY();
        proceed(p);
        if (propertyName.equals("X")){
      firePropertyChange(p, propertyName, oldX, p.getX());
        } else {
      firePropertyChange(p, propertyName, oldY, p.getY());
        }
  }

  void firePropertyChange(Point p,
                          String property,
                          double oldval,
                          double newval) {
        p.support.firePropertyChange(property,
                                     new Double(oldval),
                                     new Double(newval));
  }
```

.

Bon courage et bonne continuation.