Développement avancé de logiciels: patrons et modèle - TP 1

Configuration: 1 module par question

Question 1

Nous avons créé un fichier `BeanFactory.xml` où nous déclarons nos beans/

Premièrement, nous déclarons nos beans de quizz :

```
<bean id="umlQuizz" class="UMLQuizz"/>
<bean id="cmmQuizz" class="CMMQuizz"/>
```

Nous déclarons ensuite un bean qui permettra d'injecter les dépendances via le constructeur.

```
<bean id="quizzConstructor" class="QuizzMasterService" laz
y-init="true">
    <constructor-arg ref="cmmQuizz"/>   <!--Select the cmm
Quizz or umlQuizz-->
</bean>
```

```
<bean id="quizzSetter" class="QuizzMasterService" lazy-ini
```

```
t="true">
    <property name="quizMaster" ref="umlQuizz"/> <!--Selec
t the cmmQuizz or umlQuizz-->
</bean>
```

Il suffit de remplacer

```
ref=cmmQuizz
```

par

```
ref=umlQuizz
```

et inversement .

Nous récupérons le contexte dans le fichier `QuizProgram.java`

```
ApplicationContext context = new ClassPathXmlApplicationCo
ntext("BeanFactory.xml");
```

Nous récupérons ensuite le bean voulu. Pour changer la méthode d'injection, remplacer quizzConstrucor par quizzSetter.

```
QuizzMasterService quizzMasterService = (QuizzMasterServic
e) context.getBean("quizzConstructor");
```

Question 2

`Notifier.aj`

Cet objet nous permet de notifier des changements. C'est notre observable

```
private PropertyChangeSupport support = new PropertyChange
Support(this);
```

Nous ajoutons un pointcut lorsque la banque ajoute un nouveau client. Après l'ajout, on ajoute le client en tant qu'observer.

```
pointcut clientAdd(Customer customer, Bank bank) :
  call(void Bank.addCustomer(Customer)) && args(customer)
&& target(bank);


after(Customer customer, Bank bank) : clientAdd(customer,
bank) {
  support.addPropertyChangeListener(customer);
}
```

Voici notre pointcut pour notifier les clients que les intérêts changent. Dès que la valeur changera, les observers seront notifiés.

```
  pointcut interetsChangent(double value, Bank bank) :
```

```
    call(void Bank.setTauxInteret(double)) && args(value)
&& target(bank);
  before(double value, Bank bank) : interetsChangent(value
, bank) {
  support.firePropertyChange("tauxInteret", bank.getTauxIn
teret(), value);
}
```

Ce sera alors cette méthode dans `Customer.java` qui sera appelé

```
@Override
public void propertyChange(PropertyChangeEvent propertyCha
ngeEvent) {
  this.tauxInteret = (double) propertyChangeEvent.getNewVa
lue();
}
```

Question 3

Tout d'abord, un code doit être maintenable. Avec l'apparition des différents pattern comme le pattern MVC, les entreprises ont compris comment séparer les différents éléments afin de pouvoir répartir les différentes responsabilités de manière claire.
Premièrement le code se doit d'être maintenable et facile à corriger. C'est pourquoi l'on opère des refontes ou "refactoring". Plus l'architecture et le code est simple, plus l'erreur est facile à retrouver.
Ensuite c'est une question de sécurité. En effet une application utilisant

une technologie obsolète ou faillible s’expose à des risques non négligeables.

L’évolution constante des différentes technologies du projet évoluent et les façons de faire aussi. Si un nouveau framework permet de faire ce que notre logiciel fait et de manière plus efficace et rapide, il apparaît clair que nous devons l’injecter à notre solution.