Guide du développeur

LOG8430:TP5

**Mohammed Benbachir**

**Clément Duffau**

**Guillaume Rivest**

**07 Avril 2014**

Contenu

[Présentation du guide du développeur 3](#_Toc384412059)

[Utilisation de bibliothèques spécifique 3](#_Toc384412060)

[Explication de l’architecture et Choix de codage 3](#_Toc384412061)

[Questions de l’énoncé 3](#_Toc384412062)

[R1) 3](#_Toc384412063)

[R2) 4](#_Toc384412064)

[R3) 4](#_Toc384412065)

[R4) 5](#_Toc384412066)

[Diagramme de classe pour la partie O.O. 6](#_Toc384412067)

# Présentation du guide du développeur

Le présent document consiste en un guide à l’attention des développeurs. Ce guide explique les différents choix architecturaux ainsi que des réponses aux questions de l’énoncé.

# Utilisation de bibliothèques spécifique

Nous avons utilisé « AspectJ », qui est un « plug-in » pour Eclipse permettant de faire de la programmation orientée aspect sous la plateforme standard de Java.

# Explication de l’architecture et Choix de codage

Afin de bien répondre aux exigences demandées, nous avons créé un nouveau paquet afin d’y mettre notre aspect. Ce dernier, qui observe l’exécution de l’application, nous permet d’insérer différent point d’action (« pointcut »).

# Questions de l’énoncé

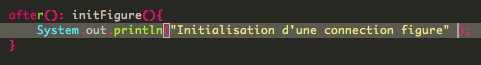
## R1)

Nous avons donc implémenté un nouvel aspect nommé : « AspectProfiler ». Cet aspect contient un seul point d’action :

Screen Shot 2014-03-27 at 12.11.26 PM.png

Ce dernier sera donc lancé à chaque initialisation d’une « ConnectionFigure » ou à l’initialisation d’une de ces dérivées, et ce peu importe les arguments du constructeur.

Ensuite notre aspect contient :



Le « after » est un greffon qui spécifie du code à exécuter après le point d’action défini précédemment a été lancé.

Lors de l’exécution, nous avons vu 3 appels au constructeur de « ConnectionFigure ». Nous avons également tenté de créer de nouvelle « ConnectionFigure » dans l’interface (par exemple le « ElbowConnection »), mais nous n’avons remarqué aucun appel de plus. Ceci nous mène à conclure que, dans le cas de ces figures, le patron « Abtract Factory » est utilisé et, lors de son utilisation, seulement une copie est retournée.

## R2)

Nous avons modifié notre greffon de la façon suivante :

Screen Shot 2014-03-27 at 12.27.02 PM.png

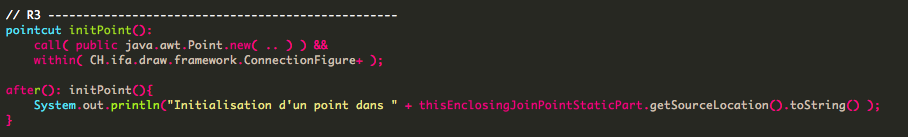
Nous allons donc chercher le nom de la classe qui est « visé » par notre point d’action. Ici, nous pouvons expliquer pourquoi nous avions 3 appels. Les 3 appels consistent en :

1. « LineConnection »
2. « ElbowConnection »
3. « ElbowConnection »

« ElbowConnection » est appelé 2 fois, car il est le « target » deux fois dans son utilisation. Une fois dans son propre constructeur, et une fois dans le constructeur de son parent : « LineConnection ».

## R3)

Ici, nous avons ajouté un nouveau point d’action ainsi qu’un greffon qui sera appelé après l’exécution du nouveau point d’action.



Nous allons donc chercher chaque initialisation de la classe « java.awt.Point » dans un contexte de classe « ConnectionFigure » ainsi que ces descendants. Une fois que ce point d’action est exécuté nous affichons la classe dans laquelle est exécuté ce point d’action.

## R4)

L'approche OOP :

Deux objets principaux « Observer » responsable d'observer, écouter des évènements, changements et « Observable », « Subject » responsable de faire un travail quelque conque et notifier les Observer abonnés lors d'un changement, évènement. En Java, celui se fait avec deux objets : l'interface Observer et la classe Observable. Il est donc nécessaire de redéfinir à pour chaque classe implémentant Observer les méthodes de celui-ci (méthode update). Pour Observable un simple héritage suffit et l'appel à « notifyObservers » est nécessaire lorsque que l'on veut informer d'un changement, évènement.

D'un point de vue concept de Observer-Observable en lui-même il s'agit donc qu'un Observable a pour abonner un ensemble d'Observer et à chaque appel à « notifyObservers », l'appel aux méthodes update des Observer est effectué.

Il est donc assez facile de mettre en place Observer-Observable en OO. L'un des principaux inconvénients, et pour la maintenabilité car pour comprendre ce que fait l'application le code d'interaction est contenu dans 2 classes différentes. L'une s'occupe de notifier les Observer et l'autre voire les autres s'occupent du traitement de cette notification. On a donc un couplage élevé et une cohésion faible. La compréhension quant à elle est plutôt aisée dans la mesure où l'on connait le patron Observer-Observable.

L'autre inconvénient est qu'au final la notion d’Observer-Observable peut être répétée au cours du programme pour chaque sujet observable. De plus, certains Observer peuvent être abonné à plusieurs Observable ce qui peut amener à du spaghetti code.

Pas vraiment réutilisable, il faut démarrer « from scratch » à chaque fois.

Côté performance, ce système nécessite une boucle for pour faire appel aux méthodes update des « Obsever » quand « notifyObserver » est appelé donc la complexité est faible (O(n)\*max(complexités update) ).

# Diagramme de classe pour la partie O.O.

Référence : R4.

