UNIVERSITÉ DE ROUEN
FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

Master 2 GIL
2017-2018
POA

Mini-projet (évalué)

— **Modalités de réalisation** : contrairement aux séances de TP, ce travail est à faire en trinôme librement formé (il y a 11 trinômes).
— **Date de remise** : une archive `projet_X_Y_Z.zip` doit être déposée sur la page UniversiTICE du cours de POA avant le lundi 8 janvier 2018 minuit (`X`, `Y` et `Z` représentent les noms des étudiants).
— **Évaluation** : attribution d'une note évaluant le code rendu.

1 Préliminaires

Le but de ce projet est de travailler sur l'application TELECOM fournie pour en retirer toutes les préoccupations transversales et les réimplanter à l'aide d'aspects.

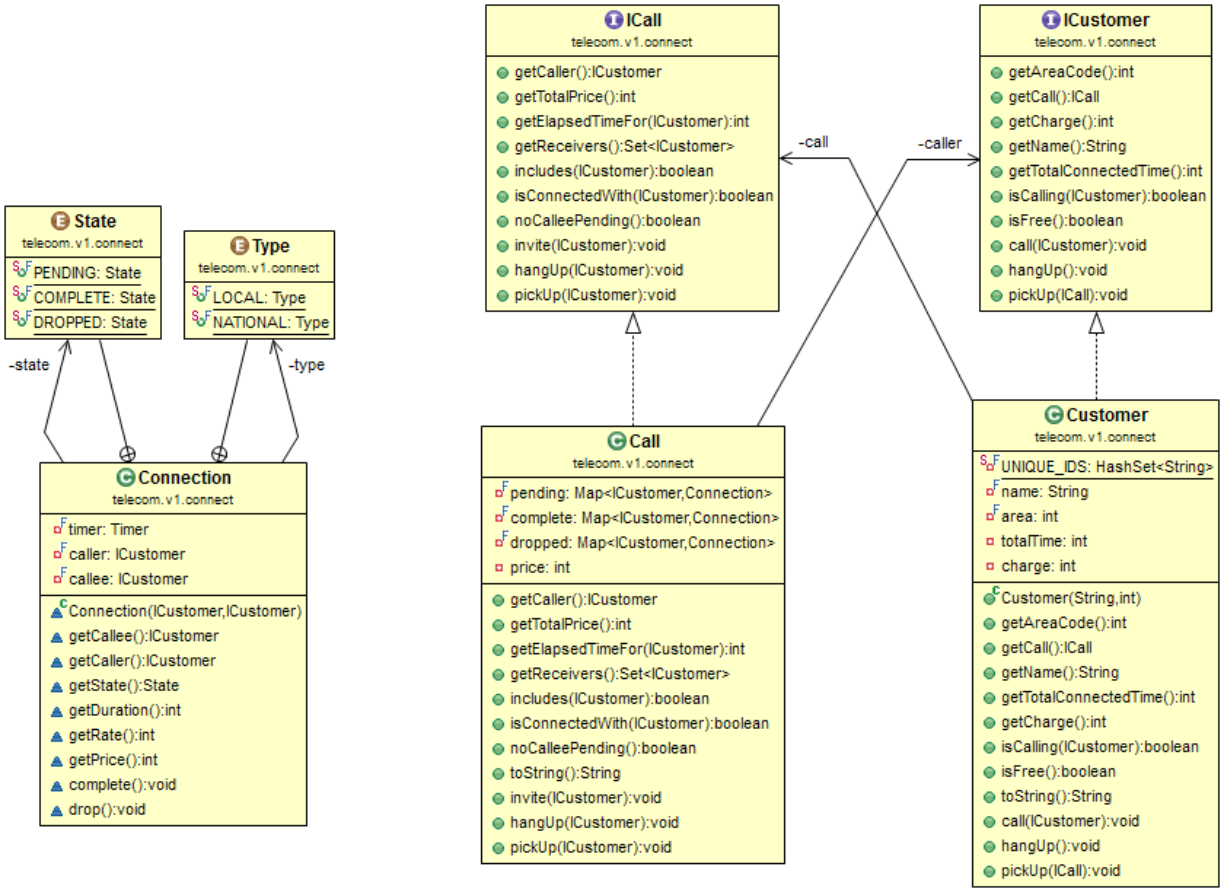
L'application est fournie dans sa version 1 complète, avec toutes les préoccupations transversales codées à l'intérieur des classes métiers. Vous devez fournir une seconde version de cette application. Pour vous aider dans votre tâche et en faciliter la correction, une version épurée des préoccupations transversales est fournie, en tant qu'ébauche de la version 2, et la structure des dossiers qui contiendront le code de cette nouvelle version vous est imposée.

Le paquetage `telecom.v1` contient une application permettant de simuler des communications téléphoniques entre les clients d'une entreprise de télécommunication :
— le sous-paquetage `telecom.v1.connect` contient les classes métiers de l'application ;
— le sous-paquetage `telecom.v1.util` contient deux classes outils (classes `Timer` et `Contract`) ainsi qu'un aspect (`Scattering`) permettant de visualiser la dispersion du code des préoccupations transversales dans le code métier de l'application ;
— le sous-paquetage `telecom.v1.simulate` contient la simulation ;
— la classe racine est `telecom.v1.Telecom`.

Les différents types du paquetage `connect` définissent les notions de client (`ICustomer`) et d'appel téléphonique (`ICall`) entre deux clients ou plus (mode conférence) ainsi que le concept local à ce paquetage de connexion (`Connect`) utilisé par la classe `Call`. À l'aide de la classe `Timer`, on peut mesurer le temps de connexion écoulé lors des appels passés entre les différents clients. Le montant des communications est, dans cette version, calculé à l'intérieur de chaque type concerné.

Quant à la simulation, elle lance un jeu de trois tests représentant chacun un scénario de communication entre plusieurs clients et affichant un certain nombre d'informations au fil des tests, puis un récapitulatif du scénario.

Les relations entre les différentes classes et interfaces du sous-paquetage `connect` de la version 1 sont portées sur le diagramme suivant :



Je vous recommande d'installer l'application TELECOM sous Eclipse en important le contenu du fichier `telecom-rsrc.zip` dans un projet tout neuf, puis d'exécuter la permière version. Observez ensuite la dispersion du code des préoccupations transversales au sein de cette version...

Les préoccupations que j'ai identifiées pour vous sont au nombre de quatre :

— `unicity` : Les identificateurs des clients doivent être uniques.

— `time` : Les temps de connexion doivent être comptabilisés pour, par la suite, être affichés.

— `billing` : On veut pouvoir gérer le tarif des communications (locale ou longue distance) et ainsi pouvoir facturer les clients en fonction de leur temps de connexion ; le montant de chaque appel doit donc lui aussi être comptabilisé.

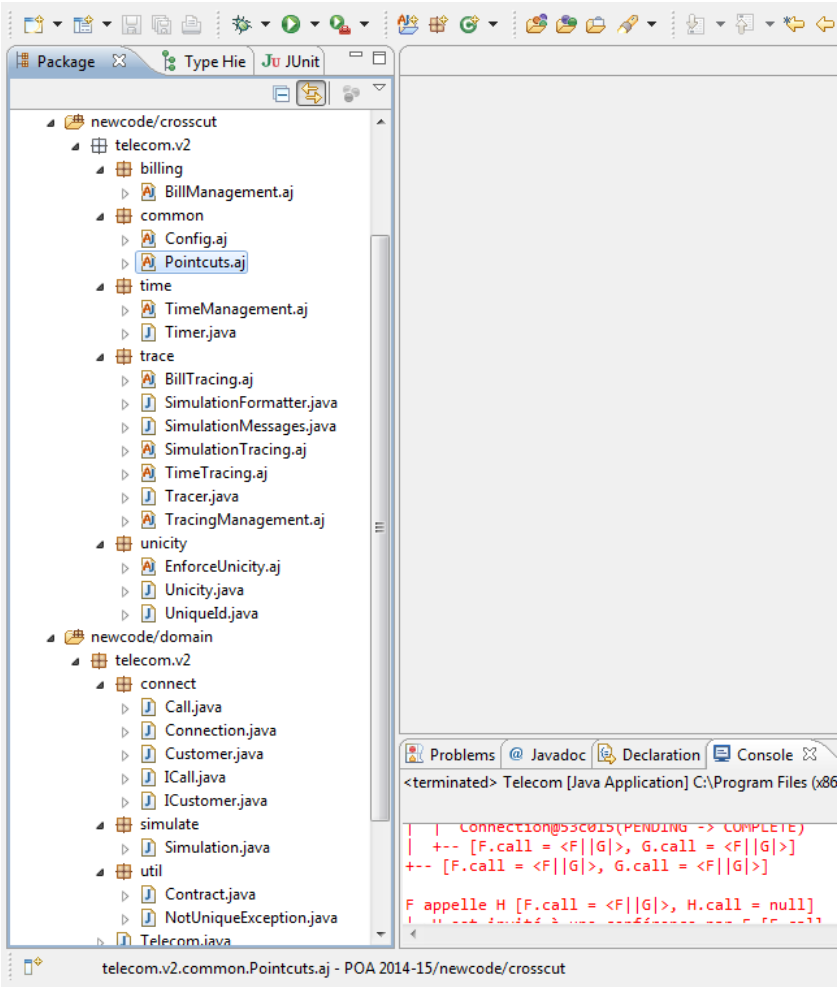
— `trace` : Un certain nombre d'informations doivent être tracées lors de la simulation, comme l'appel aux commandes de `ICustomer` et de `ICall`, les changements d'état

des connexions, les rapports finaux d'exécution, etc. (voir l'exécution de l'application donnée en annexe).

Pour chaque préoccupation transversale, je souhaite que vous procédiez ainsi :

— codez dans une ou plusieurs classes le code métier de la préoccupation transversale ;
— codez dans un ou plusieurs aspects le code qui permettra de tisser la préoccupation dans le code métier de l'application ;
— placez toutes les classes et tous les aspects se rapportant à une même préoccupation dans un paquetage spécifique (`unicity`, `trace`, `time` et `billing`).

De plus, tout ce qui concerne la configuration des aspects (notamment les déclarations de précédence) devra être codé dans un aspect `Config` dédié, et la plupart des coupes que vous utiliserez dans vos aspects seront définies et nommées dans un aspect `Pointcuts` (j'insiste une fois de plus sur l'importance de donner à vos coupes des noms bien adaptés à ce qu'elles sélectionnent !). Voici, typiquement, le genre de découpage du code que j'attends :



Les classes métiers sont dans le répertoire de sources `newcode/domain` et les préoccupations transversales dans `newcode/crosscut`. Dans le second répertoire, on trouve un sous-paquetage par préoccupation transversale.

L'annexe contient la trace d'une exécution de `telecom.v2.Telecom`. Elle donne ensuite le diagramme de classes du paquetage `connect` de la version 2, vous pourrez ainsi comparer les deux versions des classes métiers entre elles.

Une dernière recommandation : les trois préoccupations concernant l'unicité et les calculs du temps de connexion et du montant d'une connexion sont relativement simples à coder ; elles ne doivent en aucun cas comporter d'action de traçage. La quatrième préoccupation (le traçage) est, de loin, la plus difficile à mettre en œuvre. Tenez-en compte lors de la répartition du travail au sein du trinôme ; par exemple vous pourriez prendre individuellement la responsabilité de l'une des trois premières préoccupations, puis travailler ensemble sur la partie traçage...

2 Gestion de l'unicité des noms de clients : `unicity`

La politique d'unicité des noms de clients s'insère dans le contexte plus général d'une politique d'identificateurs uniques. Un identificateur est un attribut privé et constant (`final`) de type `String`. Deux identificateurs distincts ne peuvent pas prendre la même valeur, quelle que soit la classe dans laquelle ils se trouvent définis.

Le paquetage `unicity` doit contenir les éléments permettant de mettre en place une politique d'identificateur unique, que vous utiliserez sur le champ `name` de la classe `Customer`. Le principe d'utilisation est le suivant :

1. Une annotation `@UniqueId` est définie ; elle ne peut marquer que des attributs.
2. Si cette annotation est utilisée sur un attribut non `final` ou qui n'est pas de type `String`, une erreur de compilation est reportée.
3. Pour un attribut (constant et de type chaîne) marqué par `@UniqueId`, un test d'unicité est systématiquement exécuté lors de l'initialisation de cet attribut. Si la valeur donnée à l'attribut n'est pas unique, une `NotUniqueException` (donnée dans le paquetage `telecom.v2.util`) est levée.

3 Gestion des temps de connexion (`time`) et du prix des appels (`billing`)

Il n'y a pas grand chose à dire concernant ces préoccupations.

Vous pouvez récupérer la classe `Timer` de l'application TELECOM à condition d'en supprimer les commandes d'affichage... Elle détient en fait tout ce qui concerne le calcul des durées de connexion.

Vous pourriez ajouter un aspect qui injecte les attributs et les accesseurs que vous jugerez nécessaires dans les classes `Connection` et `Customer` en rapport avec la gestion des temps de connexion. Cet aspect définirait alors les actions à tisser en rapport avec cette problématique quand les connexions évoluent.

Vous pourriez faire sensiblement la même chose pour le calcul du prix des appels...

4 Gestion du traçage : `trace`

Vous utiliserez l'API *logging* de Java.

Notez bien que le traçage des appels des commandes de `ICall` et de `ICustomer` doit être indenté en fonction de la profondeur de ces appels. Le code de construction des messages à utiliser lors des appels de ces commandes est disponible à travers le type énuméré `SimulationMessages` que j'ai placé dans `telecom.v2.trace`. Ce type définit huit constantes qui calculent les messages nécessaires dans les situations suivantes :

- `CUSTOMER_CALL` : le message à afficher *avant* un appel à `call` sur une instance de `ICustomer`.
- `CUSTOMER_HANGUP` : le message à afficher *avant* un appel à `hangUp` sur une instance de `ICustomer`.
- `CUSTOMER_PICKUP` : le message à afficher *avant* un appel à `pickUp` sur une instance de `ICustomer`.
- `CUSTOMER_FINAL` : le message à afficher *après* un appel à l'une des commandes précédentes sur une instance de `ICustomer`.
- `CALL_INVITE` : le message à afficher *avant* un appel à `invite` sur une instance de `ICall`.
- `CALL_HANGUP` : le message à afficher *avant* un appel à `hangUp` sur une instance de `ICall`.
- `CALL_PICKUP` : le message à afficher *avant* un appel à `pickUp` sur une instance de `ICall`.
- `CALL_FINAL` : le message à afficher *après* un appel à l'une des commandes précédentes sur une instance de `ICall`.

La méthode statique `get` définie au sein de ce type permet, à partir d'une classe (qui sera `Customer` ou `Call`) et d'un nom de méthode (`call`, ou `hangUp`, etc.) ou du nom `"final"`, de récupérer la constante nécessaire pour le traçage. Voici un exemple basique d'utilisation de ce type énuméré :

```
pointcut test() : call(void IC*.hangUp(..));
before(Object x) : test() && target(x) {
    JoinPoint jp = thisJoinPoint;
    String methName = jp.getSignature().getName();
    SimulationMessages sm = SimulationMessages.get(x.getClass(), methName);
    System.out.println(sm.format(jp));
}
after(Object x) : test() && target(x) {
    JoinPoint jp = thisJoinPoint;
    SimulationMessages sm = SimulationMessages.get(x.getClass(), "final");
    System.out.println(sm.format(jp));
}
```

qui produira à l'exécution (ici sans log et sans indentation) :

```
A raccroche [A.call = <A||B C|>]
```

```
B raccroche d'avec A [A.call = <A||B C|>, B.call = <A||B C|>]
B se déconnecte [A.call = <A||B C|>, B.call = <A||B C|>]
+-- [A.call = <A||C|B>, B.call = <A||C|B>]
+-- [B.call = null]
C raccroche d'avec A [A.call = <A||C|B>, C.call = <A||C|B>]
C se déconnecte [A.call = <A||C|B>, C.call = <A||C|B>]
+-- [A.call = <A|||B C>, C.call = <A|||B C>]
A raccroche [A.call = <A|||B C>]
A se déconnecte [A.call = <A|||B C>]
+-- [A.call = <A|||B C>]
+-- [A.call = null]
+-- [C.call = null]
+-- [A.call = null]
-----------------------------------------------
E raccroche d'avec D [D.call = <D||E|>, E.call = <D||E|>]
E se déconnecte [D.call = <D||E|>, E.call = <D||E|>]
+-- [D.call = <D|||E>, E.call = <D|||E>]
D raccroche [D.call = <D|||E>]
D se déconnecte [D.call = <D|||E>]
+-- [D.call = <D|||E>]
+-- [D.call = null]
+-- [E.call = null]
-----------------------------------------------
G raccroche d'avec F [F.call = <F|C|G|>, G.call = <F|C|G|>]
G se déconnecte [F.call = <F|C|G|>, G.call = <F|C|G|>]
+-- [F.call = <F|C||G>, G.call = <F|C||G>]
+-- [G.call = null]
```

Vous noterez que l'affichage des appels est redéfini de sorte que, pour un appel dont l'appelant est `A`, ayant appelé `B`, `C` et `D`, `B` n'ayant pas encore répondu, `C` étant actuellement en communication avec `A`, et `D` ayant déjà raccroché, l'affichage voulu est `<A|B|C|D>`.

Enfin, vous n'oublierez pas, à chaque fin de test, de générer un rapport indiquant les temps et montants de communication pour chaque protagoniste du test (voir exécution en annexe).

Annexes

Voici l'exécution une fois le code métier de la version 2 correctement aspectisé :

```
A appelle B [A.call = null, B.call = null]
|  Connection@12fa9ae(null -> PENDING)
+-- [A.call = <A|B||>, B.call = null]

B décroche à l'appel de A [A.call = <A|B||>, B.call = null]
|  B se connecte à A [A.call = <A|B||>, B.call = <A|B||>]
|  |  Connection@12fa9ae(PENDING -> COMPLETE)
|  +-- [A.call = <A||B|>, B.call = <A||B|>]
+-- [A.call = <A||B|>, B.call = <A||B|>]

A appelle C [A.call = <A||B|>, C.call = null]
|  C est invité à une conférence par A [A.call = <A||B|>, C.call = null]
|  |  Connection@19af724(null -> PENDING)
|  +-- [A.call = <A|C|B|>, C.call = null]
+-- [A.call = <A|C|B|>, C.call = null]

C décroche à l'appel de A [A.call = <A|C|B|>, C.call = null]
|  C se connecte à A [A.call = <A|C|B|>, C.call = <A|C|B|>]
|  |  Connection@19af724(PENDING -> COMPLETE)
|  +-- [A.call = <A||B C|>, C.call = <A||B C|>]
+-- [A.call = <A||B C|>, C.call = <A||B C|>]

A raccroche [A.call = <A||B C|>]
|  B raccroche d'avec A [A.call = <A||B C|>, B.call = <A||B C|>]
|  |  B se déconnecte [A.call = <A||B C|>, B.call = <A||B C|>]
|  |  |  Connection@12fa9ae(COMPLETE -> DROPPED)
|  |  |  temps de connexion : 6 s
|  |  |  montant de la connexion locale : 18
|  |  +-- [A.call = <A||C|B>, B.call = <A||C|B>]
|  +-- [B.call = null]
|  C raccroche d'avec A [A.call = <A||C|B>, C.call = <A||C|B>]
|  |  C se déconnecte [A.call = <A||C|B>, C.call = <A||C|B>]
|  |  |  Connection@19af724(COMPLETE -> DROPPED)
|  |  |  temps de connexion : 3 s
|  |  |  montant de la connexion longue distance : 30
|  |  +-- [A.call = <A|||B C>, C.call = <A|||B C>]
|  |  A raccroche [A.call = <A|||B C>]
|  |  |  A se déconnecte [A.call = <A|||B C>]
|  |  |  +-- [A.call = <A|||B C>]
|  |  +-- [A.call = null]
|  +-- [C.call = null]
+-- [A.call = null]

B[76000] a été connecté 6 s pour un montant de 0
C[34000] a été connecté 3 s pour un montant de 0
A[76000] a été connecté 9 s pour un montant de 48
-----------------------------------------------
D appelle E [D.call = null, E.call = null]
```

```
|  Connection@cff10d(null -> PENDING)
+-- [D.call = <D|E||>, E.call = null]

E décroche à l'appel de D [D.call = <D|E||>, E.call = null]
|  E se connecte à D [D.call = <D|E||>, E.call = <D|E||>]
|  |  Connection@cff10d(PENDING -> COMPLETE)
|  +-- [D.call = <D||E|>, E.call = <D||E|>]
+-- [D.call = <D||E|>, E.call = <D||E|>]

E raccroche d'avec D [D.call = <D||E|>, E.call = <D||E|>]
|  E se déconnecte [D.call = <D||E|>, E.call = <D||E|>]
|  |  Connection@cff10d(COMPLETE -> DROPPED)
|  |  temps de connexion : 3 s
|  |  montant de la connexion locale : 9
|  +-- [D.call = <D|||E>, E.call = <D|||E>]
|  D raccroche [D.call = <D|||E>]
|  |  D se déconnecte [D.call = <D|||E>]
|  |  +-- [D.call = <D|||E>]
|  +-- [D.call = null]
+-- [E.call = null]

E[76000] a été connecté 3 s pour un montant de 0
D[76000] a été connecté 3 s pour un montant de 9
-----------------------------------------------
F appelle G [F.call = null, G.call = null]
|  Connection@1a59e87(null -> PENDING)
+-- [F.call = <F|G||>, G.call = null]

G décroche à l'appel de F [F.call = <F|G||>, G.call = null]
|  G se connecte à F [F.call = <F|G||>, G.call = <F|G||>]
|  |  Connection@1a59e87(PENDING -> COMPLETE)
|  +-- [F.call = <F||G|>, G.call = <F||G|>]
+-- [F.call = <F||G|>, G.call = <F||G|>]

F appelle H [F.call = <F||G|>, H.call = null]
|  H est invité à une conférence par F [F.call = <F||G|>, H.call = null]
|  |  Connection@4ab70a(null -> PENDING)
|  +-- [F.call = <F|H|G|>, H.call = null]
+-- [F.call = <F|H|G|>, H.call = null]

G raccroche d'avec F [F.call = <F|H|G|>, G.call = <F|H|G|>]
|  G se déconnecte [F.call = <F|H|G|>, G.call = <F|H|G|>]
|  |  Connection@1a59e87(COMPLETE -> DROPPED)
|  |  temps de connexion : 4 s
|  |  montant de la connexion locale : 12
|  +-- [F.call = <F|H||G>, G.call = <F|H||G>]
+-- [G.call = null]

G[76000] a été connecté 4 s pour un montant de 0
F[76000] est en attente de H et son montant sera supérieur à 12
-----------------------------------------------
```

Et voici les relations entre les différentes classes métier de la version 2 :