

UE Conception Orientée Objet

TP Téléphonie

Des fichiers utiles sont disponibles sur le portail.

Le but du problème est de simuler à haut niveau différentes fonctionnalités d’opérateurs de téléphonie et les modes de paiement d’utilisation de leur service.

Un *opérateur de téléphonie* doit permettre à un utilisateur d’ouvrir une *connexion* en fournissant un *mode de paiement* qui sera débité d’une certaine quantité en fonction de la politique tarifaire de l’opérateur. On décide de décrire ces trois entités par les interfaces et classes suivantes qui appartiennent au paquetage *telephonie*.

telephonie::Connexion
- mode : ModeDePaiement - operateur : Operateur - dateDebut : Date - dateFin : Date
+ Connexion(Operateur op, Date debut, ModeDePaiement m) + Operateur getOperateur() + void finConnexion(Date fin) + int heureDebutConnexion() + Date dateDebutConnexion() + Date dateFinConnexion() + int dureeConnexion() + ModeDePaiement mode()

```
public interface Operateur {
    /** crée une Connexion pour cet opérateur à la date indiqué, le mode
     * de paiement précisé sera utilisé
     * @param debut la date de début de la connexion
     * @param modeP le mode paiement qui sera utilisé pour cette connexion
     * @exception OperateurSatureException si le nombre limite de connexions simultanées
     * pour cet opérateur est dépassé
     * @exception ModeDePaiementInvalideException si le mode de paiement associé
     * n'est pas valide <code>modeP.valide()</code> vaut <code>false</code>
     */
    public Connexion seConnecter(Date debut, ModeDePaiement modeP)
        throws OperateurSatureException, ModeDePaiementInvalideException ;
    /** déconnecte la connexion <code>cnx</code> à la date indiquée, le mode de paiement
     * indiqué à la création est débité
     * @param cnx la connexion à deconnecter
     * @param fin la date de fin de cette connexion
     * @exception PasDeConnexionException si cet opérateur n'est pas l'opérateur de
     * la connexion <code>cnx</code>
     */
    public void seDeconnecter(Connexion cnx, Date fin) throws PasDeConnexionException ;
    /** fournit la durée comptabilisée pour la connexion considérée en
     * fonction de la politique tarifaire de l'opérateur
     * @param cnx la connexion dont on veut connaitre la duree
     * @return nombre entier de minutes comptabilisée pour la connexion,
     */
    public int getDureeComptabilisee(Connexion cnx);
    /** fournit le tarif unitaire appliquée par l'opérateur pour chaque minute comptabilisée
     * @param cnx la connexion concernée
     * @return le tarif unitaire d'une minute comptabilisée pour la connexion fournie
     */
    public int getTarifUnitaire(Connexion cnx);
}
```

<< interface >> telephonie::ModeDePaiement
+ valide(): boolean + debiter(dureeComptabilisee : int, tarifUnitaireApplique : int)

Dans ce paquetage sont également définies les trois classes d’exception nécessaires : *ModeDePaiementInvalideException*, *PasDeConnexionException* et *OperateurSatureException*.

Informations complémentaires.

- Un opérateur ne peut accepter qu’un nombre limité de connexions simultanées. Ce nombre est fourni à sa construction. Si ce nombre est atteint, une exception *OperateurSatureException* est levée lors d’une demande de création de connexion. Il est donc nécessaire de mémoriser à la fois le nombre de connexions maximum et le nombre de connexions actives.
- La méthode *seConnecter* d’un opérateur permet de débiter une conversation téléphonique, on obtient ainsi l’objet *Connexion* correspondant. Pour se connecter il faut préciser le mode de paiement qui sera débité lors de la déconnexion (appel à la méthode *seDeconnecter*).
- Un opérateur refuse d’établir une connexion si le mode de paiement associé n’est pas valide en levant une exception *ModeDePaiementInvalideException*.

La fin de communication se manifeste par une déconnexion auprès de l’opérateur concerné (méthode *seDeconnecter*). Lors d’une déconnexion, l’opérateur vérifie qu’il s’agit bien d’une connexion qui le concerne (càd qu’il a créée), sinon une exception *PasDeConnexionException* est levée.

Si c’était bien le cas, la fin de connexion est notifiée à la connexion (méthode *finConnexion*) et le mode de paiement associé à la connexion est débité en fonction de la durée de la connexion comptabilisée et du tarif unitaire appliqué.

La durée comptabilisée est exprimée en **nombre entier de minutes**¹ et le tarif unitaire (coût d’une minute de communication) est exprimé en nombre entier de centimes d’euros.

Ces informations sont fournies par les paramètres *dureeComptabilisee* et *tarifUnitaireApplique* de la méthode *debiter* de l’interface *ModeDePaiement*.

Le débit est toujours effectué (voir comportement ci-dessous).

- On distingue deux types d’opérateurs en fonction de leur politique tarifaire qui correspond à un calcul différent de la durée comptabilisée et du tarif unitaire appliqué à chaque minute comptabilisée :

– les opérateurs à tarif fixe :

Durée comptabilisée Il s’agit de la durée réelle de communication si la communication dure moins de cinq minutes, au-delà on applique un coefficient réducteur de 0,80 en arrondissant à l’entier supérieur le plus proche.

Tarif unitaire appliqué Celui-ci est fixe et vaut 30 centimes d’euro par minute. Toute minute commencée est due.

– les opérateurs à tarif variable :

Durée comptabilisée Il s’agit de la durée réelle de communication.

Tarif unitaire appliqué Celui-ci est fonction de l’heure de la date de début de connexion (quelqu’en soit la durée) en suivant le tableau suivant :

Tarif unitaire	Heure de connexion <i>h</i>
15c	20h ≤ <i>h</i> < 8h
30c	8h ≤ <i>h</i> < 12h ou 14h ≤ <i>h</i> < 20h
45c	12h ≤ <i>h</i> < 14h

- il existe deux types de modes de paiement : les *cartes pré-payées* et les *cartes bancaires*. Chacun de ces modes de paiement implémente de façons différentes l’interface *ModeDePaiement*.

- les cartes bancaires sont toujours valides. Elle dispose d’un solde, exprimé en nombre entiers de euros, qui peut être négatif. Lorsqu’elles sont utilisées comme mod de paiement elles sont donc débitées d’une somme correspondant à la durée de communication comptabilisée multipliée par le tarif unitaire appliqué.
- les cartes pré-payées correspondent à un certain nombre de minutes de communications disponibles. Ce nombre de minutes ne peut jamais être négatif et une carte pré-payée est invalide dès que ce nombre est nul. Initialement, une carte pré-payée offre (si on peut dire) 50 minutes de communications au tarif unitaire de 15 centimes d’euros par minute. Une carte bancaire est toujours associé à une carte pré-payée à la construction de celle-ci pour deux raisons :

– la carte bancaire doit être immédiatement débitée du montant correspondant au coût des 50 minutes de communication.

¹chaque minute commencée est comptabilisée et due à l’opérateur

- cette carte bancaire sert de caution en cas de dépassement de durée de communication comptabilisée lorsque la carte pré-payée est utilisée comme moyen de paiement. Si la durée de communication comptabilisée est inférieure au nombre de minutes restant sur la carte pré-payée, pas de problème, le nombre de minutes disponible sur la carte est diminué de cette durée. Dans le cas contraire, après avoir utilisée toutes les minutes qui restaient sur la carte, la durée restante, correspondant à la différence entre la durée de communication comptabilisée et le nombre de minutes encore disponibles sur la carte pré-payée, est facturée sur la carte bancaire associée à celle-ci au tarif unitaire propre à l'opérateur. La carte pré-payée sera ensuite invalide.

Vos classes devront se conformer, aux noms des méthodes près, à la simulation présentée à la question 4. Il est conseillée d'étudier les tests proposés par la classe `Simulation.java` avant de répondre aux questions, cela devrait pouvoir vous aider.

Le code la classe `Connexion` est fournie.

Pour gérer les heures de connexion et déconnexion on utilise la classe suivante dont le code est également fourni (cf. fichiers fournis sur le portail) :

<code>telephonie::util::Date</code>
...
<code>+Date(...)</code>
<code>+getHeures() : int // l'heure entre 0 et 23</code>
<code>+getMinutes() : int // les minutes entre 0 et 59</code>
<code>+differenceEnMinutes(d:Date) : int // ... "this-d" en mn entières (arrondi au sup)</code>
<code>+addMinutes(mn: int) : Date</code>

On supposera (sans vérification) qu'aucune connexion ne dure plus de 24h.

Q 1 . Codez les classes permettant de modéliser les deux types d'opérateur (et donc conformes à l'interface `Opérateur`).

NB : pour cet exercice de simulation, il est inutile de mémoriser dans les opérateurs les connexions qu'ils créent. Il suffit simplement d'en mémoriser le nombre (sans oublier de le décrémenter en cas de connexion).

Q 2 . Codez les entités permettant la modélisation des modes de paiement (il faut évidemment réutiliser le type `ModeDePaiement`).

Q 3 . *Simulation.* Afin de tester les différentes classes précédentes, vous pouvez utiliser la simulation proposée dans `Simulation.java`, éventuellement après avoir renommé certains identificateurs.

Vous ajouterez éventuellement des traces permettant de mettre en évidence le bon fonctionnement des classes que vous avez définies.

Cette simulation est définie selon le scénario ci-dessous :

- création d'un opérateur à tarif fixe acceptant au plus 3 connexions,
- création d'un opérateur à tarif variable acceptant au plus 4 connexions,
- création des modes de paiement : une carte bancaire (avec un solde très élevé) et une carte pré-payée associée à cette carte bancaire.
- tentative de création 5 connexions sur chacun des opérateurs (ces connexions sont stockées dans des tableaux).

Utilisation des 2 modes de paiement créés précédemment.

Les exceptions levées sont mises en évidence par un affichage.

Les dates de début de connexion sont créées aléatoirement : jour identique pour toutes les connexions, heures entre 0 et 23 et minutes entre 0 et 59².

- déconnexion de son opérateur chacune des "premières" connexions de chaque opérateur. Les dates de fin de connexion sont créées aléatoirement en ajoutant un nombre de minutes aléatoire (et pas trop grand) à l'heure de début de connexion (méthode `addMinutes` de `Date`).

NB : vous ajouterez dans les méthodes `debiter` des modes de paiement, une trace permettant de visualiser l'opération de débit.

- création pour chacun des opérateurs d'une nouvelle connexion,

- tentative de déconnexion d'une connexion du mauvais opérateur, l'exception levée est mise en évidence par un affichage (rappel : on peut visualiser la pile des appels de méthodes au moment de l'exception par la méthode `printStackTrace()` de `Exception`).
- déconnexion toutes les connexions (qui ont été réellement créées), l'exception levée lors de l'invalidité d'un mode de paiement est mise en évidence par un affichage.

²On rappelle qu'il existe principalement deux manières de produire des séquences aléatoires : la méthode statique `random()` de la classe `Math` et les méthodes `nextXXX()` de la classe `java.util.Random`