LU3IN002 : Programmation par objet
Licence de Sciences et Technologies, Sorbonne Université
Année 2019–2020

Partiel : gestionnaire de paquets

13 mars 2020

durée : 2h

Déroulement de l'épreuve

- L'examen est individuel et se déroule sur machine.
- Vous devez fournir des sources Java qui compilent et donnent les résultats demandés.
- À la fin de l'épreuve, sauvegardez vos fichiers, quittez les applications (Eclipse, etc.) et fermez votre session. Les fichiers Java laissés sur votre compte seront automatiquement ramassés.
- Vos réponses seront notées par une banque de tests JUnit. La compilation sans erreur et chaque test passé avec succès vous rapportera un certain nombre de points.
- Le barème est donné à titre indicatif et pourra être révisé lors de la correction.
- Durant l'épreuve, vous n'avez pas accès à internet. Néanmoins, vous pouvez consulter le site de l'UE et une copie locale de la documentation de l'API Java 8.
- Les documents papier (transparents du cours, notes personnelles, livres, etc.) sont autorisés, ainsi qu'une clé USB personnelle en lecture seule au format FAT. Les appareils électroniques et moyens de communication (téléphones, ordinateurs personnels, etc.) sont interdits.

Mise en place : projet Eclipse

Une archive **`PkgMan.zip`** est fournie. Elle contient un projet Eclipse nommé **`PkgMan`**, avec quelques classes et interfaces, ainsi que des tests pour vous aider à vérifier votre réponse. Les tests utilisés lors de la notation seront beaucoup plus nombreux et complexes.

- Téléchargez le fichier **`PkgMan.zip`** sur le site de l'UE et décompressez-le dans la racine de votre répertoire personnel. Il crée un répertoire `eclipse-workspace/PkgMan`.
- Lancez Eclipse pour Java en tapant `./eclipse` dans un terminal.
  Spécifiez `eclipse-workspace` comme répertoire de travail d'Eclipse.
  Il est possible qu'Eclipse se bloque en essayant d'installer un connecteur SVN. Dans ce cas, tapez dans un terminal : `killall eclipse; killall java` et relancez Eclipse.
- Importez le projet Java contenu dans `eclipse-workspace/PkgMan` : allez dans « File / Open Projects From Filesystem... », cliquez sur « Directory... », choisissez le répertoire `eclipse-workspace/PkgMan`, puis « Finish ».
- Nous travaillons dans le projet **`PkgMan`**, dans le package **`pobj.pkgman`** et ses sous-packages. Les fichiers à fournir seront donc à placer dans des sous-répertoires de `eclipse-workspace/PkgMan/src/pobj/pkgman/`

Assurez-vous de :

- respecter les noms de classe, de package et de chemin de fichier indiqués dans l'énoncé ;
- fournir des sources qui compilent, même si elles ne répondent pas complètement aux questions (les méthodes qui fonctionnent pourront vous rapporter une partie des points) ;
- implanter les interfaces et hériter des classes spécifiées dans l'énoncé ;
- ne pas modifier les interfaces fournies ; sinon, les tests de notation, qui sont programmés vis-à-vis de ces interfaces, risquent de ne plus compiler ;

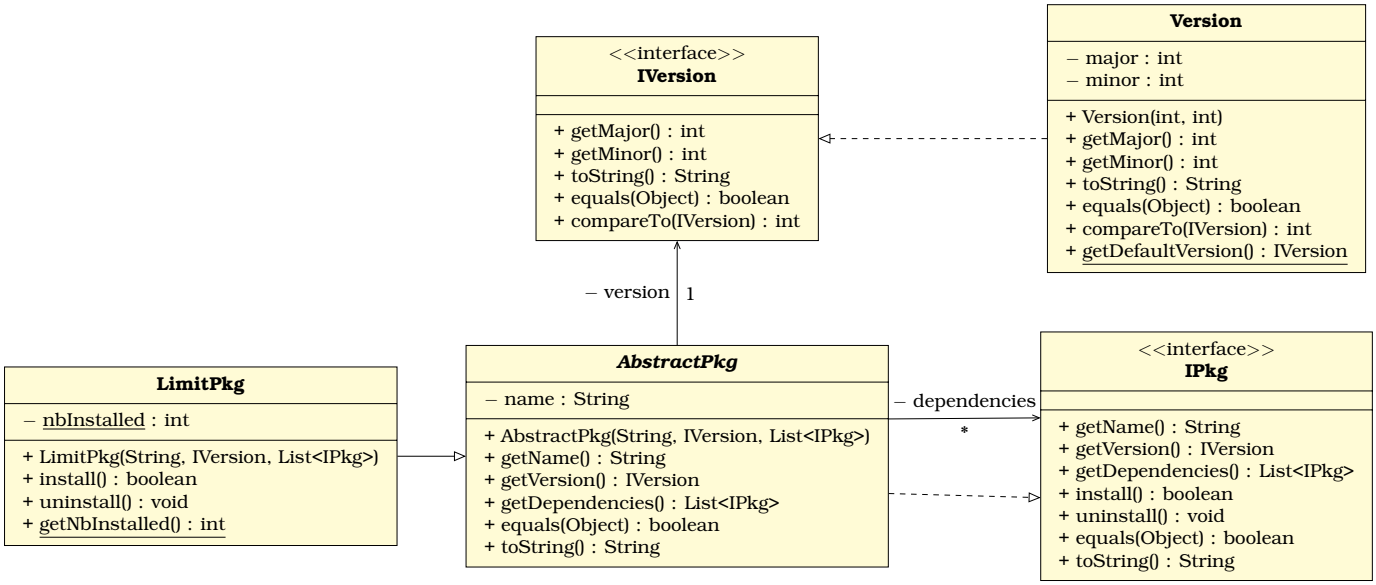
- respecter les consignes de visibilité : les méthodes seront publiques et les attributs privés.

Introduction

Le thème de l'examen est la programmation d'un gestionnaire de paquets (*package manager*), tel que trouvé dans certains systèmes d'exploitation ou langages (`apt` pour Debian, `pip` pour Python). Il permet l'installation, la désinstallation, la mise à jour de bibliothèques et applications, en gérant les *dépendances*. Les dépendances d'un paquet sont les autres paquets qui doivent être également installés pour que celui-ci fonctionne : quand l'utilisateur demande l'installation d'un paquet, le gestionnaire se chargera d'installer toutes ses dépendances (si elles ne sont pas déjà installées), mais également les dépendances des dépendances, etc., récursivement.

1 Paquets

Un paquet implante l'interface `IPkg` (fournie). Nous en construirons d'abord une implantation abstraite : `AbstractPkg`, puis une implantation concrète : `LimitPkg`. Elles utilisent la classe `Version`, d'interface `IVersion` (fournie), pour représenter un numéro de version. Pour référence, nous implanterons le diagramme UML suivant :



a) Classe `Version` (2.5 pt)

Une version est une paire d'entiers : un numéro majeur et un numéro mineur.

À faire : Programmez une classe `pobj.pkgman.Version` respectant le diagramme UML et ayant :

- des attributs entiers `major` et `minor`, et les getters associés `getMajor()`, `getMinor()` ;
- un constructeur `Version(int major, int minor)` fixant ces attributs.

Les objets `Version` sont immuables : une fois construits, la valeur des attributs ne peut plus être modifiée. Vous vous en assurerez en utilisant le modificateur adéquat dans la déclaration.

La classe redéfinira les méthodes standard suivantes :

- le test d'égalité `boolean equals(Object o)` vérifiant que `this` et `o` ont le même numéro majeur et le même numéro mineur ;
- la conversion en représentation textuelle `String toString()`, la chaîne retournée contiendra le numéro majeur, puis un point, puis le numéro mineur ; par exemple : `"1.0"` ou `"19.10"`.

La classe implantera l'interface `Comparable<IVersion>` et aura donc une comparaison d'ordre `int compareTo(IVersion v)`. Elle retourne 1 si `v` est antérieure à `this`, -1 si elle est

postérieure, et 0 si elles sont égales. Nous considérons que la version *major1.minor1* est antérieure à *major2.minor2* si *major1* < *major2*, ou bien *major1* == *major2* et *minor1* < *minor2* (donc `1.1` est antérieure à `1.2` et à `2.0`).

Vous ajouterez enfin une méthode statique `IVersion getDefaultVersion()` qui retourne la version `1.0`. Pour économiser la mémoire, vous éviterez de construire un nouvel objet à chaque appel. Les valeurs retournées par `getDefaultVersion()` doivent donc être égales pour ==.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/Version.java`

Fourni : l'interface `pobj.pkgman.IVersion` et la classe de test `pobj.pkgman.test.VersionTest`

b) Classe abstraite `AbstractPkg` (1.5 pt)

Un paquet doit implanter l'interface (fournie) `pobj.pkgman.IPkg` et avoir :

- un attribut nom (`String`) ;
- un attribut numéro de version (`IVersion`) ;
- un attribut liste de dépendances (`List<IPkg>`) ;
- une méthode d'installation retournant `true` si l'installation du paquet a réussi, `false` sinon ;
- une méthode de désinstallation, qui ne retourne rien (elle n'échoue jamais) ;
- une comparaison d'égalité : le nom et le numéro de version doivent correspondre ; la comparaison ignore les dépendances ; `compareTo` n'est pas demandé ;
- une conversion en chaîne : elle retourne une chaîne composée du nom et de la version, séparés par un tiret ; par exemple : `"java-1.8"`.

Les méthodes d'installation et de désinstallation varient d'un paquet à l'autre, mais la gestion des attributs est semblable. Nous allons factoriser le code commun dans une classe abstraite.

À faire : Programmez une classe abstraite `pobj.pkgman.AbstractPkg` qui implante `IPkg` en fournissant : les attributs nom, version et liste de dépendances ; leurs getters ; le constructeur `AbstractPkg(String nom, IVersion version, List<IPkg> dependencies)` les fixant ; les méthodes `equals` et `toString`. Vous laisserez les méthodes `install` et `uninstall` abstraites.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/AbstractPkg.java`

Fourni : l'interface `pobj.pkgman.IPkg`
Nous ne fournissons aucun test car la classe `AbstractPkg` n'est pas instanciable.

c) Classe concrète `LimitPkg` (1.5 pt)

À faire : Programmez une classe `pobj.pkgman.LimitPkg` qui implante `IPkg` en héritant de `AbstractPkg`. Elle se contente de compter le nombre global de paquets installés, et d'interdire l'installation si 10 paquets ou plus sont déjà installés. Elle aura :

- un constructeur de même signature que celui d'`AbstractPkg` ;
- un compteur statique entier qui démarre à 0, et son getter `getNbInstalled()` statique ;
- une méthode `install` qui retourne `false` si le compteur est supérieur ou égal à 10, et incrémente le compteur et retourne `true` sinon ;
- une méthode `uninstall` qui décrémente le compteur ;
- les autres méthodes demandées par `IPkg` sont héritées d'`AbstractPkg`.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/LimitPkg.java`

Fourni : la classe de test JUnit `pobj.pkgman.test.LimitPkgTest`

2 Chargement d'une base de paquets : classe `Sample` (1 pt)

Nous fournissons dans le package `pobj.pkgman.loader` une classe `PkgLoader` capable de charger une liste de paquets `List<IPkg>` depuis un fichier texte simple. Le constructeur de `PkgLoader` prend en argument un nom de fichier texte, qu'il ouvre, et charge la liste des paquets. Cette liste peut ensuite être retrouvée par un appel à `getPackages()`. `PkgLoader` utilise les classes `LimitPkg` et `Version` de la question précédente pour créer les paquets.

Le fichier texte est composé d'une liste de lignes. Chaque ligne a la forme "`pkg: dep1 ... depN`". Elle indique la présence du paquet `pkg`, qui dépend des paquets `dep1` à `depN`. Chaque élément `pkg`, `dep1`, etc., précise le nom et la version du paquet, avec la convention utilisée par `toString()` : *`nom-version`*. Le fichier `example.txt` du package contient l'exemple suivant :

```
java-7.0:
java-8.0:
junit-4.0:  java-8.0
eclipse-4.6:  java-7.0 junit-4.0
```

Il indique l'existence de 4 paquets : `java-7.0`, de nom `java`, de version `7.0`, sans dépendance ; `java-8.0`, de nom `java`, de version `8.0`, sans dépendance ; `junit-4.0`, de nom `junit`, de version `4.0`, qui dépend de `java-8.0` ; et enfin `eclipse-4.6`, de nom `eclipse` et de version `4.6`, qui dépend de `java-7.0` et de `junit-4.0`.

Nous programmons ici un exemple d'utilisation de `PkgLoader`.

À faire : Programmez une classe `pobj.pkgman.loader.Sample` exécutable qui a :

- une méthode statique `void printPkgs(List<IPkg> lst)` qui affiche à l'écran la liste des paquets `lst` avec leurs dépendances, sous forme de lignes "`pkg: dep1 ... depN`" identiques au format d'entrée ;
- un point d'entrée qui charge le contenu du fichier `example.txt` avec la classe `PkgLoader`, puis affiche à l'écran la liste des paquets grâce à la méthode `printPkgs` ; l'exécution de `Sample` affichera donc à l'écran un contenu identique au fichier `example.txt` lu.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/loader/Sample.java`

Fourni : la classe `pobj.pkgman.loader.PkgLoader` ; aucun test JUnit n'est fourni : vous pouvez vérifier visuellement le résultat de l'exécution de `Sample`.

3 Gestionnaire de paquets : classe `Manager` (2.5 pt)

Nous programmons maintenant un gestionnaire de paquets qui, lors de l'installation d'un paquet, se charge d'installer récursivement toutes ses dépendances. Il maintient une liste des paquets déjà installés, ce qui évite les installations et désinstallations inutiles.

À faire : Programmez une classe `pobj.pkgman.Manager` qui implante `IManager` (fournie) et a :

- un constructeur sans argument ;
- une méthode `List<IPkg> getInstalled()` retournant la liste des paquets actuellement installés ; elle est initialement vide ;
- une méthode `boolean installPkg(IPkg pkg)` qui installe le paquet `pkg` et ses dépendances, s'il n'est pas déjà installé : elle commence par s'appeler récursivement pour installer toutes les dépendances de `pkg`, puis elle appelle sa méthode `install()` ; la méthode retourne `false` à la première installation qui échoue, et `true` si toutes les installations ont réussi ; la liste des paquets installés est mise à jour en y ajoutant les paquets installés avec succès ; installer un paquet déjà présent retourne immédiatement `true`, sans installer ses dépendances ni appeler sa méthode `install()` ;

- une méthode `void uninstallPkg(IPkg pkg)` qui désinstalle le paquet `pkg` en appelant sa méthode `uninstall()`, si celui-ci était installé ; les dépendances ne sont pas désinstallées (elles pourraient être utiles à d'autres paquets qui restent installés) ; la liste des paquets installés est mise à jour ; si le paquet n'était pas installé, `uninstall()` n'est pas appelée.

En reprenant l'exemple de la question 2, l'installation d'`eclipse-4.6` à partir d'une liste vide de paquets installera, dans l'ordre : `java-7.0`, `java-8.0`, `junit-4.0`, puis `eclipse-4.6` et retournera `true`. Installer ensuite `junit-4.0` n'aura aucun effet et retournera `true`. Ici, deux paquets de même nom mais de version différente (comme `java-7.0` et `java-8.0`) sont considérés comme différents, et peuvent être installés simultanément. Ceci sera corrigé en question 6.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/Manager.java`

Fourni : la classe de test `pobj.pkgman.test.ManagerTest`

4 Adaptation d'interface : classe `SimplePkgAdapter` (2 pt)

L'interface `pobj.pkgman.ISimplePkg` fournie propose une description simplifiée des paquets :

- la méthode `String name()` retourne le nom du paquet ;
- la méthode `void install()` installe le paquet ; elle réussit toujours et ne retourne donc rien ;
- il n'y a pas de méthode de désinstallation, ni de numéro de version, ni de dépendance.

Nous souhaitons utiliser ces paquets dans notre application, qui est programmée vis-à-vis d'`IPkg`. Nous programmons une classe pour adapter les classes implantant `ISimplePkg` à l'interface `IPkg`. Elle va implanter `IPkg` par délégation à un objet d'interface `ISimplePkg` fixé à la construction.

À faire : Programmez une classe `pobj.pkgman.SimplePkgAdapter` qui implante `IPkg` et :

- a un attribut `ISimplePkg`, et un constructeur `SimplePkgAdapter(ISimplePkg)` qui le fixe ;
- délègue `install()` à l'attribut et retourne `true` ;
- a une méthode `uninstall()` qui ne fait rien ;
- a pour version `Version.getDefaultVersion()` et une liste vide de dépendances.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/SimplePkgAdapter.java`

Fourni : l'interface `pobj.pkgman.ISimplePkg` et la classe de test `pobj.pkgman.test.SimplePkgAdapterTest`

5 Ajout d'une fonctionnalité de log

Nous ajoutons à nos paquets une fonction de *log* (journal) qui garde une trace de tous les appels à `install()` et `uninstall()`. Pour éviter de modifier les classes existantes, nous allons créer une classe décorateur. Elle a en attribut une référence sur l'objet d'interface `IPkg` à enrichir, et lui délègue les méthodes après avoir mis à jour le log. Pour pouvoir supporter plusieurs types de log (stockage en mémoire, sortie écran ou dans un fichier, etc.), le décorateur prendra également en attribut un objet d'interface `ILogger` (fournie). Il appellera sa méthode `log(String)` pour ajouter des messages au log. Nous travaillons ici dans le package `pobj.pkgman.log`.

a) Implantation du log : `LogBuffer` (1 pt)

Nous programmons une implantation de `ILogger` qui accumule en mémoire les chaînes de caractères passées à `log(String)`. Vous prendrez garde à la performance de la méthode d'ajout en choisissant judicieusement le type de l'attribut stockant l'accumulation des messages.

À faire : Programmez une classe `pobj.pkgman.log.LogBuffer` implantant `ILogger` et ayant :

- un constructeur sans argument ; le log est initialement vide ;
- une méthode `void log(String msg)` qui ajoute la chaîne `msg` à la fin du log, suivie d'un retour à la ligne `"\n"` ;
- une méthode `String getLog()` qui retourne le contenu du log sous forme de chaîne.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/log/LogBuffer.java`

Fourni : l'interface `pobj.pkgman.log.ILogger` et la classe de test `pobj.pkgman.test.LogBufferTest.java`

b) Classe décorateur : `PkgLogger` (2 pt)

À faire : Programmez une classe `pobj.pkgman.log.PkgLogger` qui :

- implante l'interface `IPkg` ;
- a un attribut de type `IPkg` et un de type `ILogger`, fixés par le constructeur `PkgLogger(IPkg pkg, ILogger log)` ;
- délègue à `pkg` tous les appels aux getters, ainsi que `install`, `uninstall`, `equals` et `toString` ;
- lors de l'installation, ajoute au log le message `"Installing` *`paquet`*`"`, suivi du message `"Success"` ou `"Failure"` en fonction de la valeur de retour de `install()` ; *`paquet`* est la chaîne retournée par `toString()`, donc de la forme *`nom-version`* ;
- ajoute au log le message `"Uninstalling` *`paquet`*`"` lors de désinstallation.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/log/PkgLogger.java`

Fourni : la classe de test `pobj.pkgman.test.PkgLoggerTest`

6 Gestion correcte des versions : classe `ManagerVersion` (2.5 pt)

Nous revenons sur le gestionnaire `Manager` de la question 3, mais nous supposons maintenant qu'une seule version à la fois d'un paquet d'un nom donné peut être installée sur le système. Une dépendance sur un paquet *`nom-version`* indique maintenant que le paquet de nom *`nom`* doit être installé avec la version *`version`* ou une version supérieure.

L'appel à `installPkg(pkg)` où `pkg` est un paquet *`nom-version`* correspond alors à installer ou mettre à jour le paquet :

- si aucun paquet de nom *`nom`* n'est installé, alors le paquet *`nom-version`* est installé : toutes ses dépendances sont d'abord récursivement installées (ou mises à jour) par `installPkg`, puis sa méthode `install()` est appelée ;
- si un paquet de nom *`nom`* et de version égale ou supérieure à *`version`* est déjà installé, rien n'est fait ;
- si un paquet de nom *`nom`* et de version strictement inférieure à *`version`* est déjà installé, alors il est mis à jour : le paquet installé est d'abord désinstallé, sans toucher à ses dépendances ; les dépendances du paquet *`nom-version`* sont ensuite installées (ou mises à jour) récursivement, et enfin la méthode `install()` du paquet *`nom-version`* est appelée.

En reprenant l'exemple de la question 2, l'installation d'`eclipse-4.6` à partir d'une liste vide de paquets installera d'abord `java-7.0` (première dépendance d'Eclipse), le désinstallera, puis installera `java-8.0` (car `java-7.0` n'est pas suffisant pour JUnit, deuxième dépendance d'Eclipse), et enfin installera `junit-4.0` puis `eclipse-4.6` (ce n'est pas très efficace, mais nous ne cherchons pas ici à optimiser la séquence d'installation).

À faire : Programmez une classe `pobj.pkgman.ManagerVersion` implantant la même interface `IManager` que celle de la question 3 et obéissant à cette nouvelle spécification. Il ne sera pas utile d'hériter de la classe `Manager` de la question 3.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/ManagerVersion.java`

Fourni : la classe de test `pobj.pkgman.test.ManagerVersionTest`

7 Mise à jour automatique

Un gestionnaire réel maintient le système à jour en téléchargeant périodiquement la liste des paquets existants et en installant les dernières versions des paquets déjà installés. Nous programmons ici une classe `Updater` qui implante ce mécanisme. Elle utilise en interne la classe `ManagerVersion` de la question 6, à qui elle délègue l'installation effective et la gestion des dépendances. Elle utilise aussi un objet d'interface `IConnector`, implanté ci-dessous, pour télécharger la liste des paquets.

a) Base de paquets : classe `Connector` (1.5 pt)

L'interface `IConnector` (fournie) a une seule méthode : `IDatabase getDatabase()`. Celle-ci télécharge une nouvelle liste de paquets et la rend disponible via l'objet `IDatabase` retourné. Elle retourne `null` en cas d'échec du téléchargement.

L'interface `IDatabase` (fournie) a deux méthodes :

- `IPkg getPackage(String name)` retourne un paquet dont le nom (`getName`) est `name` et qui a la version disponible la plus élevée, ou `null` si aucun paquet de ce nom n'existe ;
- `void close()` est appelée quand la base de données n'est plus utilisée.

Dans la réalité, `getDatabase()` initiera un protocole client-serveur avec un serveur de mises à jour, et `close()` terminera la connexion. Pour simplifier notre implantation, `getDatabase` téléchargera entièrement la liste des paquets grâce à la classe `PkgLoader` fournie (question 2), si bien que `close` n'aura aucun travail à faire.

À faire : Programmez une classe `pobj.pkgman.updater.Connector` implantant l'interface `IConnector`, ayant un constructeur `Connector(String file)` spécifiant un nom de fichier, et une méthode `IDatabase getDatabase()` qui :

- utilise `PkgLoader` pour télécharger les paquets du fichier `file` précisé dans le constructeur ;
- à partir de la liste retournée, construit une `Map` associant à chaque nom de paquet un objet `IPkg` correspondant à la version la plus élevée ; l'exemple de la question 2 montre que le fichier peut contenir plusieurs versions d'un même paquet ;
- retourne un objet implantant l'interface `IDatabase` permettant d'explorer cette `Map` via la méthode `IPkg getPackage(String)` ; la méthode `close` sera vide.

Vous pouvez définir la classe implantant `IDatabase` dans une classe interne à `Connector`, mais ce n'est pas obligatoire.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/updater/Connector.java`

Fourni : les interfaces `pobj.pkgman.updater.IConnector` et `IDatabase`, la classe `pobj.pkgman.loader.PkgLoader` et la classe de test `pobj.pkgman.test.ConnectorTest`

b) Classe d'exceptions : `UpdateException` (0.5 pt)

Pour indiquer l'échec de la mise à jour, nous définissons ici une nouvelle classe d'exceptions.

À faire : Programmez une classe d'exceptions `pobj.pkgman.updater.UpdateException`. Son constructeur prendra en argument une chaîne précisant le message d'erreur.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/updater/UpdateException.java`

c) Mise à jour : classe `Updater` (1.5 pt)

À faire : Programmez une classe `Updater` qui a :

- des attributs `IConnector` et `IManager` fixés par le constructeur `Updater(IConnector c, IManager man)` ;
- une méthode `List<String> getInstalled()` donnant la liste des noms des paquets installés ; vous pourrez obtenir la liste des paquets par le gestionnaire `man`, puis extraire leur nom ;
- une méthode `void installPkgs(List<String> pkgs)` qui permet d'installer des paquets à leur dernière version ou de mettre à jour des paquets, connaissant uniquement leur nom : la méthode obtiendra une nouvelle base `IDatabase` par l'attribut `c` pour convertir les noms de paquet en objets `IPkg`, puis utilisera le gestionnaire `man` pour leur installation effective, et appellera enfin `close` ; en cas d'erreur d'installation, d'échec de téléchargement de la base, ou de paquet non trouvé, une exception `UpdateException` sera signalée ; `close` sera également appelée en cas d'erreur ;
- une méthode `void updateWorld()` qui met à jour tous les paquets installés ; elle se programme très simplement en passant à `install` le résultat retourné par `getInstalled`.

À fournir : `eclipse-workspace/PkgMan/src/pobj/pkgman/updater/Updater.java`

Fourni : la classe de test `pobj.pkgman.test.UpdaterTest`