TD JAVA n°11: Compléments persistance des objets : fichier properties et présentation fabrique de DAO

Exercice 1 : Lecture d'un properties File …

Les caractéristiques d'un composant pouvant être modifiées au moment de la conception s'appellent des *propriétés* [properties]. Ces informations peuvent être stockées dans de simples fichiers textes manipulables par n'importe quel éditeur de texte **:** on les appelle les **Properties Files.**

Les fichiers de propriétés doivent être formatés de la manière suivante :

```
NomPropriété = ValeurPropriété
```

La première chaîne précise le label de la *propriété* (qui ne doit pas être modifié), la seconde sa *valeur.*
Pour notre application, on crée le fichier d'initialisation (`.properties`) suivant :

CabMed.properties

```
driver = sun.jdbc.odbc.JdbcOdbcDriver
url = jdbc:odbc:cabinetMedical
```

En java, il existe la classe `java.util.Properties`
Cette classe gère une *collection d'objets* au travers d'une table de hachage particulière dont les clés et les valeurs sont de types prédéfinis à savoir tous deux des `String`.

java.util
Class Properties

java.lang.Object
  └ java.util.Dictionary
    └ java.util.Hashtable
      └ **java.util.Properties**

On peut voir cette classe comme un dictionnaire qui associe deux chaînes de caractères c.a.d qu'un objet **`Properties`** ne garde que des objets **`String`** pour à la fois la clé et les valeurs

1. Ecrire une classe **`ChargeProperty`** sous la forme ***d'un singleton***.

Cette classe aura un attribut de type `Properties` qui mémorisera la liste des propriétés du fichier cabMed.properties :

```
private Properties tableProprietes = new Properties();
```

Il n'est pas nécessaire d'écrire de getteur /setteur pour cet attribut, en revanche :
↳ **1.a** Comme tout attribut, la `tableProprietes` doit être initialisée dans le constructeur (c-a-d que le ***chargement*** des valeurs dans la `tableProprietes`, effectué à partir de la lecture des données du ***fichier*** **CabMed.properties** ne s'effectuera qu'une seule fois dans le constructeur).
Si quelque chose d'anormal se produit, lancer une **`CabinetTechniqueException`.**

Constructor Summary

`Properties()`
Creates an empty property list with no default values.

Method Summary

String	**getProperty**(String key) Searches for the property with the specified key in this property list.
void	**load**(InputStream inStream) Reads a property list (key and element pairs) from the input stream.

Rappel :

```
public abstract class InputStream
extends Object
implements Closeable
```

Direct Known Subclasses:
AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

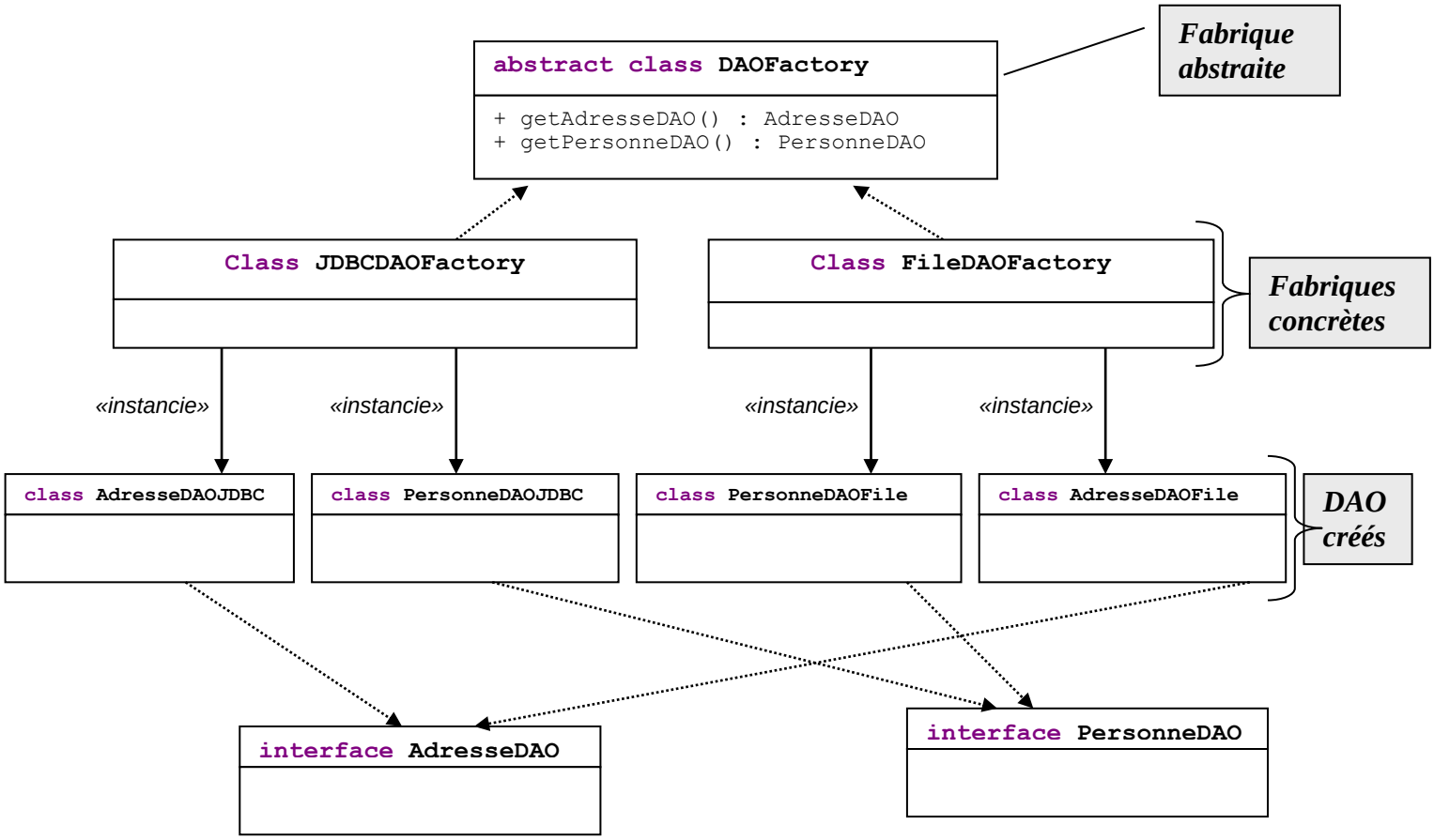
↳ **1.b** une propriété de la `tableProprietes` sera récupérée grâce à la méthode :

```
public String getPropriete(String nomPropriete) throws CabinetTechniqueException
```

qui retournera la valeur de la propriété dont le nom est passé en paramètre.
La méthode `getProperty` de la classe `Properties` renvoie `null` si la propriété n'existe pas dans le fichier : on souhaite que la méthode `getPropriete` de la classe `ChargeProperty` renvoie une `CabinetTechniqueException` si un tel cas se produit.

2. Quelle est la syntaxe à utiliser pour récupérer l'url de la connexion en utilisant un objet `ChargeProperty` ?

Et pour finir…Présentation du pattern fabrique abstraite …



Ce schéma, réalisé dans le cadre de notre implémentation, rejoint tout à fait la figure n°9.8 du tutoriel de Sun sur le `Data Access Object`(http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html)

Principe DAO Factory : *Utilisation d'une fabrique pour créer/récupérer un* `DAO` *et donc cacher le type concret de la classe d'une instance en création*

- **Fabrique abstraite :** pour cacher le type réel d'un ensemble *de fabriques concrètes*
- **Fabriques concrètes :** pour fournir tous les `DAO` (de chaque objet métier) associés à une certaine source de données

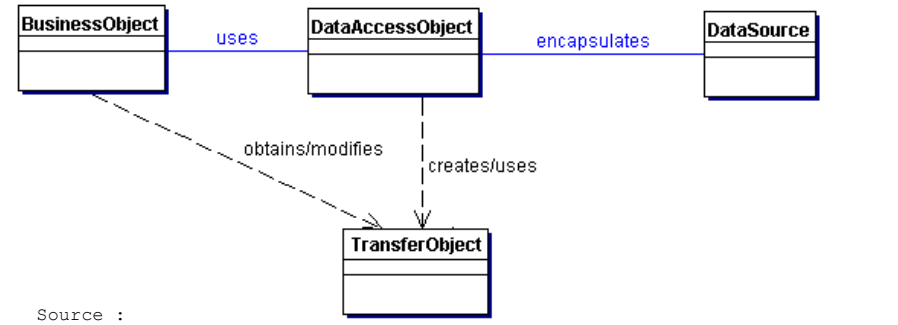
Cette implémentation est basée sur le Design Pattern du Gof : ***fabrique abstraite*** (ou ***abstract factory***) .
Une **fabrique** est un endroit du code où sont construits des objets (voir Annexe I)
Avantage : Le but de ce patron de conception est ***d'isoler la création des objets de leur utilisation***.
On peut ainsi ajouter de nouveaux objets dérivés sans modifier le code qui utilise l'objet de base.
Avec ce patron de conception, on peut interchanger des classes concrètes sans changer le code qui les utilise, même à l'exécution.
Inconvénient : Ce patron de conception exige un travail supplémentaire lors du développement initial, et apporte une certaine complexité….

… Vous trouverez une explication détaillée pour les DAO les pages qui suivent…

Explications détaillées concernant le pattern fabrique abstraite…

Le pattern DAO permet d'encapsuler les accès aux données. Le DAO masque complètement les détails de la source de données à ses clients. Il propose une interface qui ne change pas en fonction de la source de données.
Le DAO agit comme une façade entre les objets métiers et la source de données(en masquant les détails de l'implémentation de l'accès aux bases de données)

Le site de Sun nous fournit ***le diagramme de classes*** suivant qui montre les relations entre les objets qui sont nécessaires à la mise en place du pattern DAO.



Source :
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

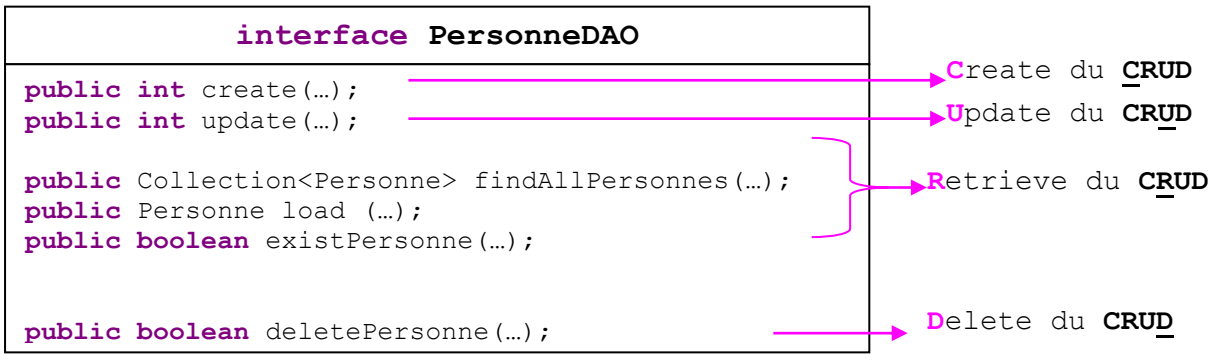
Ainsi à la classe métier `Personne` devraient être associées :
- les classes DataAccessObject : `PersonneDAOFichier` et `PersonneDAOJDBC`
- la classe TransferObject : `PersonneDTO`

Jusqu'à présent en TD/TP, nous avons écrit un pattern DAO simplifié puisque l'accès aux bases de données était directement codé dans le `DataAccessObject` et non délégué au `DataSource` comme le préconise le pattern ci-dessus.

Le pattern DAO peut fournir un plus haut niveau de flexibilité en adoptant le ***Design Pattern de fabrique abstraite*** (**`Abstract Factory`** du Gof) . C'est cette stratégie de `factory` pour les DAO que nous allons présenter maintenant …

1. Interface des DAO :

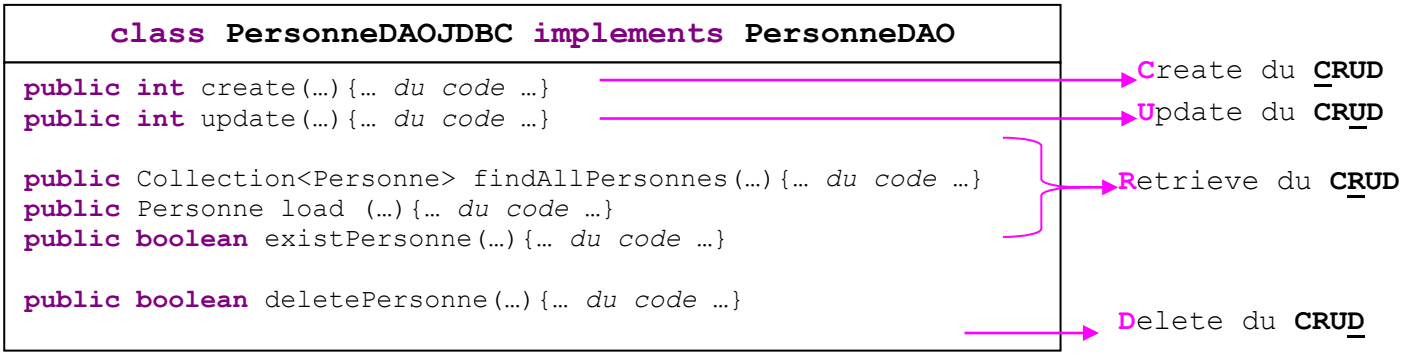
En effet, le pattern DAO consiste à isoler dans un objet dédié toutes les opérations de persistance d'un objet métier. L'utilisation d'un outil de persistance permet de fournir une implémentation générique aux opérations élémentaires de type **CRUD** : **C**reate (création), **R**etrieve(lecture), **U**pdate (modification), **D**elete (suppression).
Les « verbes » de persistance peuvent alors être définis dans une interface.



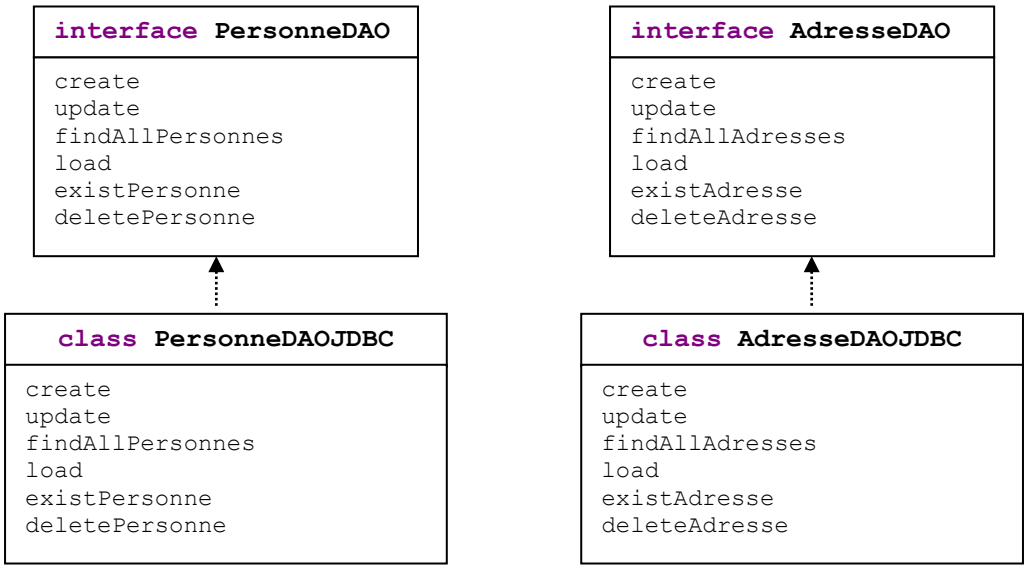
Remarque : comme on travaille sur une interface, les méthodes ne doivent pas être statiques ! ! ! !

2. Implémentation de l'interface DAO dans des classes DAO concrètes

L'implémentation de cette interface encapsule l'utilisation de l'API de bas niveau correspondant au périphérique de stockage. Par exemple, lors de l'utilisation d'une base de données relationnelles, nous utilisons l'API `JDBC` et nous devons créer une ***classe concrète*** appelée : **`PersonneDAOJDBC`** qui implémentera toutes les méthodes de l'interface

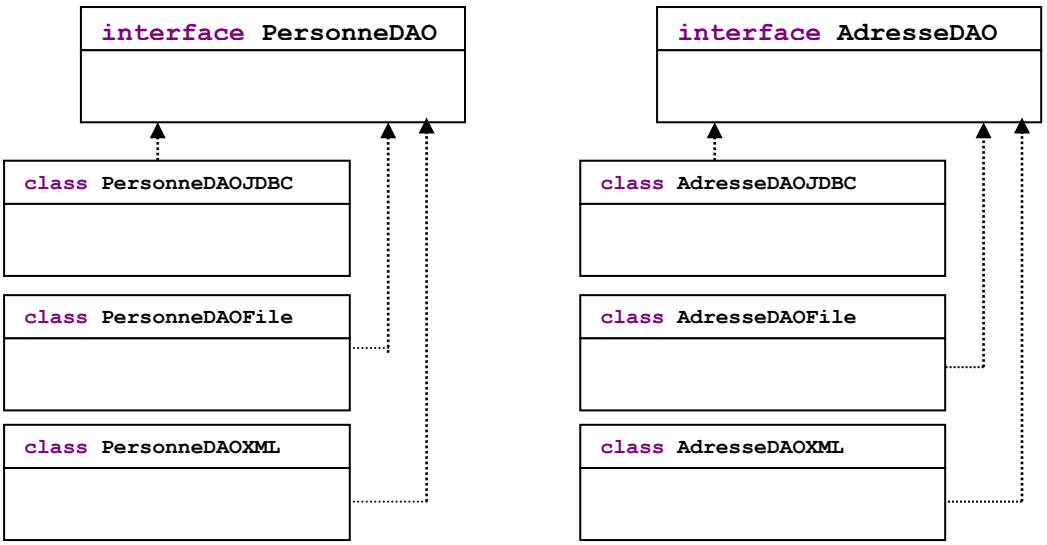


↳ Il est à noter qu'à chaque BusinessObject (classe métier) correspond un DAO spécifique.
Ainsi, il faudrait créer de la même manière l'interface `AdresseDAO` et la classe `AdresseDAOJDBC`.



↳ …mais `PersonneDAOJDBC` n'est pas la seule classe concrète que peut implémenter l'interface `PersonneDAO`.
→ Si on souhaite travailler avec des fichiers comme support de persistance, on devrait implémenter une classe DAO concrète qu'on appellerait : `PersonneDAOFile`.
→ Si on souhaite travailler avec des bases XML comme support de persistance, on devrait implémenter une classe DAO concrète qu'on appellerait : `PersonneDAOXML`.

De même pour les objet de type `Adresse`….



Remarque : On aura autant de classes DAO concrètes que de supports de persistance…
Ainsi, les objets (`PersonneDAOJDBC, PersonneDAOFile, PersonneDAOXML`) peuvent être vus comme une ***famille de « produits » issus de l'interface `PersonneDAO`****.*
De même les objets (`AdresseDAOJDBC, AdresseDAOFile, AdresseDAOXML`) peuvent être vus comme une ***famille de « produits » issus de l'interface `AdresseDAO.`***

3.Implémentation des fabriques concrètes de DAO:

→ 3.1 Principe de fonctionnement :

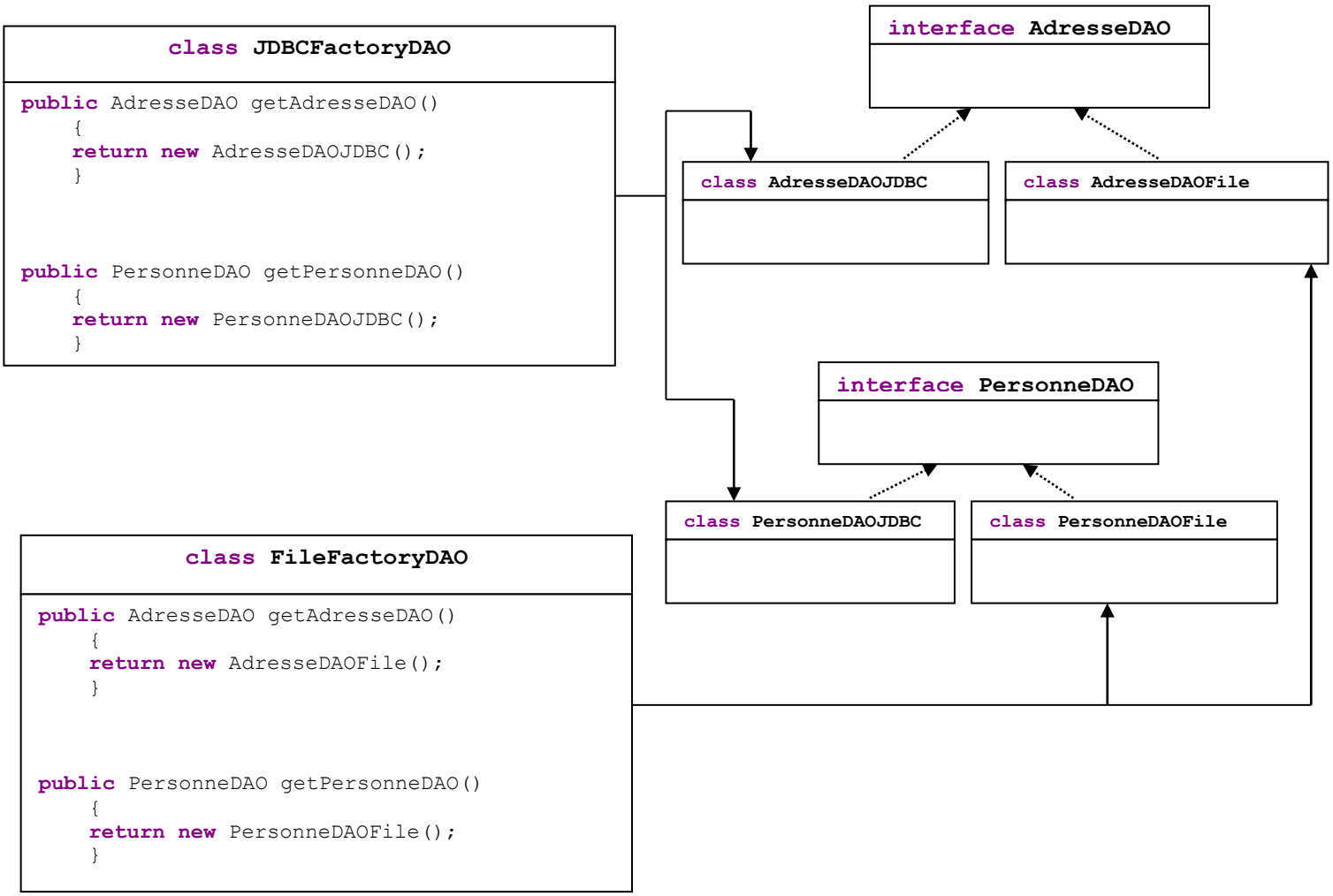
En principe, une application donnée n'utilisera ***qu'un seul support de persistance*** (c-a-d une seule sources de données : soit une base relationnelle, soit un fichier, soit une base XML, …).

A ce stade du développement, il serait intéressant de regrouper tous les DAOs associés à une même source de données dans une même classe. Une telle classe concrète s'appelle une ***fabrique concrète de DAO***.

Dans une application **JDBC**, son rôle sera de fournir au programme tous les objets DAO nécessaires (en les récupérant par une instanciation sur les objets de type `XxxDAOJDBC`): on appellera cette classe `JDBCFactoryDAO`

Dans une application travaillant avec les **fichiers**, son rôle sera de fournir au programme tous les objets DAO nécessaires (en les récupérant par une instanciation sur les objets de type `XxxDAOFile`): on appellera cette classe `FileFactoryDAO`

Le schéma précédent (en ne s'intéressant qu'aux bases de données relationnelles et aux fichiers) pourrait s'organiser désormais de la manière suivante :



Chaque fabrique concrète (**JDBCFactoryDAO, FileFactoryDAO**) peut donc créer tout un ensemble de produits (ici les « produits » sont les « DAO »).
Chaque fabrique concrète propose donc les même services : à savoir un même ensemble de méthodes pour créer des produits (objet DAO pour nous). Dans le cas de notre application, ce seront les méthodes :

```
public AdresseDAO getAdresseDAO() ;
public PersonneDAO getPersonneDAO() ;
```

Il paraît donc possible de ***factoriser ce code*** dans une classe mère qui pourrait définir l'interface que toutes les fabriques concrètes doivent implémenter Cette classe mère est aussi appelée ***fabrique abstraite*** (dans notre cas, ce sera la classe ***DAOFactory***)

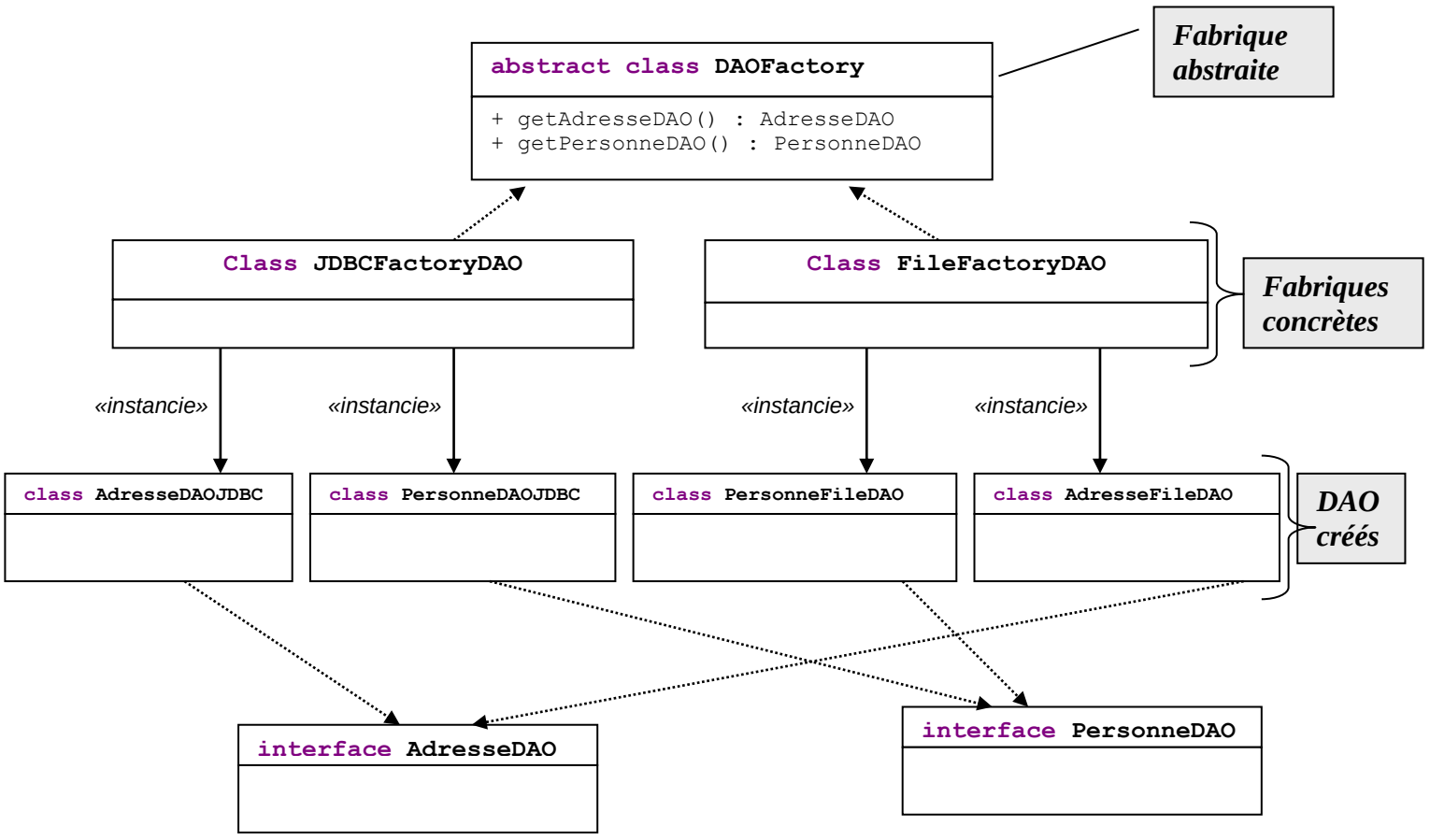
Une ***fabrique abstraite*** encapsule donc un groupe de fabriques ayant une thématique commune (ici la création d'objets DAO) :

- D'une part, elle propose un ensemble de méthodes abstraites (pour créer des produits) que toutes les fabriques concrètes implémenteront.
- D'autre part, la fabrique a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique. La fabrique proposera également une méthode `getDAOFactory`. Comme en général, les fabriques sont uniques dans un programme, on utilise souvent le singleton.

Le code de la classe ***DAOFactory*** ressemble donc à :

```
public abstract class DAOFactory {
    // Liste des DAO supportés par la factory
    public enum TypeDAO {JDBC, Fichier, XML};


    // La fabrique abstraite définit une interface de services
    // Chaque fabrique concrète doit implémenter toutes ces méthodes
    public abstract AdresseDAO getAdresseDAO();
    public abstract PersonneDAO getPersonneDAO();

    //La fabrique a également pour rôle l'instanciation d'objets divers
    public static DAOFactory getDAOFactory(TypeDAO whichFactory) {
              switch (whichFactory) {
              case JDBC:
                  return new JDBCDAOFactory();
                       // passer par getInstance si implémentation
                       // sous forme de Singleton
              case Fichier:
                  return new FileDAOFactory();
              case XML:
                  return new XMLDAOFactory();
              default           :
                  return null;
            }// fin switch
          }// getDAOFactory
```

L'implémentation de notre fabrique de DAO peut donc bien être représentée par la schéma complet suivant.



→ 3.2 Utilisation de la fabrique de DAO dans le projet `cabinetMedical`

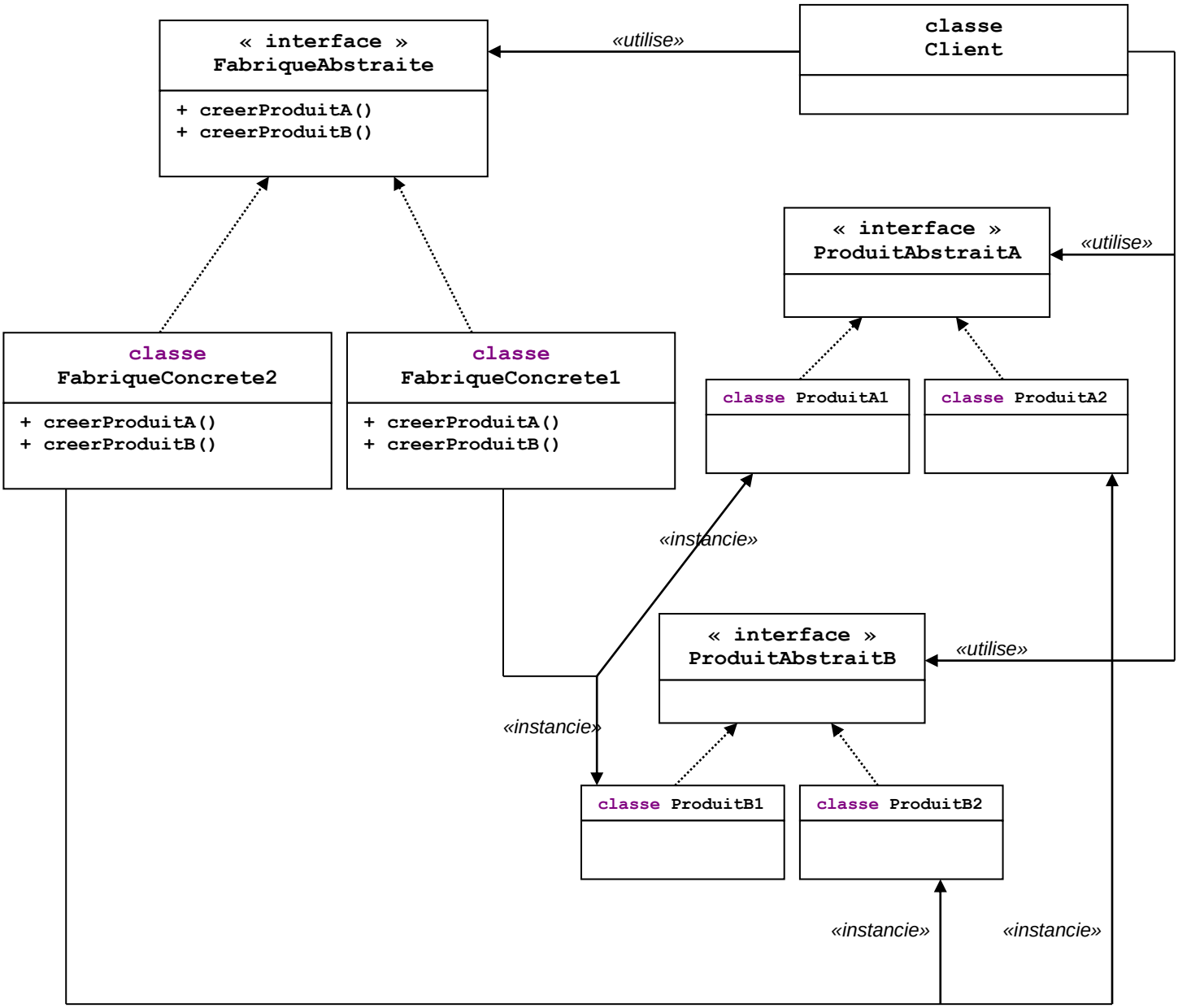
Le schéma précédent nous montre bien que le programme (c-a-d le « client » de la fabrique) qui doit manipuler la couche de d'accès aux données (c-a-d les objets de type DAO) n'aura plus aucun lien avec la classe qui sera physiquement instanciée puisqu'il passera directement par la **`DAOFactory`**.
Les objets DAO seront manipulées dans le programme au travers d'une référence sur une interface (**`AdresseDAO PersonneDAO`**)
Le client a seulement besoin de savoir manipuler les interfaces **`AdresseDAO`** ou **`PersonneDAO`** , et non chaque version particulière obtenue de la fabrique concrète.

```
public class TestFactory {

     public static void main(String[] args) {

   // Création de la fabrique requise !!!
    DAOFactory maFactory = DAOFactory.getDAOFactory(DAOFactory.TypeDAO.JDBC);

    // Création d'un objet de type PersonneDAO
    // manipulé au travers d'une référence sur Personne DAO
    PersonneDAO personneDAO= maFactory.getPersonneDAO();

    // Rechercher toutes les personnes
    // à l'aide de la méthode findAll du DAO
    Collection<Personne> maListe = personneDAO.findAllPersonnes();
    System.out.println(maListe);
}
```

Remarque : **1 seule ligne de code à changer pour changer de support de persistance**

Annexe 1 : Pattern Fabrique Abstraite

La fabrique abstraite (`abstract factory`) est un ***patron de conception*** (design pattern) ***créationnel*** utilisé en génie logiciel.
Ce pattern permet de créer des familles de produits, voici sa définition officielle :
Le pattern fabrique abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leur classes concrètes…



Le pattern fabrique abstraite permet donc à un client d'utiliser une interface abstraite pour créer un ensemble de produits apparentés sans connaître les produits concrets qui sont réellement créés (ni même sans s'en soucier). Ainsi le client est découplé de tous les détails des produits concernés.

Ainsi, on utilise l'`AbstractFactory` lorsque :
→ un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
→ un système repose sur un produit d'une famille de produits
→une famille de produits doivent être utilisés ensemble, pour renforcer cette contrainte
→on veut définir une interface unique à une famille de produits concrets

Correction TD JAVA n°11: Compléments persistance des objets : fichier properties et présentation fabrique de DAO

Le singleton dans sa forme « pure » :

- **1.** un **attribut statique** destiné à recevoir la référence de l'instance unique pour l'ensemble du logiciel,
- **2.** une **méthode** renvoyant la valeur d'instance. Si l'instance est vide, cette méthode crée une nouvelle instance de la classe en la stockant dans l'attribut .
- **3.** bloquer le **constructeur** en le mettant en **private**. Sinon ce n'est pas vraiment un singleton puisqu'il pourra être instancié comme n'importe quel autre objet.

↳ Pattern Singleton appliqué à la classe `ChargeProperty`

```
public class ChargeProperty {
    // attribut statique
    private static ChargeProperty chargeProperty=null;


    // constructeur privé
    private ChargeProperty ()
        {
        }

    // Point accès unique : renvoie une instanciation unique
    // en vérifiant si cette instance existe déjà
    // ou si elle doit vraiment être créée
    public static synchronized ChargeProperty getInstance()
        {
        if (chargeProperty==null)
            {chargeProperty=new ChargeProperty();}
        return chargeProperty;
    }
}
```

↳ Récupération du contenu du fichier => intérêt du singleton ! ! ! ! (à faire 1 seule fois puis consulter `tableProprietes`)

La classe `java.util.Properties` possède la méthode `load`, or cette méthode attend en paramètre un `InputStream`

⇒ Rappel sur le cours E/S en JAVA : la classe `InputStream` est une classe abstraite, il faut donc utilisé la classe `FileInputStream` et réaliser un **flux filtré** ! ! !

```
String fichierResource = "cabMed.properties";

FileInputStream fis
     = new FileInputStream(fichierResource);
```

étant susceptible de lever une `FileNotFoundException`, soyons plus généraliste une `Exception` ! ! ! ⇒ dans un bloc `try…catch`

```
tableProprietes.load(fis);
```

→ `IOException` et `IllegalArgumentException`

```
fis.close();
```

→ `IOException`,

… soit on spécialise avec les catchs
… soit on généralise avec directement bloc **`try…catch (Exception e )`** ! ! ! si on ne connaît pas les exceptions susceptibles d'être levées…

↳ Classe `ChargeProperty` complète

```
import java.io.FileInputStream;
import java.util.Properties;
import com.iut.cabinet.metier.CabinetTechniqueException;

public class ChargeProperty {

    // attribut statique nécessaire au bon fonctionnement du singleton ...
    private static ChargeProperty chargeProperty=null;

    //attribut supplémentaire de la classe
    // pour stocker les propriétés
    private Properties tableProprietes= new Properties();


    // constructeur privé
    private ChargeProperty () throws CabinetTechniqueException
        {
        // Chargement de la table de propriété avec
        // les informations contenues dans le fichier
         String fichierResource = "cabMed.properties";

         FileInputStream fis;
        try {
            fis = new FileInputStream(fichierResource);
            tableProprietes.load(fis);
            fis.close(); // Ne pas oublier de fermer le fichier ....
            System.out.println("fichier chargé :"+fichierResource);
            } catch (Exception e) {
                    throw new CabinetTechniqueException("PB lors de la lecture des données
                                                        dans le fichier "+fichierResource);
            }
        }


    // Point accès unique : renvoie une instanciation unique
    public static synchronized ChargeProperty getInstance()
                                                    throws CabinetTechniqueException
        {
        if (chargeProperty==null)
            {chargeProperty=new ChargeProperty();}
        return chargeProperty;
        }

    // retournera la valeur d'une propriété stockée dans tableProprietes
    // dont le nom est passé en paramètre
    public String getPropriete(String nomPropriete) throws CabinetTechniqueException
    {
    String resultat = tableProprietes.getProperty(nomPropriete);
    if (resultat==null)
        throw new CabinetTechniqueException("Propriete "+nomPropriete +
                                " est inexistante dans la table des propriétés.");
    else return resultat;
    }

} // fin classe ChargeProperty
```

Remarque ⇒ Révision `Exception` ! ! ! !

```
    public String getPropriete(String nomPropriete) throws CabinetTechniqueException
```

Si la propriété choisie n'est pas présente dans « `tableProprietes` », `getPropriete` renvoie null...

Pour ce qui est du contenu du fichier, au lieu de renvoyer `null` si la propriété est vide, **on demande de renvoyer une exception technique `CabinetTechniqueException`, car si les propriétés ne sont pas correctement récupérées, la connexion ne peut pas exister par la suite…. Et c'est un blocage MAJEUR de l'application.**

Test

```
public static void main(String args[])
    {

        try {

        // Récupération de l'URL Correcte …
        System.out.println("url recuperee : " +
                   ChargeProperty.getInstance().getPropriete("url")); //OK


        // Récupération de l'URL INCORRECTE …
        // ... car attention aux majuscules et minuscules
        System.out.println("URL recuperee : " +
                      ChargeProperty.getInstance().getPropriete("URL")); //PB

        }

        catch (CabinetTechniqueException e) {
            System.out.println("Pb avec la recherche d'une propriété...");
            e.printStackTrace();
        }
    }
```

Remarques : Modifications à effectuer pour le TP dans la classe `SimpleConnection` :

→ **lors du chargement du driver :**

```
Class.forName(driver);
```

devient

```
Class.forName(ChargeProperty.getInstance().getPropriete("driver"));
```

→ **lors de la demande de la connection :**

```
connection = DriverManager.getConnection(url);
```

devient

```
connection = DriverManager.getConnection(ChargeProperty.getInstance().getPropriete("url"));
```

Et pour finir…Présentation du pattern fabrique abstraite …

↳ La **fabrique abstraite** encapsule un groupe de fabriques ayant une thématique commune.

Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique.
Le client ne se préoccupe pas de savoir laquelle de ces fabriques a donné un objet concret, car il n'utilise que les interfaces génériques des objets produits.

Ce patron de conception sépare les détails d'implémentation d'un ensemble d'objets de leur usage générique
→ Fournit une interface pour créer des familles de d'objets sans spécifier leur classe d'implémentation
→ Fournit un moyen de se passer de l'opérateur new

↳ Une **fabrique** est un endroit du code où sont construits des objets. Le but de ce patron de conception est d'isoler la création des objets de leur utilisation. On peut ainsi ajouter de nouveaux objets dérivés sans modifier le code qui utilise l'objet de base.
Avec ce patron de conception, on peut interchanger des classes concrètes sans changer le code qui les utilise, même à l'exécution. Toutefois, ce patron de conception exige un travail supplémentaire lors du développement initial, et apporte une certaine complexité qui n'est pas forcément souhaitable.

Relations avec les autres patterns
→ Factory Method : l'Abstract Factory peut utiliser le pattern Factory Method.
→ Singleton : les classes concrètes de l'Abstract Factory sont en général implémentées par des Singletons.

Méthaphore : Un MVC est comme une entreprise qui fait des logiciels.
On a des ***programmeurs*** (Models), des ***représentants*** (Vues) et un ***boss*** (Controleurs).
Contrôleur : Le ***boss*** : il y en a toujours un, est celui qui fait rien…. Il ***fait juste prendre des décisions***. Il décide qui fait quoi et quel représentant est choisi. C'est lui qui peut congédier son personnel. *Il n' y a jamais 2 boss* en même temps, mais il peut avoir des supérieurs ou des adjoints qui sont dans d'autre domaine (d'autre use case). C'est lui qui ***envoie*** *les demandes de mises à jour* et qui ***reçoit*** *les questions des représentants* qui sera redirigé au bon programmeur pour faire le calcul.
→ **le contrôleur :** reçoit les événements de l'interface utilisateur et les traduit :
en changement dans la vue s'ils agissent sur le côté visuel ou
en changement dans le modèle s'ils agissent sur le contenu du modèle