Master Informatique

Génie Logiciel 2

178UD01

Cours n°4

Etude de cas (3/3) + *Framework* de persistance

Plan du cours

- Conception du rafraîchissement du total vente
- Conception du relayage par les services locaux
- Introduction à un *framework* de persistance

Plan du cours

- Conception du rafraîchissement du total vente
- Conception du relayage par les services locaux
- Introduction à un *framework* de persistance

Rafraîchissement du total de la vente (1/2)

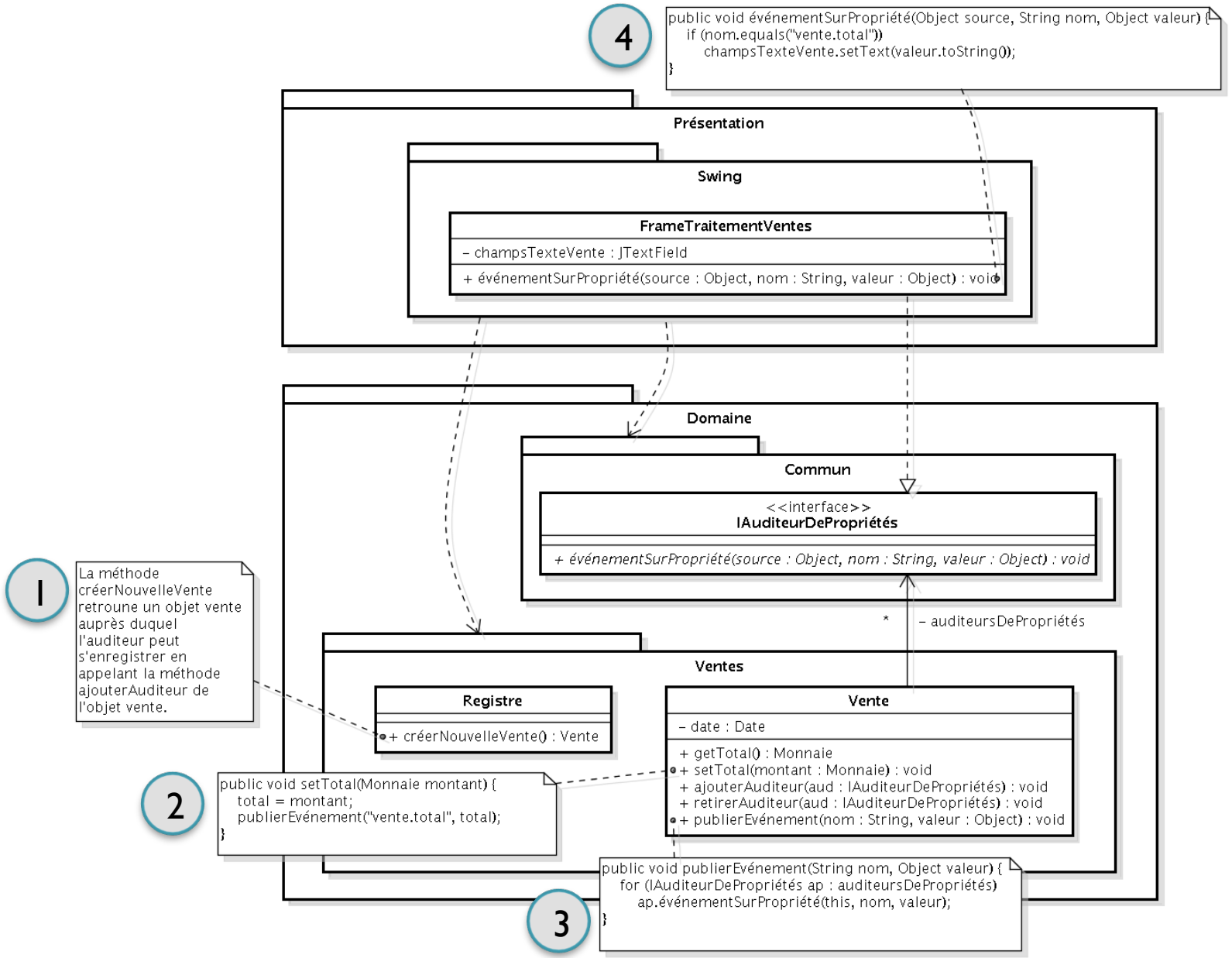
- L'affichage du total de la vente (couche présentation) doit être mise à jour après chaque article saisi (couche domaine)

- C'est le problème de la communication ascendante : des informations d'une couche inférieure doivent être remontées vers la couche supérieure

- Appliquer le pattern GoF Observateur :
  - Définition d'une interface *IAuditeur* connue à la fois par les auditeurs et le diffuseur
  - Implémentation de *IAuditeur* par les auditeurs (couche supérieure)
  - Liaison dynamique entre les auditeurs et le diffuseur (par passage de paramètre ou valeur de retour d'une méthode)
  - A chaque modification d'une propriété du diffuseur, diffusion du changement auprès des auditeurs (couche inférieure)

- En résumé, « un Observateur permet un couplage lâche avec le diffuseur. Ce couplage lâche peut être en plus dynamique. »

Rafraîchissement du total de la vente (2/2)



Plan du cours

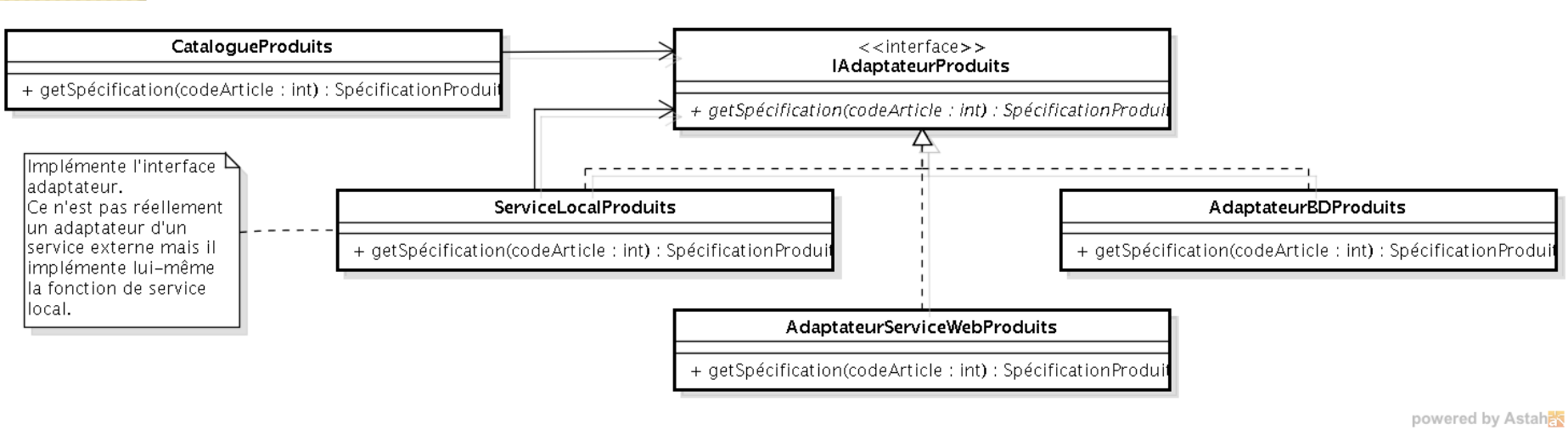
- ✓ Conception du rafraîchissement du total vente
- ➢ Conception du relayage par les services locaux
- • Introduction à un *framework* de persistance

Relayage par les services locaux (1/9)

- Problème/solution n° 1 :
  - Défaillance d'accès au service externe des spécifications des produits
  - La solution doit aussi améliorer les performances des temps de recherche
  - Comme il peut exister plusieurs types de service externe (différents SGBDR) et différents moyens d'y accéder (réseau TCP/IP, service web…), il faut mettre en place une interface *IAdaptateurProduits* unique implémentée par des adaptateurs
  - Le bon adaptateur sera choisi et fourni par la fabrique de services *FabricationServices* (cf. cours 3)
  - Pour pallier à une défaillance d'accès au service externe, il faut mettre en place un service local (cache des produits les plus courants)
  - ➢ Solution : application des patterns Adaptateur et Fabrication + réplication partielle en local

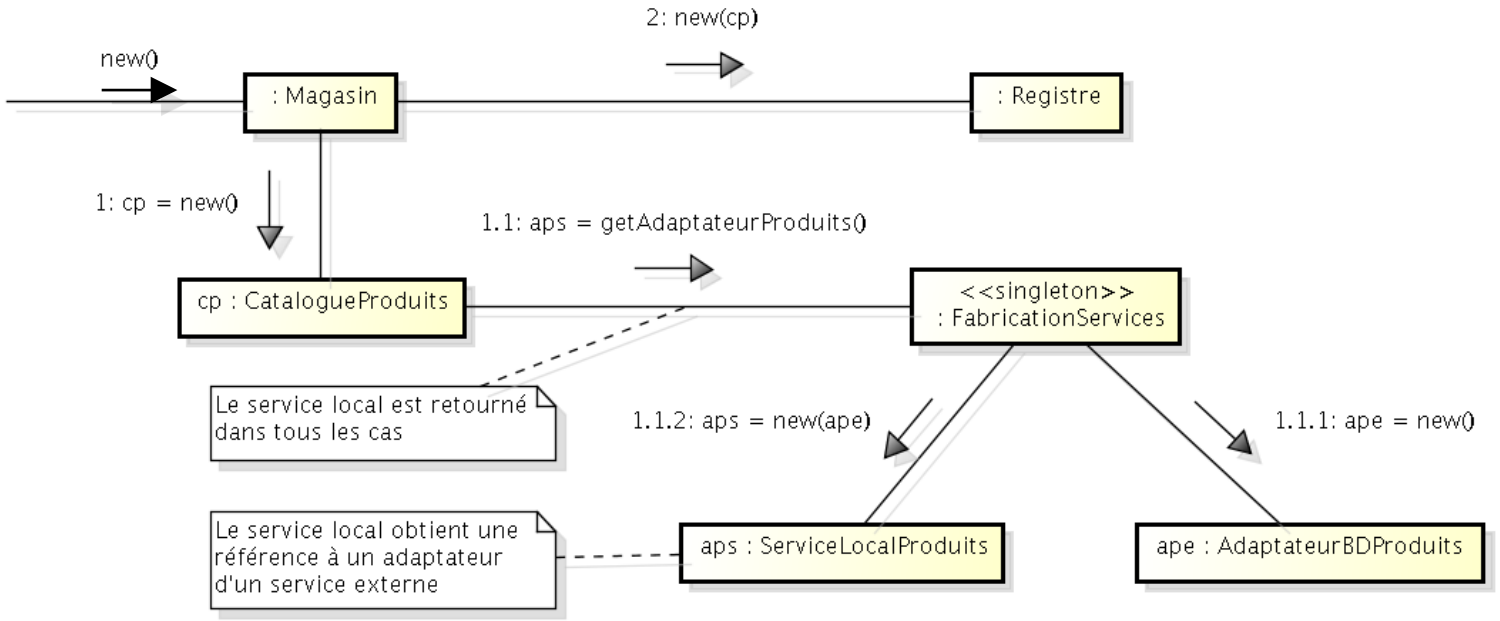
Relayage par les services locaux (2/9)

- Problème/solution n°1 (suite) :
  - Fonctionnement :
    - La *FabricationServices* retourne toujours un adaptateur à un service local sur les produits
    - L'adaptateur au service local fournit lui-même le service (faux adaptateur)
    - L'adaptateur au service local délègue le travail à un second « vrai » adaptateur lorsque celui est accessible et lorsqu'il ne trouve pas de réponse dans son cache local
  - Deux niveaux de cache :
    - L'adaptateur au service local possède un cache local sur disque
    - L'objet *CatalogueProduits* possède un cache local en mémoire extrait du service externe



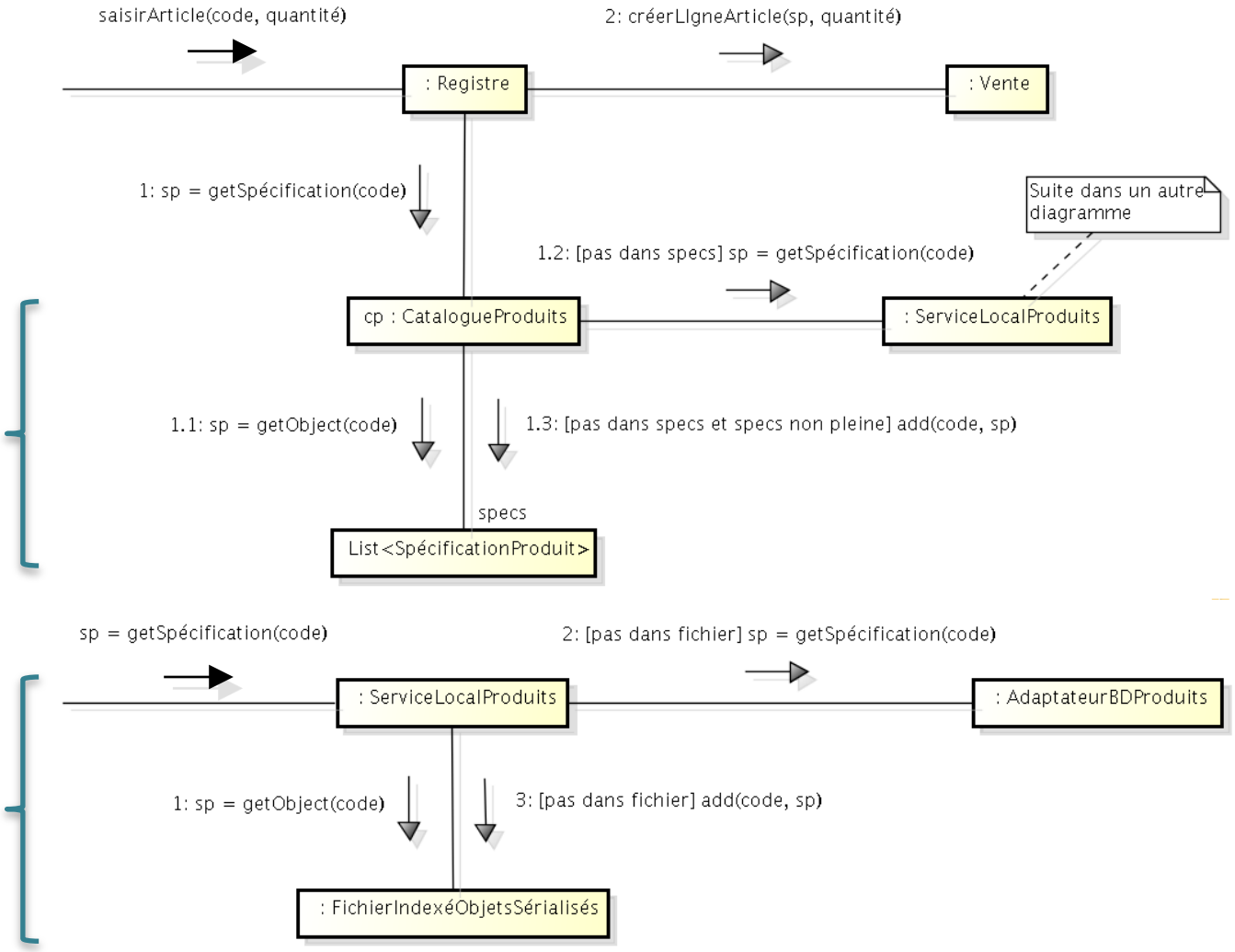
Relayage par les services locaux (3/9)

- Problème/solution n° 1 (suite) :
  - Initialisation :
    - Création de l'objet *CatalogueProduits*
    - Création des adaptateurs *via* la fabrique
      - Création de l'adaptateur du service externe
      - Création de l'adaptateur du service local (connaît l'adaptateur du service externe)
      - Retourner l'adaptateur du service local



Relayage par les services locaux (4/9)

- Problème/solution n°1 (suite) :
  - Collaboration :

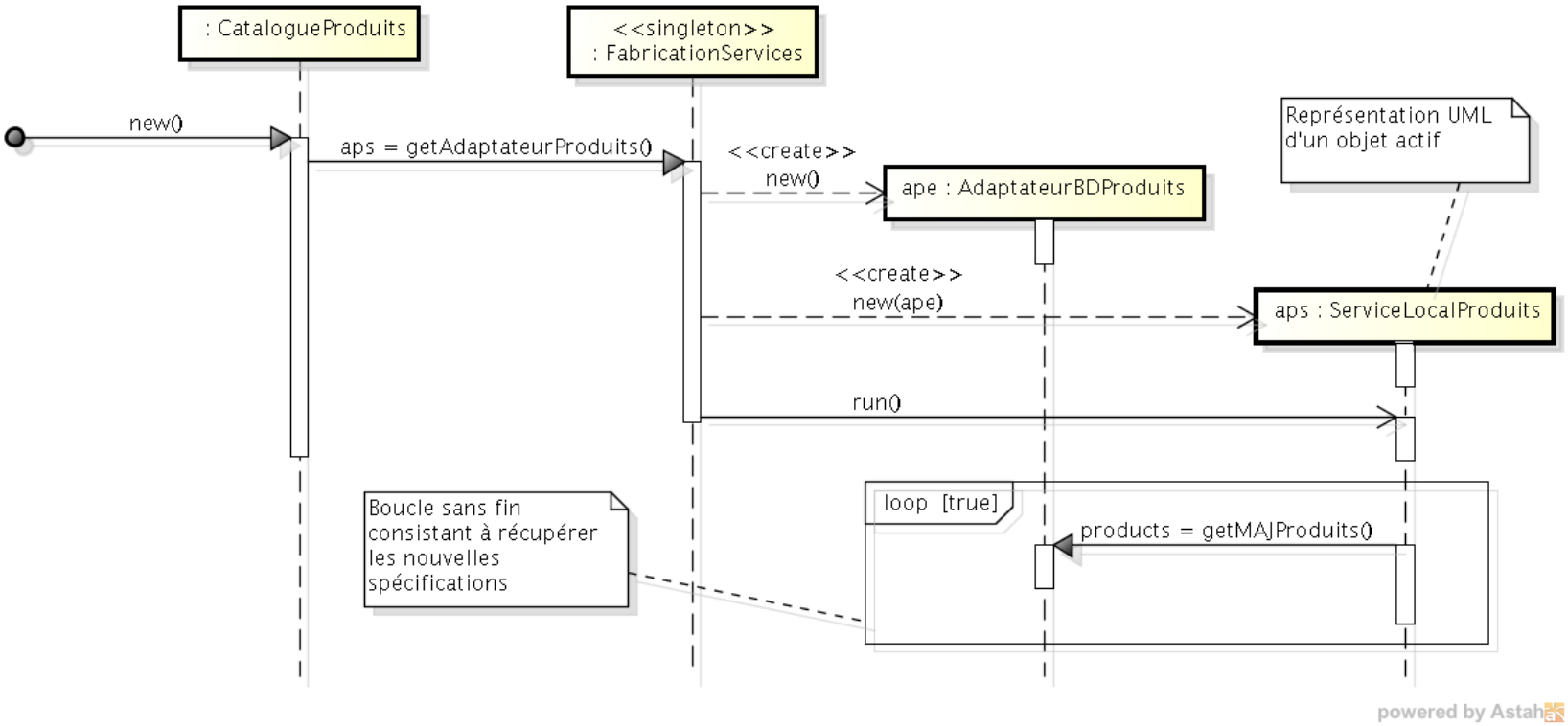


Premier niveau de cache (en mémoire)

Second niveau de cache (sur disque)

Relayage par les services locaux (5/9)

- Problème/solution n°1 (suite et fin) :
  - Péremption du cache :
    - Si, par exemple, le prix des produits changent rapidement, les données en cache deviennent vite invalides
    - Une solution consiste à mettre ne place une mise-à-jour automatique en rendant l'adaptateur du service local actif (thread ou *runnable*)
    - Tous les x minutes, le thread va requêter l'adaptateur du service externe pour obtenir les nouvelles spécifications

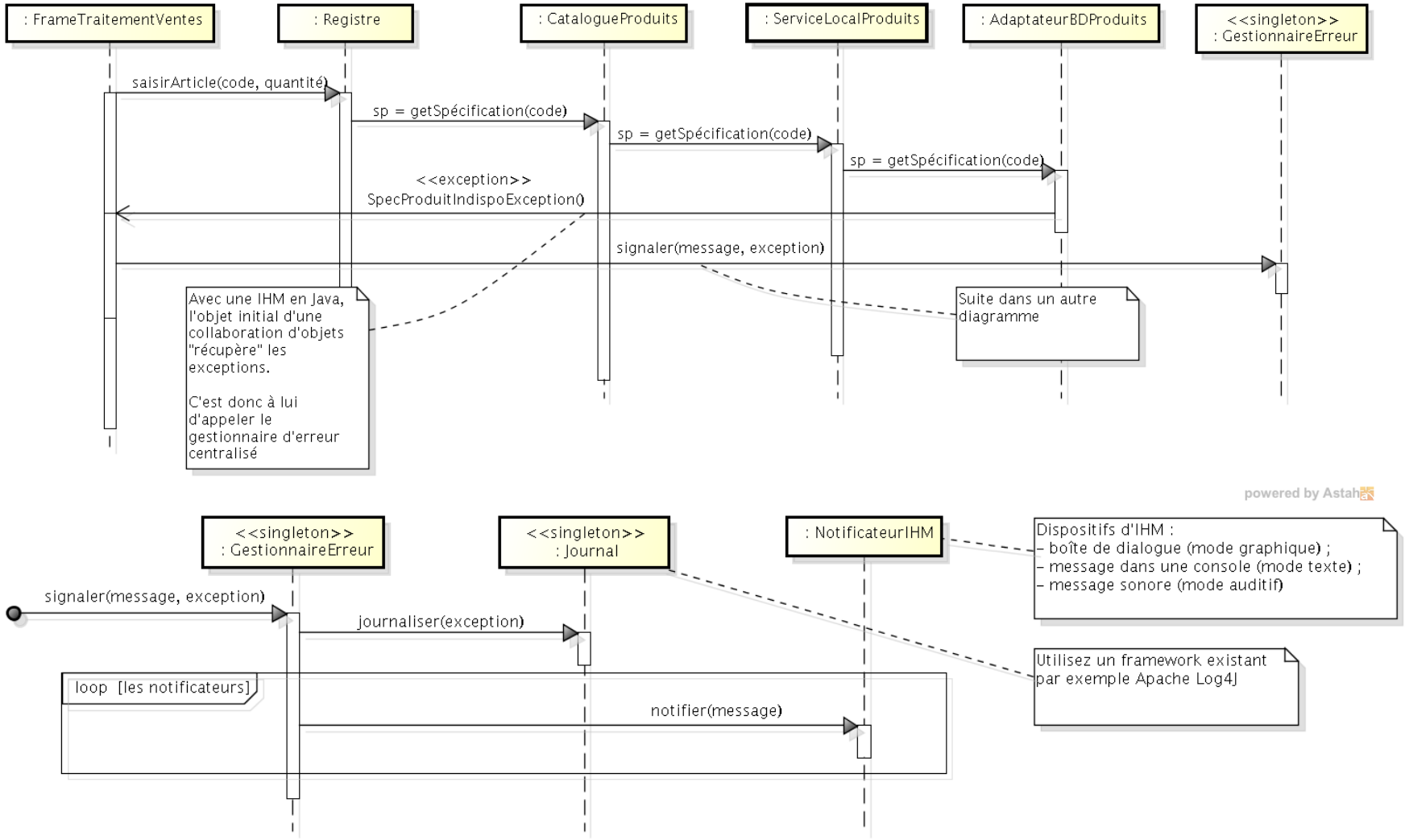


Relayage par les services locaux (6/9)

- Problème/solution n°2 :
  - Comment gérer, par exemple, une recherche infructueuse d'une *SpécificationProduit* à partir d'un *codeArticle* ?
  - Traitement des erreurs :
    - Mémorisation des erreurs :
      - Centraliser la journalisation des erreurs au niveau local
      - Accessible depuis n'importe quel endroit de l'application
      - Dans le cas d'une application distribuée : collaboration des journaux locaux avec un système de journal central
    - Alerter les utilisateurs des erreurs :
      - Le système de gestion des erreurs doit être indépendant de l'IHM
      - Accessible depuis n'importe quel endroit de l'application
      - Ce système de gestion notifie un ou plusieurs dispositifs d'IHM
    - Une fabrique lit les paramètres et crée les objets d'IHM à notifier
  - ➢ Utiliser une journalisation singleton + un gestionnaire d'erreurs singleton

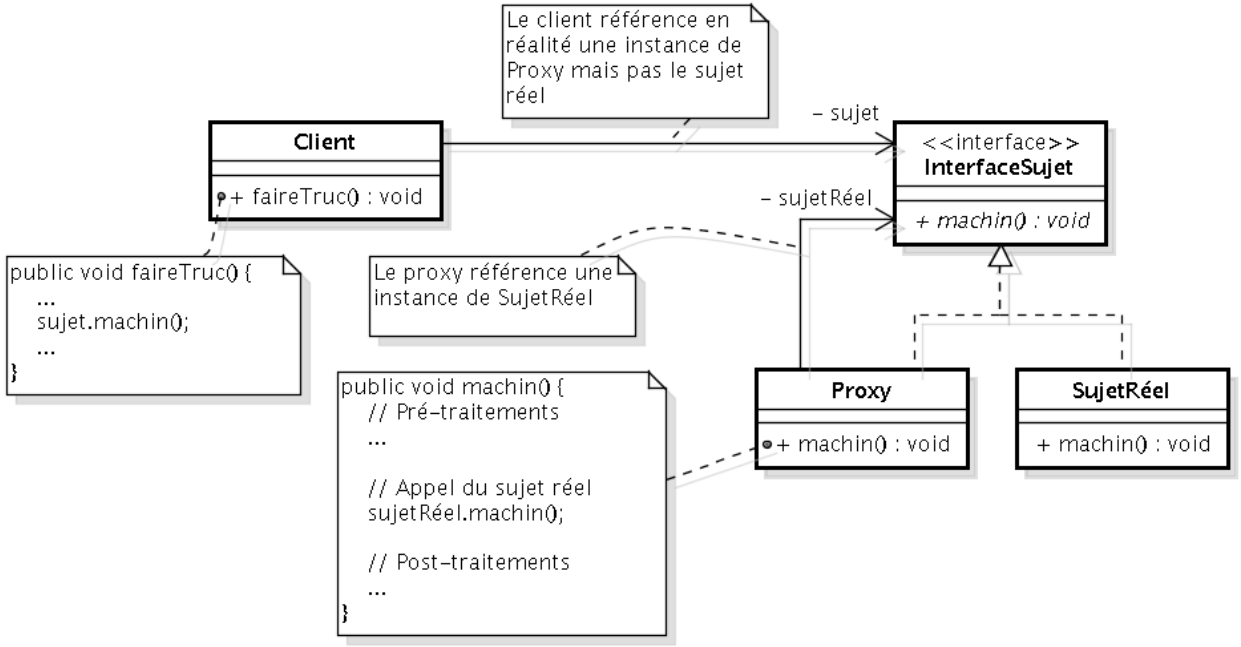
Relayage par les services locaux (7/9)

- Problème/solution n°2 (suite) :
  - Exemple de fonctionnement :



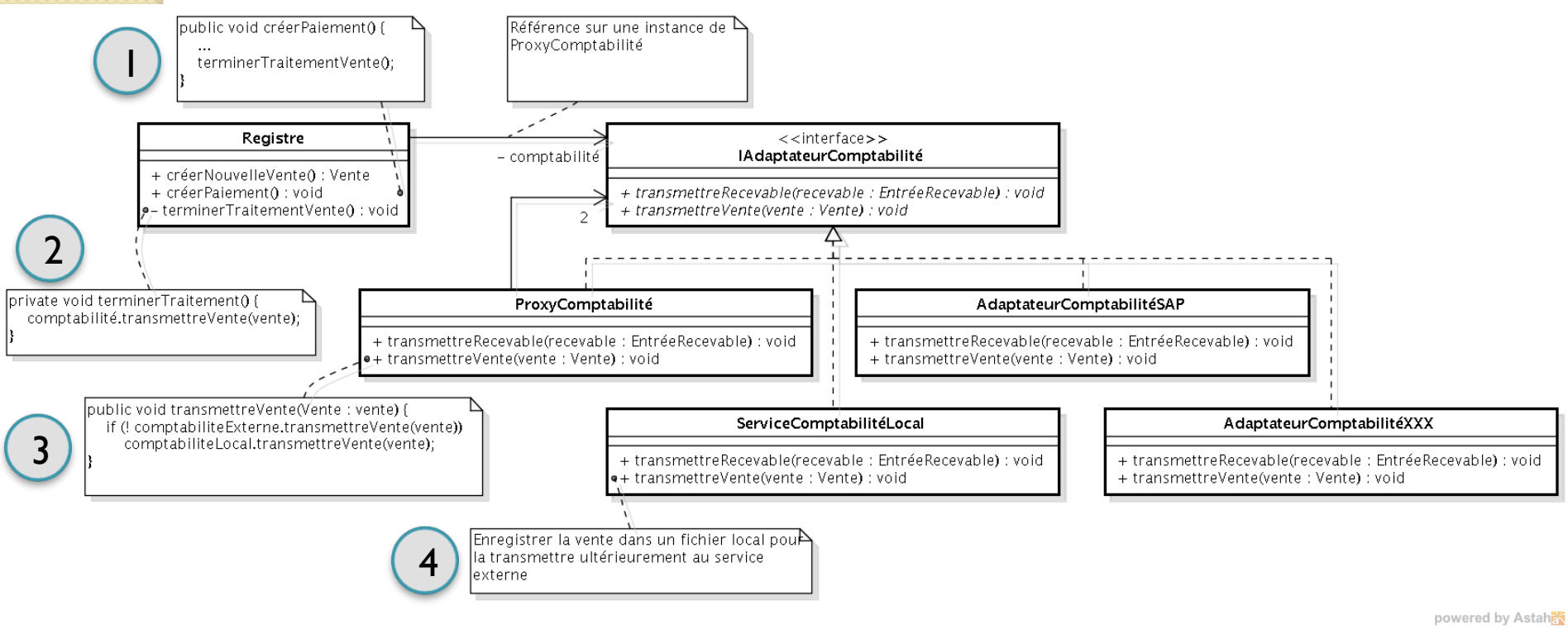
Relayage par les services locaux (8/9)

- Problème/solution n°3 :
  - Défaillance d'accès au service externe de comptabilité
  - Même démarche que pour l'accès au service des spécifications produits :
    - On utilisera un service local pour pallier à une défaillance du service externe
    - Les différents types de service externe de comptabilité seront uniformisés par une interface unique implémentée par des adaptateurs
  - Mais on améliore la solution en séparant :
    - L'instanciation du service tiers : pris en charge par le service local
    - Du contrôle d'accès au service tiers : pris en charge par une procuration (proxy)
  - ➢ Application du pattern GoF Adapatateur + pattern GoF Proxy



Relayage par les services locaux (9/9)

- Problème/solution n°3 (suite) :
  - La procuration prend la place du service externe
  - La procuration essaie d'abord le service externe
  - Si elle échoue, elle utilise le service local



Plan du cours

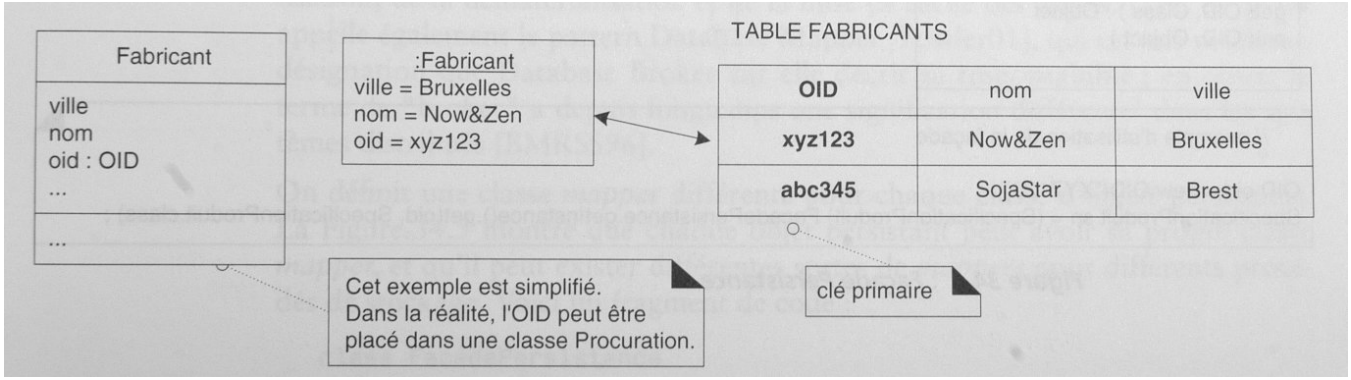
- ✓ Conception du rafraîchissement du total vente
- ✓ Conception du relayage par les services locaux
- ➢ Introduction à un *framework* de persistance

Framework de persistance (1/13)

- Problème/solution :
  - Problème
    - Les données des SpécificationsProduit sont mémorisées dans une base de données relationnelle.
    - Il faut un service de correspondance relationnel-objet qui fonctionne en lecture et en écriture (ORM : *Object-Relation Mapping)*.
    - Il faut limiter les accès à la base externe.
  - Solution
    - Il s'agit d'un problème récurrent : la solution doit donc être générique pour être réutilisée. En particulier, il devra pouvoir prendre en charge différents formats de stockage (SGBDR, fichiers XML, fichiers plats). Un cache et des transactions limiteront l'accès aux données externes.
    - Une solution générique est un framework de persistance.
    - Il faut utiliser un composant commercial ou libre mais pour le cours on va proposer la conception d'un framework.
    - Ce framework sera personnalisé pour le projet NextGen. Il sera intégré dans la couche Services techniques comme un sous-système.

Framework de persistance (2/13)

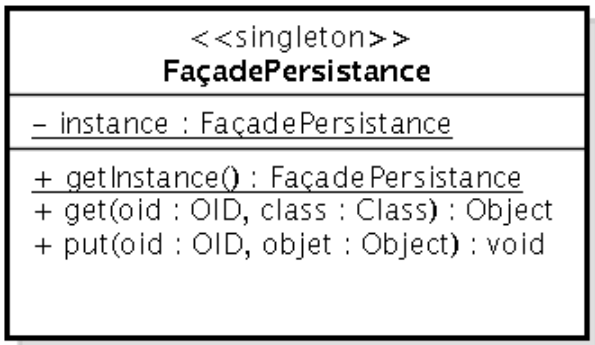
- Exigences techniques du framework de persistance :
  - Il doit exister une ***correspondance*** entre une classe et son stockage (table de la base de données). Le pattern *Représentation des objets en table* est le suivant :
    Il y a une table par classe et les attributs sont les colonnes de la table.
  - Pour faciliter la correspondance et éviter les doublons, les objets et les enregistrements auront un ***identifiant*** unique (un OID : *Object IDentifier*).



  - La matérialisation et la dématérialisation est effectuée par un ***mapper*** de base de données.
  - La ***matérialisation*** consiste à transformer un enregistrement d'une table en objet. La ***dématérialisation*** est l'opération inverse.
  - On utilisera un ***cache*** local pour des raisons de performance.
  - Les ***transactions*** seront annulées (*rollback*) ou validées (*commit*).
  - ***L'état*** des objets devra être connu. Par exemple, seul les objets modifiés devront être enregistrés après validation de la transaction.
  - Les instances associées seront ***matérialisées à la demande*** (c'est-à-dire si nécessaire). On pourra utiliser une variante du pattern GoF Procuration.

Framework de persistance (3/13)

- Conception du framework :
  - Définir une Façade d'accès et de masquage du sous-système de persistance. Cette Façade sera également un Singleton pour assurer son unicité et son accès global. La classe FaçadePersistance délègue le travail.



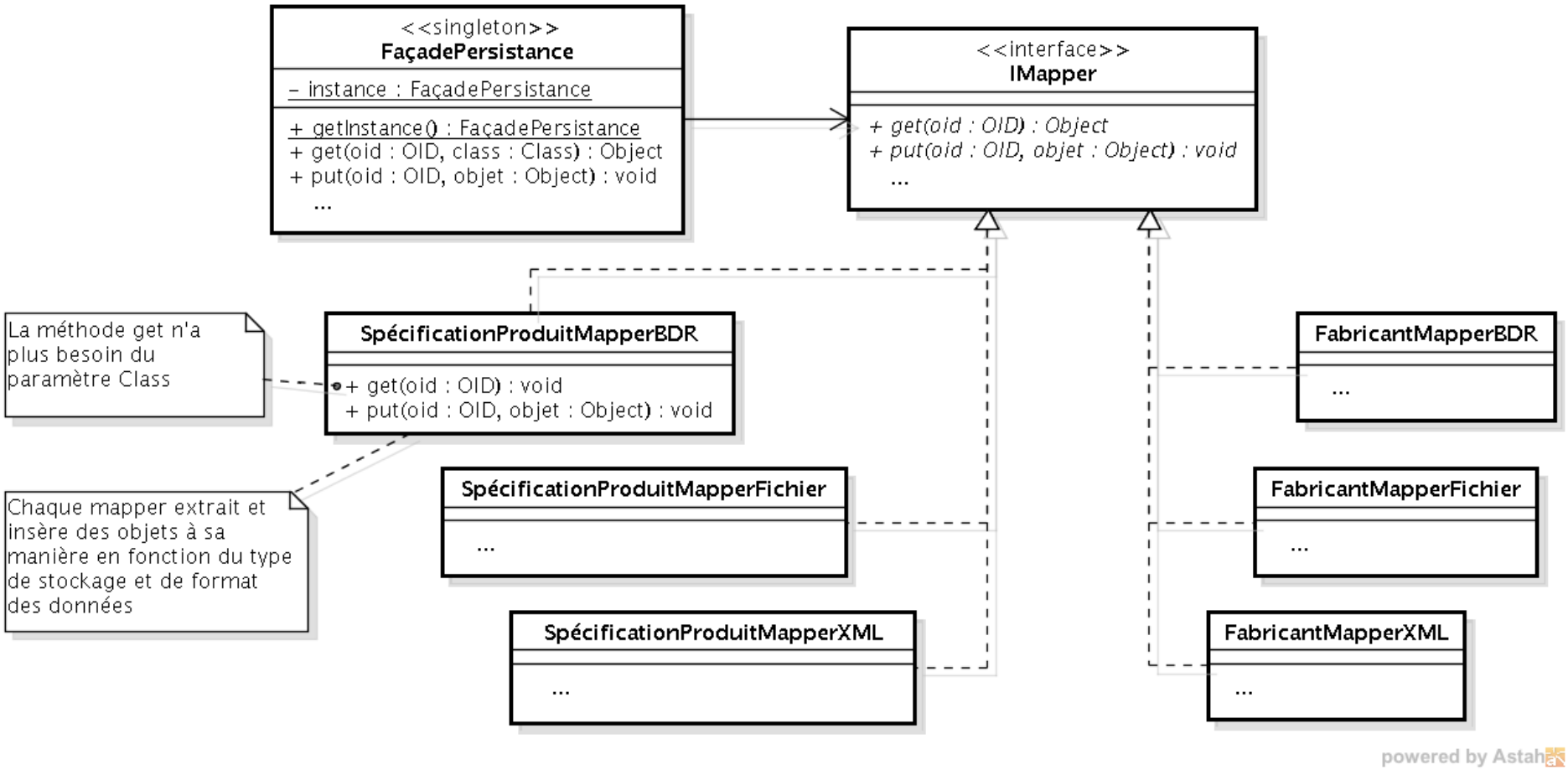

```
// Exemple utilisation de la façade

OID oid = new OID("XYZ123");
SpecificationProduit sp = (SpecificationProduit) FaçadePersistance.getInstance().get(oid, Specification.class);
```

Framework de persistance (4/13)

- Conception du framework (suite) :
  - La responsabilité de la matérialisation et de la dématérialisation ne doit pas être affectée aux classes du domaine car cela entraîne une faible cohésion de ces classes et un fort couplage entre le domaine et la persistance.
  - La solution est d'utiliser le pattern *Database Mapper* : définir une classe mapper pour chaque classe d'objets persistants.
  - Les classes mappers possèdent tous une méthode get quasiment identique :

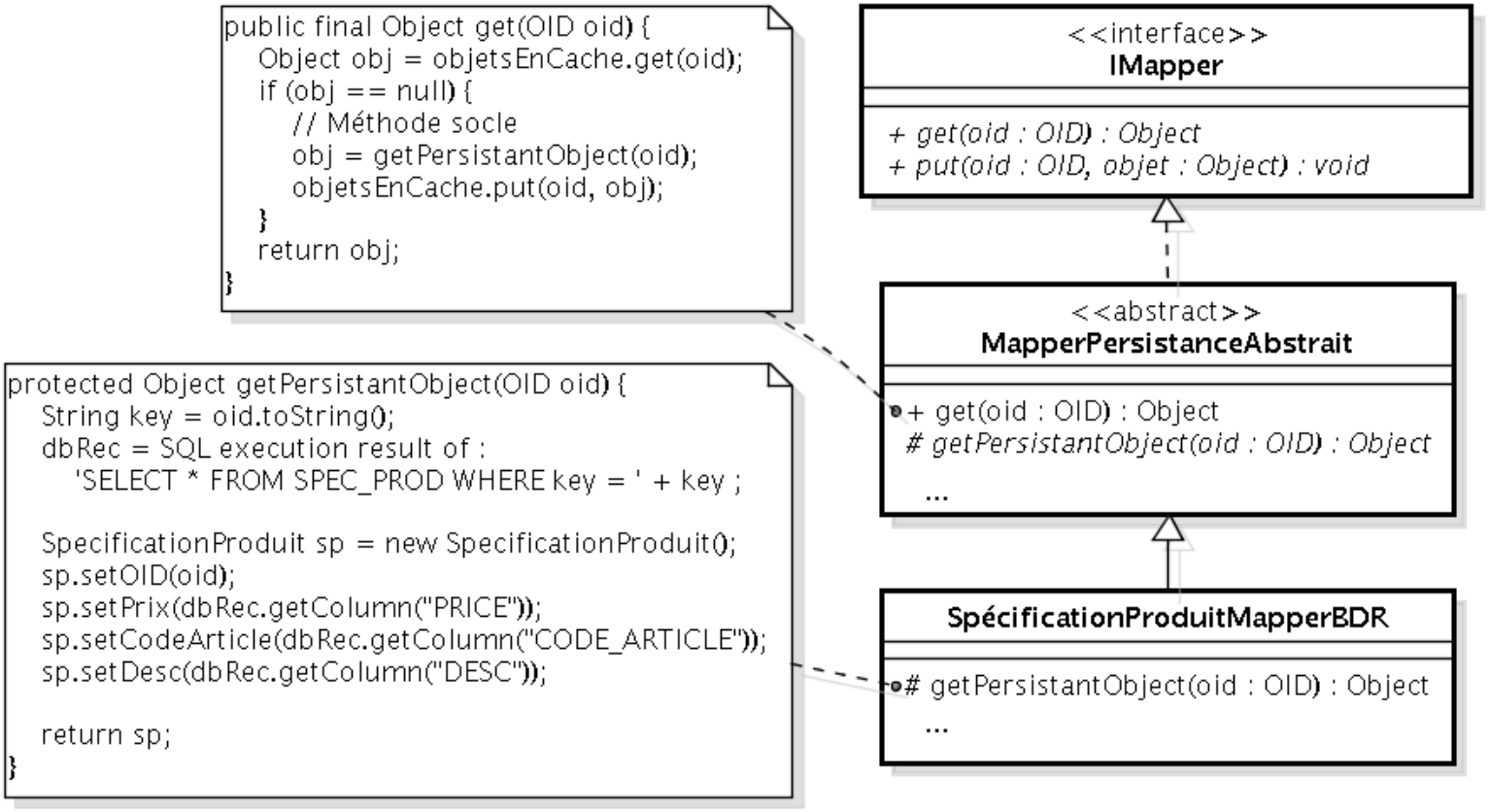
```
Si (objet est en cache)
    le retourner
Sinon
    Créer l'objet à partir de sa représentation dans le stockage (partie variable)
    Mettre l'objet dans le cache
    Le retourner
```

Framework de persistance (5/13)



Framework de persistance (6/13)

- Conception du framework (suite) :
  - Pour **éviter la répétition de code**, utilisons le pattern GoF Patron de méthode : créez une super-classe contenant le code avec des parties variables et invariables et redéfinissez les parties variables dans des sous-classes.



Framework de persistance (7/13)

- Conception du framework (suite) :
  - On peut configurer la FaçadePersistance avec un ensemble d'objets IMapper par l'entremise d'un objet FabricationMappers.
  - FabricationMappers est une fabrique singleton en charge de créer les objets mappers en fonction de la configuration du système.

```
class FabricationMappers {
    …
    public Map getTousMappers {…}
    …
}

class FaçadePersistance {
    …
    private Map mappers = FabricationMappers.getInstance().getTousMappers();
    …
}
```

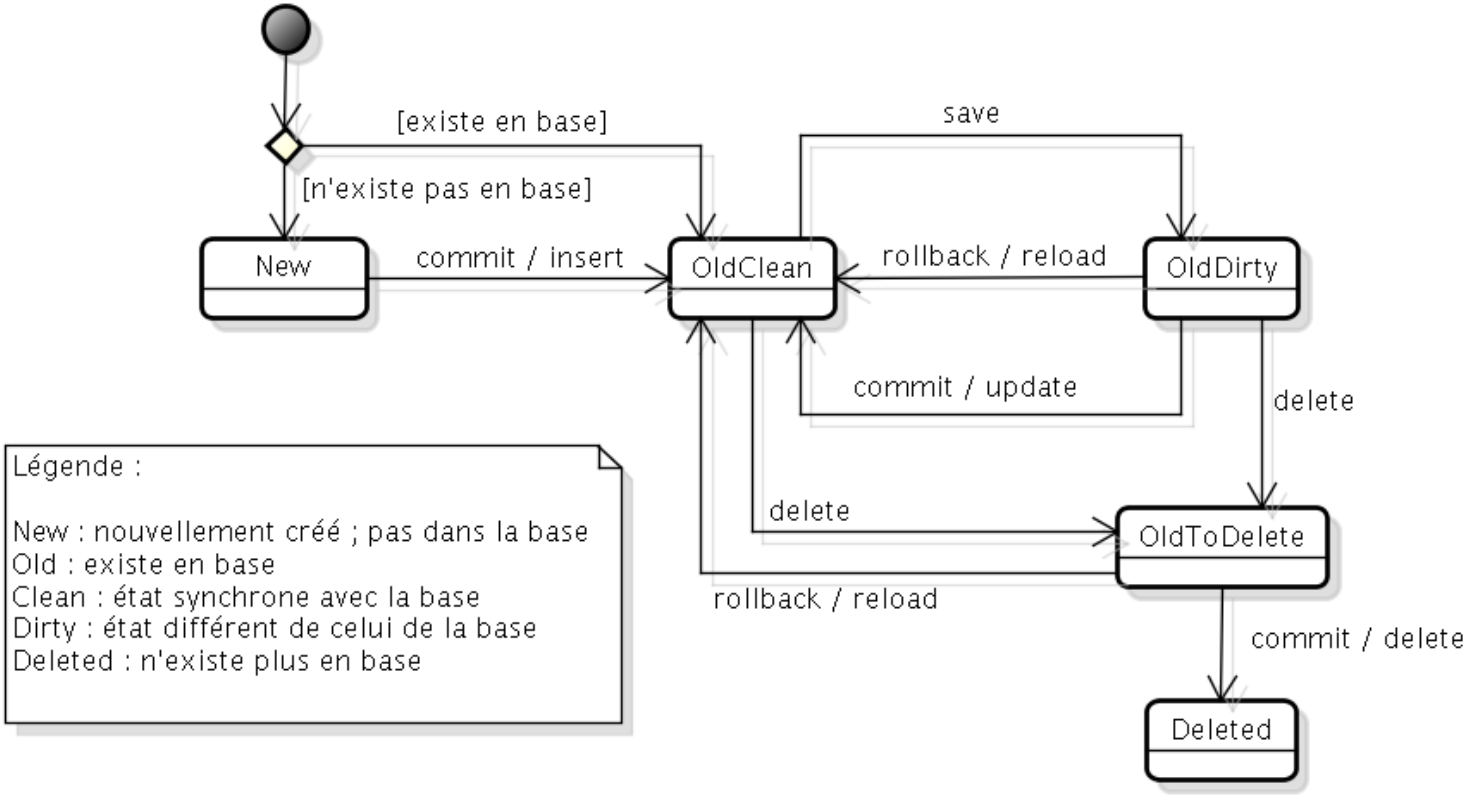
Framework de persistance (8/13)

- Conception du framework (suite) :
  - On peut regrouper et masquer les instructions SQL dans une classe unique.
  - Cela facilite la maintenance et le développement.
  - Les mappers collaborent avec elle pour obtenir les enregistrements de la base de données.

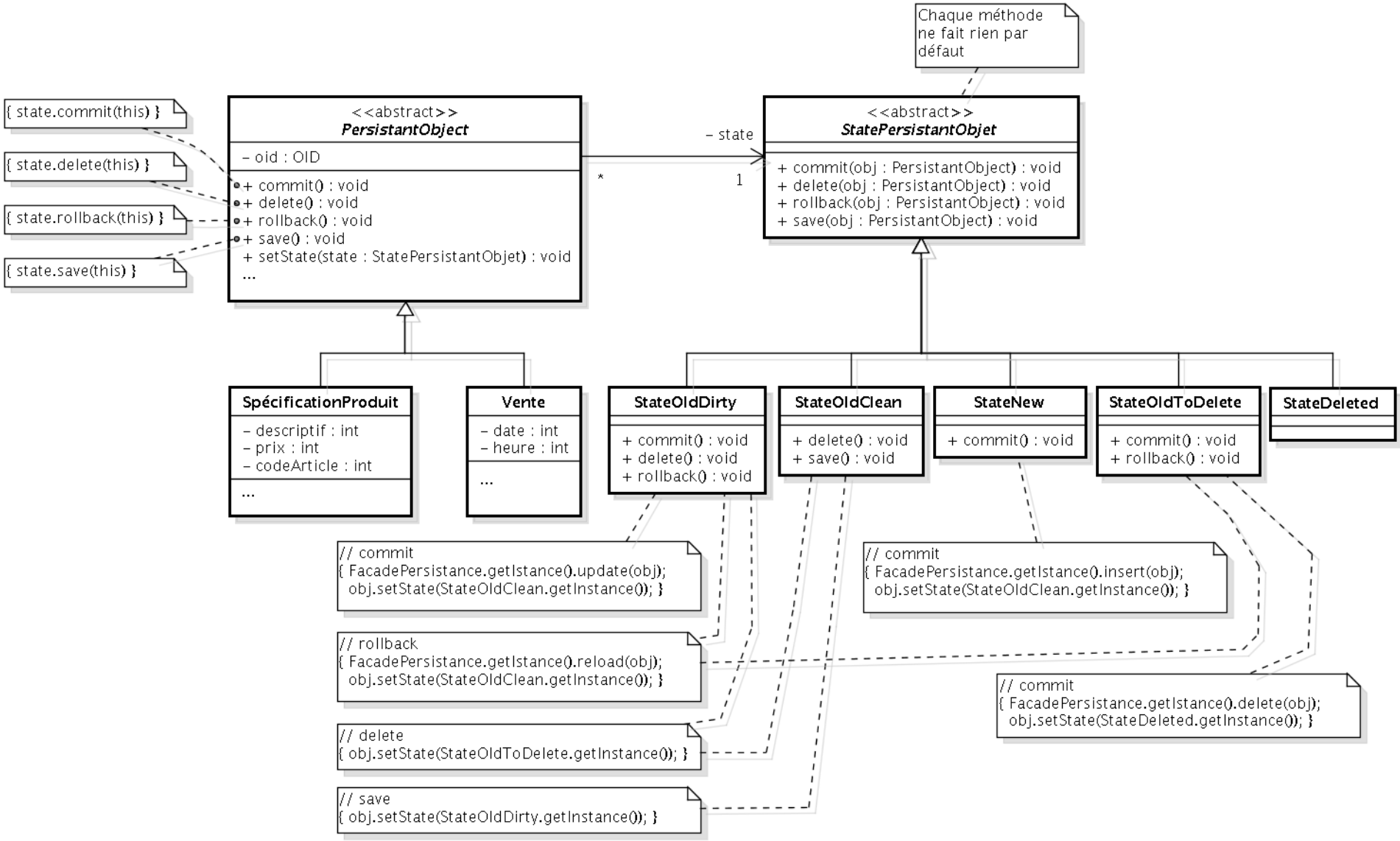
```
class OperationBDR {
    public ResultSet getDonneesSpecificationProduit(OID oid) {…}
    public ResultSet getDonneesVente(OID oid) {…}
    …
}

class SpecificationProduitMapperBDR extends MapperPersistanceAbstrait {
    …
    protected Object getPersistantObject(OID oid) {
        ResultSet rs = OperationBDR.getInstance().getDonneesSpecificationProduit(oid);
        SpecificationProduit sp = new SpecificationProduit();
        …
        return sp;
    }
}
```

Framework de persistance (9/13)

- Conception du framework (suite) :
  - Les objets persistants passent par différents états et ces états donnent des comportements différents aux commandes d'une transaction.
  - On va utiliser le pattern GoF Etat qui permet de représenter explicitement les états.

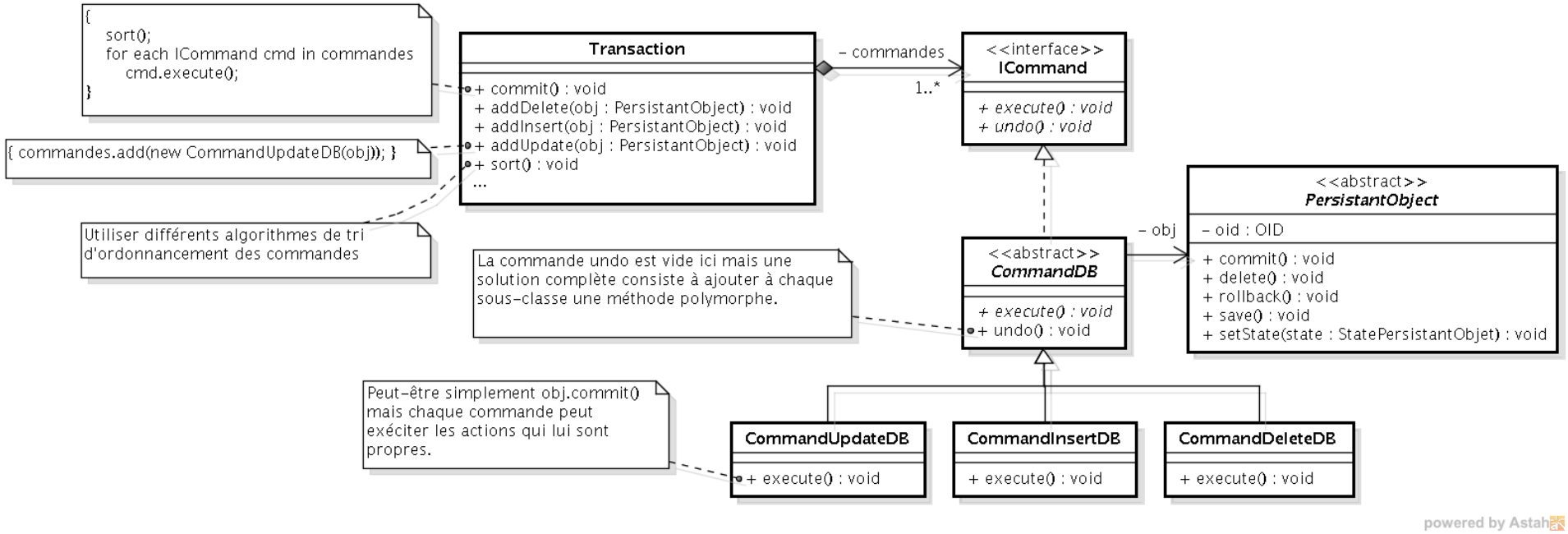


Framework de persistance (10/13)



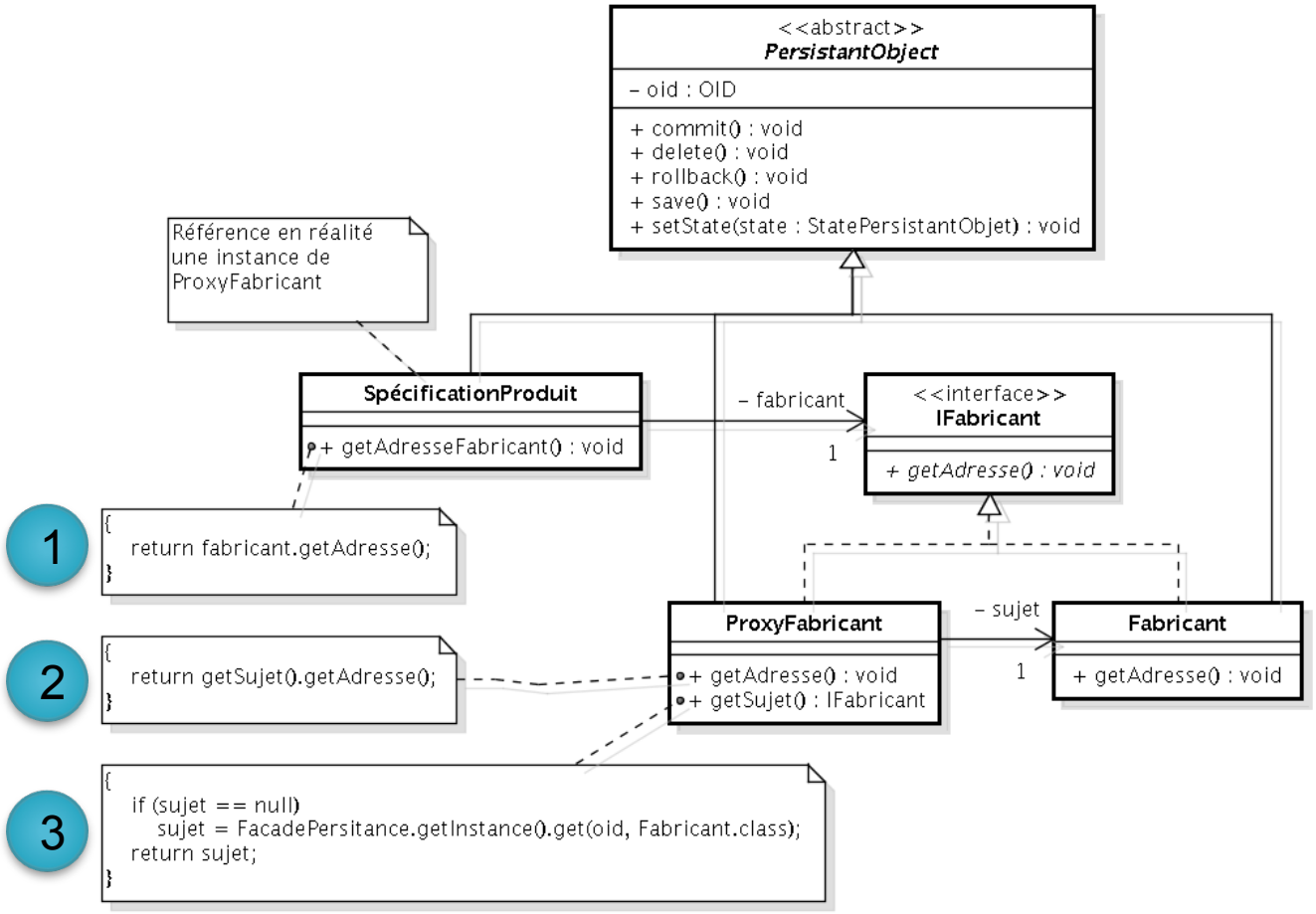
Framework de persistance (11/13)

- Conception du framework (suite) :
  - Une transaction est une unité de travail : un ensemble de tâches dont toutes doivent être accomplies ou dont aucune ne doit l'être.
  - Le pattern GoF Commande permet de représenter explicitement des tâches.
  - Une classe Transaction connaîtra la liste des tâches de la transaction en cours sur laquelle des tris d'ordonnancement pourront être effectués.



Framework de persistance (12/13)

- Conception du framework (suite) :
  - La matérialisation des objets associés à un objet matérialisé n'est pas systématique.
  - Par exemple, par défaut, la matérialisation d'une SpécificationsProduit n'a pas besoin de matérialiser son lien avec une instance de Fabricant.
  - Il faut utiliser le pattern Procuration.



Framework de persistance (13/13)

- Conception du framework (suite) :
  - Exemple de code : la procuration virtuelle est créée par le mapper de l'objet matérialisé

```
class SpecificationProduitMapperBDR extends MapperPersistanceAbstrait {
    …
    protected Object getPersistantObject(OID oid) {
        ResultSet rs = OperationsBDR.getInstance().getDonneesSpecificationProduit(oid);
        SpecificationProduit sp = new SpecificationProduit();
        …
        String cleEtrangereFabricant = rs.getString("OID_FAB") ;
        OID fabOID = new OID(cleEtrangereFabricant) ;
        sp.setFabricant(new ProxyFabricant(fabOID));
        …
        return sp;
    }
    …
}
```