Master Informatique

Génie Logiciel 2

178UD01

Cours n°3

Etude de cas (2/3) + Six patterns GRASP

Plan du cours

- Deux patterns GRASP
- Conception d'opérations du cas principal
- Quatre patterns GRASP
- Conception de l'accès aux services tiers
- Conception de la logique de tarification

Plan du cours

- Deux patterns GRASP
- Conception d'opérations du cas principal
- Quatre patterns GRASP
- Conception de l'accès aux services tiers
- Conception de la logique de tarification

Deux patterns GRASP

- *Expert en information (ou **Expert**)* : affecter une responsabilité à la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité.
- ***Créateur*** : affecter à la classe *A* la responsabilité de créer une instance de la classe *B* si une ou plusieurs des conditions suivantes est vraie :
  - *A* agrège *B*
  - *A* contient *B*
  - *A* enregistre des instances d'objets *B*
  - *A* utilise étroitement des objets *B*
  - *A* a les données d'initialisation qui seront passées aux objets *B* lors de leur construction (*A* est un Expert de la création de *B*).

Plan du cours

- ✓ Deux patterns GRASP
- ➢ Conception d'opérations du cas principal
- Quatre patterns GRASP
- Conception de l'accès aux services tiers
- Conception de la logique de tarification

Opération *créerNouvelleVente* (1/2)

- Contexte : déclenchement par le caissier à l'arrivée d'un nouveau client

Contrat d'opération C01 :
- Opération : *créerNouvelleVente()*
- Références croisées : cas d'utilisation *Traiter une vente*
- Pré-conditions : aucune
- Post-conditions :
  - Une instance *v* de *Vente* a été créée (création d'instance).
  - *v* a été associée au *registre* (formation d'association).
  - Des attributs de *v* ont été initialisés.

Un contrat d'opération est la description d'une opération système en termes de concepts métier (issus du modèle du domaine)

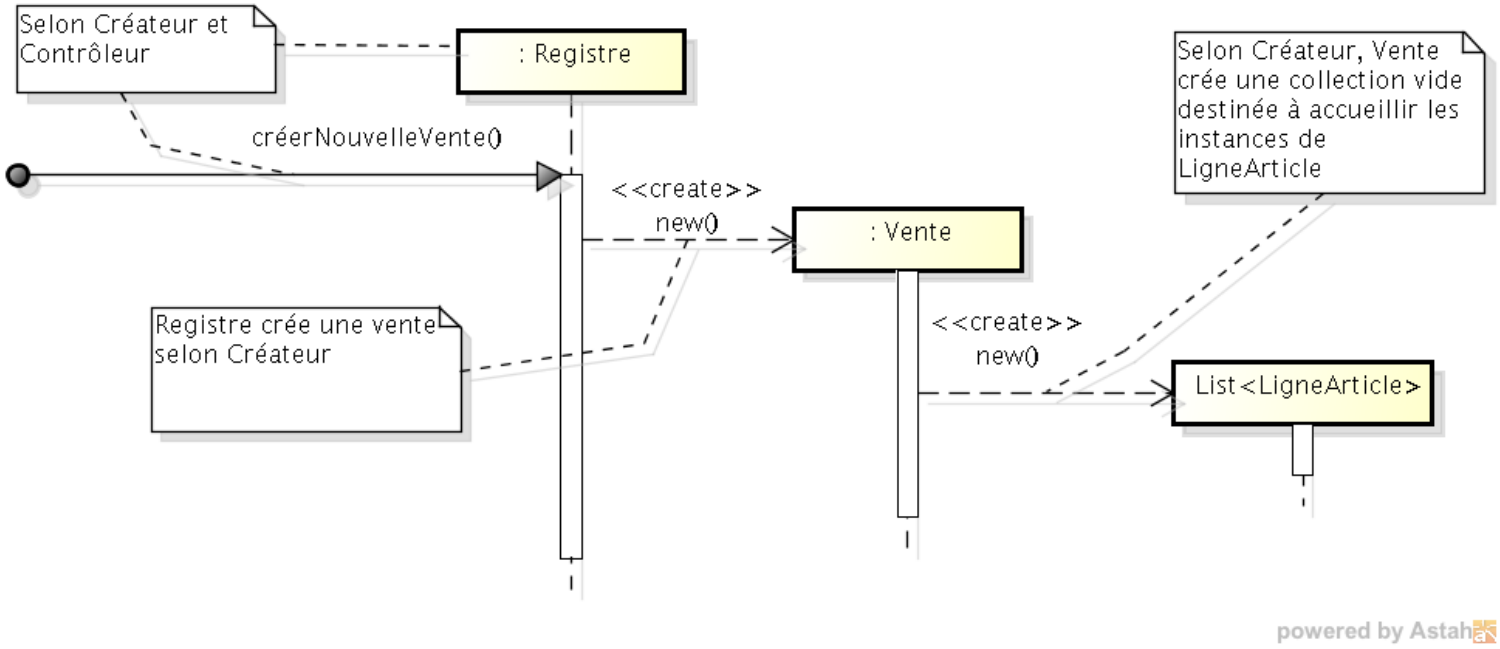
- Choix de la classe contrôleur :
  - Comme il s'agit d'une opération système, il faut choisir la classe contrôleur qui réceptionne l'opération depuis la couche *Présentation*

Représente le système, dispositif ou sous-système	*Registre* => classe inspiré du modèle du domaine
Représente un récepteur ou un gestionnaire de tous les événements système d'un scénario de cas d'utilisation	*GestionnaireTraitementVentes* *GestionnaireSessionVentes* => classes sans lien avec le modèle du domaine

  - Nous choisissons de créer une classe *Registre* car il n'y a peu d'opérations système

Opération *créerNouvelleVente* (2/2)

- Création d'une vente :
  - Il faut appliquer le pattern GRASP Créateur : affecter la responsabilité de la création à une classe qui agrège, contient ou enregistre l'objet à créer.
  - Le modèle du domaine suggère que le registre est une sorte de base de données des ventes : il sert à consigner toutes les ventes.
  - D'autre part, on faisant créer la vente par le registre il est facile de les associer (post-condition au contrat C01).
  - ➢ Donc, *Registre* est un bon créateur de *Vente*



Opération *saisirArticle* (1/2)

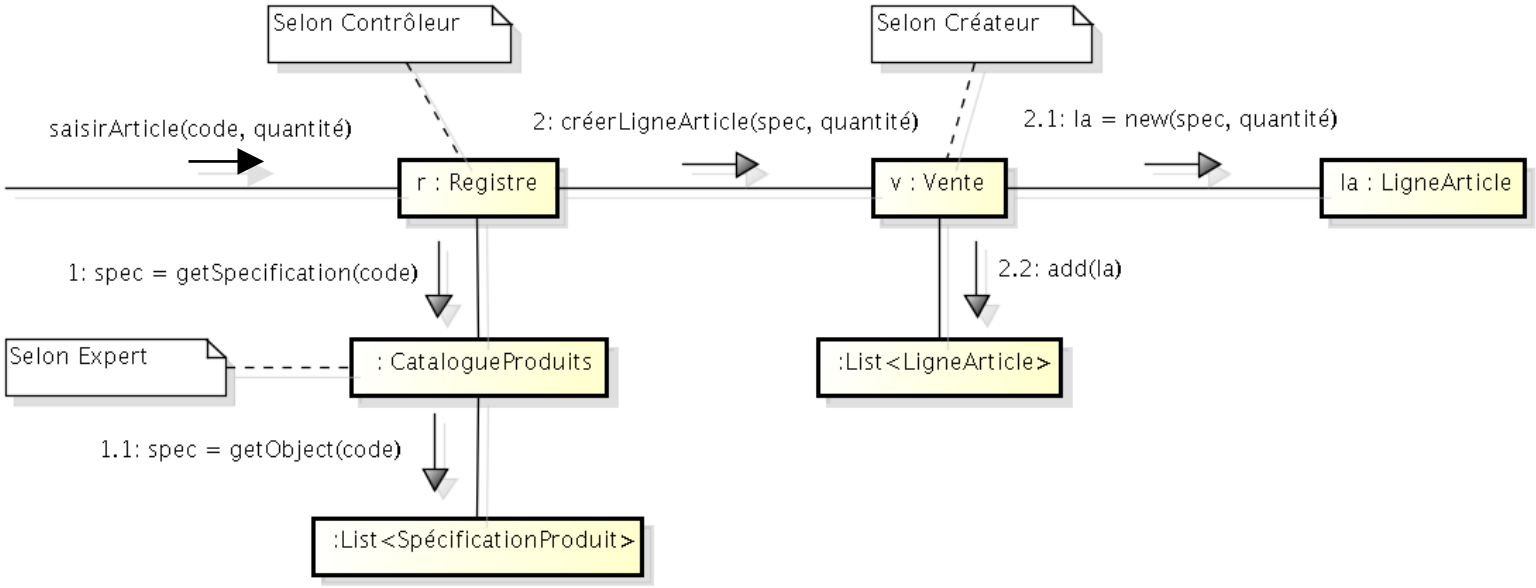
- Contexte : déclenchement par le caissier pour chaque article acheté

Contrat d'opération C02 :
- Opération : *saisirArticle(codeArticle : CodeArticle, quantité : entier)*
- Références croisées : cas d'utilisation *Traiter une vente*
- Pré-conditions :
  - Il existe une vente en cours
- Post-conditions :
  - Une instance *la* de *LigneArticle* a été créée (création d'instance).
  - *la* a été associée à la *Vente* en cours (formation d'association).
  - *la.quantité* est devenue *quantité* (modification d'attribut).
  - *la* a été associée à une *SpécificationProduit* sur la base d'une correspondance avec *codeArticle* (formation d'association).

- Choix de la classe contrôleur :
  - Comme il s'agit d'une opération système, il faut choisir la classe contrôleur qui réceptionne l'opération depuis la couche *Présentation*
  - Nous choisissons de garder la classe *Registre* pour les mêmes raisons que précédemment
- Périmètre de la responsabilité : afficher la description + prix des articles ?
  - Cela n'est pas du ressort de la couche *Domaine* mais de la couche *Présentation*
  - Utilisation du pattern Observateur (non conçu ici) : des notifications de changement d'état seront faites à des observateurs (couche *Présentation*) par certaines classes diffuseurs (couche *Domaine*).

Opération *saisirArticle* (2/2)

- Création d'une nouvelle *LigneArticle* :
  - Application du pattern GRASP Créateur
  - En s'inspirant du modèle du domaine, on choisit *Vente* comme créateur car la vente contient des objets *LigneArticle*.
  - L'association entre *Vente* et *LigneArticle* est aussi assurée.
  - Ce choix renforce la conception déjà commencée.
- Association d'un objet *SpécificationProduit* :
  - La classe *Registre* doit transmettre une *SpécificationProduit* à partir de *codeArticle*
  - Application du pattern Expert : trouver la classe qui détient l'information
  - En s'inspirant du modèle du domaine, on crée deux classes : *CatalogueProduits* et *SpécificationsProduit*



Opération *créerPaiement* (1/2)

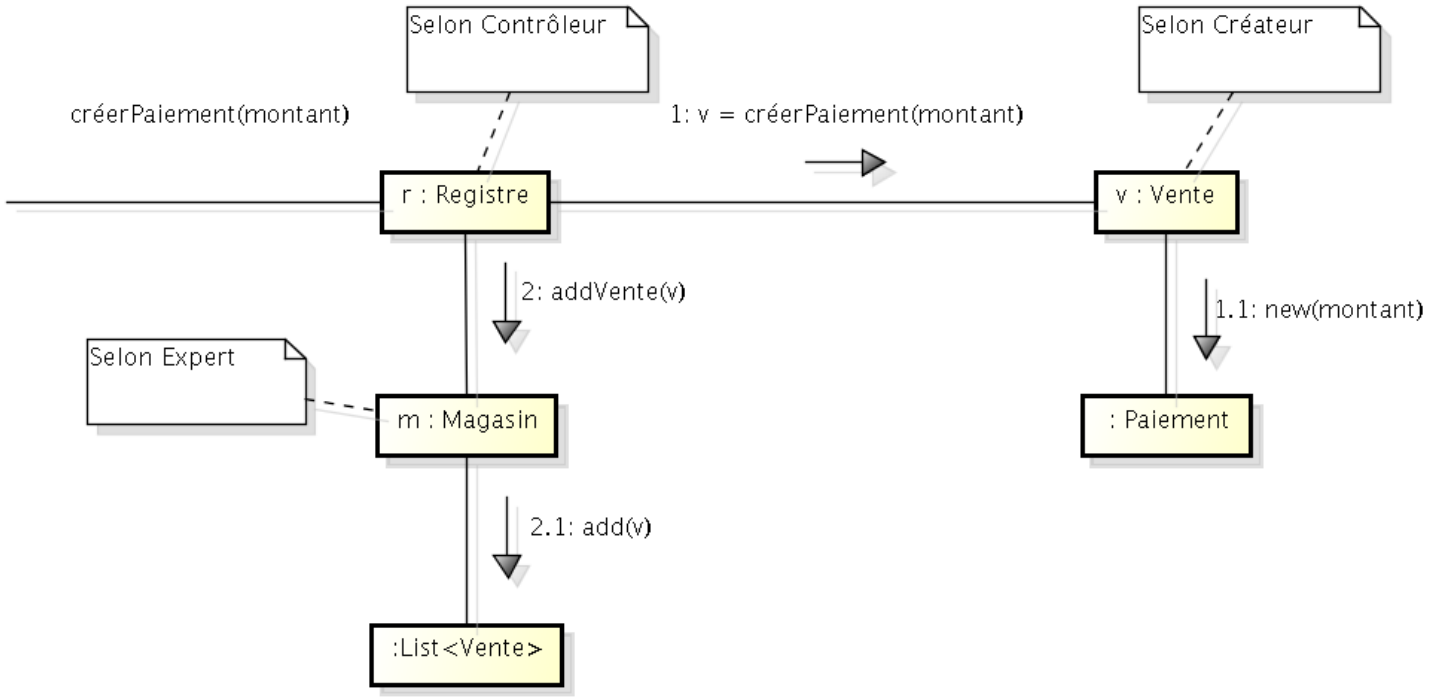
- Contexte : déclenchement par le caissier lorsqu'il saisit le montant des espèces versées

Contrat d'opération C04 :
- Opération : *créerPaiement(montant : Monnaie)*
- Références croisées : cas d'utilisation *Traiter une vente*
- Pré-conditions :
  - La vente en cours vient d'être terminée.
- Post-conditions :
  - Une instance *p* de *Paiement* a été créée (création d'instance).
  - *p.montant* est devenu *montant* (modification d'attribut).
  - *p* a été associé avec la *vente* en cours (formation d'association).
  - La *vente* en cours a été associée au *Magasin* (formation d'association).
    (Pour ajout au journal historique des ventes effectuées).

- Choix de la classe contrôleur : classe *Registre*
- Création du paiement :
  - Application du pattern GRASP Créateur
  - D'après le modèle du domaine, le registre consigne les informations comptables. De plus, d'après le pattern GRAP Expert, il détient le premier les données d'initialisation de l'objet à créer.
  - Cependant, le paiement doit être associé à la vente et non au registre car le registre n'a pas à connaître ultérieurement le paiement.
  - En choisissant *Vente* comme créateur, on réduit aussi le couplage de Registre (qui sert déjà de Façade) et on augmente sa cohésion (moins de responsabilité).

Opération *créerPaiement* (2/2)

- Consignation des ventes :
  - Application du pattern GRASP Expert
  - Le modèle du domaine indique que c'est le magasin qui connaît toutes les ventes terminées. Le registre ne connaît que la vente en cours.
  - Il suffit donc que l'objet *Registre* passe l'objet *Vente* terminée à l'objet *Magasin* (dernière post-condition au contrat C04).



Plan du cours

- ✓ Deux patterns GRASP
- ✓ Conception d'opérations du cas principal
- ➢ Quatre patterns GRASP
- Conception de l'accès aux services tiers
- Conception de la logique de tarification

Quatre patterns GRASP (1/2)

- ***Polymorphisme*** :
  - Quand des comportements varient en fonction du type, affectez les responsabilités du comportement (avec des opérations polymorphes) aux types dont le comportement varie.
  - Exemple : opération polymorphe *voler()* des types de canards

- ***Fabrication pure*** :
  - Affecter un ensemble de responsabilités cohésif à une classe « artificielle » afin de garder une forte cohésion et un faible couplage à une classe représentant un concept du domaine.
  - Exemple NextGen : responsabilités de sauvegarder les instances de Vente dans une fabrication pure StockagePermanent à la place de la classe Vente elle-même (transgression du pattern Expert pour répondre aux patterns Cohésion et Couplage).

Quatre patterns GRASP (2/2)

- ***Indirection*** :
  - Affecter la responsabilité à un objet qui sert d'intermédiaire entre deux composants pour éviter de les coupler directement.
  - Exemple de patterns d'indirection : Adaptateur, Façade, Proxy…

  « *En informatique, on peut résoudre la plupart des problèmes en ajoutant un niveau d'indirection* » mais « *on peut résoudre la plupart des problèmes de performance en supprimant un autre niveau d'indirection.* »

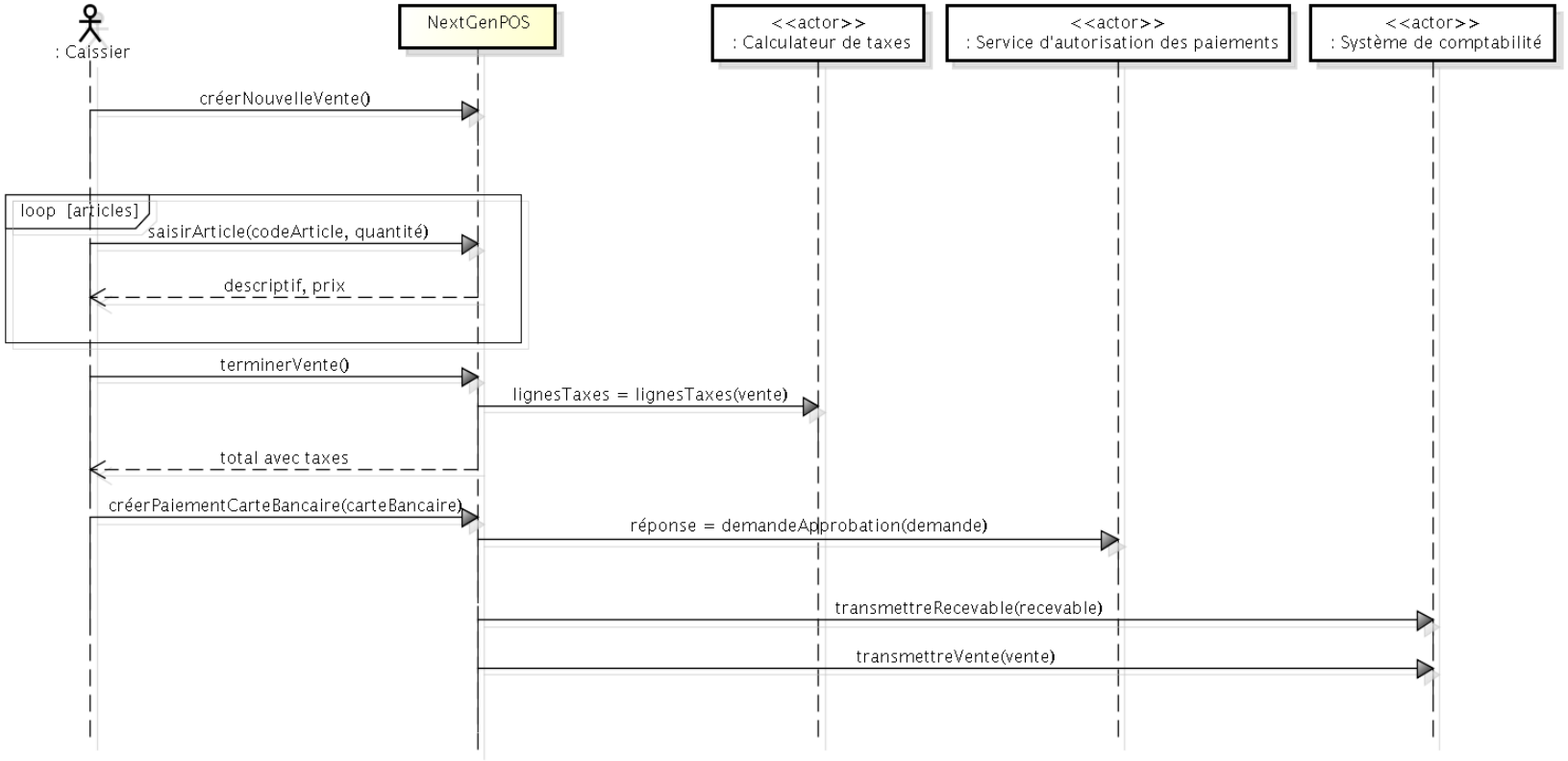
- ***Protection des variations*** (PV) :
  - Identifier les points de variation ou d'instabilité. Affecter les responsabilités pour créer une « interface » stable autour d'eux.
  - Exemple dans NextGen : point d'instabilité au niveau des différentes API des calculateurs de taxes externes résolu par une classe interface stable.
  - L'idée est de concevoir des classes, des sous-systèmes ou des systèmes de telle sorte que les variations ou l'instabilité de ces éléments n'aient pas d'impact sur d'autres éléments.
  - Exemples de mécanismes issus de PV :
    - Utiliser des classes interfaces à la place des classes concrètes (principe de substitution).
    - « Ne pas parler aux inconnus » : évitez d'envoyer des messages à des objets qui ne sont pas familiers en dehors de l'objet lui-même, des paramètres des méthodes, des attributs, des éléments d'une collection qui est un attribut, des objets locaux.
    - Externaliser les variations de comportement dans un interpréteur de règles logiques externes

Plan du cours

- ✓ Deux patterns GRASP
- ✓ Conception d'opérations du cas principal
- ✓ Quatre patterns GRASP
- ➢ Conception de l'accès aux services tiers
- • Conception de la logique de tarification

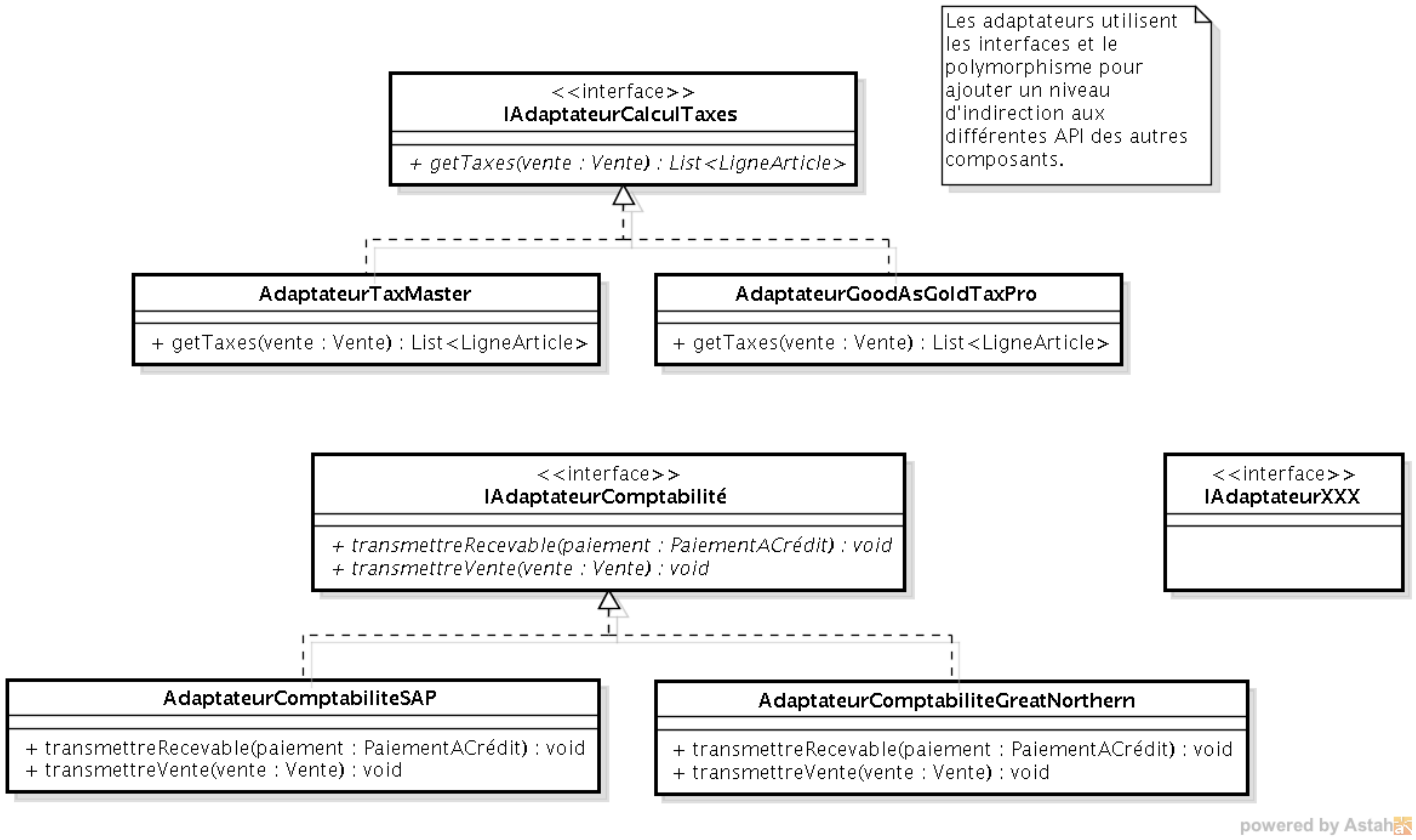
Accès aux services tiers (1/6)

- Problème n°1
  - NextGenPOS doit accepter plusieurs types de services tiers : calcul des taxes, autorisation de crédit, système de gestion comptable, gestion des stocks dont chacun possède une API distincte
  - Qui est responsable de l'accès aux services externes possédant des API distincts ?



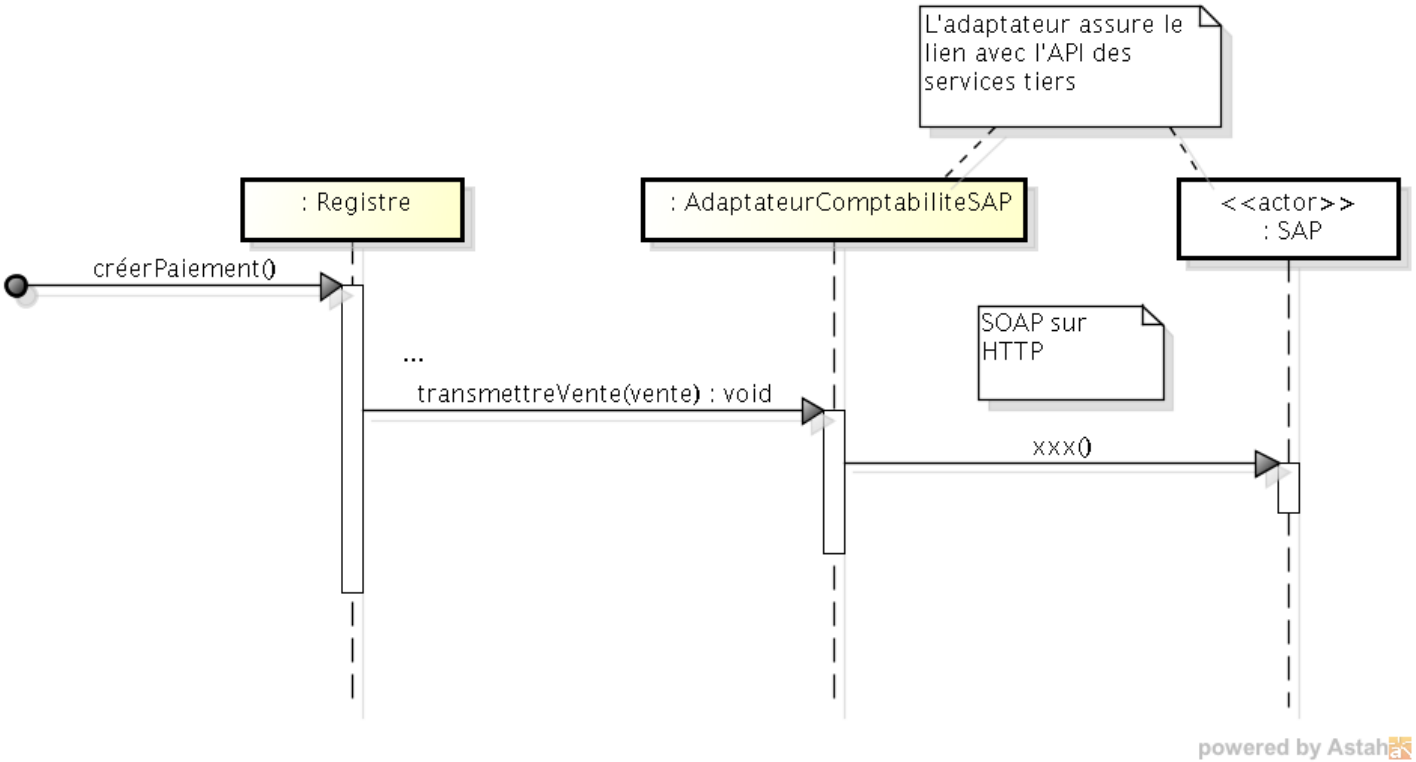
Accès aux services tiers (2/6)

- Solution n°1
  - Il faut adapter chaque API de chaque service tiers en uniformisant l'accès via des interfaces uniques par type de service
  - Appliquer le pattern GoF Adaptateur + le pattern GRASP Indirection.



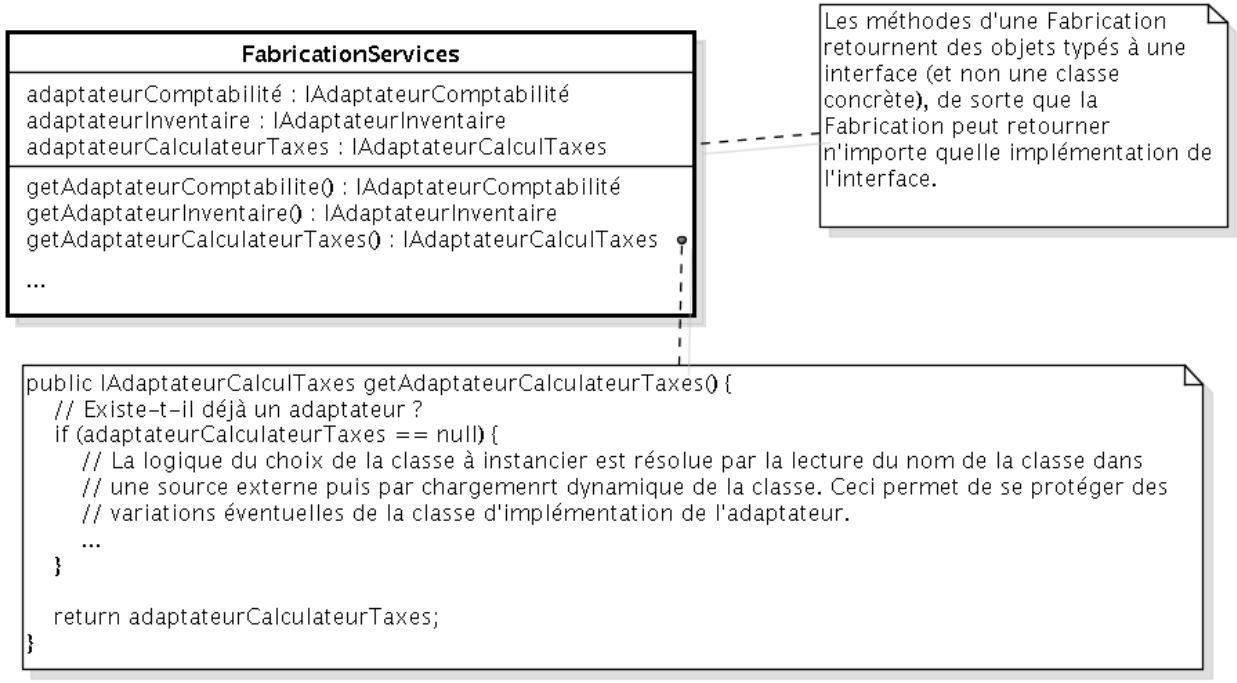
Accès aux services tiers (3/6)

- Solution n°1 (suite)
  - Les différentes API des différents services tiers sont masquées par l'accès uniformisé via la déclaration d'interfaces communes
  - Les adaptateurs implémentent l'interface commune et convertissent les requêtes au service réel externe



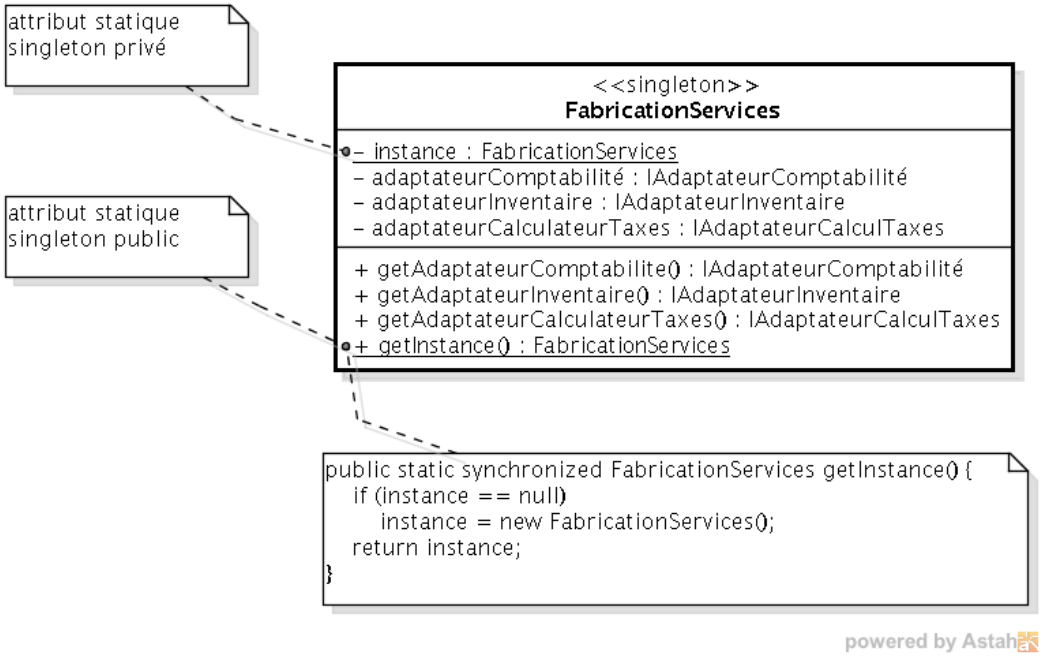
Accès aux services tiers (4/6)

- Problème/solution n°2
  - Qui est responsable de la création des objets adaptateurs et de la détermination des classes adaptateurs à utiliser ?
  - Cette responsabilité ne doit pas être attribué à un objet lié au domaine (*Magasin*, *Registre*, *Vente* ?) car celle-ci dépasse la pure logique applicative.
  - Cette responsabilité concerne un problème de connectivité aux systèmes externes (acteurs).
  - Il faut appliquer le pattern GRASP Fabrication pure.



Accès aux services tiers (5/6)

- Problème/solution n°3
  - Qui est responsable de la création de l'objet *FabricationServices* et comment y accéder ?
  - Les adaptateurs doivent être accessibles de plusieurs endroits dans l'architecture
  - À chaque service externe il ne doit y avoir qu'un seul adaptateur de créer : avoir plusieurs adaptateurs pour le même service peut conduire à des dysfonctionnements ou des problèmes de sécurité
  - Appliquer le pattern GoF Singleton



Accès aux services tiers (6/6)

- Problème/solution n°3 (suite)
  - En environnement multithread, il faut s'assurer de la synchronisation de l'appel à *getInstance* (surtout si la création nécessite de lire un fichier de configuration)
  - Le développeur a le choix entre :
    - Initialisation différée : création lors du premier appel
    - Initialisation immédiate : création au chargement de la classe

```
public static synchronized FabricationServices getInstance() {
    if (instance == null) {
        // Section critique pour une application multithreads
        instance = new FabricationServices() ;
    }
    return instance ;
}
```

```
public class FabricationServices {
    // Initialisation immédiate
    private static FabricationServices instance = new FabricationServices();

    public static synchronized FabricationServices getInstance() {
        return instance ;
    }
    …
}
```

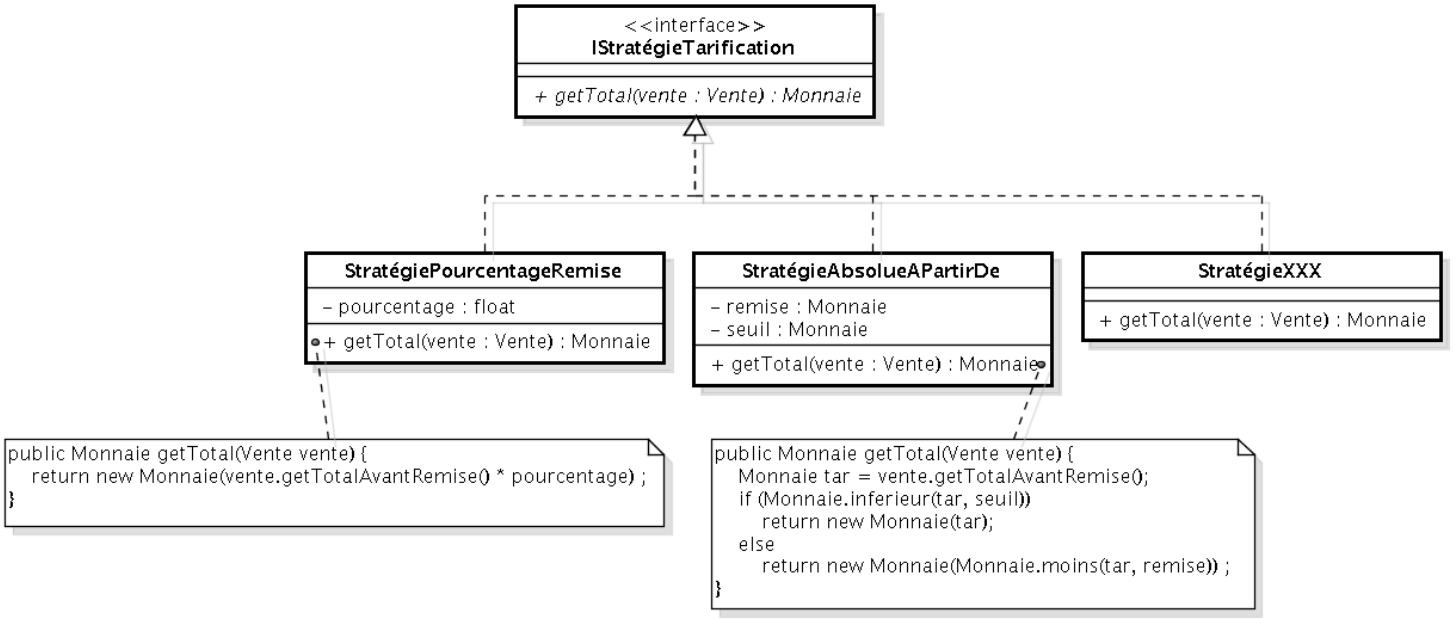
- En résumé, le concepteur « a traité le problème des variations des interfaces externes avec des Adaptateurs issus d'une Fabrication Singleton »

Plan du cours

- ✓ Deux patterns GRASP
- ✓ Conception d'opérations du cas principal
- ✓ Quatre patterns GRASP
- ✓ Conception de l'accès aux services tiers
- ➢ Conception de la logique de tarification

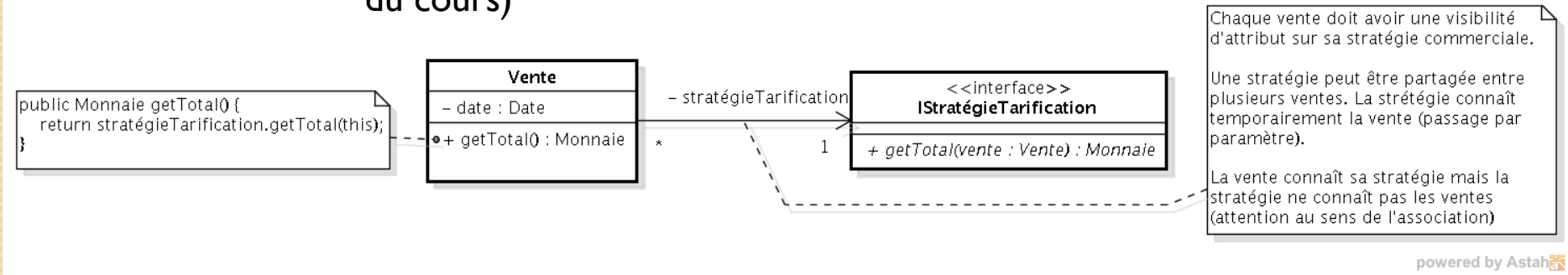
Logique de tarification (1/7)

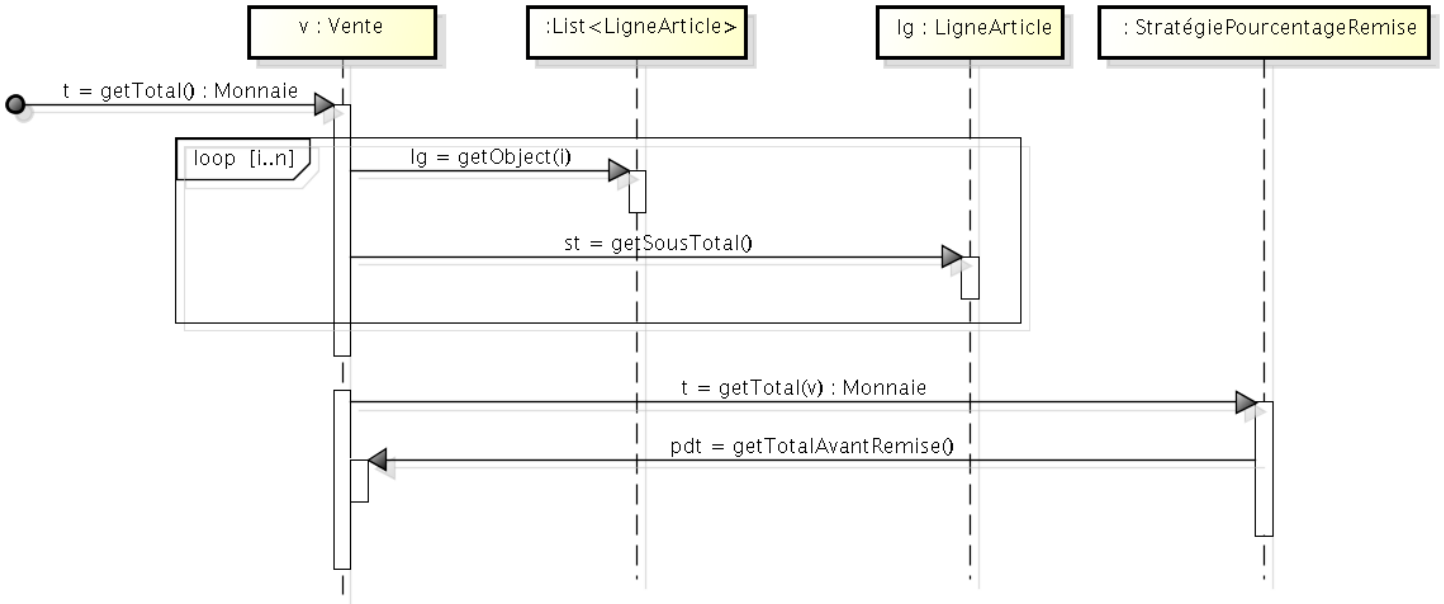
- Problème/solution n°1
  - Qui est responsable de la logique de stratégie de tarification ?
  - Logique de tarification : mise en place de la politique des réductions et d'application des réductions
  - Exemples de réductions :
    - Remise d'un pourcentage pour tous les achats sans exception
    - Remise d'un pourcentage sur certains produits durant un certain temps
    - Remise d'un pourcentage pour les clients privilégiés (avec carte abonné)
    - Etc.
  - Appliquer le pattern GoF Stratégie : une stratégie concrète = une remise



Logique de tarification (2/7)

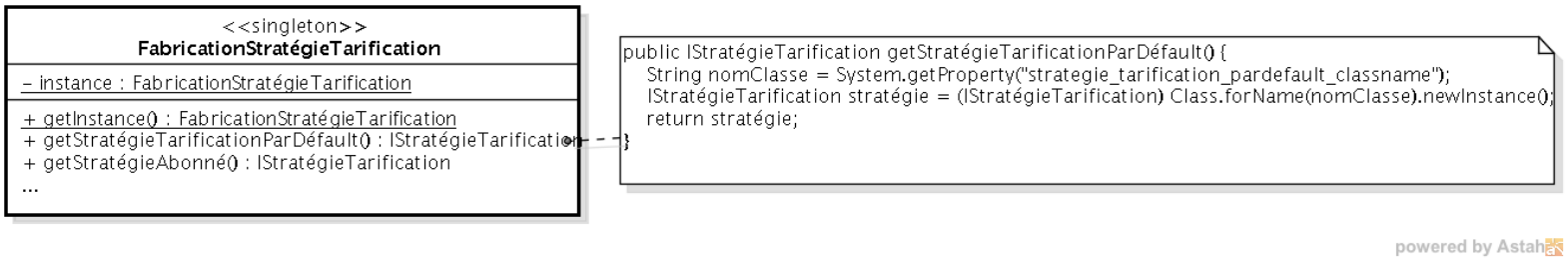
- Problème/solution n°1 (suite)
  - Comme les remises applicables sont potentiellement dépendantes du montant, de l'instant de la vente et du client…
  - …chaque vente doit être associée à une stratégie (ou plusieurs cf. la suite du cours)

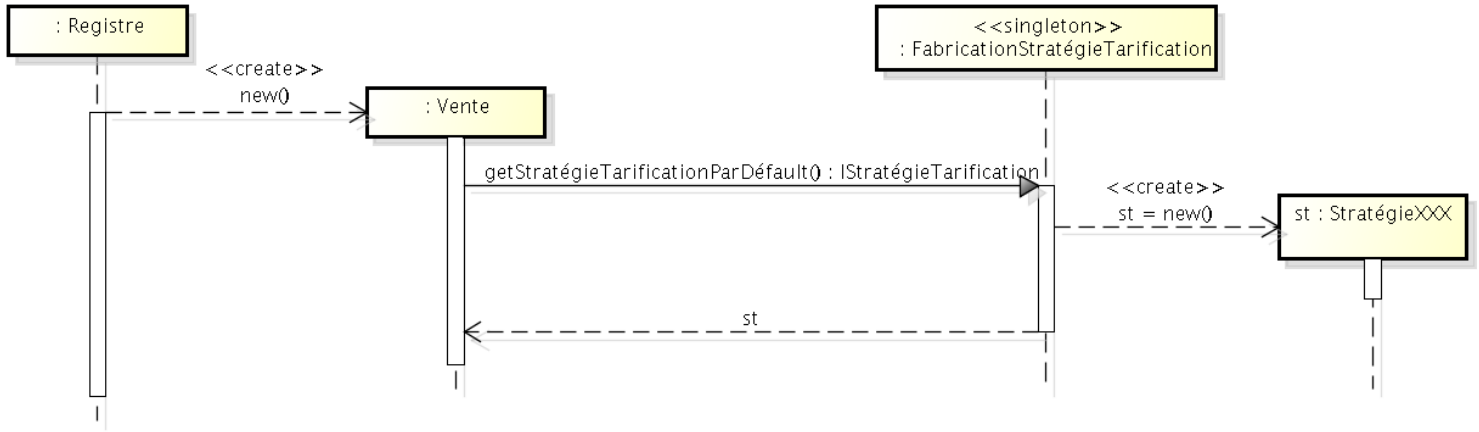




Logique de tarification (3/7)

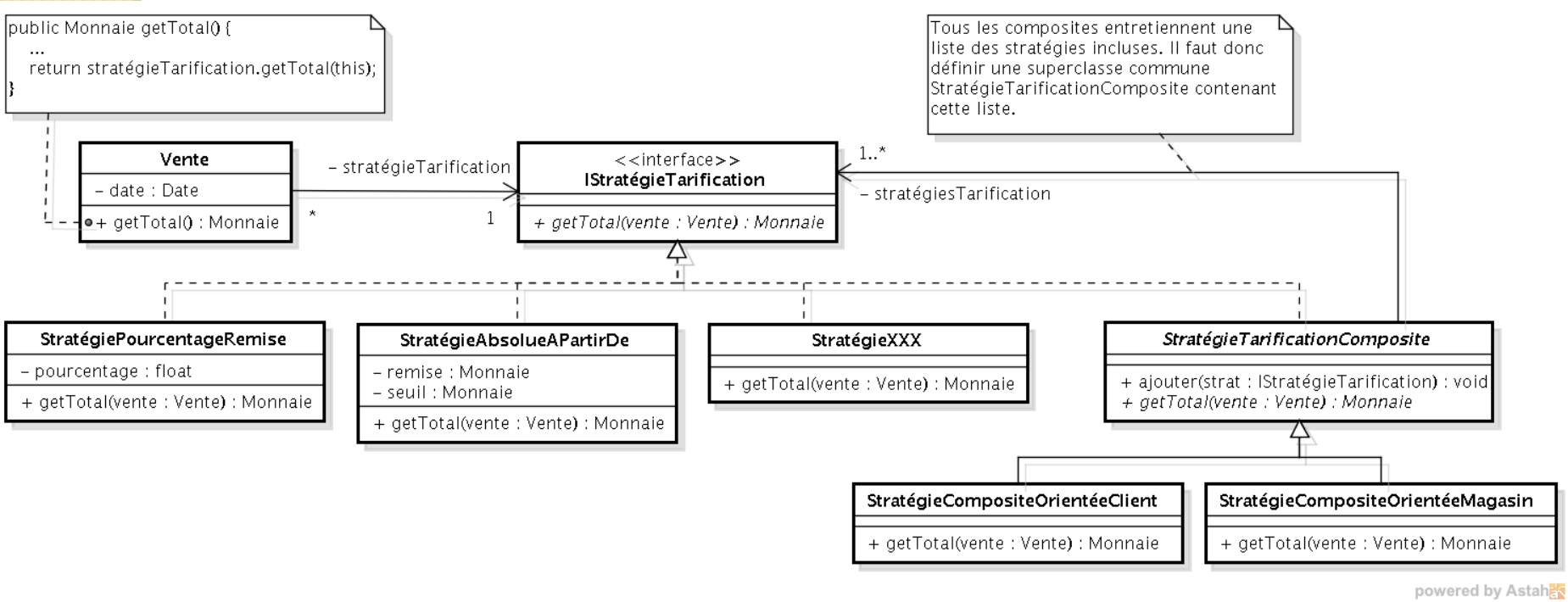
- Problème/solution n°2
  - Qui est responsable de la création de l'objet stratégie de la vente ?
  - Il existe différentes stratégies de tarification et elles varient rapidement avec le temps.
  - Il faut mettre en place une stratégie par défaut (= pas de remise commerciale par exemple)
  - Affecter cette responsabilité à la vente est une erreur : ce n'est pas la vente qui décide de sa stratégie. C'est le magasin qui affecte la bonne stratégie à la vente en cours.
  - Appliquer le pattern GRASP Fabrication pure + pattern GoF Singleton : une classe spécifique est responsable de la logique de tarification





Logique de tarification (4/7)

- Problème/solution n°3
  - Qui est responsable de la logique de tarifications multiples et conflictuelles ?
  - Par exemple, le lundi le magasin peut appliquer l'une des remises suivantes :
    - Réduction de 20% pour les personnes âgées ;
    - Réduction de 15% pour les abonnés si montant > 400 €
    - Réduction de 50% ce lundi si montant > 500 €
  - Pour un même client, plusieurs remises peuvent s'appliquer mais laquelle ?
  - Il faut appliquer une stratégie *orientée client* (avantageuse pour le client) ou *orientée magasin*
  - Appliquer le pattern GoF Composite : une stratégie tarifaire réelle doit tenir compte de toutes les remises applicables pour la vente en cours (composition de stratégies atomiques)



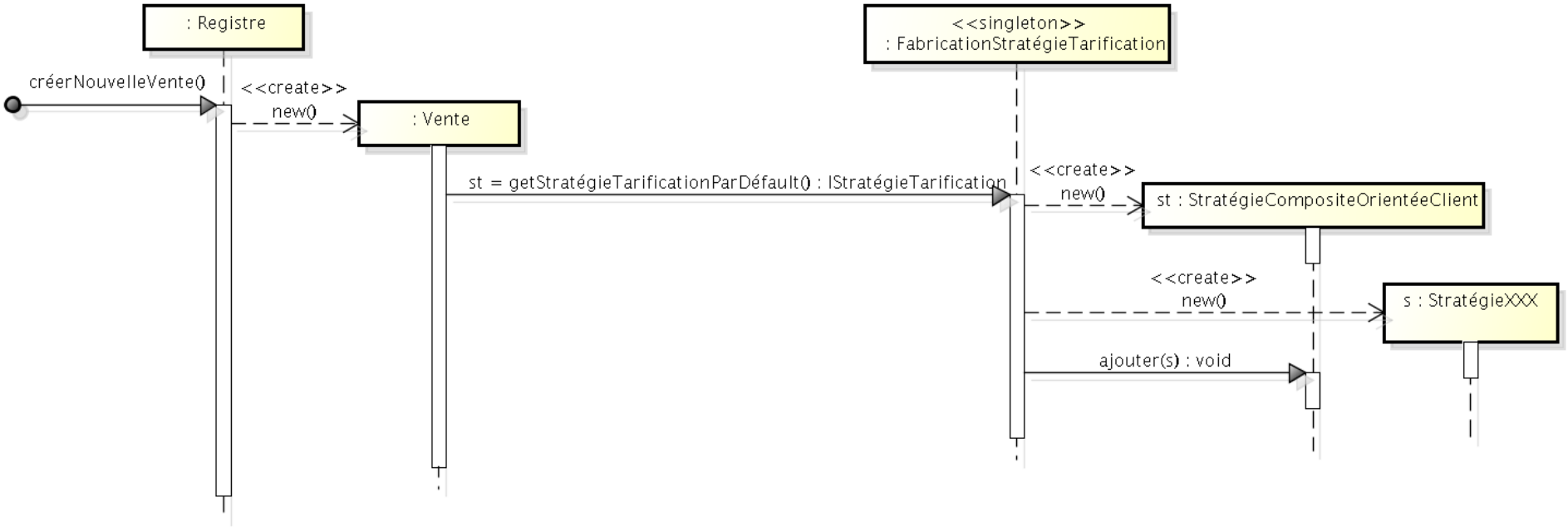
Logique de tarification (5/7)

- Problème/solution n°3 (suite)
  - Exemple de code Java :

```
// Super-classe de sorte que toutes les sous-classes puissent hériter d'une liste de stratégies
public abstract class StratégieTarificationComposite implements IStratégieTarification {

	protected List stratégiesTarification = new ArrayList();

	public void ajouter(IStratégieTarification s) {
		stratégiesTarification.add(s);
	}

	public abstract Monnaie getTotal(Vente vente);
}

// Stratégie composite renvoyant le total le plus bas de ses StratégieTarification internes
public class StratégieCompositeOrientéeClient extends StratégieTarificationComposite {
	public Monnaie getTotal(Vente vente) {
		Monnaie plusPetitTotal = new Monnaie(Integer.MAX_VALUE);
		// Itérer sur toutes les stratégies intérieures
		for (Iterator i = strategiesTarification.iterator() ; i.hasNext() ;) {
			IStratégieTarification strategie = (IStratégieTarification)i.next();
			Monnaie total = strategie.getTotal(vente);
			plusPetitTotal = total.min(plusPetitTotal);
		}
		return plusPetitTotal ;
	}
}
```

Logique de tarification (6/7)

- Problème/solution n°4
  - Quand créer les objets stratégies composites ?
  - Concevoir le fonctionnement de la *FabricationStratégieTarification* :
    - Créer une stratégie composite (nom donné par un fichier de configuration)
    - Ajouter à ce composite une stratégie atomique par défaut (par fichier de configuration)
    - Lors de la saisie des articles, ajouter de nouvelles stratégies atomiques au composite



- En résumé, « avec une Stratégie Composite et une Fabrication Singleton nous avons assuré la Protection aux Variations de la politique tarifaire. »

Logique de tarification (7/7)

- Problème/solution n°4 (suite)
  - Certaines stratégies nécessitent d'identifier le client : il faut concevoir une nouvelle opération système *saisirClient*