Patrons de Conception (Design Patterns)

Introduction

Motivation

- Il est difficile de développer des logiciels efficaces, robustes, extensibles et réutilisables
- Il est essentiel de comprendre les techniques bien éprouvées, qui ont déjà montré leur capacité à résoudre des problèmes de développement récurrents
- Les Patrons de Conception (« Design Patterns ») sont des canevas (« frameworks ») qui aident à saisir, spécifier et mettre en œuvre ces techniques éprouvées

Observations

- Les développeurs de logiciel se confrontent à des problèmes qui sont largement indépendants de l'application elle même
  - Ex: logiciel réparti – problèmes de communication, tolérance aux fautes, gestion de la concurrence, initialisation de services
- Les bons développeurs résolvent ces problèmes en s'appuyant sur les patrons de conception appropriés
- Par contre, ces patrons de conception ont été traditionnellement :
  - Implicites dans les connaissances des experts
  - Mélangés avec le code source des applications

Définition

- Patron /modèle /motif de conception (design pattern) :
  - une solution réutilisable à un problème récurrent de conception logiciel, dans un certain contexte
  - la documentation d'une expérience de conception validée par la spécification d'une architecture réutilisable
  - un concept de génie logiciel, associé (la plupart du temps) avec la programmation orientée objet

Utilité

- Aide au développement de logiciels par la réutilisation de l'expérience collective des ingénieurs expérimentés en informatique
- Aide à promouvoir les bonnes pratiques de conception, en capturant les expériences existantes et bien validées en développement logiciel
- Aide à la gestion de la complexité du logiciel
- Facilite la communication entre les développeurs

Historique

- Les patrons de conception
  - Tirent leur origine des travaux de l'architecte Christopher Alexander dans les années '70
  - Formalisés pour la première fois en 1995 dans le livre du « Gang of Four » (GoF)
    - GoF : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides
    - « Design Patterns - Elements of Reusable Object-Oriented Software »

Constitution

- Un patron de conception est défini par:
  - un Contexte : une situation qui engendre le problème
  - un Problème : le problème récurrent qui apparaît dans ce contexte
  - une Solution : une résolution validée du problème

- Le patron de conception extrait les aspects statiques et dynamiques de la structure et de la coopération entre les participants clés de la conception d'application

Sommaire

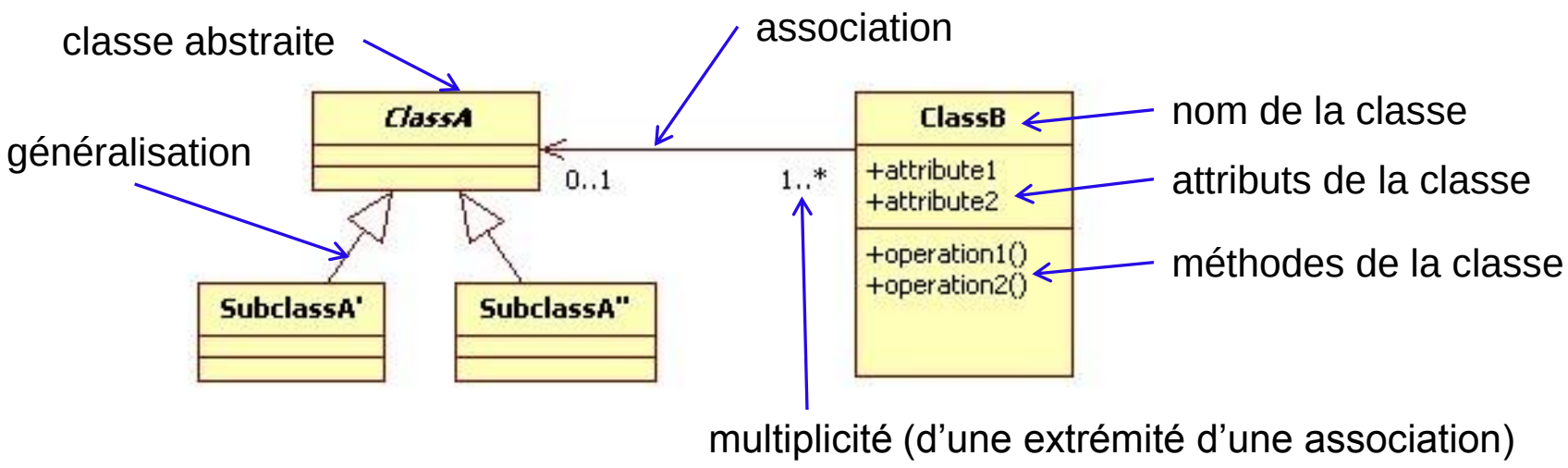
- Proxy
- Usine (« Factory »)
- Adaptateur (« Adapter »)
- Intercepteur (« Interceptor »)
- Chaîne de Responsabilités (« Chain of Responsibility »)

Une très brève introduction à UML

- Unified Modelling Language
- Spécification de l'OMG (Object Management Group)
- Langage standard de modélisation d'applications à objets:
  - Structure, comportement, architecture, processus métier, structures de données, …
- Plusieurs notations pour plusieurs diagrammes / modèles:
  - Diagramme de cas d'utilisation
  - Diagramme de classes
  - Diagramme de séquence
  - Diagramme d'activités
  - Diagramme d'état
  - ...

Diagramme de classes

- Modélise la structure statique de l'application:
  - Les classes et les relations entre les classes
    - Qui interagit avec qui? (mais pas comment)



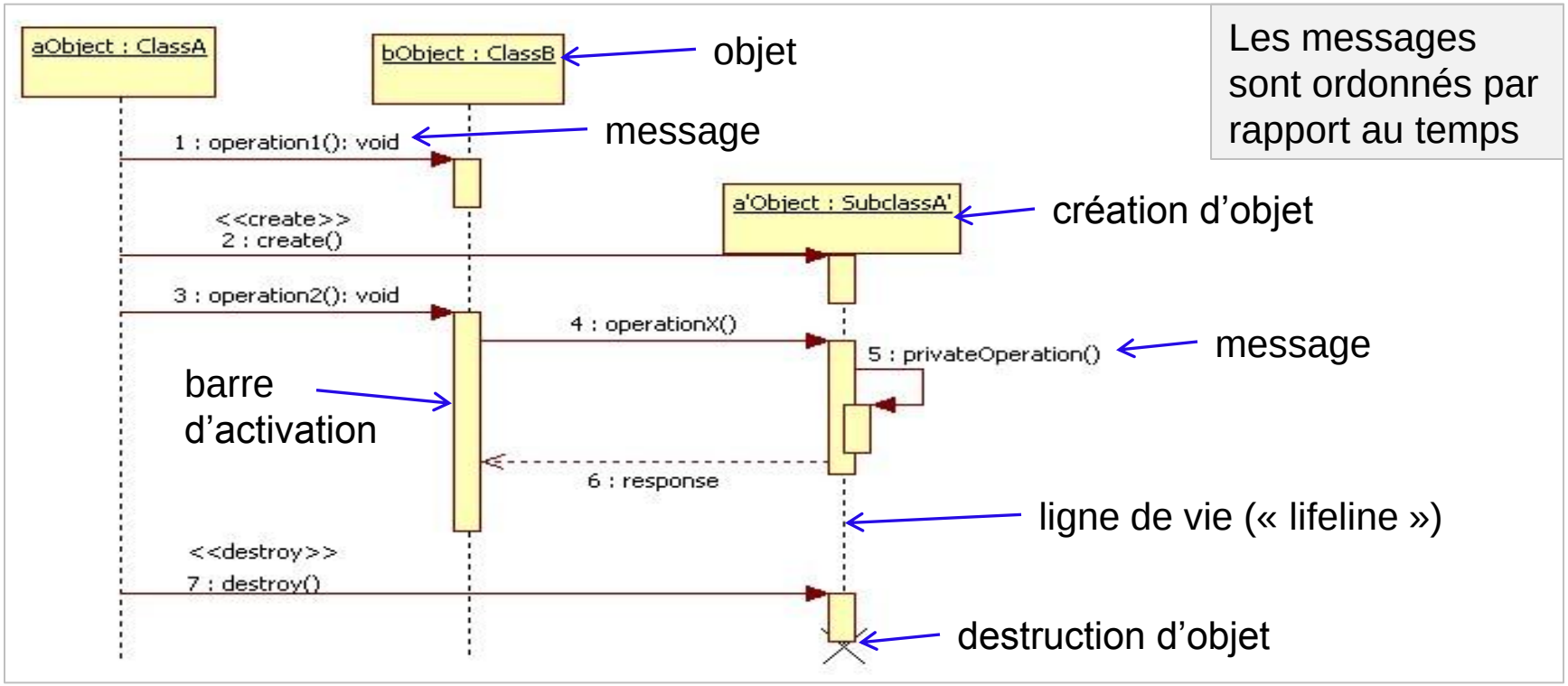
> **association** : une instance d'une classe doit connaître une instance de l'autre classe

> **généralisation** : un lien d'héritage – une classe est la super classe de l'autre

> **multiplicité** : le nombre d'instances de la classe qui peuvent être associés avec une instance de l'autre classe

Diagramme de séquence

- Modélise le comportement dynamique de l'application :
  - Les objets et les interactions entre les objets
    - Quels messages sont échangés et quand?



Proxy

- Offre un substitut ou représentant d'un autre objet, afin de contrôler l'accès à cet objet

Proxy

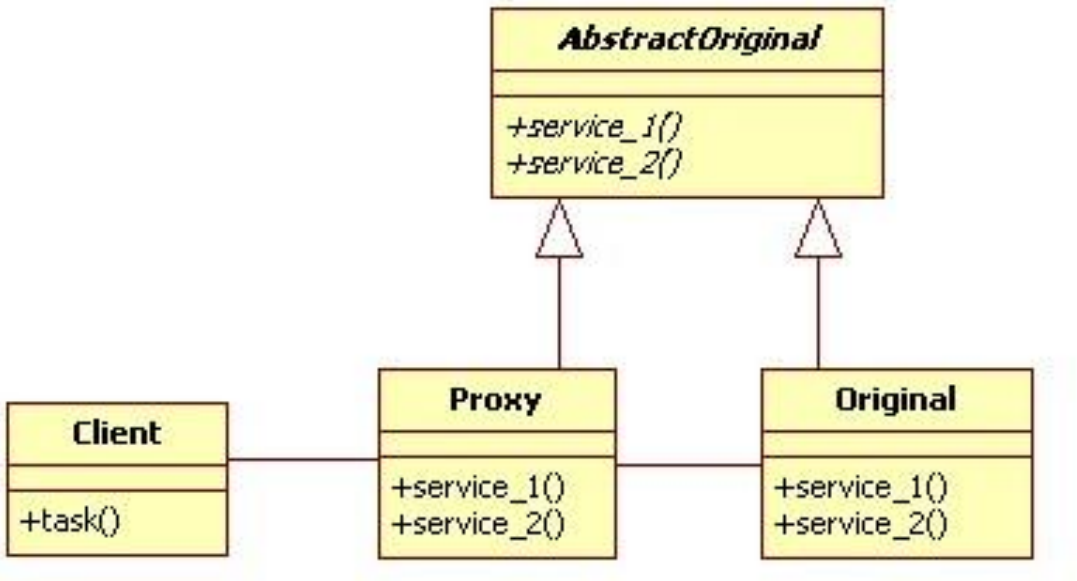
- Exemples
  - Proxy distant
    - Est le représentant local d'un objet situé dans un autre espace d'adressage
    - Ex: le Stub de RMI, CORBA, Java EE, .NET, …
  - Proxy virtuel
    - Crée des objets coûteux à la demande (lazy loading, caching, …)
    - Ex: reporter la création d'un client email jusqu'à la première utilisation
    - Ex: utiliser un cache local d'images
  - Proxy de protection
    - Contrôle l'accès au composant (plusieurs droits d'accès sont possibles)
    - Ex: Serveur d'Application – le traitement d'une requête client n'est pas démarré si les droits d'accès du client sont insuffisants

Proxy

- Contexte
  - Le client a besoin d'accéder aux services d'un autre composant (ex. objet, base de données, page html ou image)
  - L'accès direct est possible du point de vue technique mais sans être la meilleure solution
- Problème
  - L'accès direct à un composant n'est souvent pas pratique – des procédures additionnelles de contrôle sont nécessaires (ex. authentification, localisation)
  - Le code client doit rester simple et l'accès aux composants transparent et efficace
- Solution
  - Le client communique avec le représentant (proxy) plutôt qu'avec le composant
  - Le proxy offre l'interface du composant mais exécute des procédures additionnelles avant (pre) et après (post) l'invocation du composant

Proxy

- Diagramme de classes UML

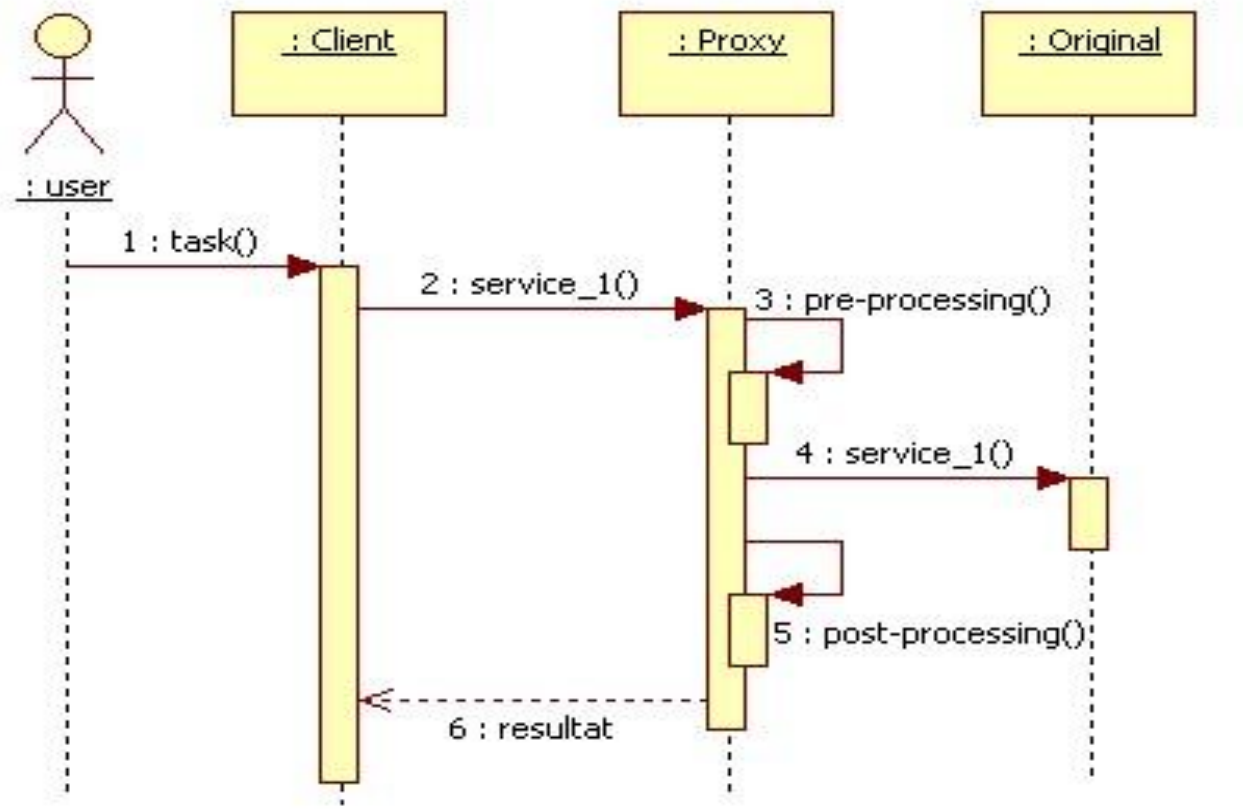


Proxy

- Rôles
  - Proxy
    - Offre aux clients l'interface de l'Original
    - Assure l'accès sécurisé, efficace et correct à l'Original
  - AbstractOriginal
    - Définit l'interface (ou la classe de base) pour l'Original et le Proxy
    - Ainsi, le Proxy peut être utilisé à tout endroit où l'Original est attendu
  - Original
    - Implémente les services décrits dans l'AbstractOriginal
    - Définit l'objet réel représenté par le Proxy
  - Client
    - Utilise l'interface du Proxy pour requérir les services de l'Original

Proxy

- Diagramme de séquence UML



Méthode Usine (« Factory Method »)

- Fournir une interface pour la création d'un objet, mais laisser les sous-classes décider quelle classe instancier
- Déléguer l'instanciation aux sous-classes

Usine (« Factory »)

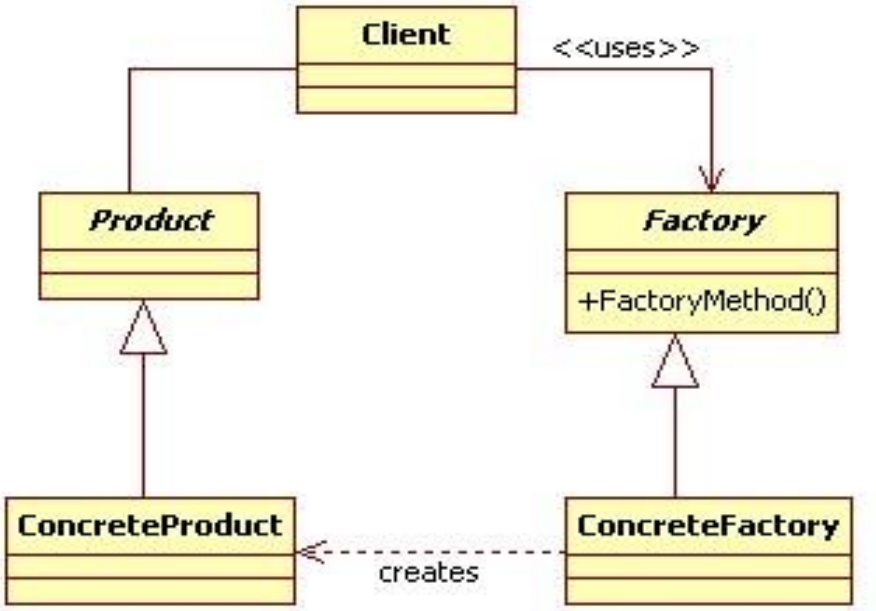
- Exemples :
  - Créer et utiliser des Connecteurs pour communiquer avec les objets distants, en utilisant différents intergiciels (ex : Sockets, RMI ou CORBA)
  - Créer et utiliser des gestionnaires de sauvegarde sans savoir à l'avance si le gestionnaire utilisé sera basé sur un système de fichiers ou une base de données

Usine (« Factory »)

- Contexte
  - Un des buts de l'orientation objet est de déléguer la responsabilité du traitement des requêtes entre plusieurs objets
- Problème
  - Parfois, une application ne peut pas anticiper la classe concrète d'objet qu'elle doit instancier (seulement l'interface ou la classe abstraite)
- Solution
  - Définir une interface pour créer des objets d'un certain type mais laisser les sous-classes décider quel sous-type de classe concrète instancier

Usine (« Factory »)

- Diagramme de classes UML

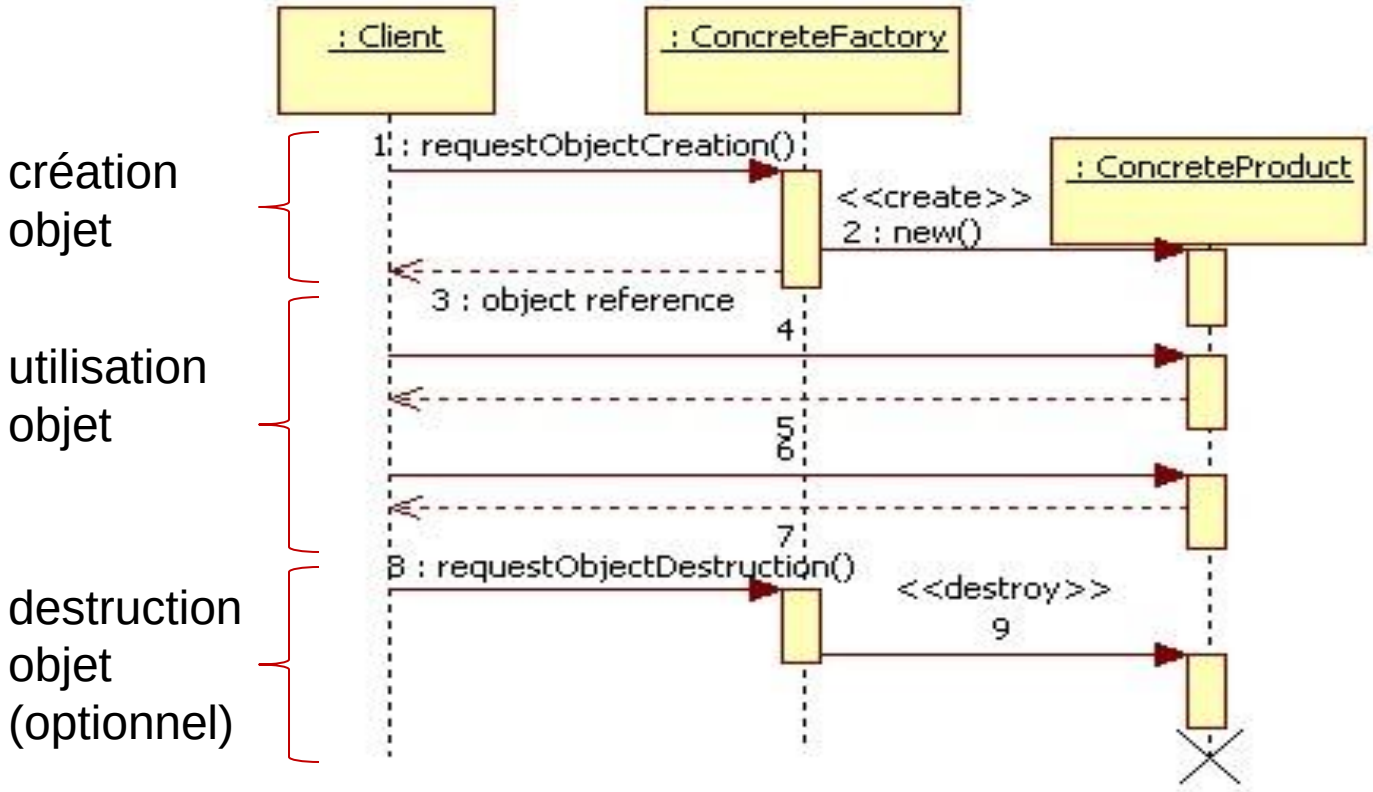


Usine (« Factory »)

- Rôles
  - Product – Interface
    - Définit l'interface des objets crées par l'Usine
  - ConcreteProduct – Classe Concrète
    - Implante l'interface Product
  - Factory – Interface
    - Définit la méthode de création qui retourne un objet de type Product
  - ConcreteFactory – Classe Concrète
    - Crée une instance d'un ConcreteProduct

Usine (« Factory »)

- Diagramme de séquence UML



Adaptateur

- Transforme l'interface d'une classe en une autre interface, connue par les clients

Adaptateur

- Exemple
  - Adaptation des logiciels « legacy » pour l'intégration dans des nouveaux systèmes
  - Intégration des systèmes hétérogènes

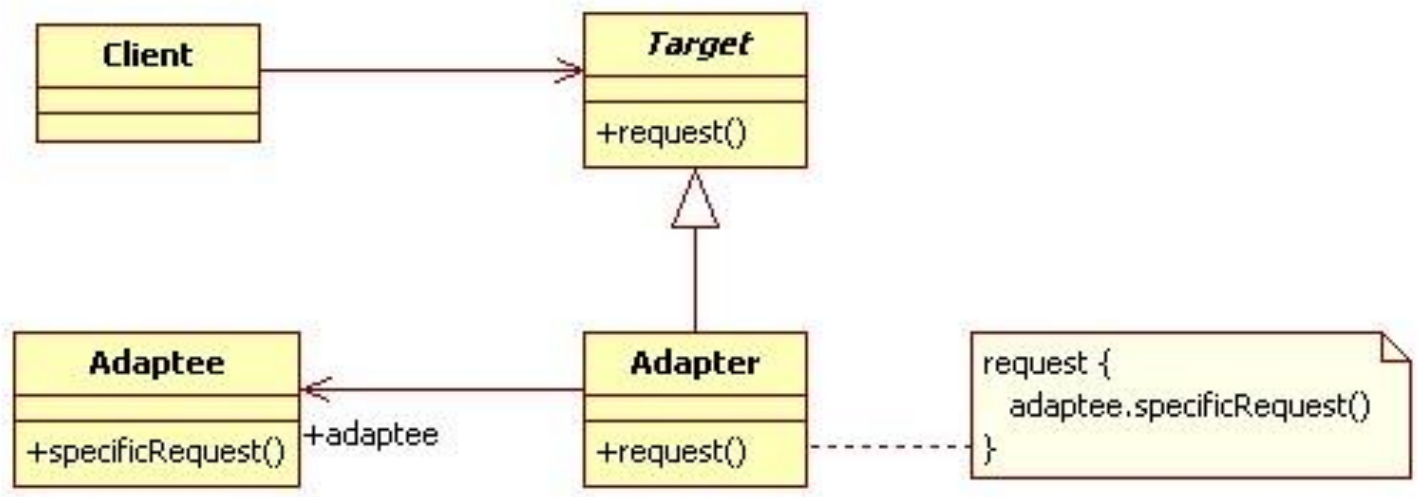
  - CORBA - Portable Object Adapters (POA)
  - Java Connector Architecture (JCA) – Resource Adapters

Adaptateur

- Contexte
  - Un service est défini par une interface; les clients appellent le service via cette interface; les servants implantent l'interface
- Problème
  - Réutiliser un servant qui implante une interface non-conforme à l'interface attendue par les clients
- Solution
  - Introduire un composant (adapter ou wrapper) représentant une couche de transformation entre les clients et le servant
  - L'adapter intercepte les requêtes et réponses du servant et les adapte de façon à les rendre conformes avec l'interface attendue

Adaptateur

- Diagramme de classes UML

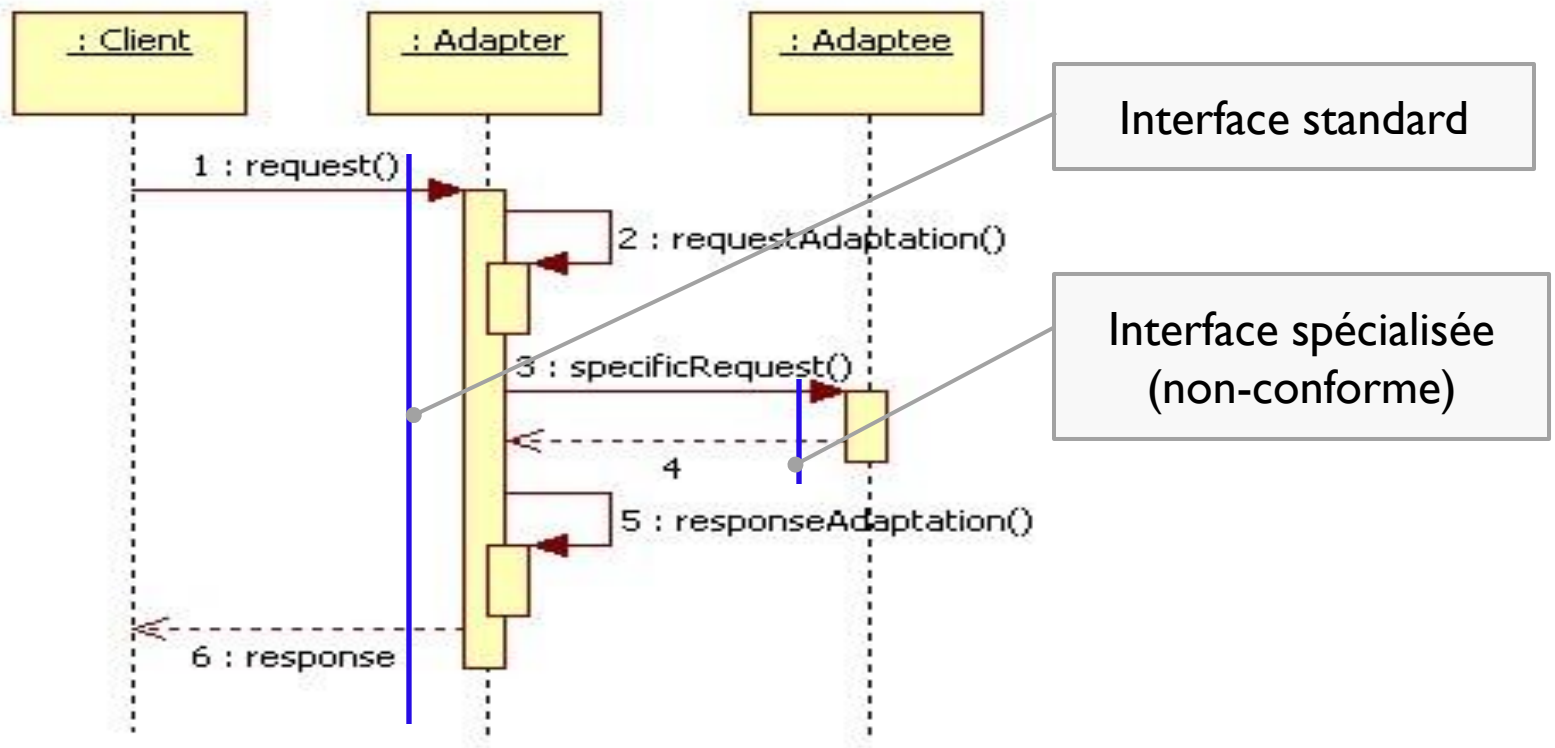


Adaptateur

- Rôles
  - Target - Interface
    - Définit l'interface utilisée par les clients
  - Client
    - Utilise des objets conformes à l'interface Target
  - Adaptee
    - Définit une interface existante qui exige une adaptation
  - Adapter
    - Adapte l'interface d'Adaptee à l'interface Target

Adaptateur

- Diagramme de séquence UML



Intercepteur

- Permet l'insertion transparente de services dans un canevas et l'activation automatique de ces services lors de l'occurrence de certains événements.

Intercepteur

- Exemple
  - Surveiller le fonctionnement d'une application
  - Remplacer un service de sécurité par un autre
  - Utiliser ou désactiver un service de journalisation (Logging)

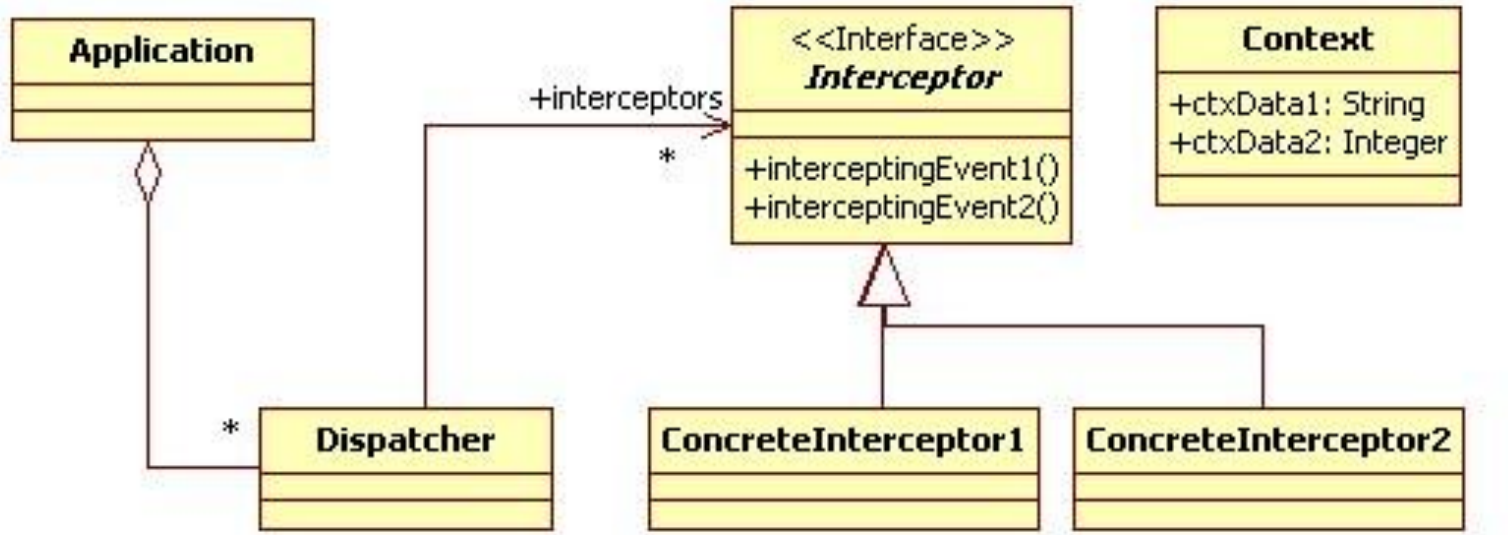
  - CORBA – Portable Interceptors
  - Aspect-Oriented Programming (AOP)

Intercepteur

- Contexte
  - Développement de canevas extensibles de façon transparente
- Problème
  - Certains canevas ne peuvent pas anticiper tous les services qu'ils doivent offrir aux clients => permettre l'intégration de nouveaux services sans avoir besoin de modifier l'architecture de base
  - L'intégration de nouveaux services ne doit pas affecter les services existants
- Solution
  - Permettre l'extension transparente d'un canevas par l'enregistrement des services via des interfaces prédéfinies
  - Permettre au canevas de déclencher automatiquement l'exécution de ces services lors de l'occurrence de certains événements
  - Définir des objets « contexte » qui permettent la transmission de l'état interne du canevas aux services

Intercepteur

- Diagramme de classes UML (simplifié)

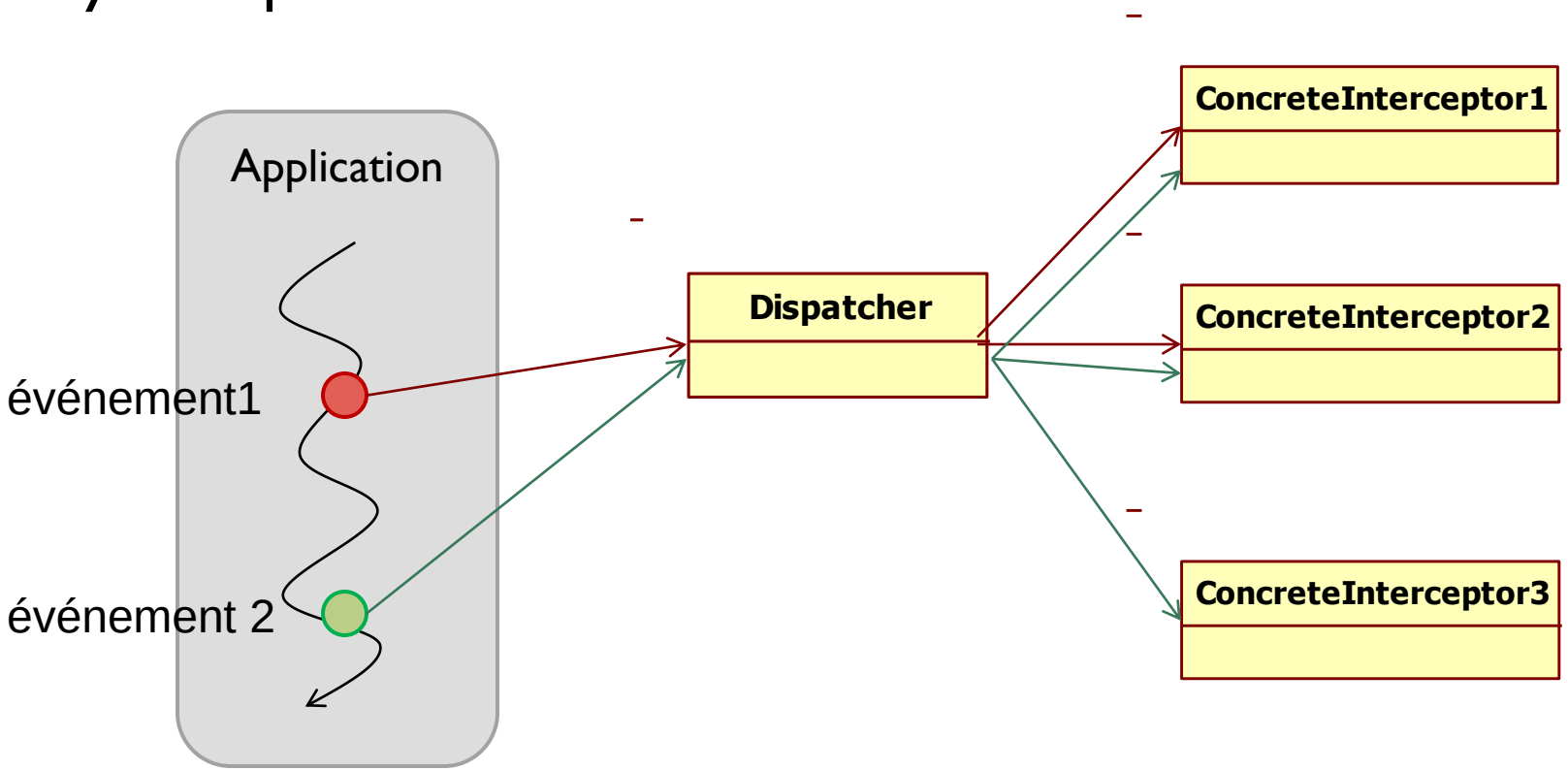


Intercepteur

- Rôles
  - Interceptor – Interface
    - Définit les méthodes de services d'interception (appelées automatiquement lors de l'occurrence de certains événements)
  - ConcreteInterceptor – Classe Concrète
    - Implante un certain service d'interception
    - Utilise l'objet Context pour connaître son contexte d'exécution
  - Dispatcher – Classe Concrète
    - Permet l'enregistrement et le retrait d'Intercepteurs
    - Distribue les appels de méthodes vers les Intercepteurs enregistrés, lors de l'occurrence d'événements
  - Context – Classe Concrète
    - Permet aux services d'obtenir des informations sur (ou de modifier) leur contexte d'exécution

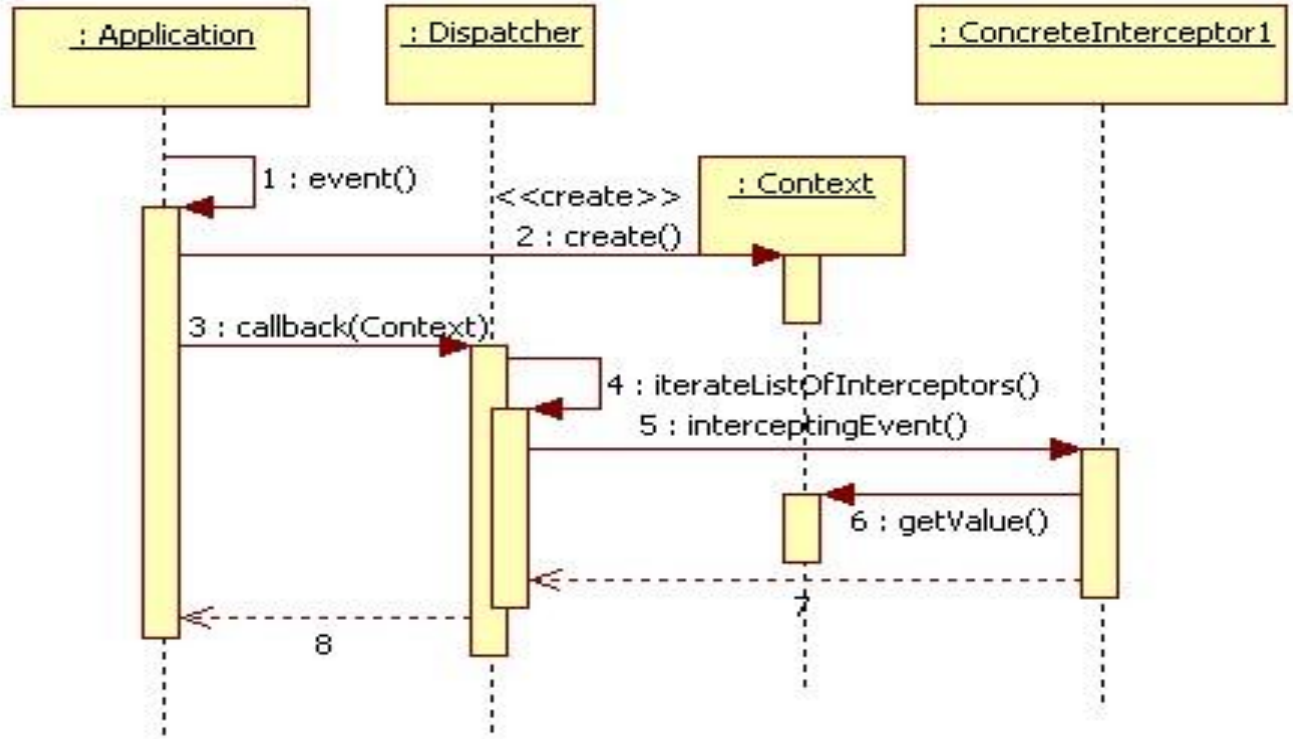
Intercepteur

- Dynamique



Intercepteur

- Diagramme de séquence UML



Chaîne de Responsabilités

- Eviter le couplage entre l'expéditeur d'une requête et son destinataire en donnant à plusieurs objets la possibilité de traiter la requête.
- Chaîner les objets traitants et faire passer la requête à travers ces objets jusqu'à ce qu'elle soit traitée.

Chaîne de Responsabilités

- Exemple - transformer le format d'un document
  - Faire passer le document initial via plusieurs filtres afin d'obtenir le document dans le format final
  - Chaque filtre exécute certaines transformations sur le document entrant et produit un nouveau fichier sortant
  - Un nouveau filtre peut être facilement inséré dans la chaîne

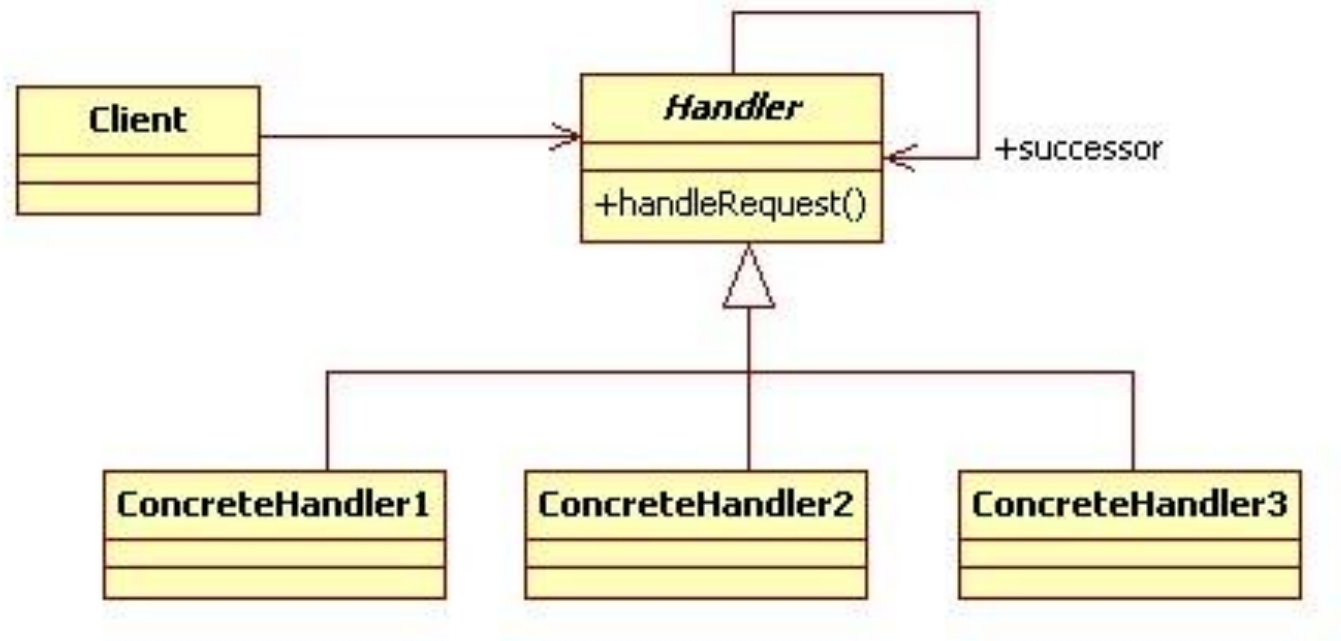
- Exemple – système de sécurité
  - Plusieurs critères possibles pour autoriser l'accès
  - Les critères peuvent changer selon la configuration donnée

Chaîne de Responsabilités

- Contexte
  - Le système doit traiter une requête
  - La requête peut être traitée de plusieurs façons (par plusieurs objets)
- Problème
  - Différents objets peuvent traiter une requête et on ne sait pas a priori lequel
  - L'ensemble des objets pouvant traiter une requête doit être facilement modifiable
- Solution
  - Isoler les différents parties d'un traitement dans différents objets
  - Faire passer la requête via une chaîne d'objets (maillons)
  - Chaque maillon peut traiter la requête et/ou la faire passer au maillon suivant

Chaîne de Responsabilités

- Diagramme de classes UML



Chaîne de Responsabilités

- Rôles
  - Handler (Maillon) – classe abstraite
    - Définit l'interface d'un maillon de la chaîne
    - Implante la gestion de la succession des maillons
  - ConcreteHandler – sous-classe concrète
    - Définit le comportement d'un maillon de la chaîne
    - A la responsabilité d'une partie d'un traitement de requête
  - Client
    - Appelle le premier maillon dans la chaîne

Bibliographie

- "Design Patterns - Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- POSA 1 - "Pattern-Oriented Software Architecture – A System of Patterns", Vol. 1, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal
- POSA 2 - "Pattern-Oriented Software Architecture – Patterns for oncurrent and Networked Objects", Vol. 2, Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann
- POSA 3, POSA 4, POSA 5, …

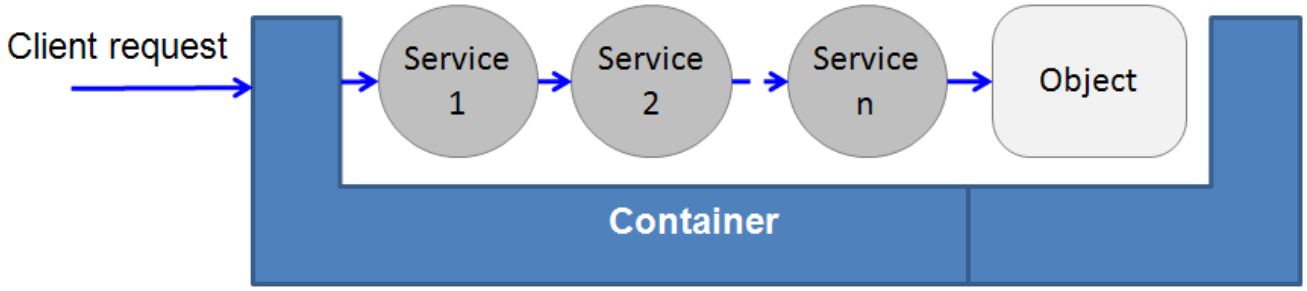
Cas d'étude

- Simulation d'un serveur d'application élémentaire

http://infres.enst.fr/~diacones/tp-patterns

Objectif

- Simulation d'un serveur d'application élémentaire
  - Chaque objet est géré par un conteneur (« container »)
  - Un conteneur peut utiliser plusieurs services non-fonctionnels (ex: sécurité, journalisation ou supervision)



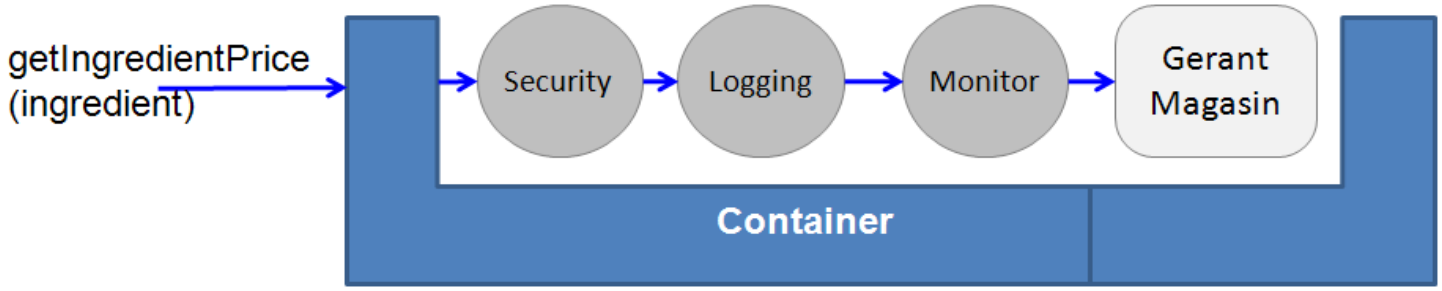
Vue générale d'un containeur

Défis

- L'utilisation du conteneur et des services non-fonctionnels doit rester transparente aux clients
- Les services utilisés par le conteneur doivent pouvoir être changés souvent et facilement
  - Insertion, modification ou retrait des services
  - Changement de l'ordre d'exécution des services

Cas d'application

- Magasin (réutilisation de l'application du TP RMI)
  - un client interroge plusieurs magasins (GerantMagasin) sur le prix d'un certain produit (ou ingrédient) afin de déterminer le magasin le moins cher (par rapport au produit)

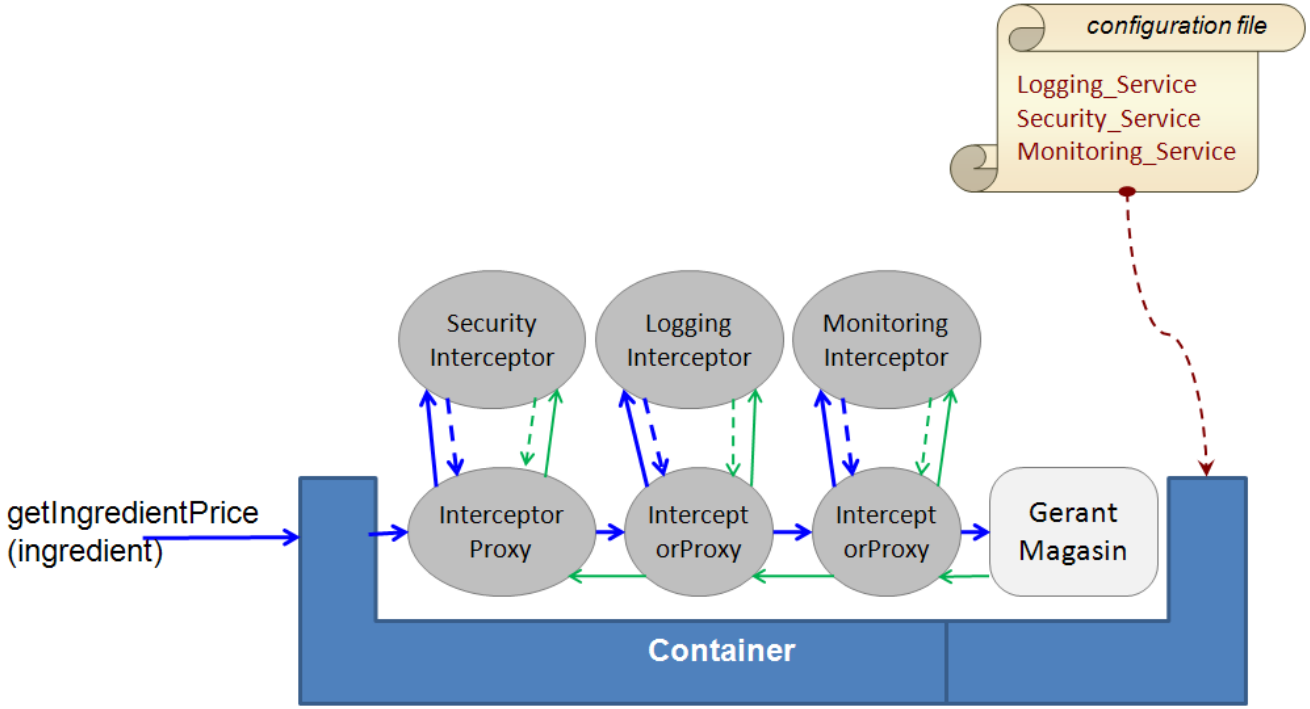


Hébergement d'un objet GerantMagasin dans un container

Conception (1)

- Combinaison de trois patrons de conception
  - Chaîne de Responsabilité
    - Chaque appel d'une méthode du Magasin passe par une chaîne de maillons (proxies)
    - Ex: Un maillon pour le service de sécurité, un maillon pour la journalisation et un pour la supervision
  - Proxy
    - Le container, ainsi que chaque maillon de la chaîne d'interception implantent l'interface Magasin – la même interface implantée par le GerantMagasin
  - Intercepteur
    - Chaque service non-fonctionnel est implanté par un intercepteur
    - Chaque maillon joue le rôle d'un Dispatcher pour un certain type d'intercepteur

Conception (2)



Conception du container du GerantMagasin

Implantation

- Squelette du code source disponible
  - www.infres.enst.fr/~diacones/tp-patterns/interceptor-tp-Project.zip
  - Code de l'application Magasin:
    - Réutilisation du code du TP RMI (mises à jour mineures)
    - Paquet `shop`
  - Code du conteneur et des services
    - Paquet `container`

Travaux Pratiques

1. Reconfiguration du serveur
   - Utilisation d'un fichier de configuration
2. Utilisation du service de sécurité
   - Modification de l'authentification client
   - Modification du service de sécurité
3. Implantation des services d'interception
   - Développement des services de supervision et de journalisation
   - Transmission de paramètres par le Context du container
4. Extension de la chaîne d'interception
   - Implantation et introduction d'un nouveau service non-fonctionnel