



École d'été sur les Intergiciels et la Construction d'Applications Réparties

Patrons et Canevas pour l'Intergiciel

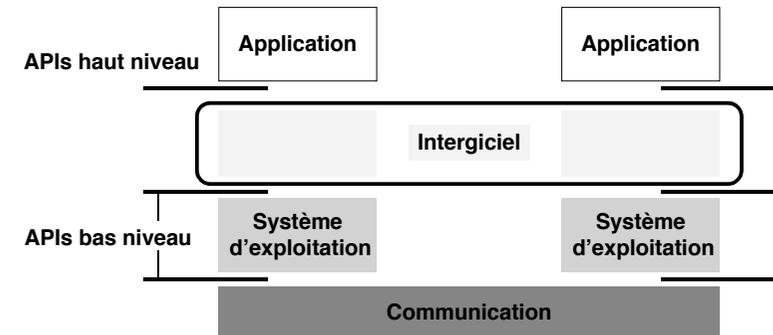
Sacha Krakowiak

Projet Sardes (IMAG-LSR et INRIA)

<http://sardes.inrialpes.fr/~krakowia>



■ L'intergiciel est la couche "du milieu" (*Middleware*)



- L'intergiciel a quatre fonctions principales
 - ◆ Fournir une interface ou API (*Applications Programming Interface*) de haut niveau aux applications
 - ◆ Masquer l'hétérogénéité des systèmes sous-jacents
 - ◆ Rendre la répartition invisible ("transparente")
 - ◆ Fournir des services répartis d'usage courant
- L'intergiciel vise à faciliter la programmation répartie
 - ◆ Développement, évolution, réutilisation des applications
 - ◆ Portabilité des applications entre plates-formes
 - ◆ Interopérabilité d'applications hétérogènes



- Objets et composants répartis
 - ◆ Java RMI, CORBA, .NET, Enterprise Java Beans, ...
- *Message-Oriented Middleware* (MOM)
 - ◆ Message Queues, *Publish-Subscribe*
- Intégration d'applications
 - ◆ *Web Services*
- Coordination
 - ◆ Jini, outils de *workflow*
- Accès aux données, persistance
- Support d'applications mobiles
 - ◆ Informatique ubiquitaire, réseaux de capteurs



■ Présenter les principaux intergiciels disponibles aujourd'hui

- ◆ Principes directeurs, organisation, usage
- ◆ Fonctionnement interne
- ◆ Exemples concrets d'utilisation

■ Indiquer les tendances de l'évolution

- ◆ Développements courants, prototypes de recherche

} Ateliers

Plusieurs exemples sont tirés d'ObjectWeb (www.objectweb.org), consortium pour le développement d'intergiciel innovant libre (*open source*)



■ Introduire les principes de base de l'architecture des intergiciels ...

- ◆ Fournir un cadre commun pour les autres présentations
- ◆ Introduire quelques tendances de la recherche courante

■ ... via une démarche systématique

- ◆ Identifier les principaux schémas d'architecture répondant aux problèmes des applications réparties
- ◆ Mettre en évidence les patrons de conception (*design patterns*) et les canevas logiciels (*software frameworks*) utiles pour la construction de l'intergiciel
- ◆ Illustrer l'usage de ces patrons et canevas sur des exemples concrets



■ Architecture de l'intergiciel

- ◆ Schémas d'interaction
- ◆ Patrons élémentaires : *Proxy, Factory, Adapter, Interceptor*
- ◆ Schémas de décomposition

■ Patrons et canevas pour l'exécution répartie

- ◆ Désignation et liaison : *Export-Bind* ; Objets répartis : *Broker*
- ◆ Coordination : *Observer, Publish-Subscribe*

■ Patrons et canevas pour la composition

- ◆ Notions de base sur les composants
- ◆ Infrastructures à composants (conteneurs, etc.)
- ◆ Déploiement



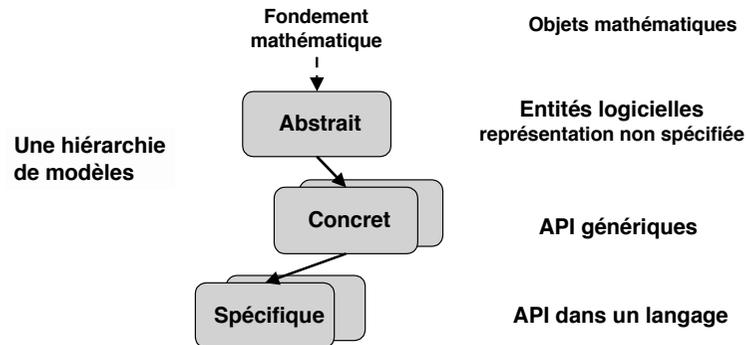
■ Principe directeur : séparation des préoccupations

- ◆ Isoler les aspects indépendants et les traiter séparément
- ◆ Examiner un problème à la fois
- ◆ Éliminer les interférences
- ◆ Permettre aux aspects d'évoluer indépendamment

■ Mise en œuvre

- ◆ Encapsulation : séparer interface et réalisation (contrat commun)
- ◆ Abstraction : décomposer en niveaux, cacher les détails non pertinents
- ◆ Séparation entre politiques et mécanismes
 - ❖ Ne pas réimplémenter les mécanismes quand on change de politique
 - ❖ Ne pas "sur-spécifier" les mécanismes
- ◆ Isolation et expression indépendante des aspects extra-fonctionnels

Définition : une description d'un aspect (point de vue, fonction) de l'architecture
Usage : compréhension, explication, prévision, preuve, guide pour la réalisation



Architecture de l'intergiciel

Services et interfaces

■ Définition

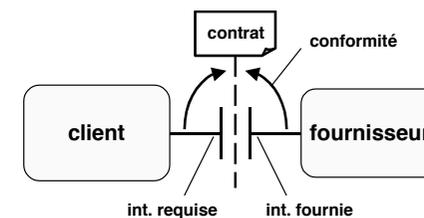
- ◆ Un système est un ensemble de composants (au sens non technique du terme) qui interagissent
- ◆ Un service est "un comportement défini par contrat, qui peut être implémenté et fourni par un composant pour être utilisé par un autre composant, sur la base exclusive du contrat" (*)

■ Mise en œuvre

- ◆ Un service est accessible via une ou plusieurs interfaces
- ◆ Une interface décrit l'interaction entre client et fournisseur du service
 - ❖ Point de vue opérationnel : définition des opérations et structures de données qui permettent l'accès au service
 - ❖ Point de vue contractuel : définition du contrat entre client et fournisseur

(*) Bieber and Carpenter, *Introduction to Service-Oriented Programming*, <http://www.openwings.org>

Définitions d'interfaces (1)



- ◆ La fourniture d'un service met en jeu deux interfaces
 - ❖ Interface requise (côté client)
 - ❖ Interface fournie (côté fournisseur)
- ◆ Le contrat spécifie la compatibilité (conformité) entre ces interfaces
 - ❖ Au delà de l'interface, chaque partie est une "boîte noire" pour l'autre (principe d'encapsulation)
 - ❖ Conséquence : client ou fournisseur peuvent être remplacés du moment que le composant remplaçant respecte le contrat (est conforme)

■ Partie "opérationnelle"

◆ Interface Definition Language (IDL)

- ❖ Pas de standard, mais s'appuie sur un langage existant
 - ▲ IDL CORBA sur C++
 - ▲ Java et C# définissent leur propre IDL

■ Partie "contractuelle"

◆ Plusieurs niveaux de contrats

- ❖ Sur la forme : spécification de types -> conformité syntaxique
- ❖ Sur le comportement (1 méthode) : assertions -> conformité sémantique
- ❖ Sur les interactions entre méthodes : synchronisation
- ❖ Sur les aspects non fonctionnels (performances, etc.) : contrats de QoS, ou SLA (*Service Level Agreement*)

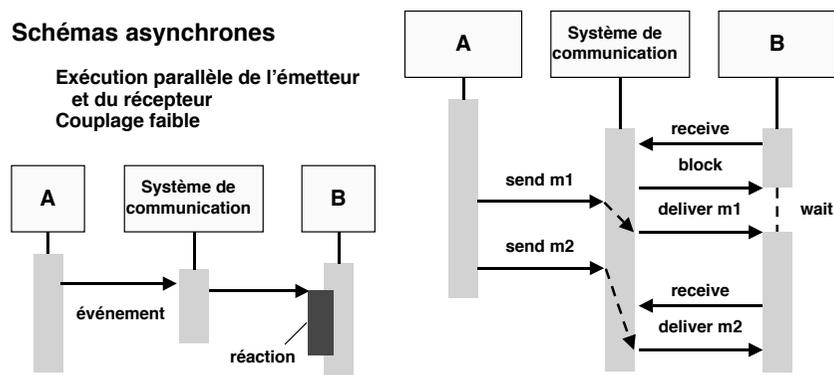
Dans un environnement réparti, le schéma abstrait cache une réalité complexe

- Le client et le fournisseur sont en général sur des sites différents
- Le client ne connaît pas au départ l'identité (ou la localisation) du fournisseur
- Le client et le fournisseur peuvent être mobiles
- Le fournisseur peut lui même être réalisé de manière répartie

Le contrat doit néanmoins être maintenu

Schémas asynchrones

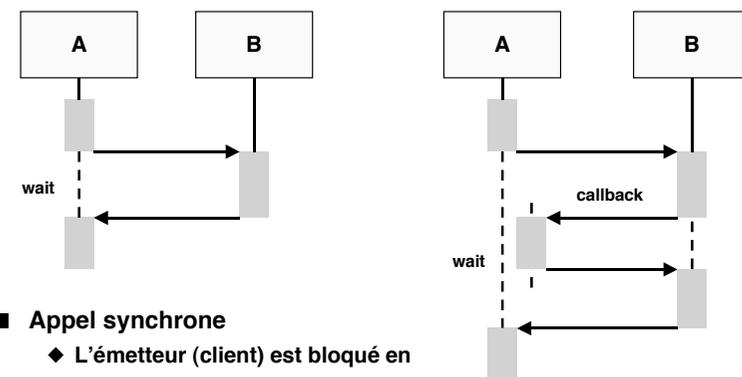
Exécution parallèle de l'émetteur et du récepteur
Couplage faible



◆ Événement-réaction

◆ Messages asynchrones

événements et messages peuvent être ou non mémorisés

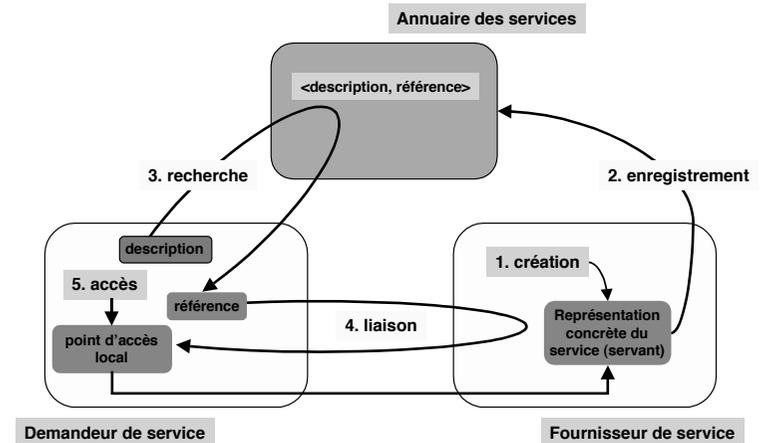
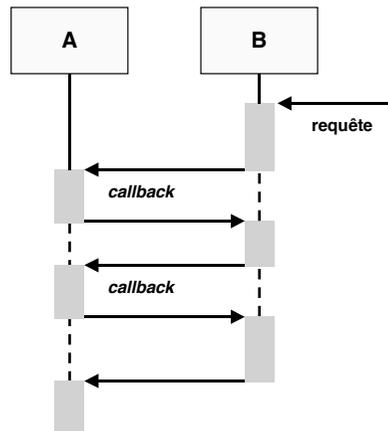


■ Appel synchrone

- ◆ L'émetteur (client) est bloqué en attendant le retour
- ◆ Couplage fort

Avec appel en retour (*callback*)

- **Inversion du contrôle**
 - ◆ Situation où B "contrôle" A
 - ◆ La requête de service pour A est déclenchée depuis l'extérieur



- **Définition** [dépasse le cadre de la conception de logiciel]
 - ◆ Ensemble de règles (définitions d'éléments, principes de composition, règles d'usage) permettant de répondre à une classe de besoins spécifiques dans un environnement donné.
- **Propriétés**
 - ◆ Un patron est élaboré à partir de l'expérience acquise au cours de la résolution d'une classe de problèmes apparentés; il capture des éléments de solution communs
 - ◆ Un patron définit des principes de conception, non des implémentations spécifiques de ces principes.
 - ◆ Un patron fournit une aide à la documentation, par ex. en définissant une terminologie, voire une description formelle ("langage de patrons")

E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture - vol. 1*, Wiley 1996
 D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture - vol. 2*, Wiley, 2000

- **Définition d'un patron**
 - ◆ Contexte : Situation qui donne lieu à un problème de conception; doit être aussi générique que possible (mais éviter l'excès de généralité)
 - ◆ Problème : spécifications, propriétés souhaitées pour la solution; contraintes de l'environnement
 - ◆ Solution :
 - ❖ Aspects statiques : composants, relations entre composants; peut être décrit par diagrammes de classe ou de collaboration
 - ❖ Aspects dynamiques : comportement à l'exécution, cycle de vie (création, terminaison, etc.); peut être décrite par des diagrammes de séquence ou d'état
- **Catégories de patrons**
 - ◆ Conception : petite échelle, structures usuelles récurrentes dans un contexte particulier
 - ◆ Architecture : grande échelle, organisation structurelle, définit des sous-systèmes et leurs relations mutuelles
 - ◆ Idiomatiques: constructions propres à un langage

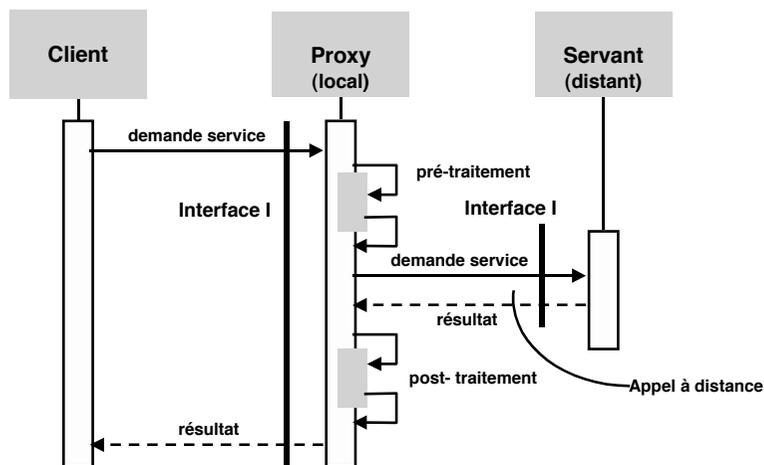
Source: F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture - vol. 1*, Wiley 1996

- **Proxy**
 - ◆ Patron de conception : représentant pour accès à distance
- **Factory (+ Pool)**
 - ◆ Patron de conception : création d'objet
- **Wrapper [Adapter]**
 - ◆ Patron de conception : transformation d'interface
- **Interceptor**
 - ◆ Patron d'architecture : adaptation de service

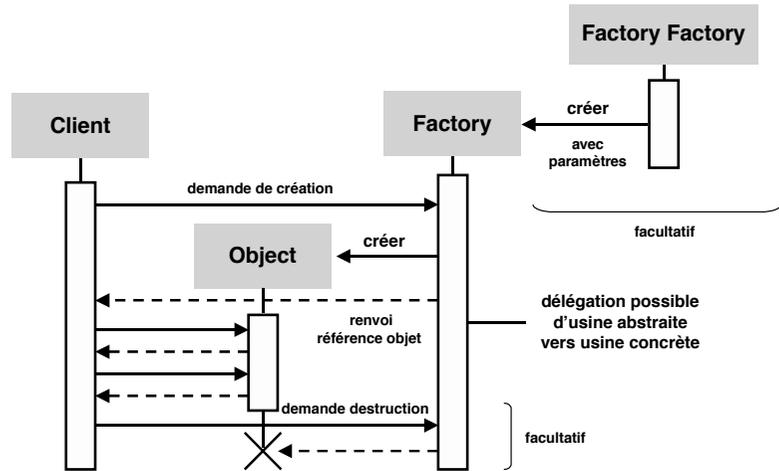
Ces patrons sont d'un usage courant dans la construction d'int logiciel

Nombreux exemples dans toute la suite

- **Contexte**
 - ◆ Applications constituées d'un ensemble d'objets répartis ; un client accède à des services fournis par un objet pouvant être distant (le "servant")
- **Problème**
 - ◆ Définir un mécanisme d'accès qui évite au client
 - ❖ Le codage "en dur" de l'emplacement du servant dans son code
 - ❖ Une connaissance détaillée des protocoles de communication
 - ◆ Propriétés souhaitables
 - ❖ Accès efficace et sûr
 - ❖ Programmation simple pour le client : accès "transparent"
 - ◆ Contraintes
 - ❖ Environnement réparti (pas d'espace unique d'adressage)
- **Solutions**
 - ◆ Utiliser un représentant local du servant sur le site client (isole le client du servant et du système de communication)
 - ◆ Garder la même interface pour le représentant et le servant
 - ◆ Définir une structure uniforme de représentant pour faciliter sa génération automatique

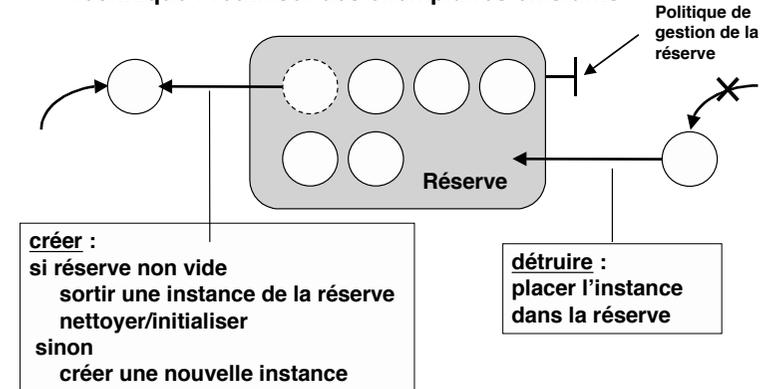


- **Contexte**
 - ◆ Application = ensemble d'objets en environnement réparti
- **Problème**
 - ◆ Créer dynamiquement des instances multiples d'une classe d'objets
 - ◆ Propriétés souhaitables
 - ❖ Les instances doivent être paramétrables
 - ❖ L'évolution doit être facile (pas de décisions "en dur")
 - ◆ Contraintes
 - ❖ Environnement réparti (pas d'espace d'adressage unique)
- **Solutions**
 - ◆ **Abstract Factory** : définit une interface et une organisation génériques pour la création d'objets ; la création effective est déléguée à des usines concrètes
 - ◆ **Abstract Factory** peut être implémentée par **Factory Methods** (méthode de création redéfinie dans une sous-classe)
 - ◆ Pour plus de souplesse, on peut utiliser **Factory Factory** (le mécanisme de création lui-même est paramétré)



■ Idée : réduire le coût de la gestion de ressources

◆ Technique : réutiliser des exemplaires existants



- Gestion de la mémoire
 - ◆ Réserve (*pool*) de zones (plusieurs tailles possibles)
 - ◆ Évite le coût du ramasse-miettes
 - ◆ Évite les copies inutiles (chaînage de zones)
- Gestion des activités
 - ◆ Réserve de *threads*
 - ◆ Évite le coût de la création
- Gestion de la communication
 - ◆ Réserve de connexions
- Gestion des composants
 - ◆ Voir plus loin (réalisation des conteneurs)

■ Contexte

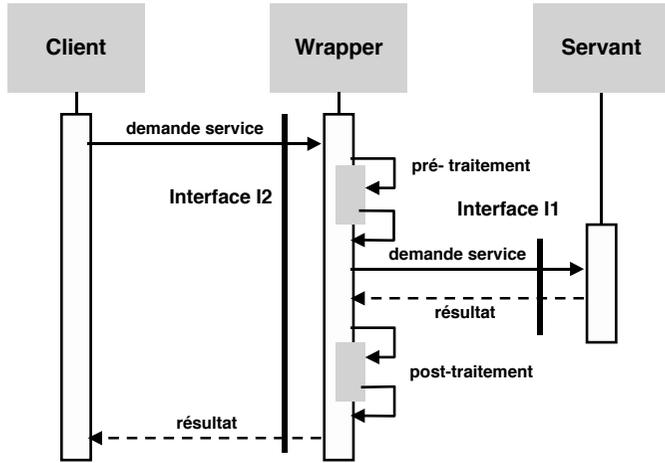
- ◆ Des clients demandent des services ; des servants fournissent des services ; les services sont définis par des interfaces

■ Problème

- ◆ Réutiliser un servent existant en modifiant son interface et/ou certaines de ses fonctions pour satisfaire les besoins d'une classe de clients
- ◆ Propriétés souhaitables : doit être efficace ; doit être adaptable car les besoins peuvent changer de façon imprévisible ; doit être réutilisable (générique)
- ◆ Contraintes :

■ Solutions

- ◆ Le *Wrapper* isole le servent en interceptant les appels de méthodes vers l'interface de celui-ci. Chaque appel est précédé par un prologue et suivi par un épilogue dans le *Wrapper*
- ◆ Les paramètres et résultats peuvent être convertis



■ Contexte

- ◆ Fourniture de services (cadre général)
 - ❖ Client-serveur, pair à pair, hiérarchique
 - ❖ Uni- ou bi-directionnel, synchrone ou asynchrone

■ Problème

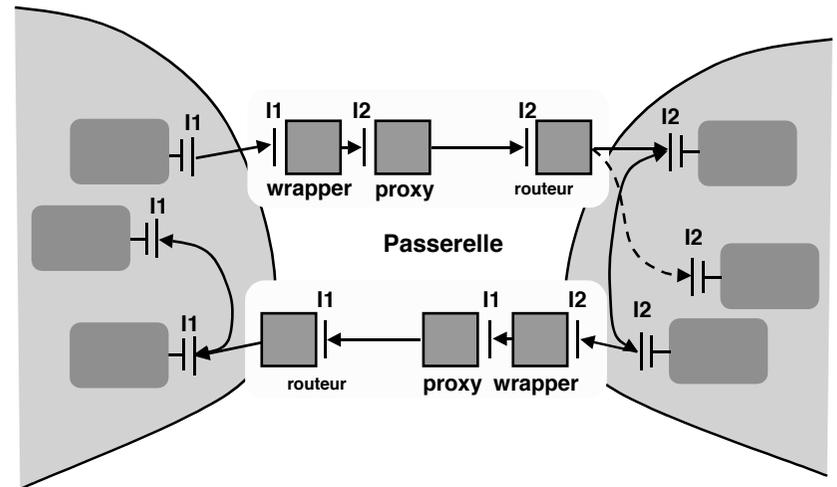
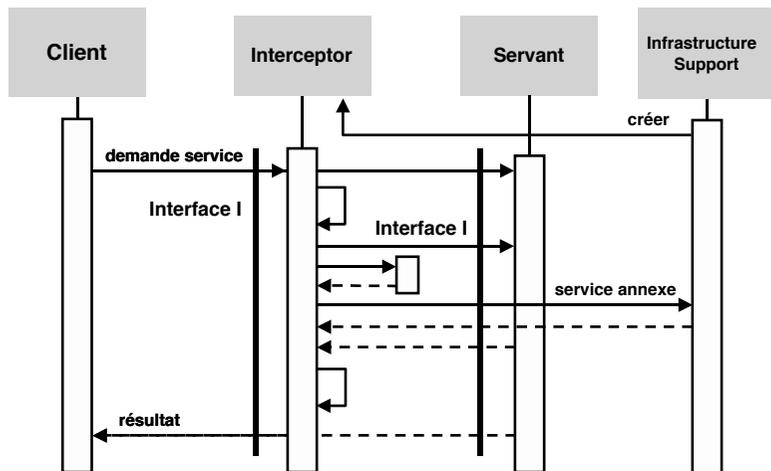
- ◆ Transformer le service (ajouter de nouvelles fonctions), par différents moyens
 - ❖ Interposer une nouvelle couche de traitement (cf. *Wrapper*)
 - ❖ Changer (conditionnellement) la destination de l'appel

◆ Contraintes

- ❖ Les programmes client et serveur ne doivent pas être modifiés
- ❖ Les services peuvent être ajoutés ou supprimés dynamiquement

■ Solutions

- ◆ Créer des objets d'interposition (statiquement ou dynamiquement). Ces objets
 - ❖ interceptent les appels (et/ou les retours) et insèrent des traitements spécifiques, éventuellement fondés sur une analyse du contenu
 - ❖ peuvent rediriger l'appel vers une cible différente
 - ❖ peuvent utiliser des appels en retour



■ Wrapper vs. Proxy

- ◆ *Wrapper* et *Proxy* ont une structure similaire
 - ❖ *Proxy* préserve l'interface ; *Wrapper* transforme l'interface
 - ❖ *Proxy* utilise (pas toujours) l'accès à distance ; *Wrapper* est en général local

■ Wrapper vs. Interceptor

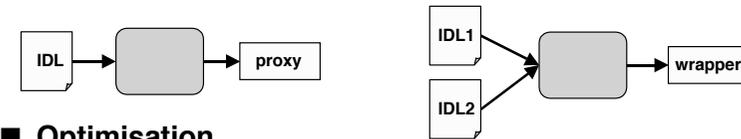
- ◆ *Wrapper* et *Interceptor* ont une fonction similaire
 - ❖ *Wrapper* transforme l'interface
 - ❖ *Interceptor* transforme la fonction (peut même complètement détourner l'appel de la cible initiale)

■ Proxy vs. Interceptor

- ◆ *Proxy* est une forme simplifiée d'*Interceptor*
 - ❖ on peut rajouter un intercepteur à un mandataire (*smart proxy*)

■ Génération automatique

- ◆ À partir d'une description déclarative



■ Optimisation

- ◆ Éliminer les indirections, source d'inefficacité à l'exécution
 - ❖ Court-circuit des chaînes d'indirection
 - ❖ Injection de code (insertion du code engendré dans le code de l'application)
 - ❖ Génération de code de bas niveau (ex. bytecode Java)
 - ❖ Techniques réversibles (pour adaptation)

■ Définition

- ◆ Un canevas est un "squelette" de programme qui peut être réutilisé (et adapté) pour une famille d'applications
- ◆ Il met en œuvre un modèle (pas toujours explicite)
- ◆ Dans les langages à objets : un canevas comprend
 - ❖ Un ensemble de classes (souvent abstraites) devant être adaptées (par ex. par surcharge) à des environnements et contraintes spécifiques
 - ❖ Un ensemble de règles d'usage pour ces classes

■ Patrons et canevas

- ◆ Ce sont deux techniques de réutilisation
- ◆ Les patrons réutilisent un schéma de conception ; les canevas réutilisent du code
- ◆ Un canevas implémente en général plusieurs patrons

■ Objectifs

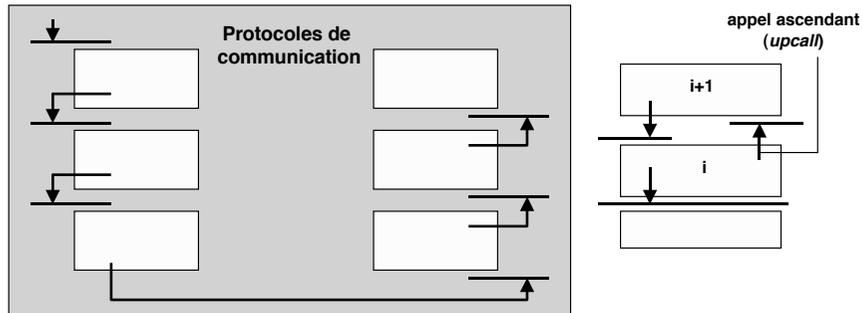
- ◆ Faciliter la construction
 - ❖ La structure reflète la démarche de conception
 - ❖ Les interfaces et les dépendances sont mises en évidence
- ◆ Faciliter l'évolution
 - ❖ Principe d'encapsulation
 - ❖ Échange standard

■ Exemples

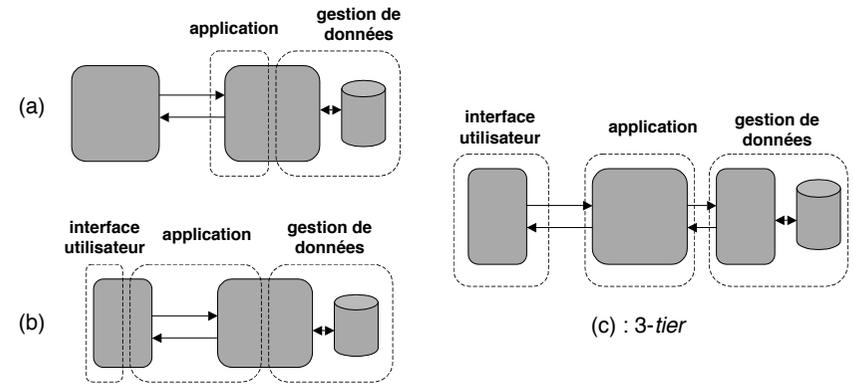
- ◆ Structures multi-niveaux
 - ❖ Décomposition "verticale" ou "horizontale"
- ◆ Canevas pour insertion de composants

■ Hiérarchie de "machines abstraites"

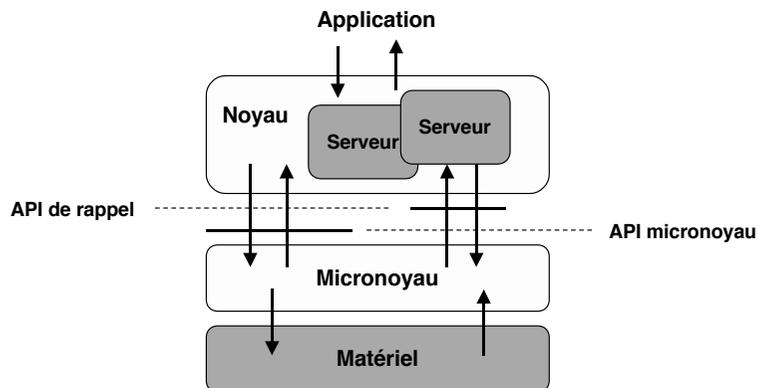
- ◆ La réalisation des niveaux $< i$ est invisible au niveau i
- ◆ Exemple : machines virtuelles (OS multiples, JVM, etc.)



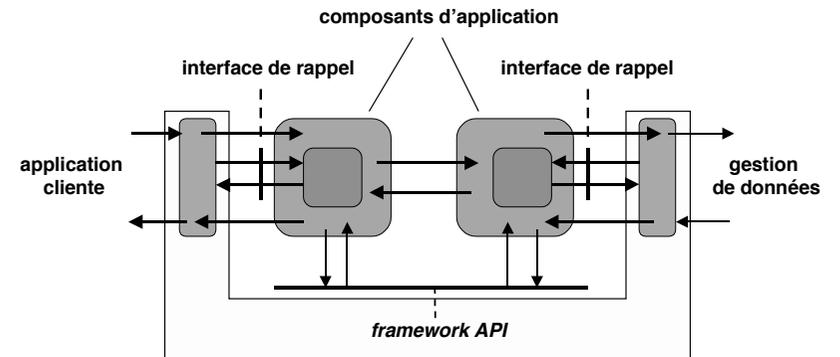
■ Exemple : évolution du schéma client-serveur



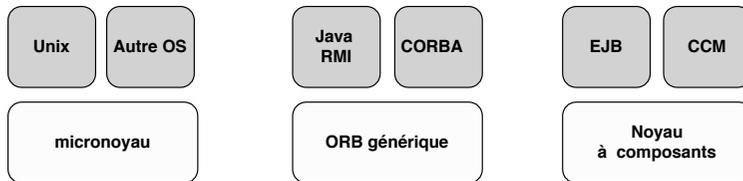
■ Architecture de micro-noyau



■ Architecture d'un canevas pour composants (middle tier)



- **Motivation : réutilisation de mécanismes génériques**
 - ◆ Un canevas de base réalise les entités définies par un modèle abstrait
 - ❖ Critères : générique, modulaire, composable, adaptable
 - ◆ Des "personnalités" utilisent les APIs du canevas de base (y compris appels en retour) pour réaliser des mises en œuvres concrètes du modèle
 - ◆ Avantages : réutilisation, unité conceptuelle, facilité de (re)configuration
 - ◆ Difficulté : efficacité
- **Exemples**

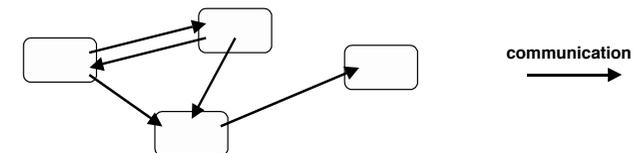


- **Qu'est ce que l'adaptation ?**
 - ◆ Changement de la structure et/ou des fonctions d'une application
 - ◆ Adaptation dynamique : réalisée sans arrêt de l'application
 - **Pourquoi l'adaptation ?**
 - ◆ Pour répondre à l'évolution
 - ❖ Des besoins : nouvelles fonctions, nouvelles qualités
 - ❖ De l'environnement d'exécution (capacités du matériel, mobilité, conditions de communications, perturbations et défaillances, etc.)
 - **Comment?**
 - ◆ Principe : système réflexif (fournit une représentation de son propre fonctionnement, pour l'inspecter ou le modifier)
 - **Techniques**
 - ◆ Techniques ad hoc (intercepteurs)
 - ◆ Protocoles à méta-objets (MOP)
 - ◆ Programmation par aspects (AOP)
- Ces techniques permettent de réaliser des actionneurs dans un système réactif. cf atelier Jade**

Patrons et canevas pour l'exécution répartie

Objets répartis

- **Schéma de base**
 - ◆ Application = ensemble d'objets répartis sur un réseau, communiquant entre eux (1 objet intégralement sur un site)



- Autres modèles (non considérés ici)**
- Objets fragmentés
 - Objets dupliqués
 - Objets mobiles
 - ...



■ Exemples

- ◆ *Java Remote Method Invocation (RMI)* : appel d'objets distants en Java - Sun
- ◆ *Common Object Request Broker Architecture (CORBA)* : support pour l'exécution d'objets répartis hétérogènes - OMG
- ◆ *DCOM, COM+* : *Distributed Common Object Model* - Microsoft

■ Schéma commun : ORB (*Object Request Broker*)

- ◆ Modèle de base : client-serveur
- ◆ Identification et localisation des objets
- ◆ Liaison entre objets clients et serveurs
- ◆ Exécution des appels de méthode à distance
- ◆ Gestion du cycle de vie des objets (création, activation, ...)
- ◆ Services divers (sécurité, transactions, etc.)



■ Schéma d'interaction

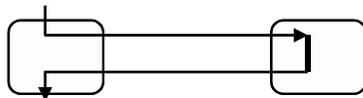
- ◆ Communication
- ◆ Synchronisation
- Désignation et localisation des objets
 - ◆ Identification et référence
- Liaison
 - ◆ Établissement de la chaîne d'accès
- Cycle de vie
 - ◆ Création, conservation, destruction des objets
- Mise en œuvre (réalisation, services)

Notre objectif

- élaborer des patrons pour les aspects ci-dessus
- proposer un canevas pour la structure d'un ORB



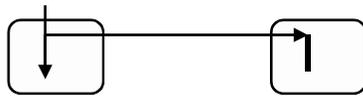
■ Synchrones



Couplage fort

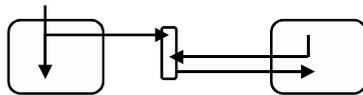
RMI, CORBA, COM, ...

■ Asynchrones



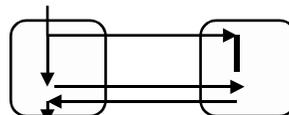
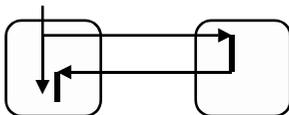
Couplage faible

Événements



Files de messages

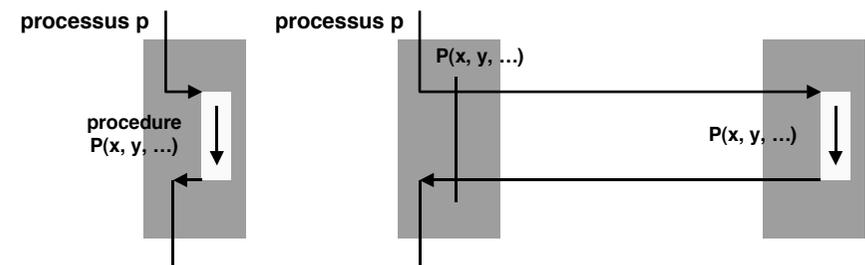
■ Semi-synchrones



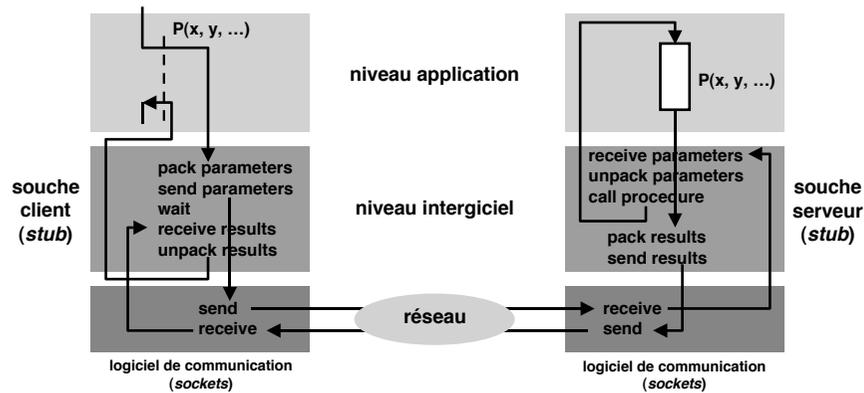
Combinaisons synchrone-asynchrone



■ L'appel de procédure à distance (RPC), un outil pour construire des applications client-serveur



L'effet de l'appel doit être identique dans les deux situations. Cela est impossible à réaliser en présence de défaillances



■ Réalisation de l'appel de procédure à distance

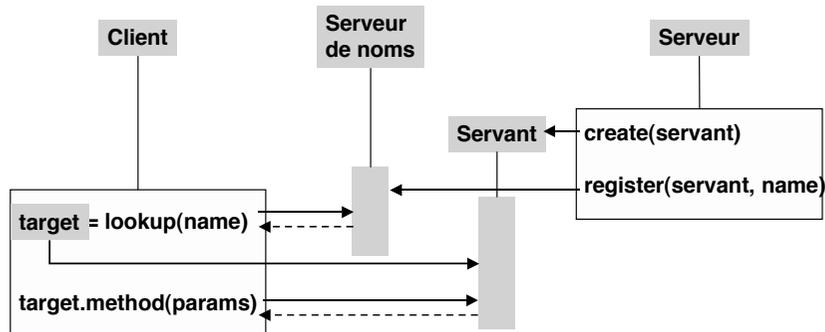
■ Pourquoi les objets répartis

- ◆ Avantages d'un modèle à objets pour la programmation
- ◆ L'encapsulation se prête bien à la répartition (objet = unité naturelle de répartition)
- ◆ Réutilisation de l'existant (par *wrappers*)

■ Différences avec RPC

- ◆ Encapsulation de données
- ◆ Création dynamique d'objets
 - ❖ Donc liaison dynamique
- ◆ Intégration de services
 - ❖ Persistance, duplication, transactions, etc.

Les étapes d'un appel d'objet réparti (vues du programmeur)



Connaissances requises :

le client et le serveur connaissent le serveur de noms
 le client et le serveur s'accordent sur le nom *name*
 le client connaît l'interface du servant

Les étapes d'un appel réparti (vues du système)

En partant de la fin...

Pour réaliser l'appel réparti, il faut avoir établi une chaîne d'accès entre le client et le servant



L'opération de liaison (*binding*) est la création de cette chaîne d'accès (également appelée "objet de liaison")



■ Désignation

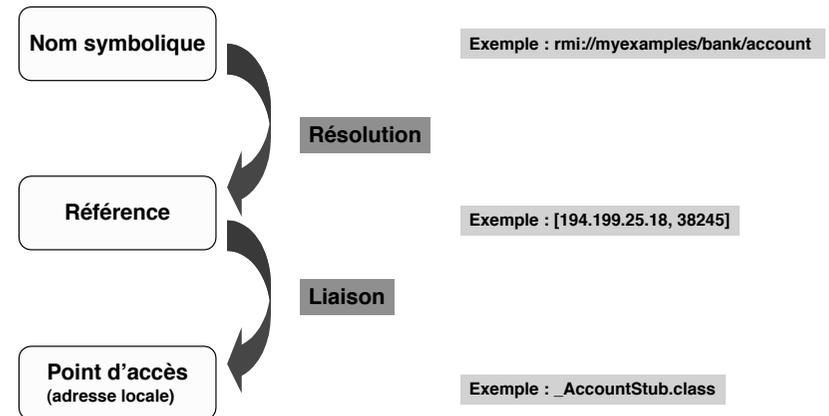
- ◆ Associer des noms à des objets
- ◆ Retrouver un objet à partir de son nom

■ Liaison

- ◆ Créer une chaîne d'accès à un objet (à partir d'un nom)

■ Spécificité de la liaison répartie

- ◆ En centralisé (très schématiquement)
 - ❖ 2 sortes de noms : nom symboliques, adresses
 - ❖ Liaison = recherche de l'adresse (souvent avec indirection)
- ◆ En réparti
 - ❖ "Adresse" = référence (exemple : [adresse IP, n° de port])
 - ❖ Mais référence ≠ chaîne d'accès !



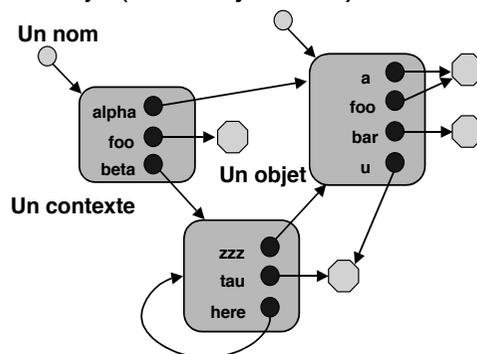
■ Deux sortes de noms

- ◆ Identificateurs : distinguer un objet des autres
- ◆ Références : localiser un objet (en vue d'y accéder)

■ Noms contextuels

Un espace plat est inutilisable
recherche inefficace
pas de localité

Contexte = partie de l'univers
Graphe de contextes
(souvent hiérarchique : arbre, etc.)



■ Résoudre un nom (dans un contexte)

- ◆ À partir du nom, trouver l'objet
- ◆ Processus récursif

`target = context.resolve(name)` [ou `name.resolve()`]

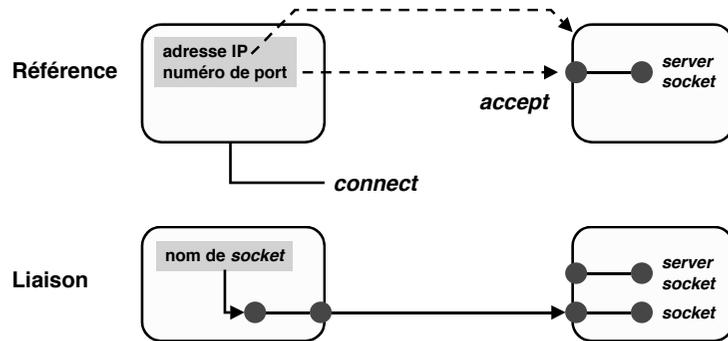
3 issues possibles pour target

- ❖ Une valeur typée : c'est l'objet
- ❖ Une référence (ex : adresse) => l'objet est localisé
- ❖ Un autre nom (dans un autre contexte) => on rappelle *resolve*

■ Lier un nom

- ◆ À partir du nom, construire une chaîne d'accès à l'objet
- ◆ Rappel : en réparti, résolution ≠ liaison !
 - ❖ Exemple : une référence (adresse IP, n° de porte) ne suffit pas pour accéder à l'objet

■ sockets



■ La liaison est établie en 2 phases

■ Côté serveur (*export*)

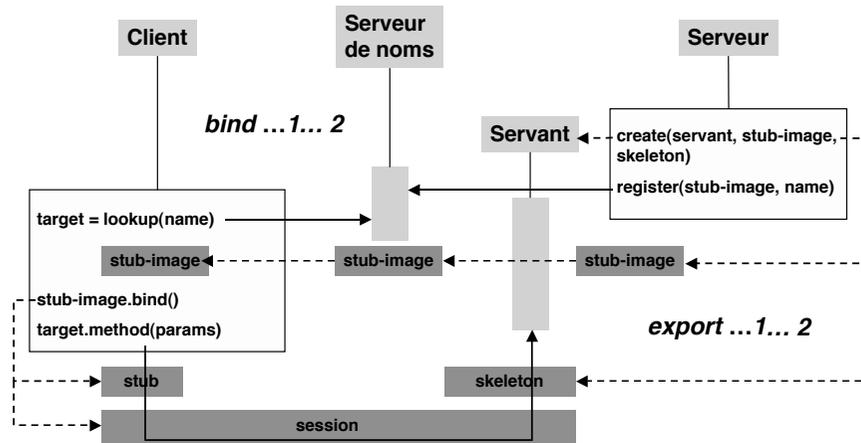
- ◆ “publication” de l’objet à lier (identification)
- ◆ préparation de certains éléments de la liaison

■ Côté client (*bind*)

- ◆ établissement de la liaison par création et assemblage des constituants de l’objet de liaison

■ Exemple

- ◆ La connexion par sockets peut être décrite en ces termes
 - ❖ *accept* = *export*
 - ❖ *connect* = *bind*

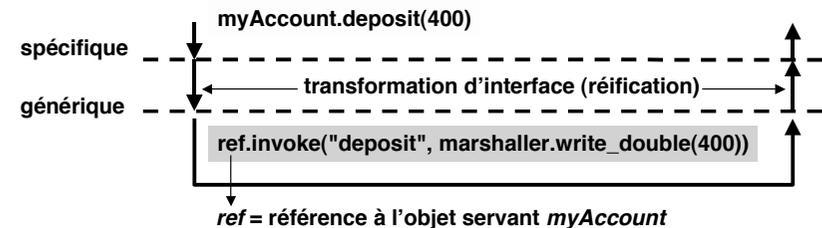


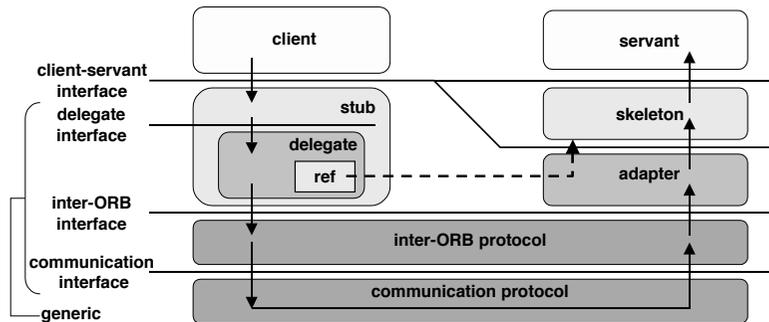
■ L’interface “de bout en bout” est spécifique

- ◆ Interface d’un objet défini par l’application
- ◆ Exprimée dans un IDL, pour faciliter la portabilité

■ Les interfaces intermédiaires (internes à l’ORB) ont intérêt à être génériques

- ◆ Pour faciliter les échanges de composants d’ORB
- ◆ Pour faciliter l’interopérabilité entre ORBs





Transformation d'interface
 côté client, dans la souche (vers le "délégué", ou souche générique)
 côté serveur, dans un adaptateur

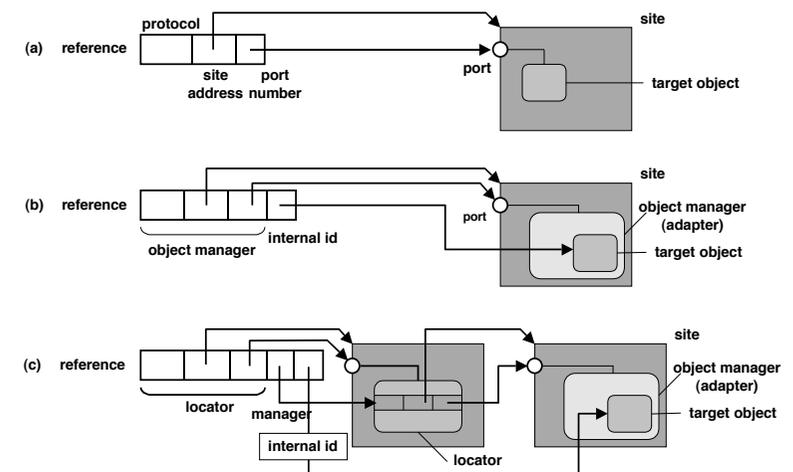
- **Gestion des objets servants**
 - ◆ Référentiel des implémentations dans CORBA
- **Gestion des références d'objets**
 - ◆ Créer une référence pour un objet (à la création de l'objet)
 - ◆ Trouver un objet, connaissant sa référence
- **Gestion des activités côté serveur**
 - ◆ Activer un objet (lui associer un *thread* pour son exécution)
- **Exemple : le POA (Portable Object Adapter) de CORBA (OMG)**
 - ◆ Permet d'isoler les politiques de gestion d'objets
 - ❖ Persistance, politique d'activation, format de références, etc.

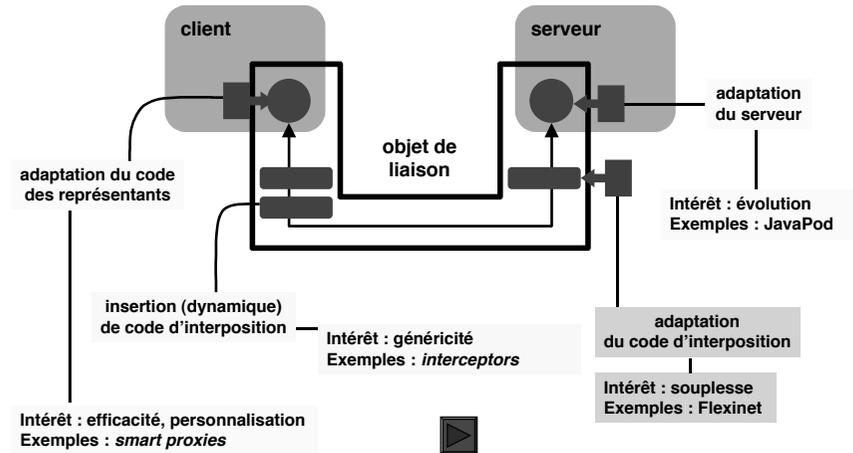
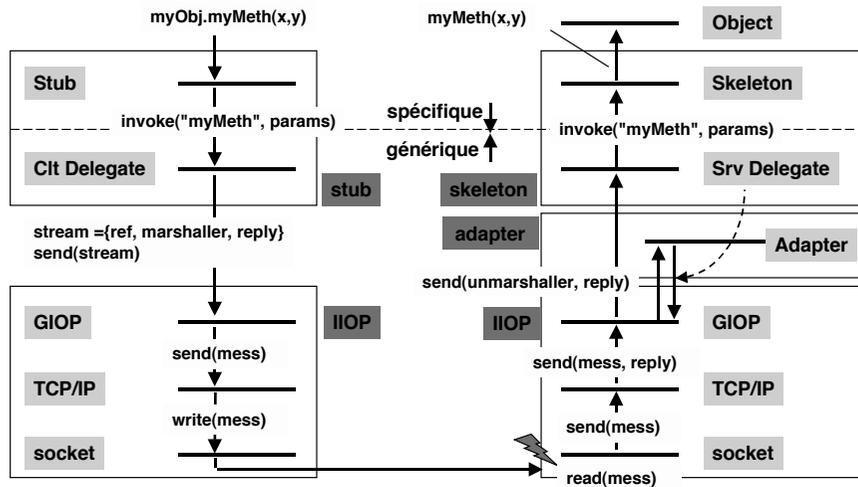
■ GIOP : *General Inter-ORB Protocol*

- ◆ Définit une interface et un protocole générique pour l'appel d'objets distants sur une couche de transport
 - ❖ Représentation commune de données (*Common Data Representation, CDR*)
 - ❖ Format standard de référence d'objet (*Interoperable Object Reference, IOR*)
 - ❖ Format des messages
 - ❖ Contraintes sur la couche de transport

■ IIOP : *Internet Inter-ORB Protocol*

- ◆ La réalisation "standard" de GIOP
 - ❖ GIOP sur TCP/IP





■ Définitions

- ◆ Méthodes et outils permettant à un ensemble d'entités de coopérer à une tâche commune
- ◆ Modèle de coordination, définit :
 - ❖ les entités coopérantes (processus, activités, "agents", ...)
 - ❖ le support (médium) de coordination : véhicule de l'interaction
 - ❖ les règles de coordination : primitives, patrons d'interaction

■ Domaine d'application

- ◆ Couplage faible (évolution indépendante des entités)
- ◆ Structure très dynamique (les entités peuvent rejoindre/quitter le système à tout instant)
- ◆ Hétérogénéité (système, environnement, administration)
- ◆ Condition requise : capacité de croissance

■ Contexte

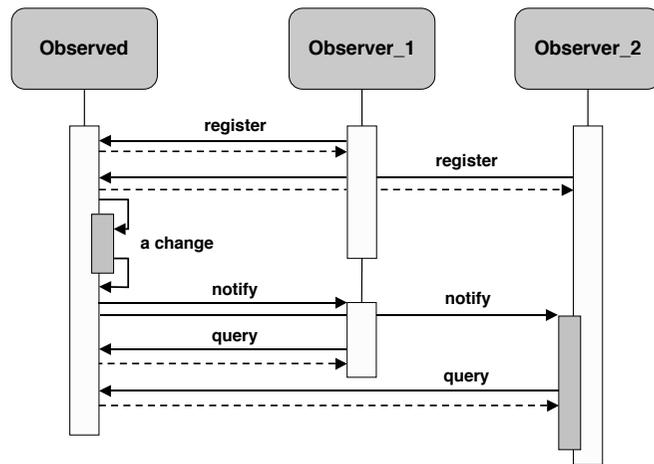
- ◆ Des objets "observés", dont l'état (visible) évolue au cours du temps
- ◆ Des objets "observateurs"

■ Problème

- ◆ Permettre aux observateurs d'être informés de l'évolution des objets observés
- ◆ Propriétés souhaitables
 - ❖ Requérir un effort minimal de la part des observateurs
 - ❖ Garantir l'indépendance mutuelle des observateurs
 - ❖ Permettre l'évolution dynamique (arrivée-départ des observateurs et observés)
- ◆ Contraintes
 - ❖ Passage à l'échelle

■ Solution

- ◆ Les observateurs enregistrent leur intérêt auprès des observés
- ◆ Les observés notifient aux observateurs enregistrés les événements pertinents, de manière asynchrone



■ Les limitations de *Observer*...

- ◆ Forte charge sur les objets observés (gèrent les observateurs et répondent aux consultations)
- ◆ Manque de sélectivité du schéma notification-consultation (l'observateur reçoit toutes les notifications de changement)

■ ... et deux réponses

◆ *Publish-Subscribe*

- ❖ Deux "rôles" : abonné (*subscriber*) et émetteur (*publisher*)
- ❖ Une entité d'intermédiation (médiateur)
- ❖ Abonnement par sujet ou par contenu

◆ Espace partagé

- ❖ Le médium de coordination est un ensemble de tuples
- ❖ Opérations : déposer, consulter avec filtrage (destructivement ou non)

■ Contexte

- ◆ Ensemble d'entités devant se coordonner par émission d'événements et réaction à ces événements (autre formulation de *Observer*)

■ Problème

◆ Propriétés souhaitables

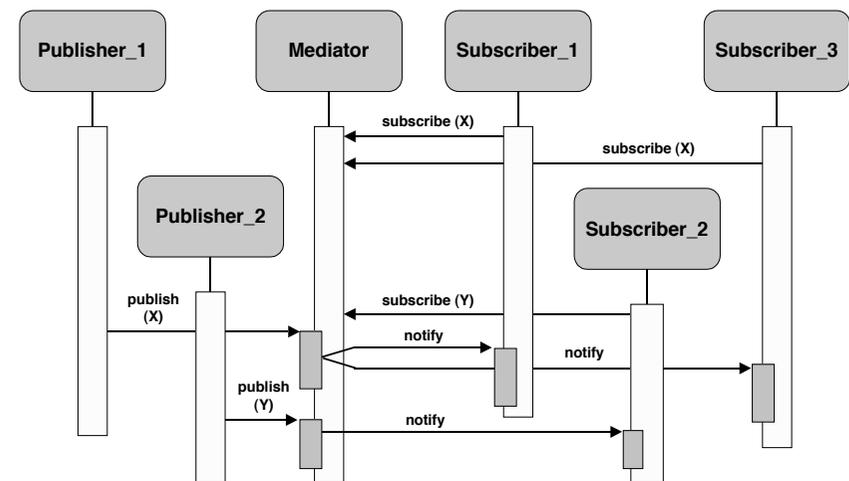
- ❖ Comme *Observer* (indépendance, évolution dynamique)
- ❖ Pas de rôle prédéfini
- ❖ Sélectivité sur la nature des événements

◆ Contraintes

- ❖ Passage à l'échelle
- ❖ Propriétés diverses : tolérance aux fautes, transactions, persistance, ordre

■ Solution

- ◆ Deux "rôles" : abonné (*subscriber*) et émetteur (*publisher*)
- ◆ Une entité d'intermédiation (médiateur)
- ◆ Abonnement par sujet (statique) ou par contenu (dynamique)





- **Message-Oriented Middleware (MoM)**
 - ◆ Regroupe *Publish-Subscribe* et *Message Queues*
 - ◆ Abonnement par sujet : nombreuses réalisations industrielles
 - ❖ Tibco, Websphere, ... ScalAgent (JORAM) -> cf. atelier
 - ❖ Un standard pour l'interface : JMS (*Java Messaging System*)
 - ◆ Abonnement par contenu : prototypes de recherche
 - ❖ Gryphon (IBM Research)
 - ❖ Siena (Univ. Colorado)
- **Espace partagé**
 - ◆ Modèle : Linda (espace de tuples), projection dans divers langages
 - ◆ Réalisation : Jini (Sun)
 - ❖ Utilisation : découverte de ressources



Patrons et canevas pour la composition



- **Notion de composant logiciel (unité de construction)**
 - ◆ Nécessité perçue très tôt [McIlroy 1968]...
 - ◆ ... mais conception et réalisation n'ont pas suivi
 - ◆ Les objets sont une première approche pour la décomposition
 - ❖ Encapsulation (séparation interface-réalisation)
 - ❖ Mécanismes de réutilisation (héritage, délégation)
- **Limitations des objets (comme unité de construction)**
 - ◆ Pas d'expression explicite des ressources requises par un objet (autres objets, services fournis par l'environnement)
 - ◆ Pas de vue globale de l'architecture d'une application
 - ◆ Pas de possibilité d'expression de besoins "non fonctionnels", par ex. persistance, performances, etc.)
 - ◆ Peu d'outils pour le déploiement et l'administration



- **Pour mettre en œuvre des composants, il faut**
 - ◆ Un modèle de composants, qui définit les entités et leur mode d'interaction et de composition
 - ❖ Deux niveaux de modèles
 - ▲ Abstrait (définition des entités de base et de leurs relations)
 - ▲ Concret (représentation particulière du modèle abstrait)
 - ◆ Une infrastructure à composants, qui met en œuvre le modèle et permet de construire, déployer, administrer et exécuter des applications conformes au modèle
- **Situation actuelle**
 - ◆ Les infrastructures industrielles (EJB, .NET, OSGi) n'ont pas de base formelle
 - ◆ Des modèles issus de la recherche commencent à être proposés (cf. cours Fractal), ainsi que des infrastructures prototypes (cf. atelier AOKell)



Que doit fournir un modèle de composants ?



- Encapsulation
 - ◆ Interfaces = seule voie d'accès, séparation interface-réalisation
 - ◆ Possibilité de définir des interfaces multiples
- Composabilité
 - ◆ Dépendances explicites (expression des ressources fournies et requises)
 - ◆ Composition hiérarchique (un assemblage de composants est un composant)
- Capacité de description globale
 - ◆ Si possible exprimée formellement (langage de description)
- Réutilisation et évolution
 - ◆ Modèles génériques de composants
 - ◆ Adaptation (interface de contrôle)
 - ◆ Reconfiguration



Que doit fournir une infrastructure à composants ?



- Couverture du cycle de vie
 - ◆ Non limitée aux phases de développement et d'exécution
 - ◆ Administration et maintenance
 - ❖ Déploiement : installation et activation des composants
 - ❖ Surveillance : collecte et intégration de données, tenue à jour de la configuration
 - ❖ Contrôle : réaction aux événements critiques (alarme, surcharge, détection d'erreur)
 - ❖ Maintien de la disponibilité de l'application
 - ❖ Évolution (redéploiement, reconfiguration) pour réagir à l'évolution des besoins et de l'environnement
- Services communs
 - ◆ Propriétés non-fonctionnelles (persistance, transactions, QoS)



Paradigmes de la composition



- Éléments de la composition
 - ◆ Composant. Unité de composition et de déploiement, qui remplit une fonction spécifique et peut être assemblé avec d'autres composants. À cet effet, il porte une description des interfaces requises et fournies.
 - ◆ Connecteur. Élément permettant d'assembler des composants en utilisant leurs interfaces fournies et requises. Remplit 2 fonctions : liaison et communication.
 - ◆ Configuration. Un assemblage de composants (peut ou non, selon le modèle, être lui-même un composant).

N.B.1 La notion de composant est préservée à l'exécution pour faciliter

- l'adaptation dynamique
- la répartition

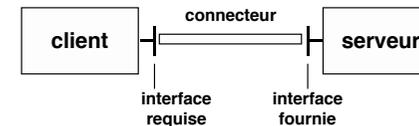
N.B.2 La différence entre composant et connecteur est une différence de fonction, non de nature : un connecteur est lui-même un composant



Exemples de schémas de composition (1)



Client-serveur

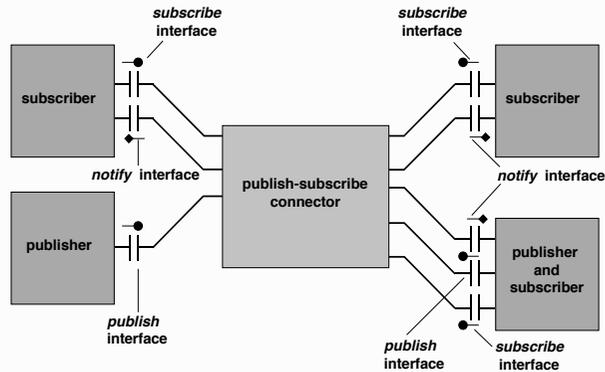


Une autre vue : le connecteur comme composant



Une autre vue : le connecteur comme composant composite





■ Pourquoi une description formelle ?

- ◆ Donner un support concret aux notions d'architecture logicielle
- ◆ Permettre un traitement formel (vérification, preuves)
- ◆ Permettre d'automatiser (ou d'assister) des opérations globales (mettant en jeu des configurations)
 - ❖ Déploiement
 - ❖ Reconfiguration
- ◆ Servir d'aide à la documentation (description globale, visualisation)

■ Situation actuelle

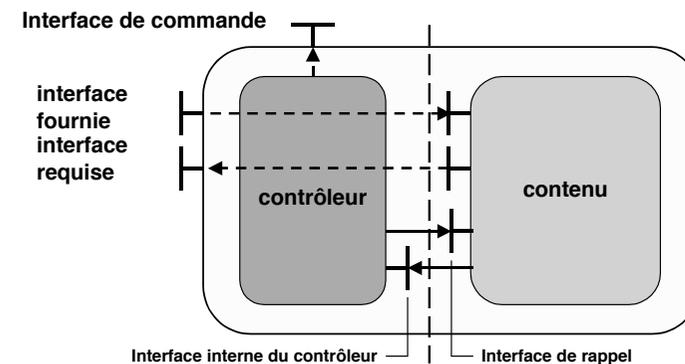
- ◆ Notion d'ADL (*Architecture Description Language*)
- ◆ Pas de standard reconnu (des tentatives)
 - ❖ ACME, ADML, Xarch, UML-2
- ◆ Vers l'utilisation de XML comme support commun (invisible) ?

■ Description des composants

- ◆ Définition des interfaces (fournies, requises)
- ◆ Mode d'interaction (synchrone, asynchrone, "porte", etc.)
- ◆ Signatures (types), contrats, annotations

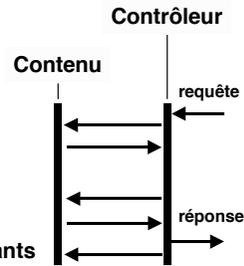
■ Description des configurations

- ◆ Association interfaces fournies-requises
 - ❖ Directe (invisible, réalisée par édition de liens)
 - ❖ Via un connecteur : description, "rôle" = interface
- ◆ Composition hiérarchique (si le modèle l'autorise)
 - ❖ Interfaces exportées/importées
- ◆ Interface graphique (visualisation, action)
- ◆ Aspects dynamiques (expérimental)



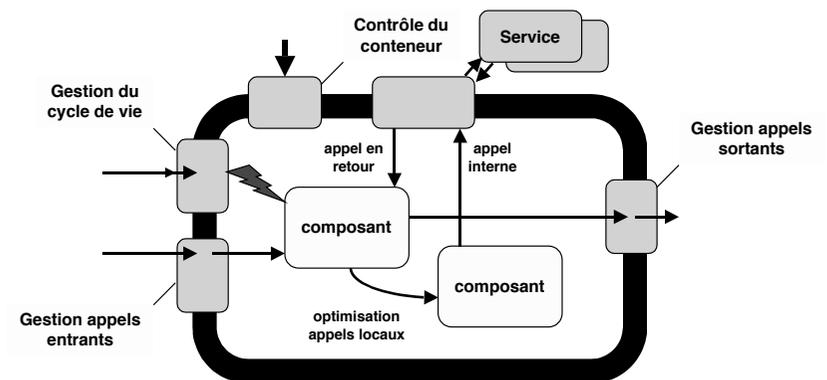
La séparation entre contrôleur et contenu est motivée par la séparation des préoccupations : isoler la partie purement fonctionnelle (propre à l'application)

- **Gestion du cycle de vie**
 - ◆ Création, destruction
 - ◆ Activation, passivation
- **Gestion des composants inclus**
 - ◆ ... si le modèle comporte des composants composites
- **Liaison**
 - ◆ Connexion avec d'autres composants
- **Médiation**
 - ◆ Gestion des interactions avec d'autres composants
 - ◆ Médiation pour la fourniture de services externes
 - ◆ Met en œuvre l'inversion du contrôle
- **Autres opérations réflexives**
 - ◆ ... si le modèle le permet

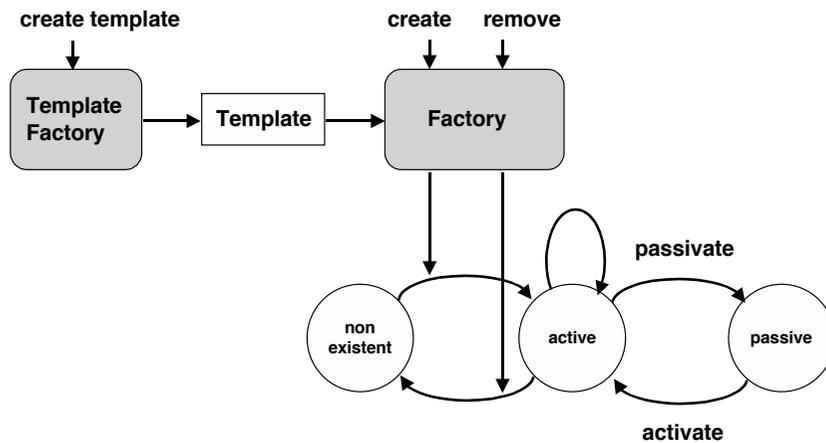


- **Infrastructures à conteneurs**
 - ◆ Canevas de base pour le support des composants
 - ◆ Séparation des préoccupations
 - ❖ La partie "contenu" réalise les fonctions de l'application
 - ❖ La partie "conteneur" fournit les fonctions de l'infrastructure
 - ◆ Fonctions du conteneur
 - ❖ Mise en œuvre des fonctions du contrôleur (gestion, médiation)
 - ❖ Allocation des ressources aux composants
 - ❖ Réalisation des aspects "non fonctionnels" (services communs)
- **Exemples (sous des formes diverses)**
 - ◆ *Enterprise Java Beans (EJB)*, cf cours J2EE
 - ◆ *CORBA Component Model (CCM)*
 - ◆ *Julia* (réalisation du modèle Fractal), cf cours Fractal
 - ◆ *OSGi*, cf cours OSGi
 - ◆ *Avalon* (Apache-Djakarta)

- **Mise en œuvre du contrôleur de composants**
 - ◆ Conteneur : infrastructure de gestion pour un ou plusieurs composants
 - ❖ Par composant ou par "type"
 - ◆ Rappel des fonctions : cycle de vie, médiation, gestion de services, composition (si composants composites)
 - ◆ Génération automatique des conteneurs à partir
 - ❖ des descriptions d'interfaces des composants (cf. talons)
 - ❖ de paramètres de configuration fournis
 - ◆ "Contrats" entre le conteneur et les composants inclus
 - ❖ Interface interne du contrôleur
 - ❖ Interfaces de rappel des composants



Illustrations spécifiques dans les autres cours : EJB, Fractal, OSGi, ...



■ Mécanismes pour économiser les ressources

◆ Activation-passivation

- ❖ *passivate* : Élimine le composant de la mémoire si non utilisé (en stockant son état) - appelé par le conteneur
- ❖ *activate* : Restaure l'état du composant, qui peut être réutilisé

◆ Pool d'instances

- ❖ Le conteneur maintient une réserve (*pool*) fixe d'instances de composants qui peuvent donner vie à des composants "virtuels"
- ❖ Un composant virtuel incarné par une instance de la réserve devient utilisable, à condition de charger son état => méthodes de rappel pour sauver- restaurer l'état
- ❖ Évidemment plus simple pour les composants sans état

◆ Différences entre passivation et *pool*

- ❖ Passivation : géré par le conteneur, pas de méthodes de gestion de l'état

■ Maison (*home*)

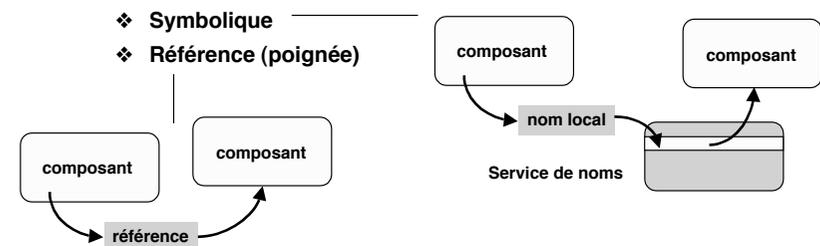
- ◆ Partie du conteneur (visible de l'extérieur) qui gère le cycle de vie des composants qu'il inclut (du même type)
 - ❖ Engendré automatiquement (à partir d'un IDL)
- ◆ Fonctions
 - ❖ Créer / détruire les composants : implémente *Factory*, utilise les mécanismes d'économie (*pool*, passivation)
 - ❖ Gérer les composants pendant leur existence (en particulier nommer, localiser)
- ◆ Interface fournie
 - ❖ *create, find, remove*
 - ❖ Utilise une interface de rappel (à implémenter par le programmeur)

■ Principe : désignation contextuelle

◆ Désignation dans un contexte global

- ❖ Service de noms (CORBA, J2EE/JNDI, ...)
- ❖ Clé pour les objets persistants (clé primaire SGBD)

◆ Désignation locale pour éviter noms globaux "en dur"

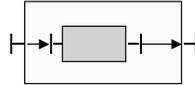




Les principes généraux de la liaison s'appliquent ; quelques spécificités.

■ Plusieurs types de liaison selon localité

- ◆ Entre composants de même niveau dans un même espace : référence dans cet espace (exemple : référence Java)
- ◆ Entre composants dans des espaces différents : objet de liaison (avec éventuellement optimisations locales)
 - ❖ Exemple EJB :
 - ▲ Dans un même conteneur : LocalObject
 - ▲ Entre deux conteneurs : RemoteObject
 - ▲ Entre client et serveur : RMI (stub + skeleton)
- ◆ Entre composants inclus (dans les modèles qui l'autorisent)
 - ❖ Exportation - Importation



■ Principe

- ◆ Le conteneur réalise une médiation pour l'appel d'une méthode d'un composant, via un objet intermédiaire (intercepteur)
- ◆ Comme le reste du conteneur, l'intercepteur est engendré automatiquement (fait partie de la chaîne de liaison)
- ◆ Le conteneur fournit en général une optimisation locale pour les appels entre composants qu'il contient

■ Fonctions réalisées

- ◆ Lien avec gestion de ressources (activation d'un composant passivé)
- ◆ Sécurité
- ◆ Fonctions supplémentaires "à la carte" (intercepteurs programmables)



Pour exécuter une application (composée), il faut :

- ❖ choisir les composants à utiliser (par ex. version) et fixer les paramètres éventuels
- ❖ vérifier la cohérence du système (dépendances entre composants)
- ❖ déterminer les sites sur lesquels l'application doit être installée, placer les composants sur ces sites, établir les liaisons
- ❖ lancer l'exécution des composants dans un ordre approprié

Configuration : les deux premières étapes
 Déploiement : les deux suivantes

Le terme de "déploiement" inclut souvent la phase de configuration
 Reconfiguration : modification d'une application déjà déployée



Le déploiement a longtemps été considéré comme une opération accessoire

- ❖ Pas de modèle conceptuel
- ❖ Réalisation ad hoc, par des scripts écrits à la main

Le déploiement a pris une place importante avec le développement d'applications réparties complexes et à grand nombre de composants

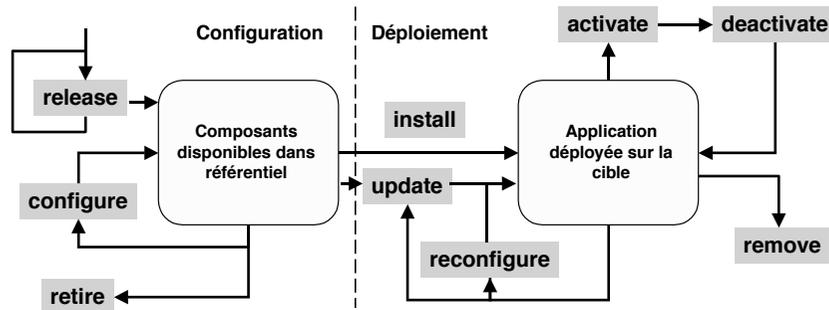
Le traitement ad hoc est inadapté : les erreurs de configuration ou de déploiement sont la cause majeure des défaillances de services sur l'Internet

Le déploiement est maintenant l'objet d'une forte activité

- ❖ Recherche (modèles et méthodes)
- ❖ Réalisations prototypes
- ❖ Premières tentatives de normalisation (OMG)

Éléments du processus de déploiement

- ❖ Référentiel (*repository*) : base de données contenant les composants à déployer
- ❖ Cible : infrastructure matérielle et logicielle sur laquelle l'application va être déployée



Motivation principale : capacité d'adaptation. Il faut donc pouvoir :

- ❖ Appliquer les changements à toutes les étapes du cycle de vie
- ❖ Retarder les changements jusqu'au dernier moment
- ❖ Pouvoir modifier les politiques d'adaptation

Problèmes :

- ❖ Gérer la multiplicité des configurations
 - Les systèmes de gestion de versions s'appliquent surtout aux composants individuels
- ❖ Préserver la cohérence
 - Nécessite d'explicitier toutes les dépendances (certaines sont "invisibles", ex. bibliothèques)
 - Peut impliquer la coexistence de plusieurs versions d'un composant

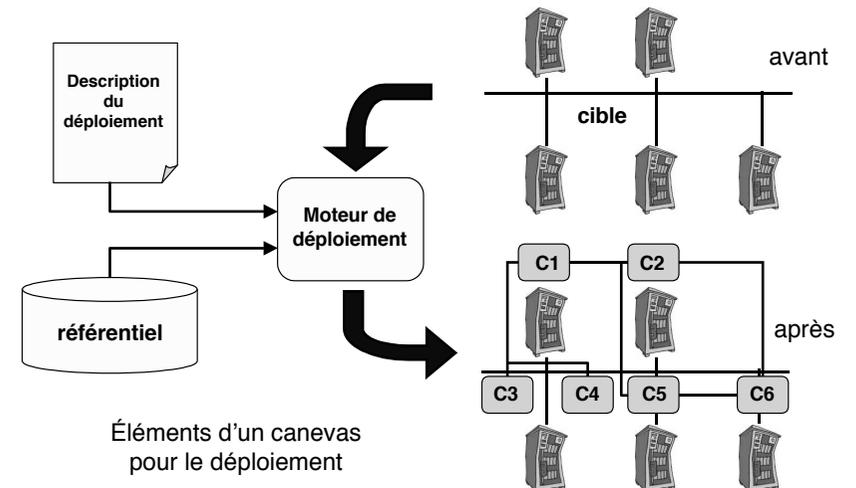
Déploiement dirigé par l'architecture

Motivation : maintenir une représentation unique du système (ou de l'application) pendant tout le cycle de vie

- ❖ La configuration et le déploiement d'un système sont décrits par une spécification fondée sur le **schéma architectural** de celui-ci
- ❖ Cette spécification dirige le processus de configuration et déploiement

C'est une application du principe de séparation des préoccupations

- ❖ La spécification du déploiement est séparée du code de l'application
- ❖ La spécification est déclarative et ne fixe pas les modalités de l'exécution du déploiement





La description du déploiement doit spécifier

- ❖ La définition du système à déployer (en utilisant la désignation des composants dans le référentiel). Analogue d'un ADL
- ❖ Les besoins et les contraintes : localisation des composants (absolue ou en termes de co-localisation, etc.)
- ❖ (éventuellement) les outils nécessaires (ex : scripts préexistants ou créés dynamiquement, etc.)

Travaux actuels : étendre un ADL existant, avec des éléments dynamiques



Le moteur de déploiement est essentiellement un moteur de *workflow*

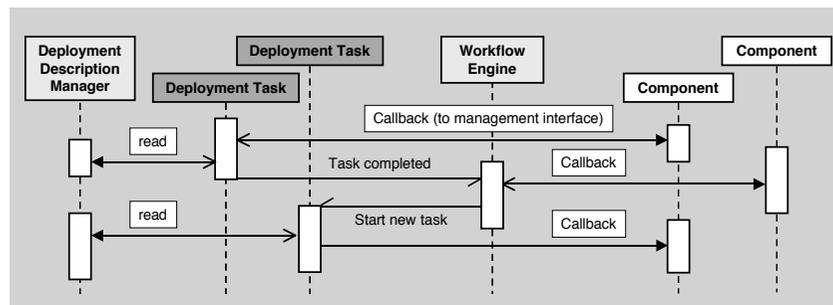
- ❖ *Workflow* (graphe de tâches) : spécification d'un ensemble de tâches, avec des contraintes de dépendance mutuelle (précédence, exécution parallèle, etc.)
- ❖ Exemples de tâches élémentaires de déploiement :
 - Choisir une version de composant et la configurer
 - Charger un composant sur un site du système cible
 - Créer une liaison entre composants (sur un même site ou sur des sites différents)
 - Lancer un composant en activant un point d'entrée spécifié

Exemple de travaux actuels : génération automatique (partielle) du moteur de déploiement à partir de règles de haut niveau (MDA)



Exécution du déploiement (en tant que moteur de *workflow*)

- ❖ L'exécution est pilotée par la description
- ❖ Les tâches impliquent souvent l'exécution d'un *callback* sur une interface de gestion d'un composant
- ❖ C'est un exemple d'inversion du contrôle



■ Fondements théoriques

- ◆ Objectif : vérification, preuve
- ◆ Voir Fractal, atelier Dream

■ Composition

- ◆ Aspects dynamiques de l'ADL
- ◆ Liaison avec déploiement, reconfiguration (voir atelier Jade)

■ Infrastructures adaptables

- ◆ Conteneurs ouverts
- ◆ Intercepteurs programmables
- ◆ Utilisation des aspects (voir atelier AOKell)



- L'architecture logicielle est un aspect fondamental de l'intergiciel et des applications réparties
- Les patrons et canevas permettent d'intégrer, de mettre en forme et de transmettre l'expérience acquise
 - ◆ Il est important d'identifier les "bons" patrons architecturaux
 - ❖ Architecture globale vs constructions locales
 - ◆ Attention à la multiplication...
- Les architectures d'intergiciel ne sont pas encore assez ouvertes et génériques
 - ◆ Compromis ouverture-efficacité
- Le logiciel libre (*open source*) est important pour la diffusion, la compréhension et l'amélioration des architectures d'intergiciel
- La documentation d'architecture est un aspect essentiel et négligé