Master M1 Informatique
Cours d'Option

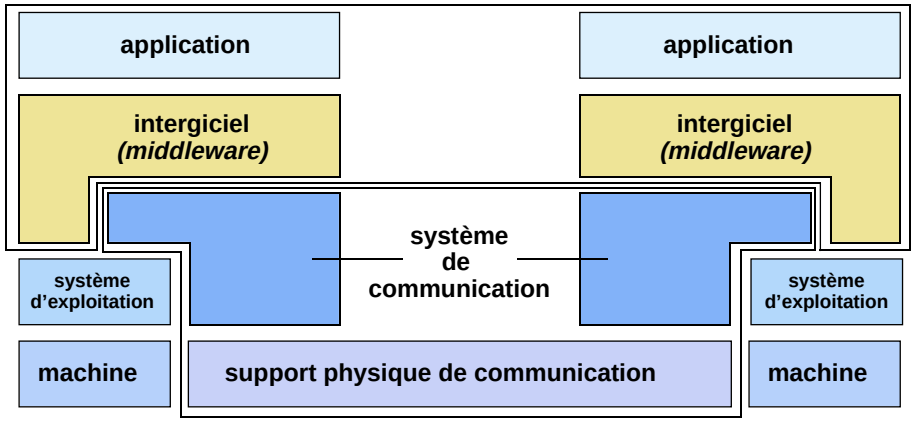
Middleware Adaptable

Introduction

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/people/krakowia

L'intergiciel (*middleware*)



Fonctions du *middleware*

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

Pourquoi le *middleware* adaptable ?

- Les besoins des applications évoluent en permanence
  - Capacité de croissance
  - Qualité de service
- Les applications réparties travaillent dans un environnement changeant
  - Mobilité
    - Logique (code et données mobiles)
    - Physique (mobilité des utilisateurs, du matériel)
  - Connexion et déconnexion dynamique
  - Qualité variable des communications
- Réponse : *adaptation* du middleware et des applications
  - Découverte dynamique de services
  - Reconfiguration dynamique
  - Comportement adaptable

Objectifs et organisation du cours

- Étude approfondie d'aspects avancés du *middleware*, avec application concrètes
- Préparation
  - À la mise en œuvre d'applications évolutives dans un cadre professionnel
  - À la recherche dans ce domaine

Deux modules (consécutifs) :

- Méthodes et outils pour l'adaptation
- Composants logiciels

Forte composante expérimentale (<1/3 cours, >2/3 pratique)

- Réalisations, études de cas

Aspects pratiques

- Page web de ce cours
  - http://sardes.inrialpes.fr/people/krakowia/Enseignement/Option-M1/MWA-M1.html
  - Contient (ou contiendra) : programme, pointeurs divers, supports
- Équipe pédagogique
  - Coordination
    - Sacha Krakowiak (Pr. UJF, projet Sardes, IMAG-LSR et INRIA)
  - Techniques d'adaptation
    - Sara Bouchenak (MCF UJF, projet Sardes, IMAG-LSR et INRIA)
  - Composants logiciels :
    - Éric Bruneton (ing. Télécom., Dépt. ASR, France Telecom R&D)
    - Didier Donsez (MCF UJF, projet Adèle, IMAG-LSR)

Plan de l'introduction

- Motivations et objectifs du cours
- Besoins des applications
  - Services et interfaces
  - Exemples de besoins
    - multimédia, variété des supports et des communications
    - caches locaux, duplication
    - découverte de services
- Introduction au *middleware*
  - Architecture logicielle, patrons de base
  - Schémas d'architecture
    - Architecture en couches, machine virtuelle (gest. ressources, langage), techniques d'interposition, exemples
- Les principales techniques d'adaptation
  - Composants et adaptation: liens entre les 2 notions
  - Interception/interposition, réflexion et métaniveau, aspects
- Travaux en cours (Sardes)

Applications réparties

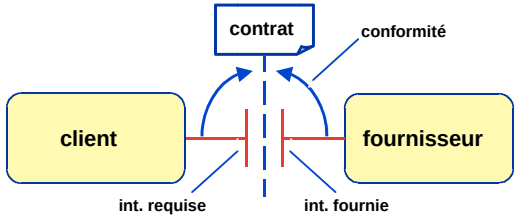
- Distinction entre "système" et "application"
  - Système : gestion des ressources communes et de l'infrastructure, lié de manière étroite au matériel sous-jacent
    - Système d'exploitation : gestion de chaque élément
    - Système de communication : échange d'information entre les éléments
    - Caractéristiques communes : cachent la complexité du matériel et des communications, fournissent des services communs de plus haut niveau d'abstraction
  - Application : réponse à un problème spécifique, fourniture de services à ses utilisateurs (qui peuvent être d'autres applications). Utilise les services généraux fournis par le système
  - La distinction n'est pas toujours évidente, car certaines applications peuvent directement travailler à bas niveau (au contact du matériel). Exemple : systèmes embarqués, réseaux de capteurs

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 concourent à la réalisation du 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)
- Le contrat peut en outre spécifier des aspects non contenus dans l’interface
  - Propriétés dites “non fonctionnelles”, ou Qualité de Service

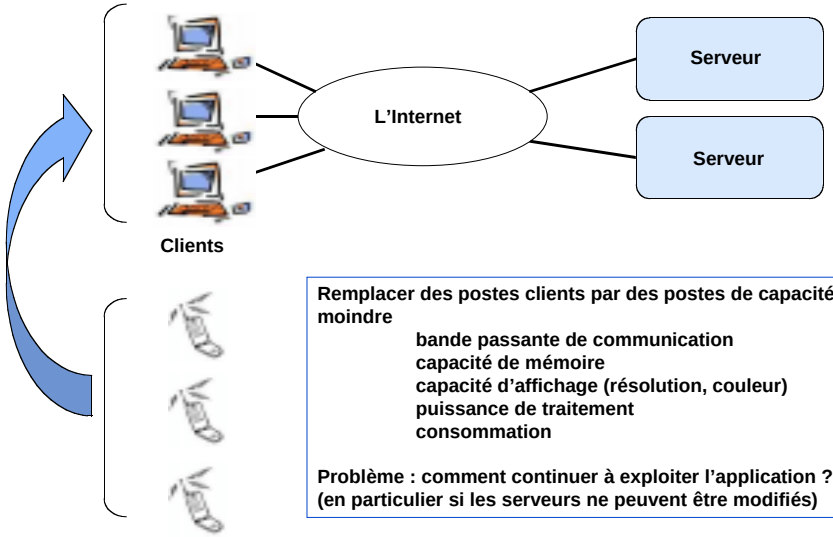
Définitions d’interfaces (2)

- **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

Besoins des applications : quelques exemples

- **Objectif commun**
  - Préserver différents aspects de la qualité de service …
    - Performances
    - Sécurité
    - Tolérance aux fautes
  - … dans un environnement variable
    - Capacités des supports
    - Conditions de communication
    - Spécifications du service
- **Principes de solution**
  - Utlisation du *middleware* pour l’adaptation

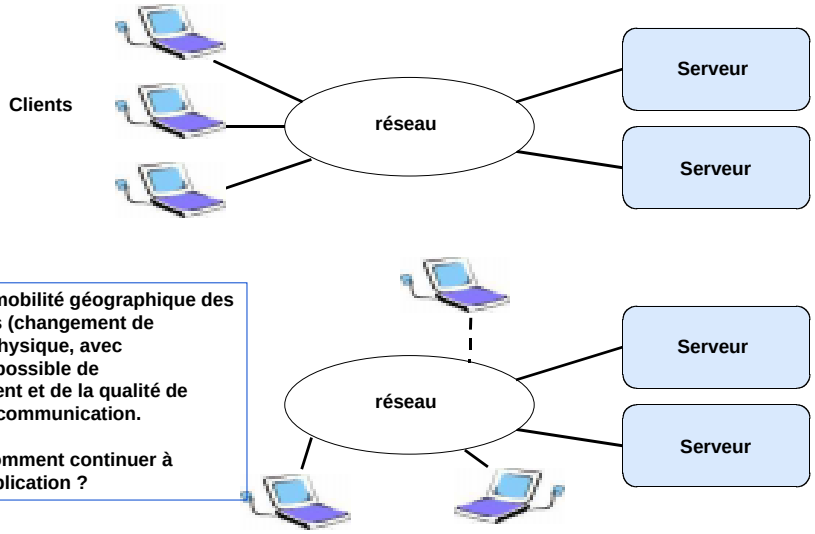
Exemple 1
Adaptation de services en fonction des capacités du poste client



Remplacer des postes clients par des postes de capacité moindre
- bande passante de communication
- capacité de mémoire
- capacité d'affichage (résolution, couleur)
- puissance de traitement
- consommation

Problème : comment continuer à exploiter l'application ? (en particulier si les serveurs ne peuvent être modifiés)

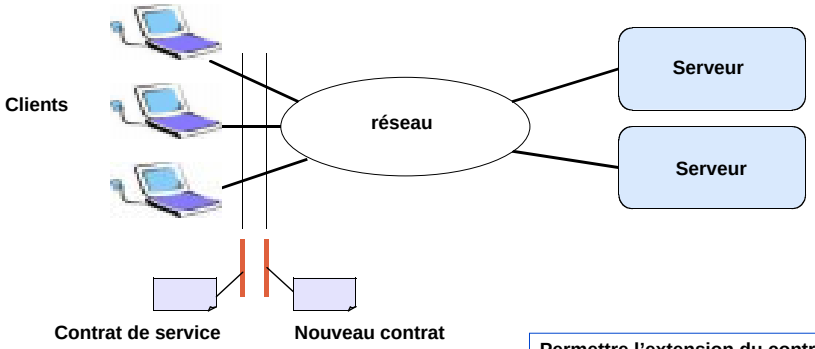
Exemple 2
Adaptation de services du fait de la mobilité



Permettre la mobilité géographique des postes clients (changement de localisation physique, avec modification possible de l'environnement et de la qualité de service de la communication.

Problème : comment continuer à exploiter l'application ?

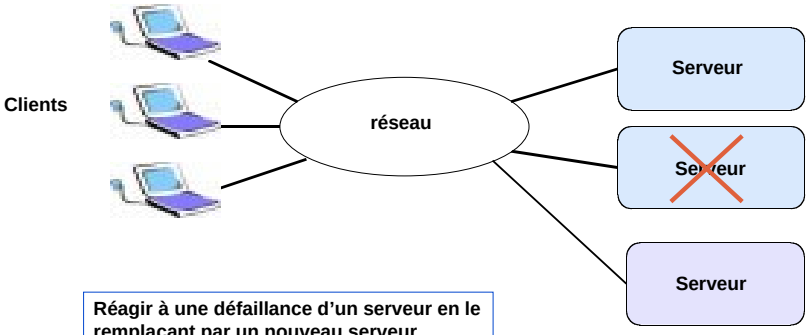
Exemple 3
Extension de services, modification de leur mise en œuvre



Permettre l'extension du contrat de service (exemple : fonctions étendues, ajout de propriétés non fonctionnelles)

Problème : comment réaliser la transition (en particulier sans arrêter l'application) ?

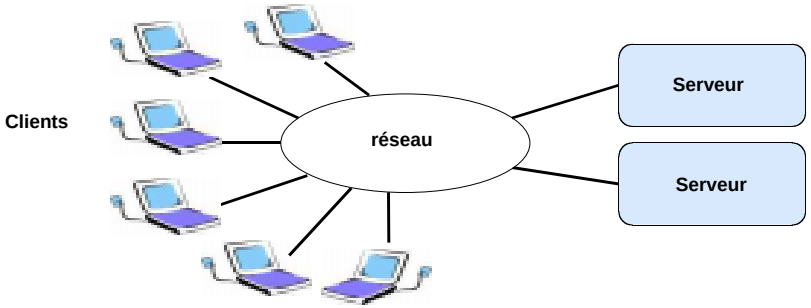
Exemple 4
Adaptation de services pour la tolérance aux fautes



Réagir à une défaillance d'un serveur en le remplaçant par un nouveau serveur équivalent

Problème : comment réaliser ce remplacement (détecter la défaillance, effectuer la transition), sans arrêter l'application ?

Exemple 5
Adaptation de services en fonction de la charge



Réagir à une augmentation de la charge (par ex. nombre de clients connectés)

Problème : comment maintenir un niveau acceptable de qualité de service (par ex. temps de réponse moyen) ?

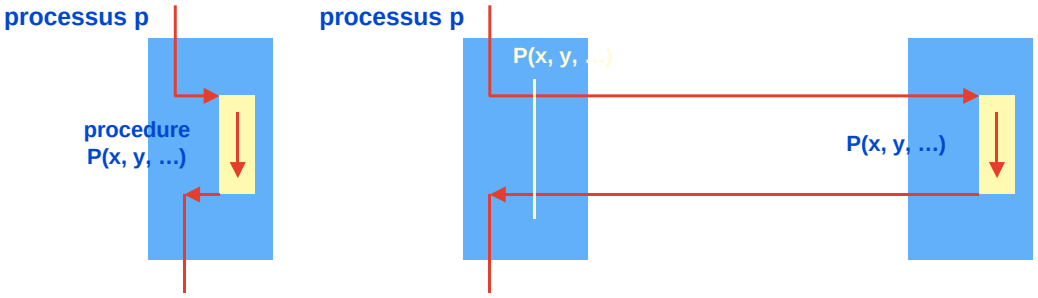
Classes de *middleware*

- **Critères de classification**
  - Nature des entités qui communiquent
    - Objets
    - Composants
    - Autres
  - Mode d'accès aux services
    - Synchrone (client-serveur)
    - Asynchrone (événements, messages)
    - Mixte
  - Support de communication utilisé
    - Entités fixes / mobiles
    - Performances / qualité de service garanties ou non

Pas de classification rigoureuse en raison de la diversité des réalisations

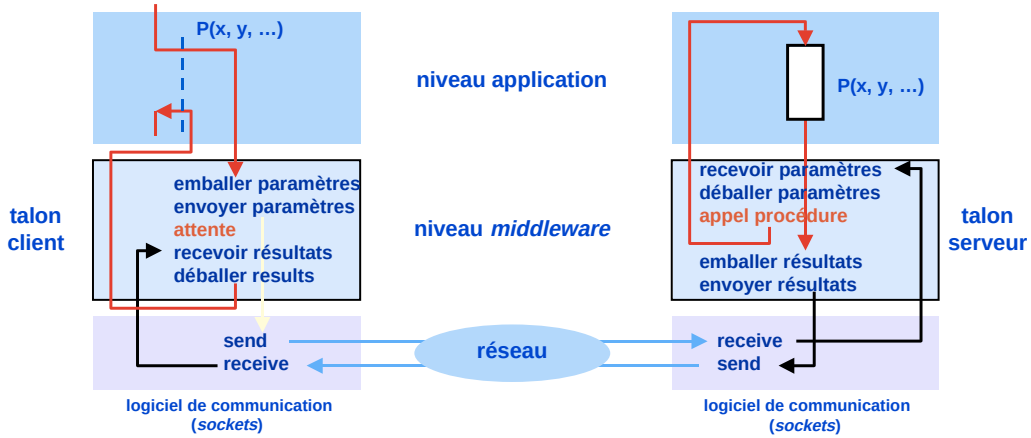
Un exemple simple de *middleware* : RPC (1/2)

- **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

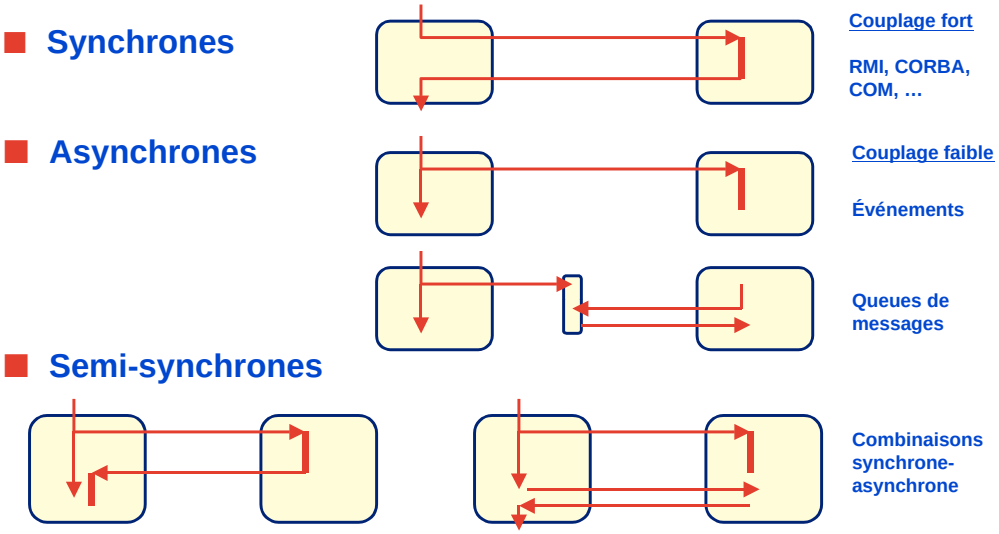
Un exemple simple de *middleware* : RPC (2/2)



- **Mise en œuvre de l'appel de procédure à distance**

Schémas d'interaction (1/4)

- **Synchrones**
- **Asynchrones**
- **Semi-synchrones**



Couplage fort

RMI, CORBA, COM, ...

Couplage faible

Événements

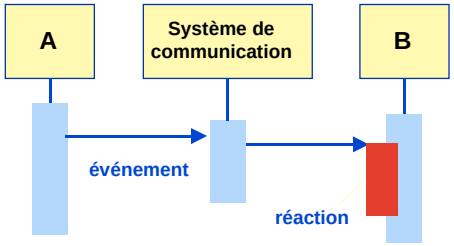
Queues de messages

Combinaisons synchrone-asynchrone

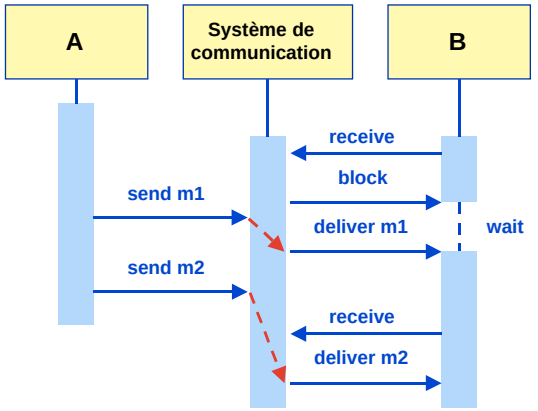
Schémas d'interaction (2/4)

Schémas asynchrones

Exécution parallèle de l'émetteur et du récepteur
Couplage faible



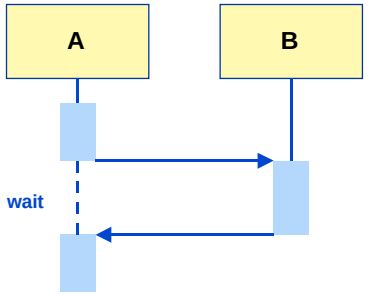
◆ Événement-réaction



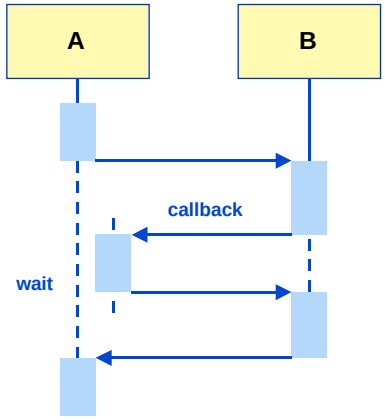
◆ Messages asynchrones

événements et messages peuvent être ou non mémorisés

Schémas d'interaction (3/4)



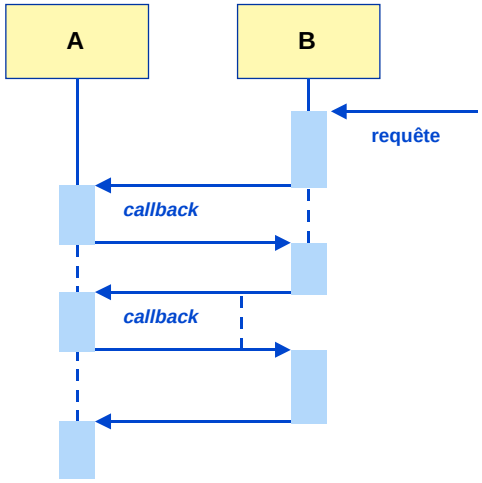
- **Appel synchrone**
  - L'émetteur (client) est bloqué en attendant le retour
  - Couplage fort



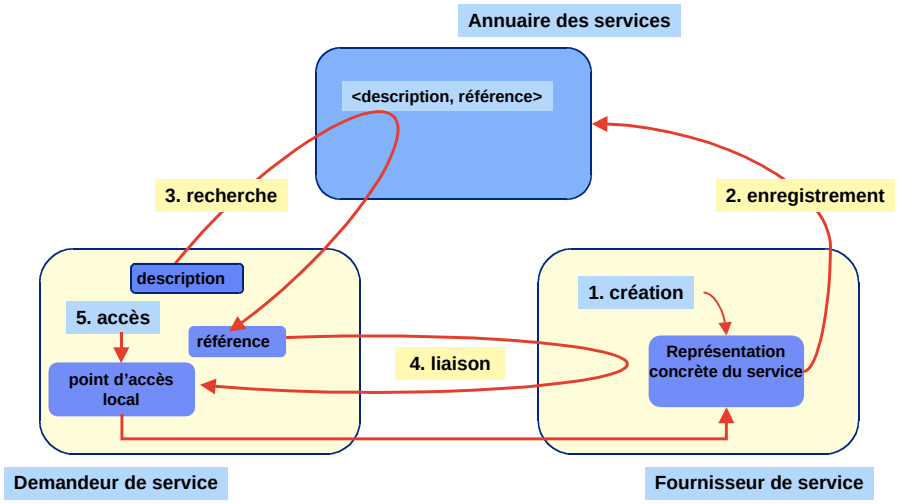
Avec appel en retour (callback)

Schémas d'interaction (4/4)

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



Accès à un service



Patrons de conception (1)

- **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

Patrons de conception (2)

- **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

Quelques patrons de base

- ***Proxy***
  - Patron de conception : représentant pour accès à distance
- ***Factory***
  - Patron de conception : création d'objet
- ***Wrapper* [*Adapter*]**
  - Patron de conception : transformation d'interface
- ***Interceptor***
  - Patron d'architecture : adaptation de service
- ***Observer***
  - Patron de base pour l'asynchronisme

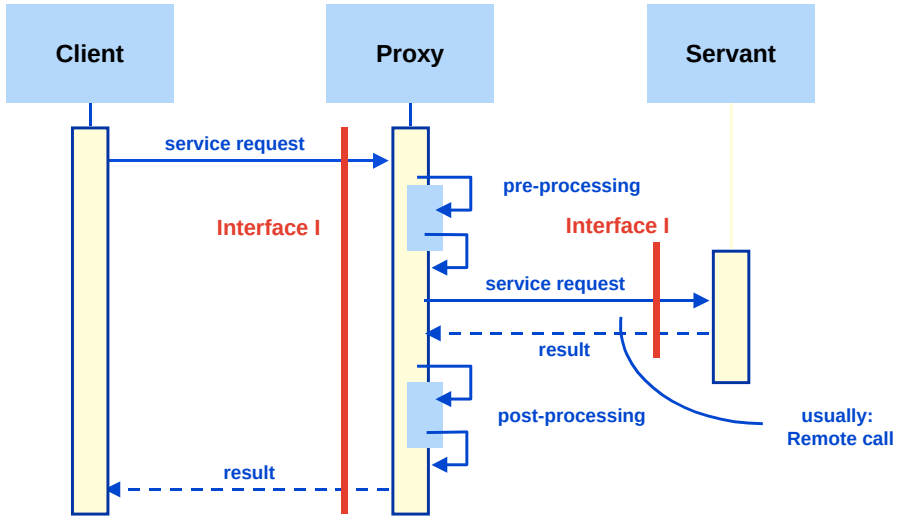
Ces patrons sont d'un usage courant dans la construction d'intergiciel

Nombreux exemples dans toute la suite

Proxy (Mandataire)

- **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 ; idéalement, pas de différence entre accès local et distant
  - 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

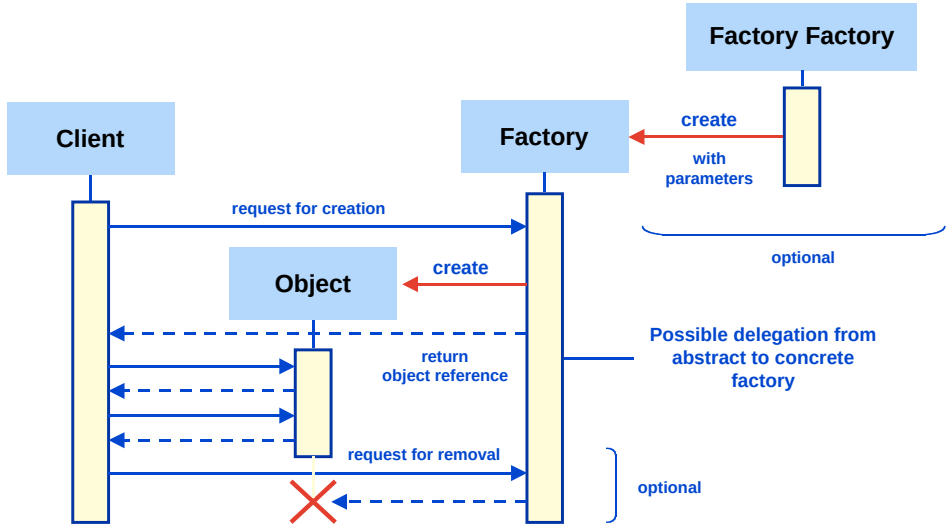
Usage de *Proxy*



Factory (Fabrique)

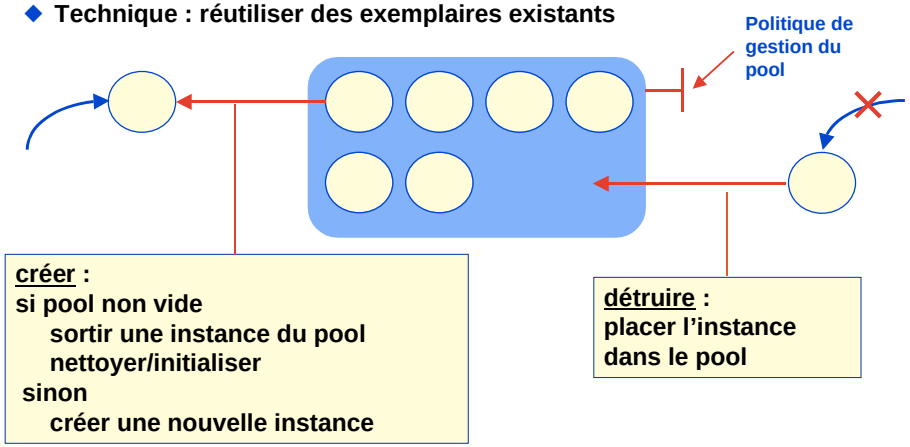
- **Contexte**
  - Application = ensemble d'objets en environnement réparti
- **Problème**
  - Créer dynamiquement des instances multiples d'une classe d'objets
  - Propriétés souhaitables
    - Les instances doivent être paramétrables
    - L'évolution doit être facile (pas de décisions "en dur")
  - 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 fabriques concrètes qui implémentent les méthodes de création
  - *Abstract Factory* peut être implémentée par *Factory Methods* (méthode de création redéfinie dans une sous-classe)
  - Pour plus de de souplesse, on peut utiliser *Factory Factory* (le mécanisme de création lui-même est paramétré)

Usage de *Factory*



Un complément à *Factory* : *Pool*

- **Idée : réduire le coût de la gestion de ressources**
  - Technique : réutiliser des exemplaires existants



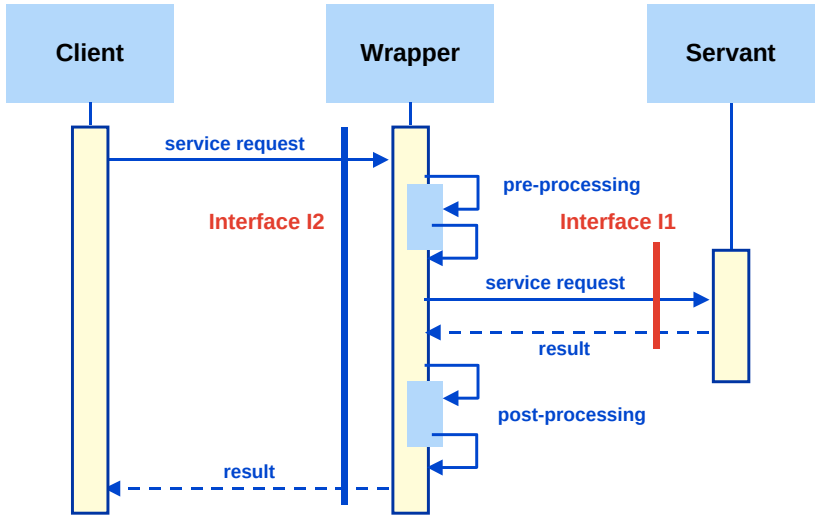
Utilisation de *Pool*

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

Wrapper (ou *Adapter*)

- **Contexte**
  - Des clients demandent des services ; des servants fournissent des services ; les services sont définis par des interfaces
- **Problème**
  - Réutiliser un servant existant en modifiant son interface et/ou certaines de ses fonctions pour satisfaire les besoins d'un client (ou 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 servant 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

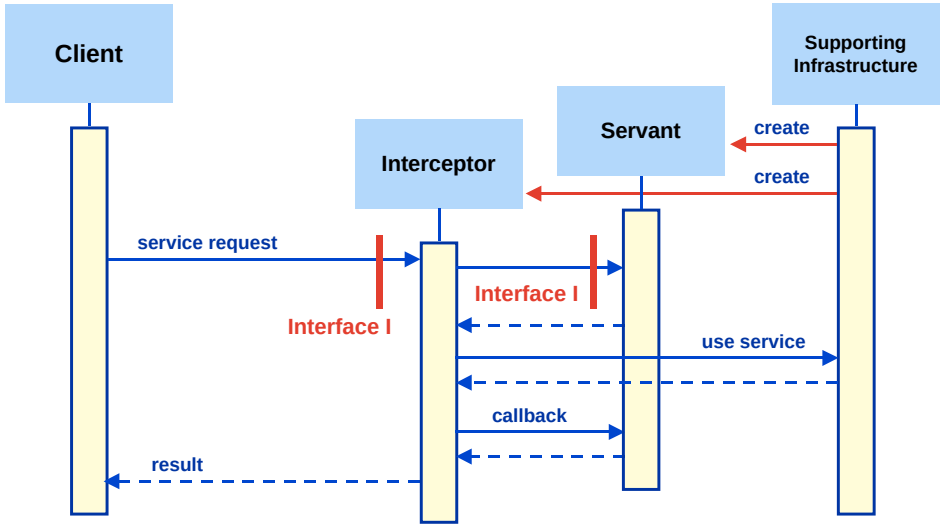
Usage du *Wrapper*



Interceptor (Intercepteur)

- **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 cleient 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

Usage d'*Interceptor*

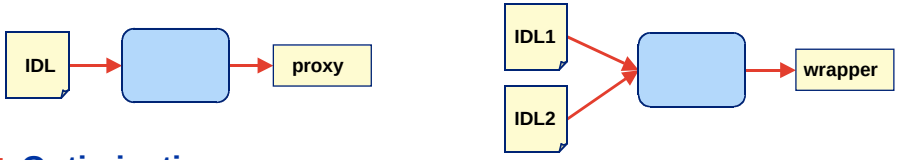


Comparaison des patrons de base

- ***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 proxy (*smart proxy*)

Mise en œuvre des patrons de base

- **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)

Canevas logiciels (*Frameworks)*

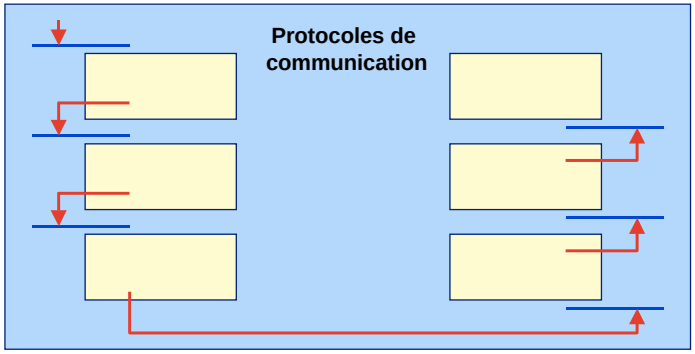
- **Définition**
  - Un canevas est un “squelette” de programme qui peut être réutilisé (et adapté) pour une famille d’applications
  - Il met en œuvre un modèle (pas toujours explicite)
  - Dans les langages à objets : un canevas comprend
    - Un ensemble de classes (souvent abstraites) devant être 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

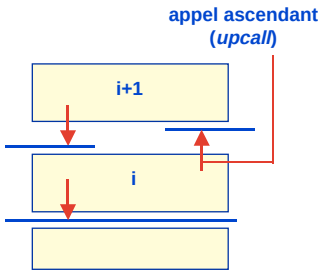
Schémas de décomposition

- **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

Décomposition en couches

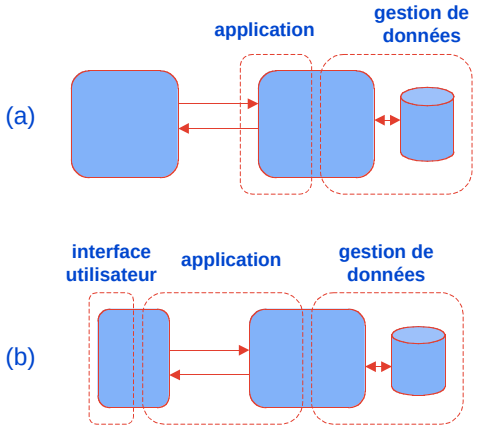
- **Hiérarchie de “machines abstraites”**
  - La réalisation des niveaux < i est invisible au niveau i
  - Exemple : machines virtuelles (OS multiples, JVM, etc.)

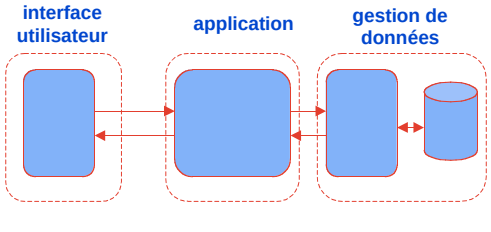




Décomposition “horizontale”

- **Exemple : évolution du schéma client-serveur**

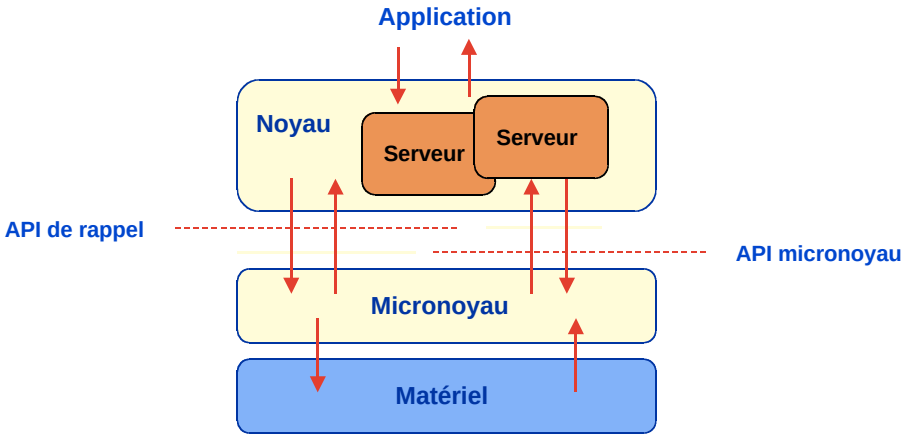




(c) : 3-*tier*

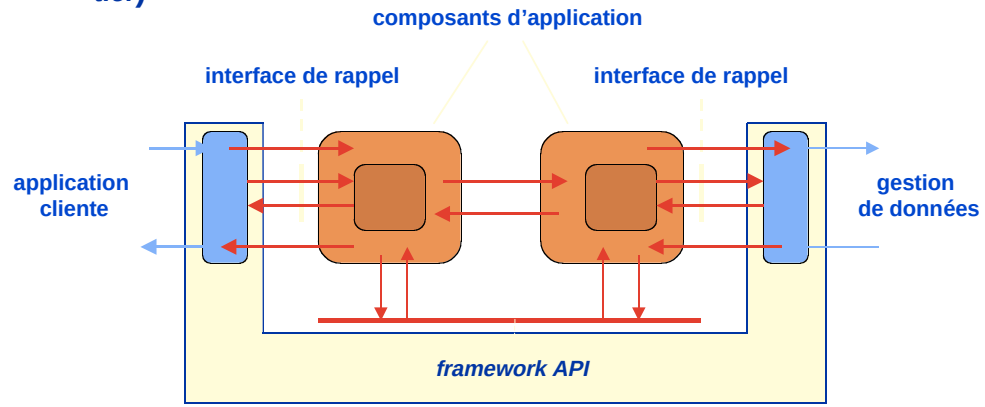
Exemple de canevas global (1)

- **Architecture de micro-noyau**



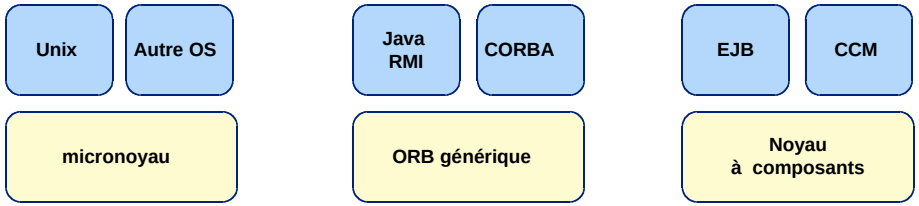
Exemple de canevas global (2)

- **Architecture d'un canevas pour composants (*middle tier*)**



Canevas de base et personnalités

- **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**



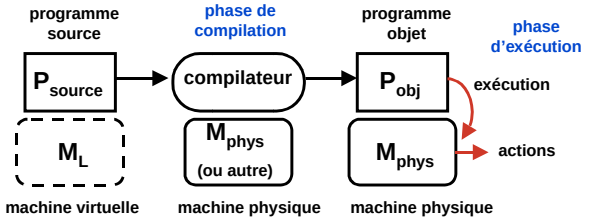
Machines virtuelles

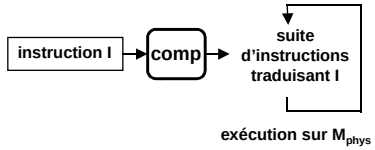
- **Définition**
  - Simulation d'une machine "abstraite" (définie par le jeu d'instructions qu'elle interprète) sur une machine physique
- **Utilité**
  - Émulation d'une machine (projet, ou machine n'existant plus)
  - Multiplexage de ressources
  - Un mode d'exécution d'un langage (i.e. Un interprète)

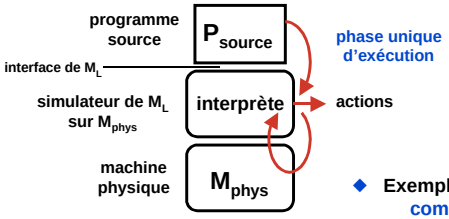
Rappel des schémas d'exécution d'un programme

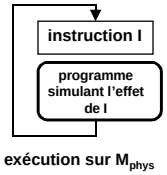
- Un programme P_{source} dans un langage (impératif) L décrit une suite d'actions à exécuter par une machine (virtuelle) M_L
  - Le programme est une suite d'instructions dont chacune spécifie une action (passage d'un état initial à un état final)
  - La machine M_L est capable d'exécuter (interpréter) le programme P_{source}, c'est-à-dire de traduire en actions de M_L la suite d'instructions de P_{source}.
- Comment faire si on ne dispose pas de la machine M_L, mais d'une machine différente, M_{phys} ?
  - Deux solutions de base
    - a) Traduire le programme P_{source} dans un programme P_{obj} "équivalent" pour la machine M_{phys}, et faire exécuter P_{obj} sur M_{phys} ("équivalent" = qui a le même effet). C'est un schéma de compilation (la traduction de P_{source} en P_{obj} est faite par un compilateur)
    - b) Construire (par programme) sur M_{phys} un simulateur de la machine M_L, et faire exécuter le programme original P_{source} sur ce simulateur. C'est un schéma d'interprétation (le simulateur de M_L sur M_{phys} est un interprète)
  - Une solution mixte est possible
    - compiler P_{source} pour une machine intermédiaire M_{int}
    - construire un interprète de M_{int} sur M_{phys}

Rappel : compilation et interprétation









- Exemples
  - compilation : C, C++, Fortran, Ada
  - interprétation : lang. machine, *shell* Unix, Tcl-TK, PostScript
  - l'un ou l'autre : Lisp, Scheme
  - schéma mixte : Java (cf plus loin), Smalltalk

Compilation et interprétation : comparaison

Compilation

- Efficacité
  - le code engendré s'exécute directement sur la machine physique
  - ce code peut être optimisé
- Mise au point
  - pas toujours facile de relier une erreur d'exécution au texte source
- Cycle de modification - rééxécution
  - toute modification du texte source impose de refaire le cycle complet (compilation, édition de liens, exécution)

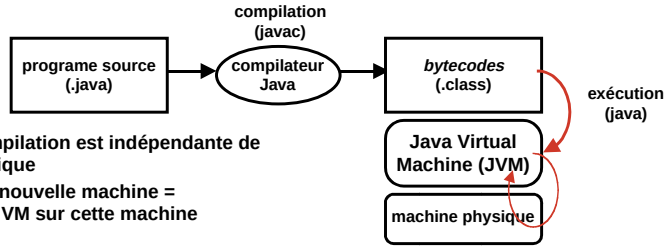
Interprétation

- Efficacité
  - l'interprétation directe est souvent longue (appel de sous-programmes)
  - pas de gain sur les boucles
  - facteur de x10 à x100 …
- Mise au point
  - lien direct entre instruction et exécution
  - possibilités étendues d'observation et trace intégrées
- Cycle de modification - rééxécution
  - cycle très court (modifier et réexécuter)

L'augmentation de puissance des processeurs amène un regain des techniques d'interprétation

Schéma mixte d'exécution

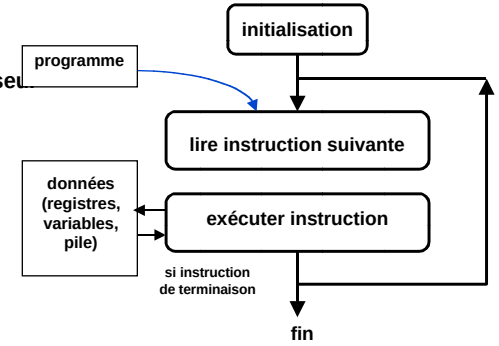
- Objectifs
  - Essayer de combiner les avantages des schémas de compilation et d'interprétation
  - Améliorer la portabilité des programmes entre machines, via un langage intermédiaire standard
- Principe
  - Définir un langage intermédiaire et une machine virtuelle capable d'interpréter ce langage
  - Écrire un compilateur du langage initial (source) vers le langage intermédiaire
  - Écrire un interprète du langage intermédiaire (c'est-à-dire un simulateur de la machine virtuelle)
- Exemple : Java



  - La phase de compilation est indépendante de la machine physique
  - Portage sur une nouvelle machine = réécriture de la JVM sur cette machine

Principe de fonctionnement d'un interprète (1)

- **Définition de la machine virtuelle**
  - Éléments du "pseudo-processeur" (analogie avec processeur physique)
    - pseudo-registres (zones de mémoire réservées)
    - pseudo-instructions (réalisées par une bibliothèque de programmes)
  - Structures d'exécution
    - allocation des variables
    - pile d'exécution
- **Cycle d'interprétation**
  - Analogie avec cycle d'un processeur
  - Pseudo-compteur ordinal



Principe de fonctionnement d'un interprète (2)

- **Raffinement de la "boîte" *exécuter instruction***
  - Format de l'ïnstruction : instruction ::= <code opération>, <suite de paramètres>
  - Opérations

    Décoder instruction (isoler code op., paramètres)
    selon code op faire
    $code_1$: exécuter programme de $code_1$ (paramètres)
    …
    $code_n$: exécuter programme de $code_n$ (paramètres)
  - Exécution du programme correspondant à un code op.

    Dépend de l'instruction, mais quelques points communs
    allouer éventuellement place en mémoire pour données
    charger paramètres dans registres ou pile
    exécuter l'opération sur les paramètres
    mettre éventuellement à jour les résultats en mémoire
    déterminer l'instruction suivante à exécuter
  - Autres opérations possibles

    Observation, mise au point (exécution pas à pas), trace
    Transformation préalable du programme dans une forme adaptée à l'exécution (postfixé, etc.)

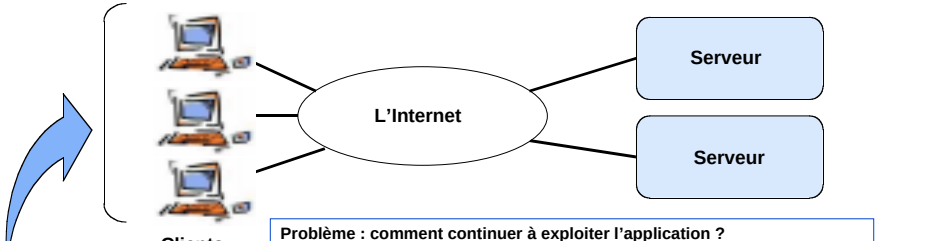
Adaptation des systèmes et applications

- **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)

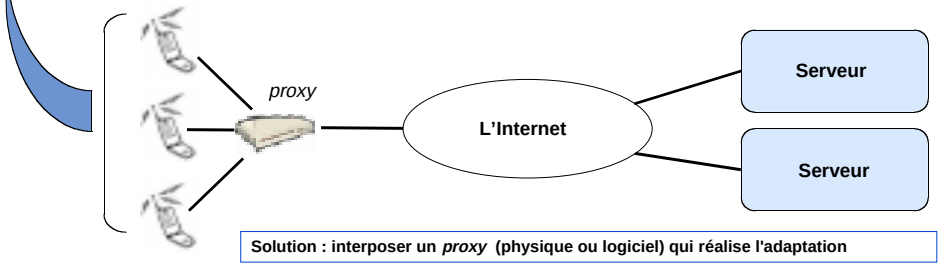
Adaptation par interception : exemples

- **Principes de résolution des problèmes d'adaptation rencontrés**
  - Adaptation de services en fonction des capacités du poste client
  - Adaptation de services du fait de la mobilité
  - Extension de services, modification de leur mise en œuvre
  - Adaptation de services pour la tolérance aux fautes
  - Adaptation de services en fonction de la charge

Exemple 1
Adaptation de services en fonction des capacités du poste client

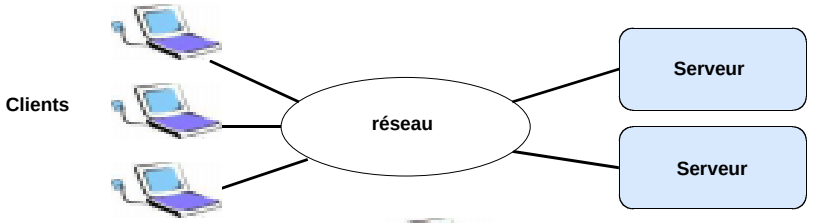


Problème : comment continuer à exploiter l'application ?
(en particulier si les serveurs ne peuvent être modifiés)



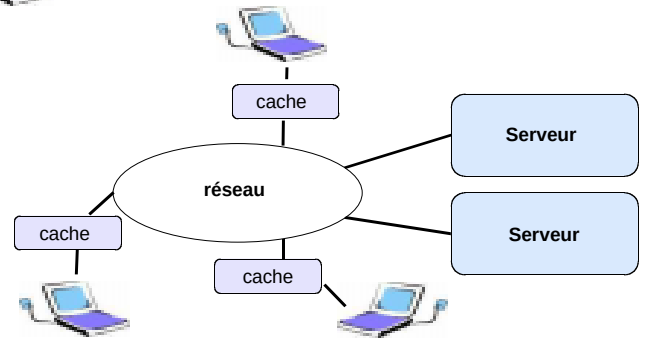
Solution : interposer un *proxy* (physique ou logiciel) qui réalise l'adaptation

Exemple 2
Adaptation de services du fait de la mobilité

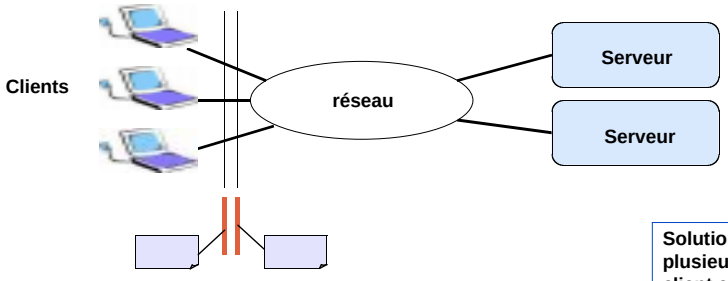


Problème : comment continuer à exploiter l'application ?

Solution : interposer un *proxy* jouant le rôle de cache client pour améliorer les performances, notamment si communication défectueuse

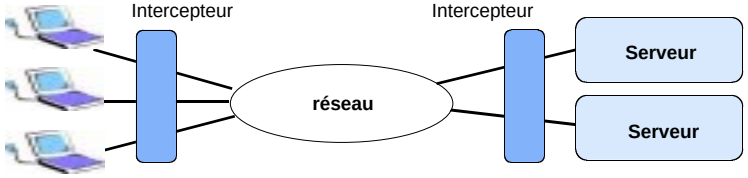


Exemple 3
Extension de services, modification de leur mise en œuvre

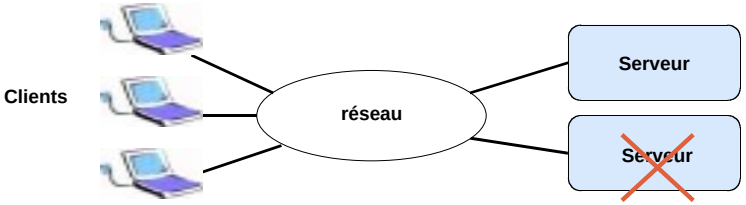


Problème : comment réaliser la transition (en particulier sans arrêter l'application) ?

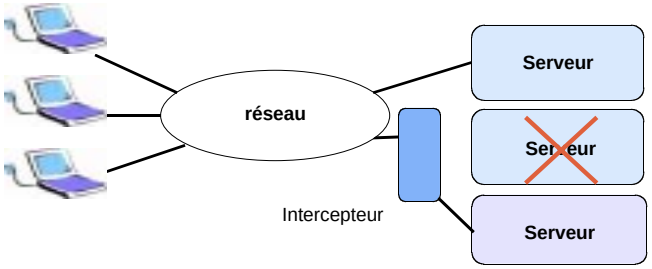
Solution : interposer un ou plusieurs intercepteurs (côté client et/ou serveur) pour réaliser les extensions



Exemple 4
Adaptation de services pour la tolérance aux fautes

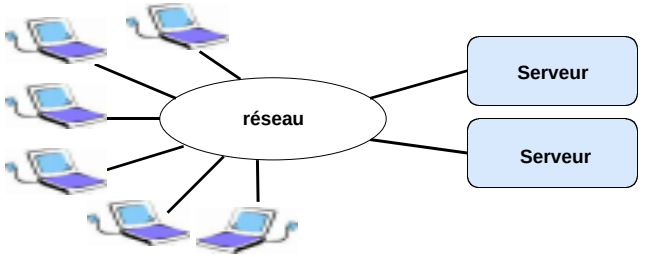


Problème : réagir à une défaillance d'un serveur en le remplaçant par un nouveau serveur équivalent, sans arrêter l'application

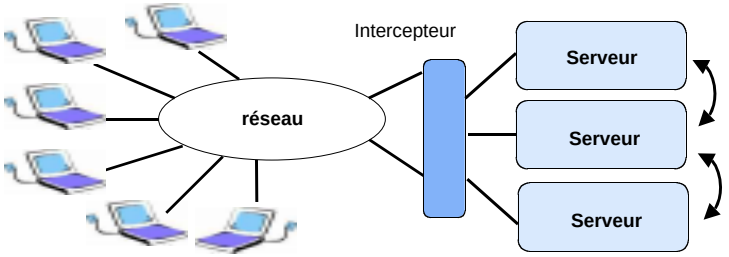


Solution : utiliser un intercepteur pour détecter la défaillance et assurer le remplacement

Exemple 5
Adaptation de services en fonction de la charge



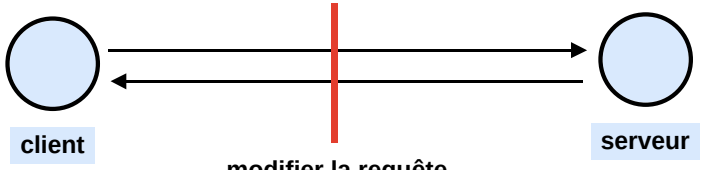
Problème : comment maintenir un niveau acceptable de qualité de service lorsque la charge augmente ?



Solution : répartir la charge sur les serveurs (éventuellement utiliser des serveurs disponibles) via un intercepteur

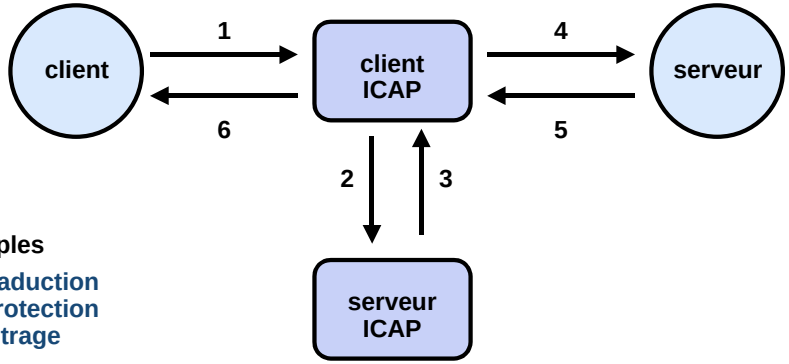
Exemple 6 : protocole ICAP (*Internet Content Adaptation Protocol*)

- **Fonction : fournir des services à valeur ajoutée**
  - utilisation de ressources (processeurs, appareils spécialisés)
  - réalisation de fonctions (filtrage, transformation, traduction, protection)
- **Mode d'action**
  - interposition dans un système client-serveur utilisant HTTP



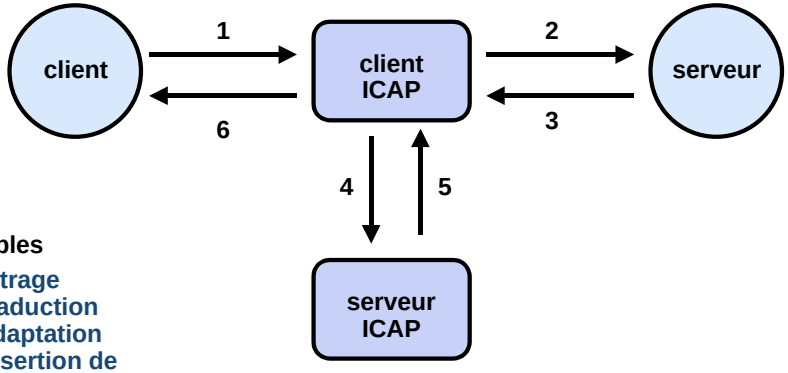
modifier la requête
modifier la réponse
ajouter une fonction

Protocole ICAP : modifier la requête



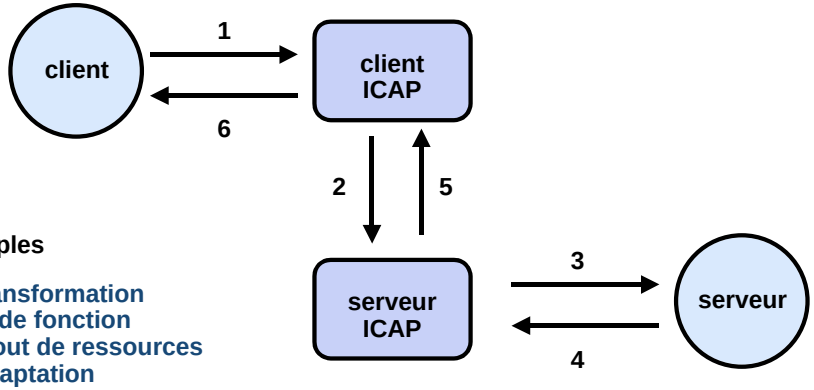
Exemples
- traduction
- protection
- filtrage

Protocole ICAP : modifier la réponse



Exemples
- filtrage
- traduction
- adaptation
- insertion de commentaires (ou de publicité)

Protocole ICAP : interposer une fonction

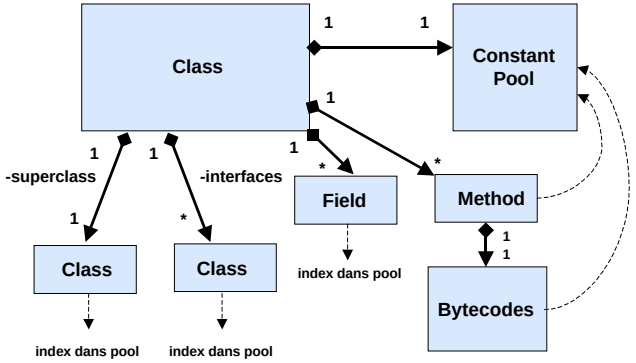


Exemples

- transformation
  - de fonction
- ajout de ressources
- adaptation

Exemple de réflexivité : API Java Reflection

Les méta-données d'une classe Java contiennent une description des différents constituants de cette classe.



Contient toutes les constantes liées àl la classe et en particulier les chaînes de caractères qui sont les noms de méthodes, de classes, etc.

Les autres éléments contiennent des index vers des entrées de ce pool

L'API Java Reflection fournit une interface Java pour l'accès à ces méta-données et pour leur manipulation

Utilisation de la réflexivité en Java

Permet de réaliser des opérations génériques (indépendantes d'une classe particulière)
Très utile si la classe n'est connue qu'au moment de l'exécution.

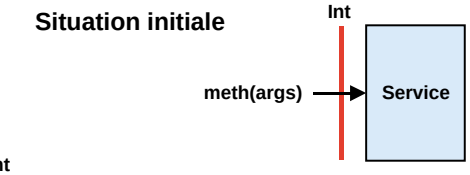
Exemples

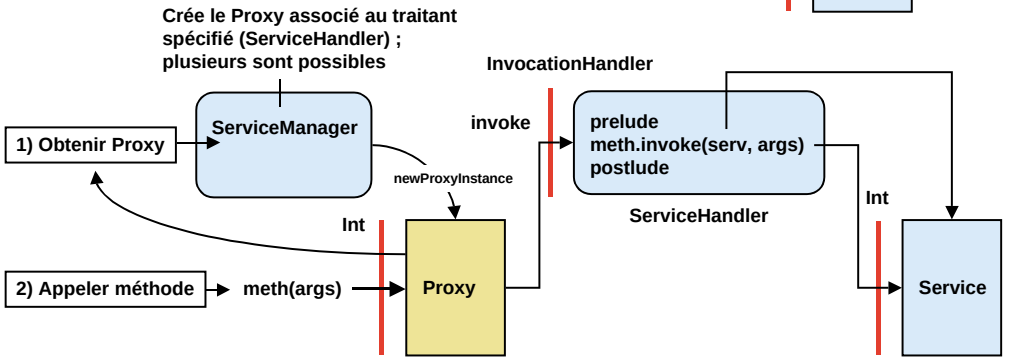
```
…
String className = "le nom d'une classe" ;
Class c = Class.forName(className) ;          // la classe n'est connue qu'à l'exécution

…
Object obj = c.newInstance()                  // constructeur standard pour c
…
methodName = "le nom d'une méthode";
Method m = c.getMethod(methodName, new Class[ ] {…}) ;
…
m.invoke(c, new Object  [ ] {…}) ;            // appel dynamique de méthode  (générique)
…
```


Application de la réflexivité en Java : *Dynamic Proxy*

On souhaite utiliser le même service (même interface), mais en modifiant son effet dynamiquement (choix entre plusieurs réalisations)





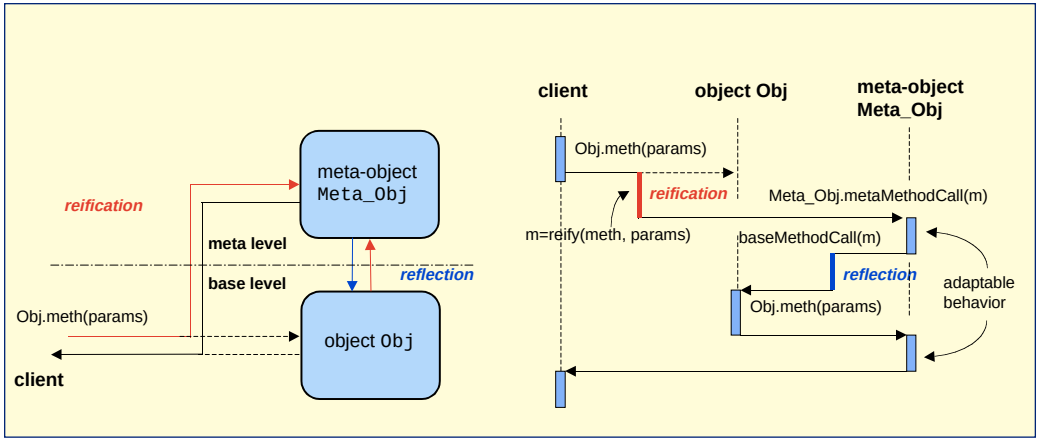
Voir java.lang.reflect.Proxy et java.lang.reflect.InvocationHandler

Protocoles à méta-objets

- Un service adaptable est organisé en deux niveaux
  - Niveau de base : réalise les fonctions définies par la spécification
  - Méta-niveau : utilise une représentation du niveau de base pour observer ou modifier le comportement de celui-ci
    - La représentation doit être causalement connectée (i.e. toujours fidèle : tout changement du niveau de base est reflété dans la représentation et vice-versa)
  - Relations entre les niveaux
    - Création d'une représentation d'une entité : réification
    - Action du méta-niveau sur le niveau de base : réflexion
- L'organisation peut être étendue récursivement
  - "Tour réflexive" : méta-méta-niveau, etc.
  - En pratique, on se limite à 2 ou 3

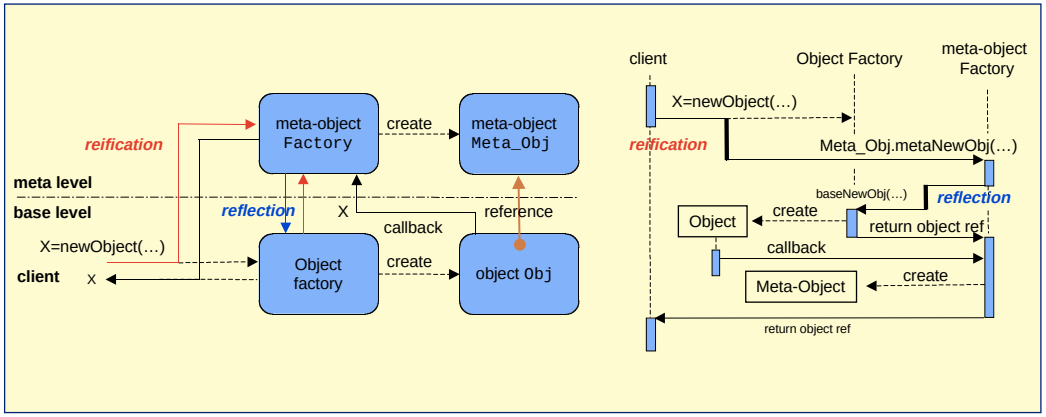
Protocoles à méta-objets : exemple 2

- Réification d'un appel de méthode



Protocoles à méta-objets : exemple 2

- Création d'un objet



Programmation par aspects

- Principe (application de la séparation des préoccupations)
  - Identifier un comportement de base et des "aspects" supplémentaires aussi indépendants que possible
  - Décrire séparément le comportement de base et les différents aspects
  - Intégrer l'ensemble dans un programme unique
- Méthode
  - Description séparée des aspects (éventuellement format spécifique)
  - Intégration ("tissage"), statique ou dynamique
- Notions de base
  - Point de jointure (*join point*) : point d'insertion de code d'aspects
  - Coupure (*cut point*) : ensemble de points de jointure logiquement corrélés
  - Indications (*advice*) : définition des relations entre le code inséré et le code de base (exemple : avant, après, …)
- Problème : réalisation efficace

Programmation par aspects : exemple

- Réalisation d'un *Wrapper* en AspectJ

```
public aspect MethodWrapping  {

/* point cut definition */
    pointcut Wrappable(): call(public * MyClass.*(..));

 /* advice definition */
    around(): Wrappable() {
            <prelude>     /* a sequence of code to be inserted before the call */
            proceed();    /* performs the call to the original method  */
            <postlude>   /* a sequence of code to be inserted after the call */
    }
}
```

Effet : encadre tout appel à une méthode publique de la classe MyClass par <prelude> et <postlude> (application possible :journalisation, test de conditions, etc.)

Adaptation et composants

- **Composants logiciels**
  - Unité de décomposition d'un système ou d'une application
    - Pour les fonctions (interfaces)
    - Pour le déploiement (installation physique)
- **Lien entre adaptation et composants**
  - Un composant peut être aussi une unité d'adaptation
    - Le mécanisme qui gère les composants (conteneur ou autre) peut aussi servir à l'adaptation
  - L'interception s'applique bien aux interactions entre composants
    - Une interface est un point de passage obligé et bien défini
  - La configuration physique est un autre outil d'adaptation
    - Redéploiement
    - Reconfiguration dynamique

Quelques recherches en cours : projet SARDES

- **Présentation**
  - Projet du laboratoire IMAG-LSR et de l'INRIA Rhône-Alpes ; lieu : INRIA, Montbonnot
  - 20 personnes (10 permanents, 10 thésards, ingénieurs…)
  - Voir http://sardes.inrialpes.fr/
- **Thème de recherche : *middleware*, systèmes et applications répartis**
  - SARDES =
    - *Systems Architecture for Reflective Distributed EnvironmentS*
    - *Self-Administrable and Reconfigurable Distributed EnvironmentS*
- **Relations**
  - Membre fondateur d'ObjectWeb, consortium pour le *middleware* en logiciel libre
    - Voir http://www.objectweb.org
  - Nombreuses relations industrielles
    - Bull
    - Microsoft
    - France Telecom
    - ST Microelectronics
    - Une start-up issue du projet : Scalagent Distributed Technologies
  - Relations internationales
    - Projets européens
    - …

Projet SARDES : thèmes de recherche

- **Modèles et outils pour les composants répartis**
  - Modèles et schémas d'architecture pour le calcul réparti
  - *Middleware* adaptable (migration, duplication, reconfiguration)
- **Administration de systèmes répartis**
  - Observation de systèmes, *monitoring*
  - Systèmes autonomes
  - Gestion de ressources et support système pour grappes de machines
- **Applications**
  - Noyaux pour systèmes et applications reconfigurables
  - Gestion de QoS pour applications multimédia
  - Optimisation d'applications sur serveurs en grappes

Plan du cours

- **Méthodes et outils pour l'adaptation**
  - Adaptation à haut niveau (aspects)
    - Étude de cas : AspectJ
  - Adaptation à bas niveau (*bytecode*)
    - Étude de cas : BCEL
- **Composants logiciels**
  - Modèles et architectures à composants
  - Étude de cas n°1 : Fractal et Julia
  - Étude de cas n°2 : OSGi