DEA *Systèmes Informatiques Répartis*
Tronc Commun *Conception par Objets*
et Prototypage d'Applications Concurrentes

Objets, Parallélisme et Répartition, Architectures Logicielles, Composants, Frameworks, Acteurs, Agents

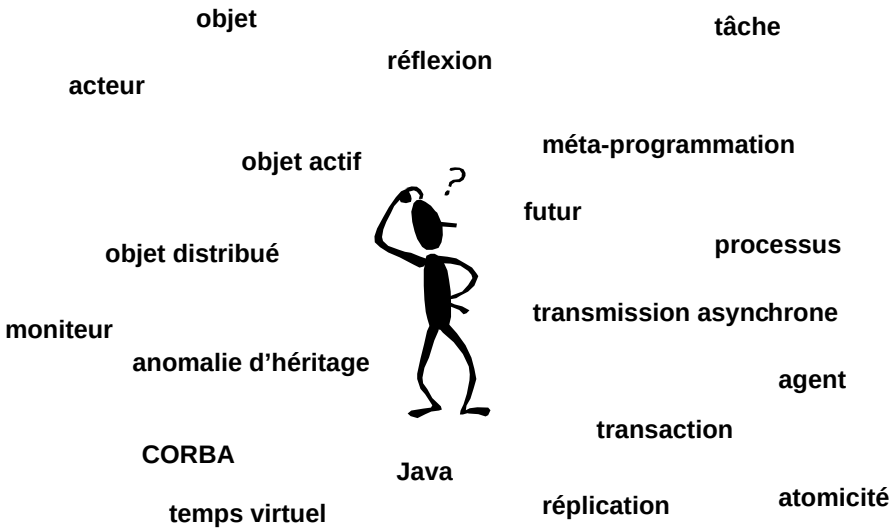
Jean-Pierre Briot

Thème OASIS
(Objets et Agents pour Systèmes d'Information et Simulation)
Laboratoire d'Informatique de Paris 6
Université Paris 6 - CNRS
Jean-Pierre.Briot@lip6.fr

Plan général

- Objets pour la Programmation Parallèle et Répartie
  - Approche applicative
  - Approche intégrée
  - Approche réflexive
- Architectures Logicielles
  - Architectures logicielles
  - Composants
  - Frameworks
  - Design patterns
- Acteurs
  - Framework d 'acteurs : Actalk
  - Exemples de programmes acteurs
- Concurrence
  - Sûreté vs vivacité
- Agents
  - Introduction aux (multiples)

Inflation des termes, et des concepts ??



Objectif

- Rappeler en quoi les concepts d'objet (de facto standard actuel de la programmation "classique, i.e. séquentielle et centralisée) offrent une bonne fondation pour la programmation parallèle et répartie
- Analyser et classifier les différents types d'articulation entre :
  - programmation par objets
  - programmation parallèle et répartie
- Nous considérons trois approches principales :
  - applicative *(structuration sous forme de bibliothèques)*
  - intégrée *(identification et unification des concepts et mécanismes)*
  - réflexive *(association de méta-bibliothèques de mise en œuvre à un programme - idée : réifier le contexte du calcul, de manière à pouvoir adapter un programme à différents environnements et contraintes de calcul)*
- Analyser
  - les limites d'une transposition naïve des concepts d'objet, ou plutôt des techniques d'implantation, à la programmation parallèle et répartie
  - de possibles solutions

Objets pour la Programmation Parallèle et Répartie

- Exposé fondé sur une étude menée en collaboration avec Rachid Guerraoui, EPFL, Suisse
- Articles de référence :
  - «Objets pour la programmation parallèle et répartie», Jean-Pierre Briot et Rachid Guerraoui, dans «Langages et modèles à objets», édité par Amedeo Napoli et Jérôme Euzenat, Collection Didactique, INRIA, 1998.
  - Initialement publié dans la revue Technique et Science Informatiques (TSI),15(6):765-800, Hermès, France, juin 1996.
  - «Concurrency and distribution in object-oriented programming», Jean-Pierre Briot, Rachid Guerraoui, et Klaus-Peter Löhr, ACM Computing Surveys, à paraître fin 1998.

Plan

- Applications informatiques : enjeux actuels et futurs
- Concepts d'objet
  - Potentiel (concurrence et répartition) et limites
- Objets, parallélisme et répartition : 3 approches
- Approche applicative
  - Principes, Exemple, Bilan
- Approche intégrée
  - Dimensions d'intégration (objet actif, objet synchronisé, objet réparti)
  - Exemples, Limitations
- Approche réflexive
  - Principes, Exemples, Bilan
- Conclusion

Enjeux actuels et futurs

- De la programmation séquentielle, centralisée, en monde clos
  ... à la programmation parallèle, répartie, de systèmes ouverts
  - ex : travail coopératif assisté par ordinateurs (CSCW)
  - ex : simulation répartie multi-agent
- Décomposition fonctionnelle (logique) : Concurrence
  vs Mise en oeuvre (physique) : Parallélisme
  - intrinsèque (ex : multi-agent, atelier flexible)
  - a posteriori (temps de calcul)
- Répartition
  - intrinsèque (ex : CSCW, contrôle de procédé)
  - a posteriori (volume de données, résistance aux pannes)
- Système ouvert
  - reconfigurable dynamiquement, ex : Internet
  - adaptation à l'environnement, ex : contraintes de ressources (temps, espace..)

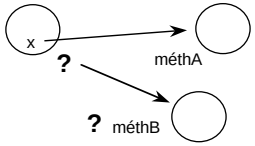
Concepts d'objet

- objet : module autonome (données + procédures)
- protocole de communication unifié : transmission de messages
- abstraction : classe (factorisation) d'objets similaires
- spécialisation : sous-classe (mécanisme d'héritage)
- encapsulation : séparation interface / implémentation
- gestion dynamique des ressources
- *concepts suffisamment forts* : structuration et modularité
- *concepts suffisamment mous* : généricité et granularité variable

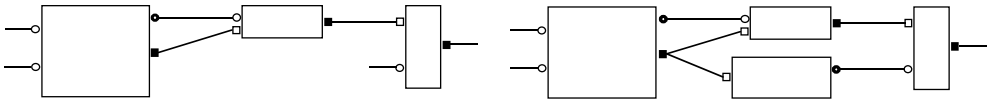
Limites des objets

- granuralité encore trop fine-moyenne
  - pas trop bien adapté à la programmation à grande échelle
- pas encore assez modulaire
  - références directes entre objets
  - donc connexion non reconfigurable sans changer l'intérieur de l'objet
    - » objet appelé
    - » nom de la méthode appelée

Idées :



- composants
  - plus «gros»
  - plus autonomes et encapsulés
- réification des relations/connexions entre composants
  - notion de connecteurs



Architectures logicielles [Shaw et Garlan 96]

- composants et connecteurs
- différents types d'architectures
  - pipes et filters, ex : Unix Shell `dvips | lpr`
  - couches, ex : Xinu, protocoles réseaux
  - événements (publish/subscribe), ex : Java Beans
  - frameworks, ex : Smalltalk MVC
  - repositories, ex : Linda, blackboards
- un même (gros) système peut être organisé selon plusieurs architectures
- les objets se marient relativement bien avec ces différentes architectures logicielles
  - objets et messages comme support d'implémentation des composants et aussi des connecteurs
  - cohabitation, ex : messages *et* événements

Concurrence potentielle

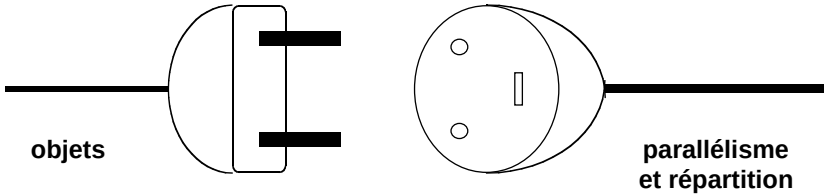
- *Simula-67* [Birtswistle et al.'73]
  - *body* d'une classe : corps de programme exécuté lors de la création d'une instance
  - *coroutines* : suspension (`detach`) et relance (`resume`)
- Objets <-> Processus [Meyer, CACM'9/93]
  - variables
  - données persistantes
  - encapsulation
  - moyens de communication
- Contraintes technologiques et culturelles ont fait régresser ces potentialités parmi les successeurs directs de Simula-67

Répartition potentielle

- Objet = unité naturelle de répartition
  - unité fonctionnelle
  - transmission de messages
    - » indépendance services / implémentation (*encapsulation*)
    - » indépendance services / localisation (*transparence*)
  - autonomie et relative complétude facilite migration/duplication
- Architecture client/serveur <-> Objet
  - analogue
  - MAIS dichotomie client/serveur est *dynamique* chez les objets
    - » un objet envoie un message : *client*
    - » le même objet reçoit un message : *serveur*

Limites

- Mais malgré ses potentialités les concepts d'objet ne sont pas suffisants pour aborder les enjeux de la programmation parallèle et répartie :
  - contrôle de concurrence
  - répartition
  - résistance aux pannes



Travaux développés en parallèle

- Diverses communautés
  - programmation parallèle
  - programmation répartie
  - systèmes d'exploitation
  - bases de données

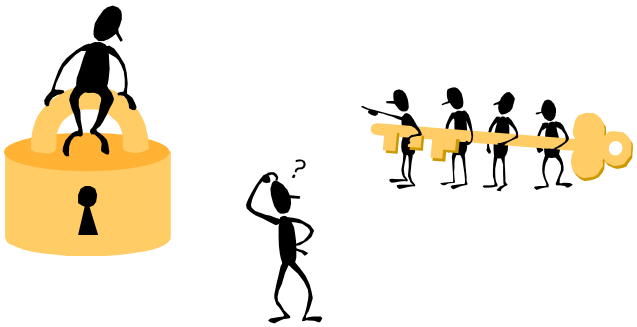
  ont développé différents types d'abstractions :
  - synchronisation
  - transactions
  - communication de groupes
  - ...

  pour aborder de tels besoins

Articulation

- La question est par conséquent :

Comment doit-on lier les concepts d'objet
aux acquis et enjeux de la programmation parallèle et répartie ?



Une classification des approches possibles

- Nous distinguons trois approches principales :
  - approche ***applicative***
    - » application *telle quelle* des concepts d'objet à la conception de programmes et systèmes parallèles et répartis
    - » processus, fichiers, ressources... sont des objets
    - » bibliothèques / frameworks
    - » Ex : *Smalltalk* [Goldberg et Robson'89], *Choices* [Campbell et al., CACM'9/93]
  - approche ***intégrée***
    - » identification et unification des concepts d'objet avec les concepts de la programmation parallèle et répartie
      - object = activité -> objet actif
      - transmission de message = synchronisation /et/ invocation distante
    - » ex : *Actors* [Agha'86], *Java RMI*
  - approche ***réflexive***
    - » séparation entre fonctionnalités (programme générique) et mise en œuvre (modèle d'exécution, protocoles de synchronisation, de répartition, de résistance aux fautes...)
    - » protocoles exprimés sous la forme de bibliothèques de méta-programmes/objets
    - » ex : *CLOS MOP* [Kiczales et al.'91], *OpenC++* [Chiba, OOPSLA'95], *CodA* [McAffer, ECOOP'95]

Complémentarité des approches

- Ces approches ne sont pas en compétition
- Elles ont des objectifs/niveaux complémentaires
  - approche applicative destinée aux concepteurs de systèmes :
    identification des abstractions fondamentales
    utilisationdes classes et de l'héritage pour structurer, classifier, spécialiser/réutiliser
  - approche intégrée destinée aux concepteurs d'applications :
    langage de haut niveau uniforme (minimum de concepts)
    -> maximum de transparence pour l'utilisateur
  - approche réflexive destinée aux concepteurs de systèmes adaptables :
    les concepteurs d'application peuvent spécialiser dynamiquement le système selon les besoins propres de leurs applications

Approche applicative

- Principes
  - appliquer *tels quels* les concepts d'objet à la structuration et la modularité de systèmes complexes
  - bibliothèques et frameworks
  - les différentes abstractions sont représentées par des classes (ex : en *Smalltalk*, processus, sémaphore, fichier...)
  - l'héritage permet de spécialiser statiquement un système générique
    (ex : dans *Choices*, sous classes concrètes correspondant à différents formats de fichiers, réseaux de communication, etc.)
  - les différents services sont représentés par différents objets/composants spécialisés (ex : systèmes d'exploitation à «micro-kernel», e.g. *Chorus [Rozier et al.'92])*
- Gains
  - compréhensibilité
  - extensibilité
  - efficacité

Smalltalk

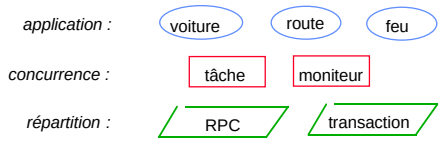
- Langage de programmation par objets minimal
- Riches bibliothèques de classes représentant :
  - constructions du langage (ex : structures de contrôle, `ifTrue:ifFalse:`)
  - ressources (messages, multi-tâche, compilateur...)
  - outils de l'environnement (ex : browser, debugger...)
- Concurrence
  - processus (tâches) (`Process`)
  - séquenceur (`ProcessorScheduler`)
  - sémaphores (`Semaphore`)
  - communication (`SharedQueue`)
  - *aisément extensibles ,* (ex :*Simtalk* [Bézivin, OOPSLA'97], *Actalk* [Briot, ECOOP'92])
- Répartition
  - communications (*sockets Unix*, *RPCTalk*...)
  - stockage (*BOSS*) -> persistance, encodage...
  - briques de base pour construire divers services répartis (*DistributedSmalltalk*, *GARF* [Mazouni et al., TOOLS'95] , *BAST* [Garbinato et al, ECOOP'96]...)

Autres exemples

- C++
  - bibliothèque de threads : C++, *ACE* [Schmid'95]
  - bibliothèque de répartition : *DC++* [Schill et Mock, DSE'93]
  - *Choices* [Campbell et al., CACM'9/93]
    - » classes abstraites : `ObjectProxy`, `MemoryObject`, `FileStream`, `ObjectStar`, `Disk`
    - » spécialisables pour des environnements spécifiques (fichiers Unix ou MS, disque SPARC, mémoire partagée...)
- Beta
  - bibliothèques de répartition [Brandt et Lehrman Madsen, OBDP-LNCS'94]
    - » Classes `NameServer`, `ErrorHandler`
- Eiffel
  - bibliothèques pour parallélisme de données (SPMD)
    - » structures de données abstraites répartissables en *EPEE* [Jezequel, JOOP'93]

Bilan

- Avantages : structuration, modularité, extensibilité
- Objectif de fond : dégager les abstractions minimales
  - synchronisation, atomicité, persistance, transaction...
- Limitations :
  - le programmeur a deux (ou même trois) tâches distinctes :
    - » programmer son application en termes d'objets
    - » gérer le parallélisme et la répartition
      - également par des objets,
      - mais ***PAS LES MÊMES !!!***



  - possible lourdeur
    - » ex : classe Concurrency [Karaorman/Bruno, CACM'9/93]
    - » encapsule activité ainsi que transmission de message distante et asynchrone
    - » ***MAIS*** impose une certaine dose de manipulation explicite des messages
  - (manque de) transparence
  - complexité (trop de dimensions différentes et indépendantes à gérer)

Approche intégrée

- Principes :
  - fusion des concepts d'objet avec les concepts de la programmation parallèle et répartie
  - offrir au programmeur un cadre conceptuel (objet) unique
  - plusieurs dimensions d'intégration possibles :
    - » objet <-> activité -> objet actif
      - ex : *Acteurs*
    - » activation <-> synchronisation -> objet synchronisé
      - transmission de messages : synchronisation appelant/appelé
      - au niveau de l'objet : synchronisation des invocations
      - ex : *Guide* [Balter et al., Computer Journal'94], *Arjuna* [Parrington et Shrivasta, ECOOP'88], *Java [Lea'97]*
    - » objet <-> unité de répartition -> objet réparti
      - ex : *Emerald* [Jul et al.'98]
- Gains
  - simplicité

Approche intégrée (suite)

- Ces trois dimensions sont relativement indépendantes entre elles
- Ex : Java

objet actif	NON	un thread est un objet mais tout objet n'est pas un thread
objet synchronisé	OUI	à chaque objet un verrou (en fait un moniteur) est associé
objet réparti	NON	OUI avec Java RMI

Objet actif

- objet = activité
  - une activité : séquentiel
  - plusieurs activités
    - » quasi-concurrent (ex : *ABCL/1* [Yonezawa'90])
    - » concurrent (ex : *Actors* [Agha'86])
    - » ultra-concurrent (ex : acteur non sérialisé)
- objet est réactif <-> activité (tâche/processus) est autonome
  - dans l'union, qui l'emporte ??
    - » objet actif réactif (ex : *Actors*)
    - » objet actif autonome (ex : *POOL* [America, OOCP'87], *CEiffel* [Löhr, OOPSLA'92])
- acceptation implicite ou explicite de messages
  - implicite (ex : *Actors*)
  - explicite
    - » concept de body (hérité de *Simula*), ex : *POOL*, *Eiffel//* [Caromel, CACM/9/93]

Objet synchronisé

- synchronisation au niveau de la transmission de messages
  - transmission de messages : synchronisation implicite appelant/appelé (transmission *synchrone*)
  - transparent pour le client
  - dérivations/optimisations :
    - » transmission *asynchrone*, ex : *Actors*
    - » transmission avec réponse anticipée (*future*), ex : *ABCL/1*, *Eiffel//*
- synchronisation (des invocations) au niveau de l'objet
  - synchronisation intra-objet
    - » en cas de concurrence intra-objet (multiples invocations)
    - » ex : multiples lecteurs / un écrivain
  - synchronisation comportementale
    - » dynamicité des services offerts
    - » ex : buffer de taille bornée, le service `put:` devient temporairement indisponible pendant que le tampon est plein
    - » transparent pour le client
  - synchronisation inter-objets
    - » ex : transfert entre comptes bancaires, transaction

Formalismes de synchronisation

- Origines : systèmes d'exploitation, parallélisme, bases de données
- Intégration relativement aisée dans un modèle objet
  - formalismes centralisés, associés au niveau des classes
    - » *path expressions* (specif. abstraite des entrelacements possibles entre invocations)
      - ex : *Procol* [Lafra'91]
    - » *body* (procédure centralisée décrivant l'activité et les types de requêtes à accepter)
      - ex : *POOL*, *Eiffel//*
    - » *comportements abstraits* (synchronisation comportementale)
      - empty = {put:}, full = {get}, partial = empty U full
      - ex : *Act++* [Kafura, ECOOP'89] *Rosette* [Tomlinson, ECOOP'88]
  - formalismes décentralisés associés au niveau des méthodes
    - » *gardes* (conditions booléennes d'activation)
    - » compteurs de synchronisation
      - ex : *Guide*
    - » *Java* : verrou (lock) au niveau de l'objet avec mot clé `synchonized` au niveau des méthodes

Objet réparti

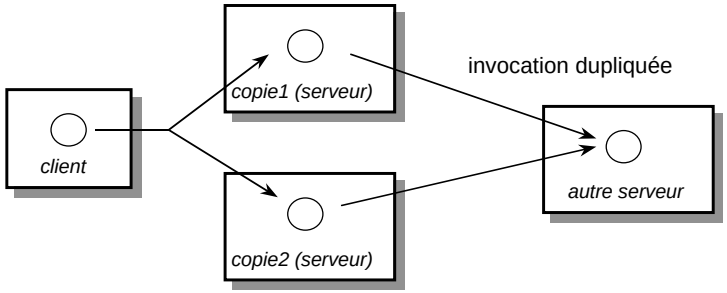
- objet : unité indépendante d'exécution
  - données, traitements, et même ressources (si objet actif)
  - transmission de messages conduit à la transparence de la localisation
  - autonomie et relative atomicité de l'objet facilite migration et duplication
- association de l'invocation distante à la transmission de messages
  - *Java RMI*
- association des transactions à la transmission de messages
  - synchronisation inter-objets et résistance aux pannes
    - » *Argus* [Liskov'83]
- mécanismes de migration
  - meilleure accessibilité, ex : *Emerald (call by move)*
- mécanismes de réplication
  - meilleure disponibilité (dupliquer les objets trop sollicités)
  - résistance aux pannes (ex : *Electra* [Maffeis'95])

Limite 1 : Spécialisation de la synchronisation

- spécialisation des conditions de synchronisation
  - approche naturelle : utiliser l'héritage
  - *MAIS* cela ne marche pas si bien ! (inheritance anomaly [Matsuoka RDOBCP'93])
  - formalismes centralisés -> le plus souvent redéfinition complète
  - formalismes décentralisés -> peut induire des redéfinitions nécessaires
    - » ex : compteurs de synchronisation
      - nouvelle méthode en exclusion mutuelle -> clause à rajouter dans toutes les méthodes
    - » ex : comportements abstraits
      - méthode `get2` retirant deux éléments d'un tampon borné -> oblige à subdiviser le comportement abstrait `partial` en deux sous-comportements : `one` et `partial`
- directions :
  - spécifications plus abstraites [McHale 94]
  - séparation entre synchronisation comportementale et intra-objet [Thomas PARLE'92]

Limite 2 : Duplication des invocations

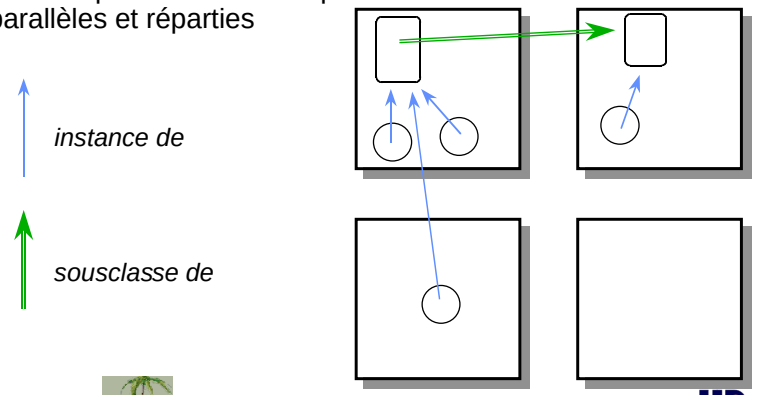
- Application directe des protocoles de duplication de serveurs (pour gérer la tolérance aux pannes) aux objets
  - *PROBLEME* : Ces protocoles font l'hypothèse qu'un serveur restera toujours un serveur simple (i.e., n'invoquera pas d'autres serveurs en tant que client)
  - Cette hypothèse ne tient plus dans le monde objet...
  - Si le serveur dupliqué invoque à son tour un autre objet, cette invocation sera dupliquée. Ce qui peut conduire à des incohérences



  - Solution possible : *pré-filtrage* par un coordinateur arbitrairement désigné (un des serveurs dupliqué) (en fait solution un peu plus complexe pour résistance aux pannes du coordinateur -> *post-filtrage*) [Mazouni et al. TOOLS-Europe'95]

Limite 3 : factorisation vs répartition

- Les stratégies standard de mise en œuvre (implémentation) des concepts d'objet (factorisation : classe et héritage) ont fait des hypothèses FORTES (séquentialité et mémoire centralisée)
  - Lien instance - classe
  - Lien classe - surclasse
- Elles ne peuvent être transposées directement à des architectures parallèles et réparties

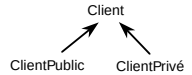


factorisation vs répartition (2)

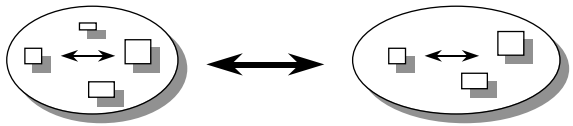
- Solution1 : dupliquer l'ensemble des classes
  - Cela suppose qu'elles sont immutables
    - » constantes de classe OK, mais pas de variables de classe
    - » problème de mise à l'échelle
- Solution2 : partitionner statiquement les classes en modules [Gransart'95]
  - Mais complexifie les possibilités de migration

factorisation vs répartition (3)

- Solution2' : méthodologie de partitionnement plus fine [Purao et al., CACM'8/98]
  - reconception d'une application existante



  - méthode semi-automatique
  - environnement aide et réalise les choix qui restent a la charge de l'utilisateur
  - modèle d'architecture : hiérarchique (à clusters)
  - distinction entre :
    - » communication inter-sites - grand coût
    - » communication inter-processeurs (intra-site) - faible coût



factorisation vs répartition (4)

phase 1 : répartition entre les sites

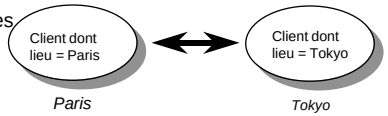
» refactorisation «roll up» des attributs/méthodes de sous-classes dans une super-classe pour éviter que l'héritage ne «traverse» PLUSIEURS sites

Client (Privé ou/et Public)

» puis fragmentation (spécialisation) des classes
- à partir de scénarios d'interaction

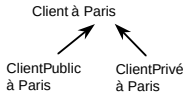
Client dont lieu = Paris
Client dont lieu = Tokyo

» allocation des fragments (= sous ensembles d'instances) sur les différents sites
- critère : minimiser les communications inter-sites

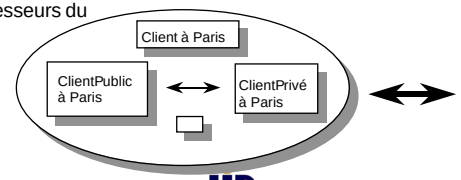


– phase 2 : répartition à l'intérieur d'un site donné

» redépliement de l'héritage «roll down» des fragments

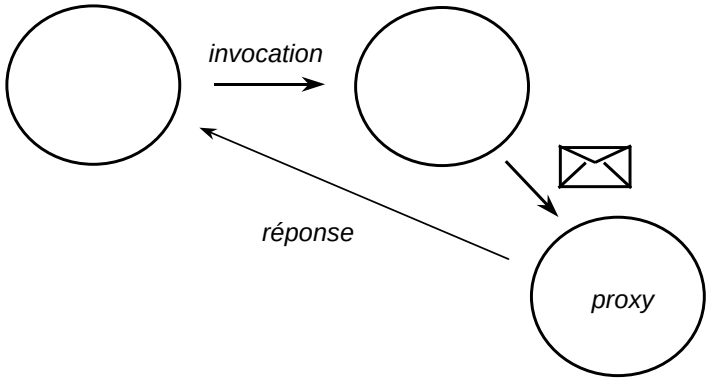


» optimisation de l'allocation des fragments sur les processeurs du site
- décision multi-critères :
  – adéquation du processeur
  – concurrence
  – communication inter-processeurs
  – réplication des instances



Délégation

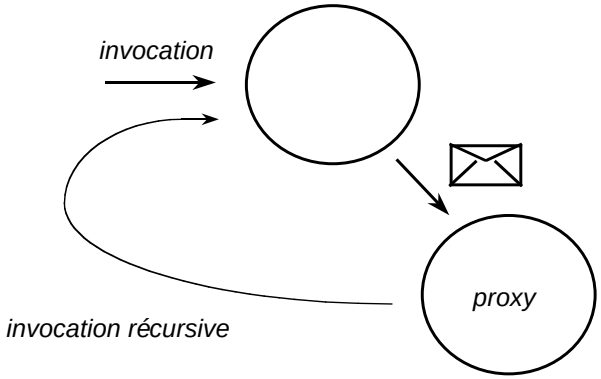
- Le mécanisme de délégation [Lieberman OOCP'87] offre une alternative a priori séduisante à l'héritage
  – Repose uniquement sur la transmission de messages, donc indépendant d'une hypothèse de mémoire centralisée



Délégation (2)

- *PROBLEME* : ordonnancement correct des invocations récursives, qui doivent être traitées AVANT les autres invocations

  -> synchronisation non triviale



Bilan

- Approche intégrée séduisante
  – nombre minimal de concepts
  – cadre unique
- Mais problèmes dans certains secteurs d'intégration
- Uniformité peut-être trop réductrice
  – Limites de la transparence et donc du contrôle
  – Problèmes d'efficacité
    » tout objet est actif
    » toute transmission de messages est une transaction
- Réutilisation des programmes standards/séquentiels existants
  – Identifier les activités et les encapsuler dans des objets actifs
  – Règles de cohabitation entre objets actifs et objets passifs

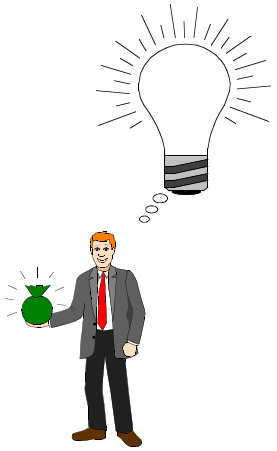
Méta-Bilan

- Objectif : réconcilier le meilleur de l'approche applicative et de l'approche intégrée
- Observation : l'approche applicative et l'approche intégrative ne sont pas au même niveau
  - approche applicative pour le concepteur
  - approch intégrative pour l'utilisateur
- Comment interfacer des bibliothèques de composants et de protocoles destinées au concepteur (approche applicative)

  avec un langage uniforme destiné à l'utilisateur (approche intégrée)??

Réflexion

- Le concept de *réflexion* (méta-programmation, architectures réflexives...) offre justement un cadre conceptuel permettant un découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre*

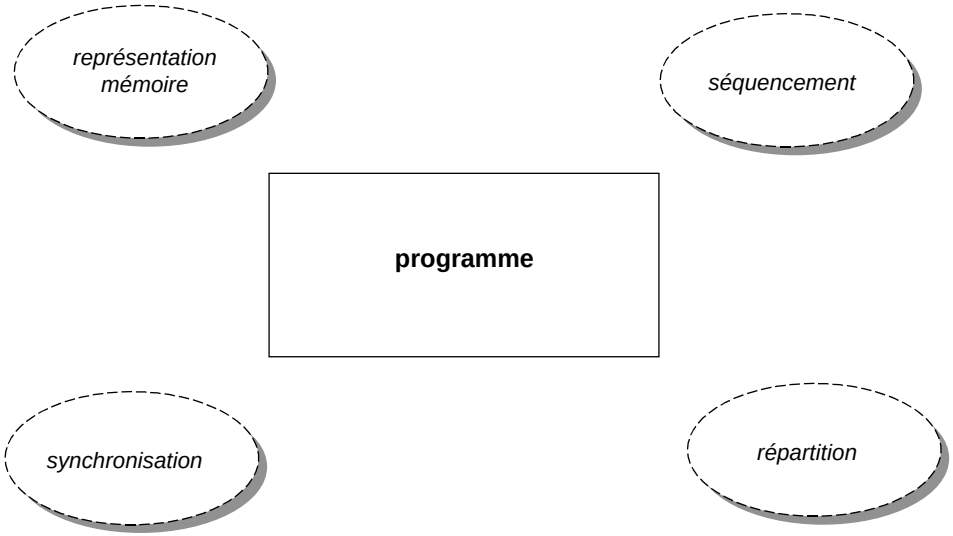


Réflexion (2)

- Diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes (*réifiées*) sous la forme de *méta-programmes*.
  - Habituellement elles sont invisibles et immuables (interprète, compilateur, moniteur d'exécution...)
- La spécialisation de ces méta-programmes permet de *particulariser* (éventuellement dynamiquement) l'exécution d'un programme
  - » représentation mémoire
  - » modèle de calcul
  - » contrôle de concurrence
  - » séquencement
  - » gestion des ressources
  - » protocoles (ex : résistance aux pannes)

  avec le minimum d'impact sur le programme lui-même

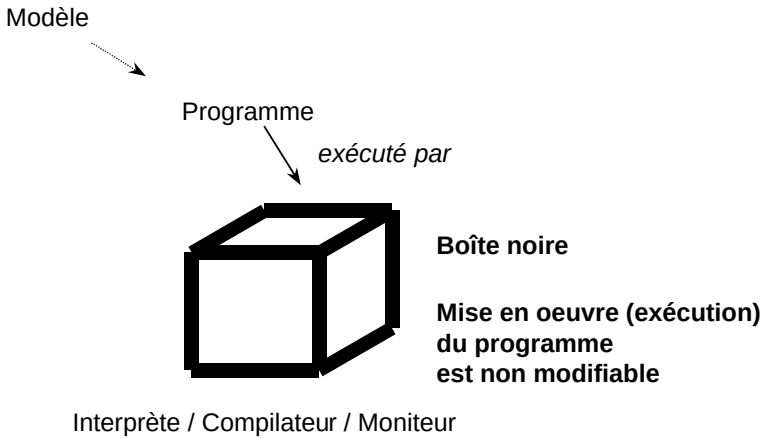
Contexte d'exécution



(Retour à un vieux) Dilemne

- Ecrire de BEAUX programmes
  - lisibles
  - concis
  - modulaires
  - abstraits
  - génériques
  - réutilisables
- Ecrire des programmes EFFICACES
  - spécialisés
  - choix optimaux de représentation interne des données
  - contrôle optimisé
  - gestion des ressources adéquate
- DILEMNE : Spécialiser/optimiser des programmes tout en les gardant génériques

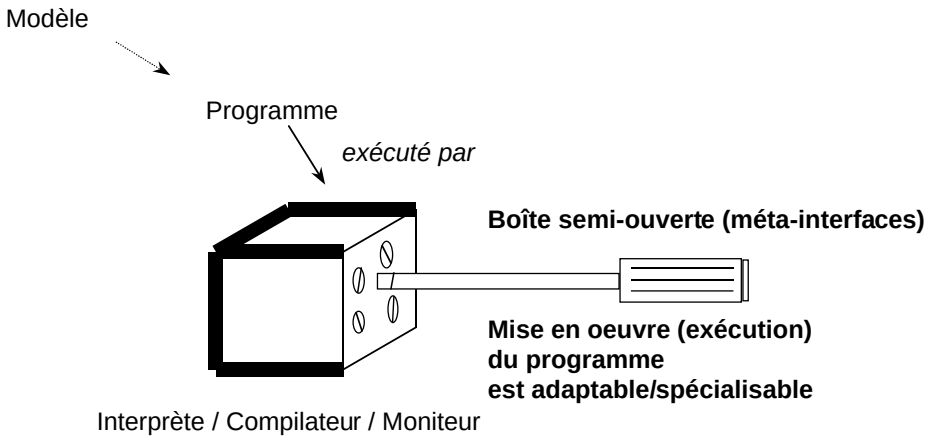
Boîte noire



Solutions Ad-Hoc

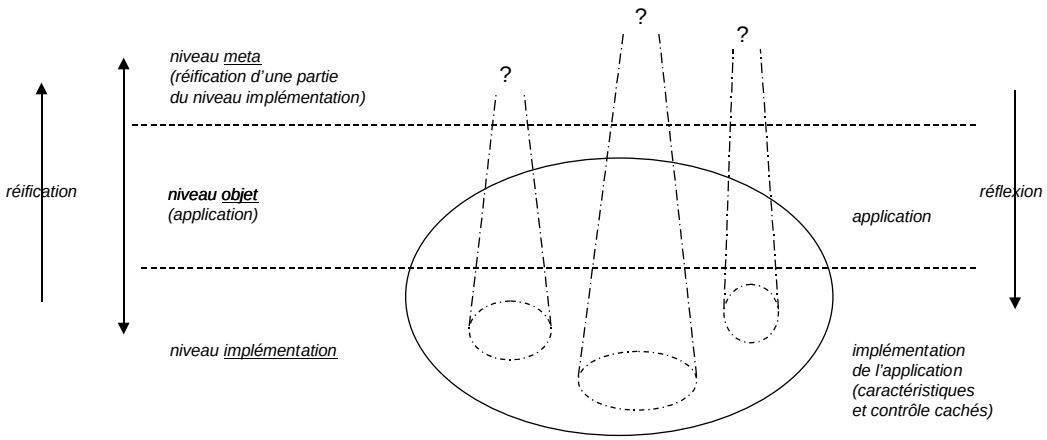
- Coder "entre" les lignes
  - difficile à comprendre
  - difficile à maintenir (hypothèses cachées)
  - peu réutilisable
- Annotations/Directives (déjà mieux)
  - ex : High Performance Fortran (HPF)
  - mais
    - » notations de plus ou moins bas niveau
    - » ensemble/effet des annotations non extensible/adaptable

Réflexion (3)



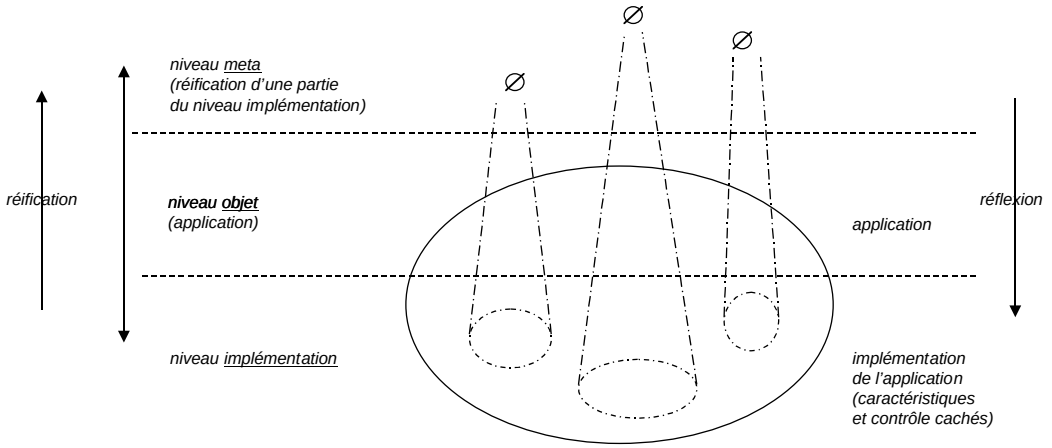
Open Implementation [Kiczales 94]

Réification/réflexion



Réification/réflexion

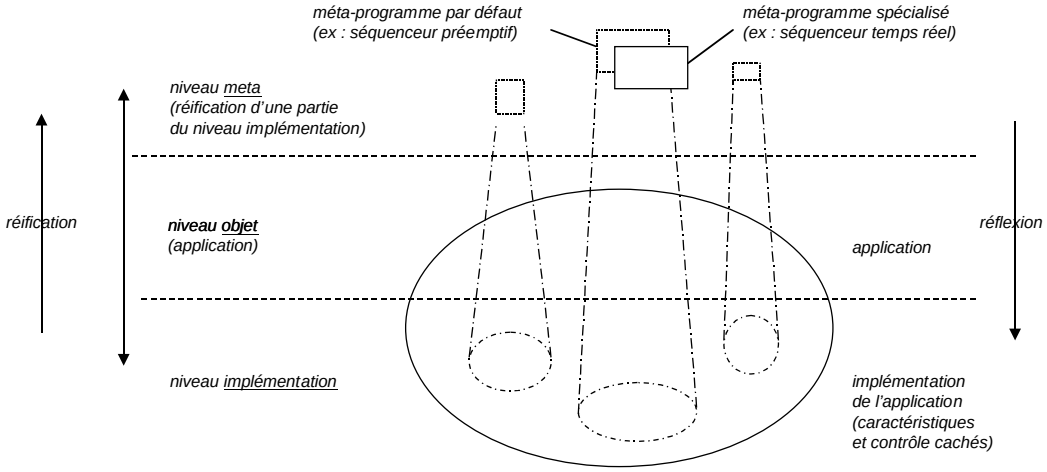
- Réification numérique (potentiomètres)
  - Ex : Options de compilation d'un compilateur
    - Efficacité vs taille du code généré



Réification/réflexion (2)

- Réification logicielle (méta-programmes)
  - Ex : algorithme de séquencement (scheduler)

plus général/flexible que des potentiomètres



Réflexion (4)

- Découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre (exécution)*
- Séparation entre programme ET méta-programme(s) favorise :
  - généricité et réutilisation des programmes
  - et des méta-programmes
- Ex :
  - changer la stratégie de contrôle pour un programme donné
  - réutiliser une stratégie de contrôle en l'appliquant à un autre programme

Structure / Dynamique

- Structure (représentation)
  - spécialiser la création des données
    - » méthodes de classe (= méthodes de métaclasses) en Smalltalk
    - » constructor member functions en C++, en Java
  - spécialiser un gestionnaire de fenêtres
    - » implantation d'une feuille de calcul en Silica [Rao]
- Dynamique (comportement/exécution)
  - implémenter des coroutines via la manipulation de continuations
    - » call/cc en Scheme
  - spécialiser le traitement d'erreur
    - » doesNotUnderstand: en Smalltalk
  - changer l'ordre de déclenchement de règles de production
    - » méta-règles en NéOpus

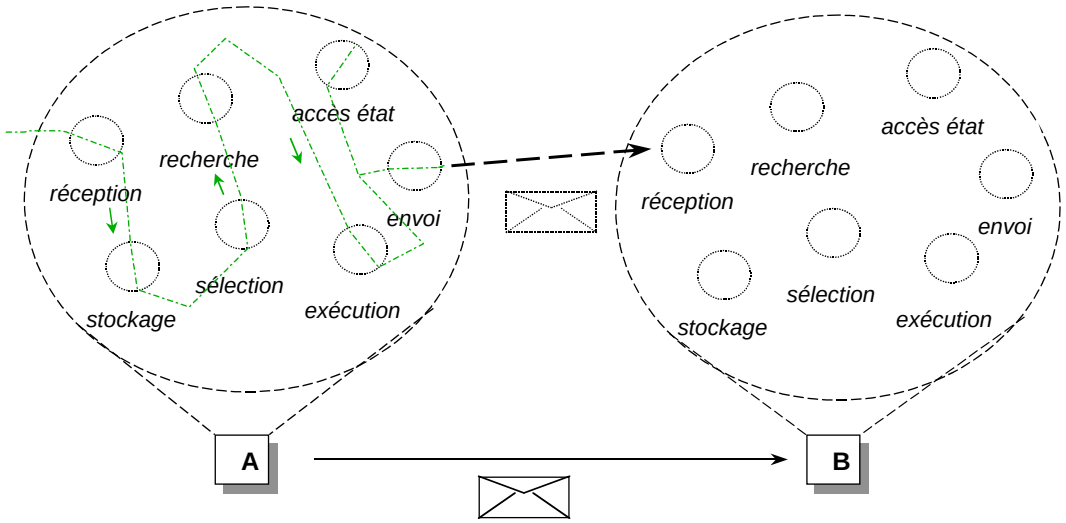
Approche réflexive

- Réflexion permet d'intégrer intimement des (méta-)bibliothèques de contrôle avec un langage/système
- Offre ainsi un cadre d'interface entre approche applicative et approche intégrée
- La réflexion s'exprime particulièrement bien dans un modèle objet
  - modularité des effets
  - encapsulation des niveaux
- méta-objet(s) au niveau d'un seul objet
- méta-objets plus globaux (ressources partagées : séquencement, équilibre de charges...)
  - *group-based reflection* [Watanabe'90]

Méta-objets/composants

- *CodA* [McAffer ECOOP'95] est un exemple de modèle relativement général d'architecture réflexive
- Sept méta-objets/composants de base :
  - envoi de message
  - réception de messages
  - stockage des messages reçus
  - sélection du premier message à traiter
  - recherche de méthode correspondant au message
  - exécution de la méthode
  - accès à l'état de l'objet
- Les méta-composants sont :
  - spécialisables
  - (relativement) combinables

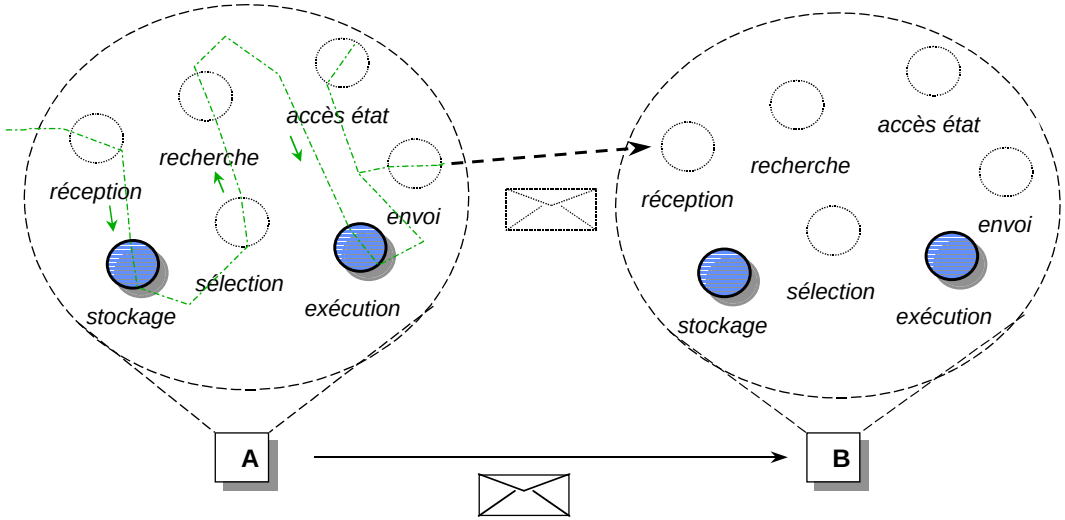
CodA



Ex : Exécution concurrente

- envoi de message
- réception de messages
- **stockage des messages reçus**
  - » file d'attente (FIFO)
- sélection du premier message à traiter
- recherche de méthode correspondant au message
- **exécution de la méthode**
  - » processus associé
  - » boucle infinie de sélection et traitement du premier message
- accès à l'état de l'objet

Exécution concurrente (2)

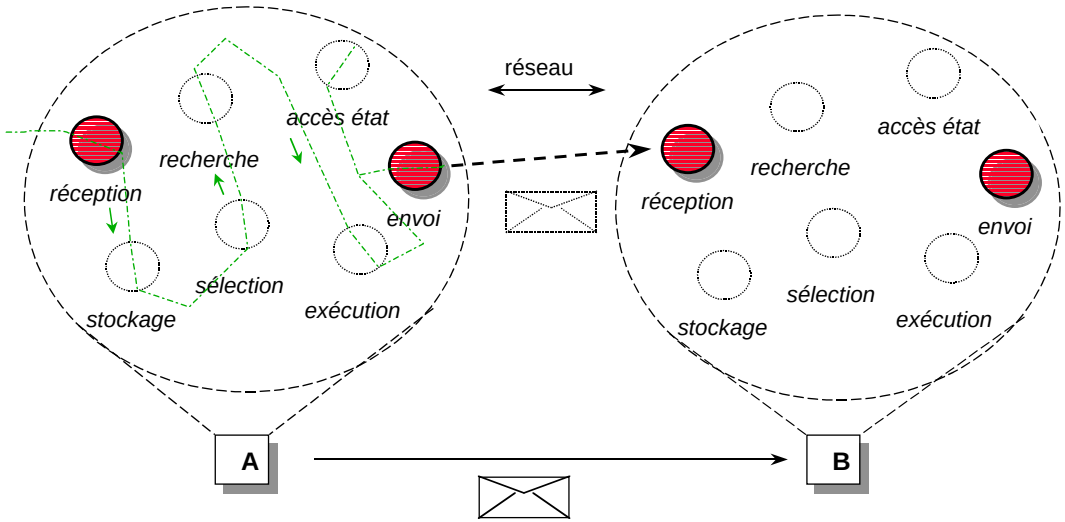


Ex : Exécution répartie

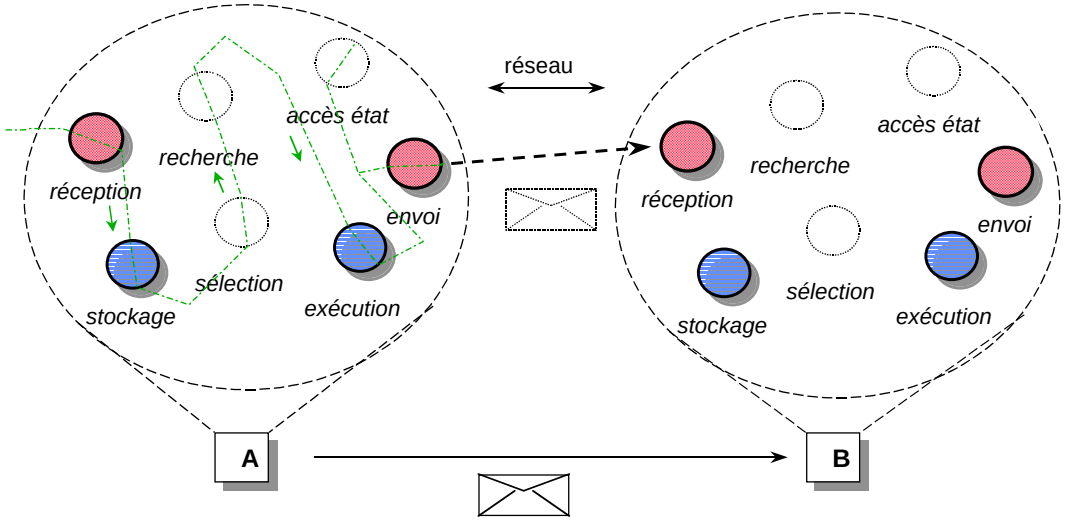
- **envoi de message**
  - » encodage des messages, envoi via le réseau
- **réception de messages**
  - » réception via le réseau, décodage des messages
- stockage des messages reçus
- sélection du premier message à traiter
- recherche de méthode correspondant au message
- exécution de la méthode
- accès à l'état de l'objet

- **encodage**
  - » discipline d'encodage (marshal/unmarshal)
- **référence distante**
- **espace mémoire**

Exécution répartie (2)



Exécution concurrente et répartie (composition)

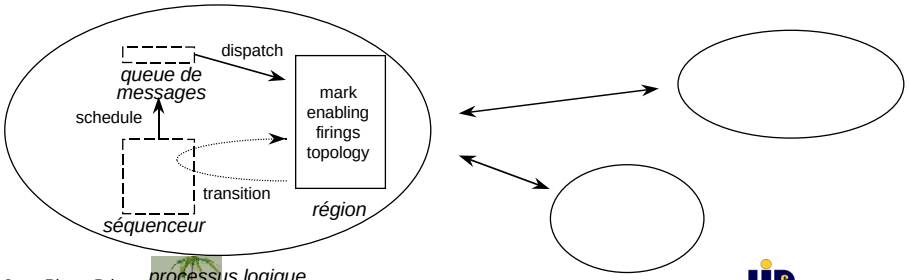


Méta-composants de plus gros grain (autres ex.)

- *Actalk* [Briot, LMO'94]
  - » activité / synchronisation (concurrence intra, acceptation explicite, gardes, compteurs de synchro...)
  - » communication (synchrone, asynchrone, future...)
  - » invocation (estampillage temporel, priorités...)
- *GARF* [Garbinato et al. 94]
  - » objet (persistant, dupliqué...)
  - » communication (multicast, atomique...)
  - » *ex : résitance aux pannes*
- *MAUD* [Agha et al. DCCA'93]
  - » envoi des messages
  - » réception des messages
  - » état
  - » *ex : installation dynamique de protocoles (duplication de serveurs, atomicité)*
- *AL1/D* [Okamura ECOOP'94]
  - » *ex : contrôle de la migration*

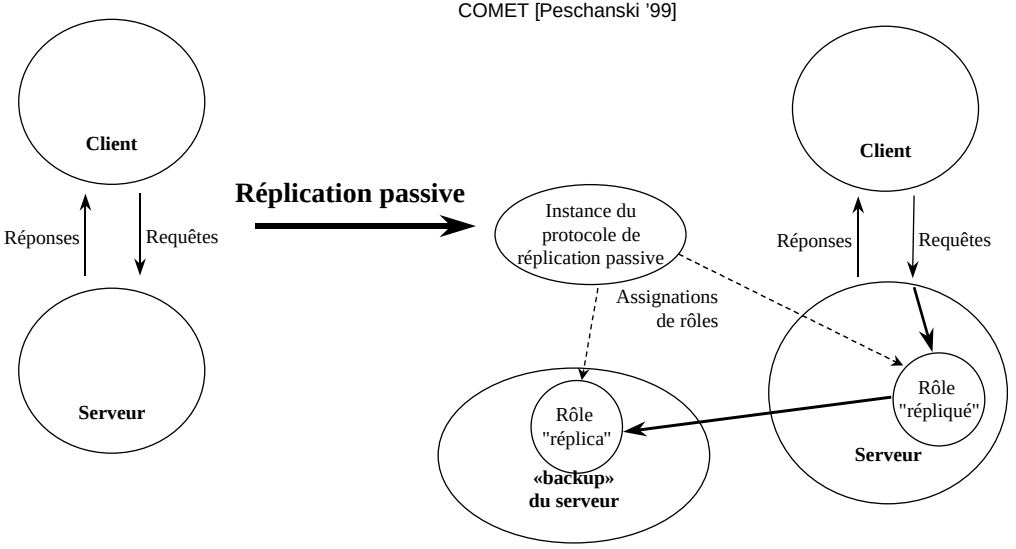
Temps réel

- méta-acteurs associés à des acteurs
  - – contrôle du temps pour du «soft real time» [Honda'92]
- machine de contrôle [Nigro et al., FMOODS'97] pour un ensemble d'acteurs
  - – méta-composants :
    - » horloge, queue de messages (= liste d'événements), contrôleur (période de simulation), séquenceur
    - » permettent de modifier les aspects temporels de l'application indépendamment de l'application elle-même
    - » ex : simulation distribuée optimiste (Time Warp) de réseaux de Petri temporels (timed Petri nets)

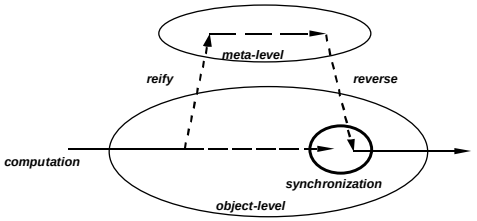


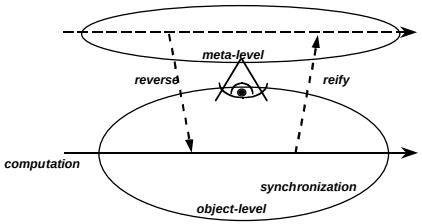
Réflexion sur un ensemble d'objets
Ex : Installation d'un protocole de réplication passive

COMET [Peschanski '99]



Réflexion dans les systèmes concurrents





Systèmes commerciaux

- *Aperios* [Yokote OOPSLA'92]
  - spécialisation dynamique de la politique de séquencement (ex : passer au temps réel)
  - application au «video on demand»
- Moniteur de transaction [Barga et Pu '95]
  - Incorporation de protocoles transactionnels étendus (relâchant certaines des propriétés standard : ACID)
  - dans un système existant
  - réification a posteriori via des upcalls
    - » (délégation de verrou, identification de dépendances, définition de conflits)

Bilan - Conclusion

- Approche réflexive prometteuse
- Architectures réflexives encore plus ou moins complexes, mais méthodologie s'établit et s'affine
- Validations en vraie grandeur en cours
- Retour du problème clé de la composition arbitraire (de méta-composants)
- (In)Efficacité
  - réduction de la portée de la réflexion (compilation)
    - » ex : OpenC++ version 2 [Chiba, OOPSLA'95]
  - transformation de programmes - évaluation partielle
    - » [Masuhara et al., OOPSLA'95]
- Ne dispense pas du travail nécessaire à l'identification des bonnes abstractions

Exemple : CORBA

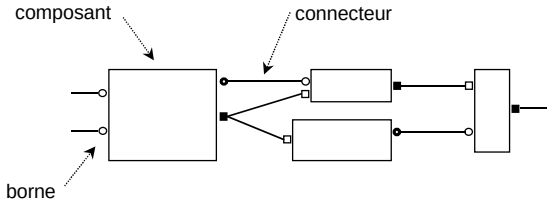
- approche applicative :
  - structuration en bibliothèques
    - » services (ex : nommage, événements, transactions..)
    - » facilités (ex : interface utilisateur, gestion de tâches...)
    - » domaine d'application
- approche intégrative
  - objet distribué
    - » intégration transmission de message avec invocation distante
    - » transparence pour l'utilisateur
- approche réflexive
  - réification de certaines caractéristiques de la communication
    - » ex : smart proxies de Orbix (IONA)
      - ex d'utilisation : implantation de transmission de messages asynchrone
      - intégration des services avec la communication distante

Architectures logicielles

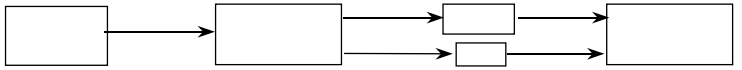
- Programmation à grande échelle
- Configuration et reconfiguration d'applications modulaires/réparties

- Composants
  - clients, serveurs, filtres, couches...

- Connecteurs
  - appels de procédure, messages, diffusion d'événements, pipes...



Ex1 : Pipes & filters



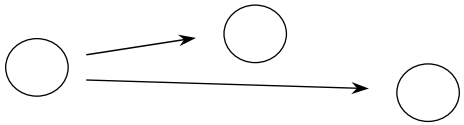
- Composants : filters
- Connecteurs : pipes

- Ex : Unix shell `dvips | lpr`

- +
  - » compositionalité (pipeline)
  - » réutilisabilité
  - » extensibilité
  - » analyses possibles (débit, deadlock...)
  - » concurrent

- -
  - » «batch», pas adéquat pour systèmes interactifs, ex : interfaces homme-machine
  - » petit dénominateur commun pour la transmissiond e données
    - performance
    - complexité

Ex2 : Objets & messages



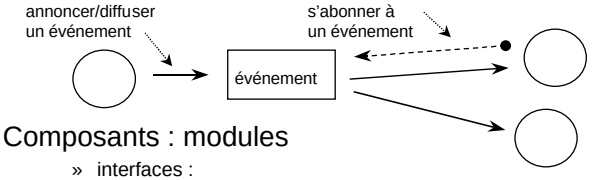
- Composants : objets
- Connecteurs : transmission de messages

- Ex : Java

- +
  - » encapsulation
  - » décomposition

- -
  - » références directes
    - nécessité de recoder les références si reconfiguration

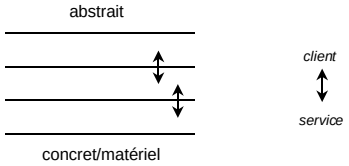
Ex3 : Diffusion d'événements (publish/subscribe)



- Composants : modules
  - » interfaces :
    - procédures
    - événements
- Connecteurs : diffusion d'événements

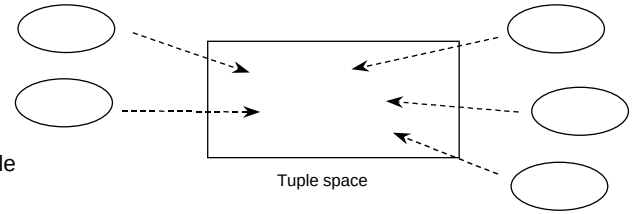
- Ex : interfaces homme machine, bases de données (contraintes d'intégrité), Java Beans
- +
  - » réutilisation
  - » évolution
- -
  - » contrôle externe aux composants
    - difficile de déterminer quels modules seront activés, dans quel ordre...
  - » validation difficile

Ex4 : Systèmes en couches (layered systems)



- Composants : couches
- Connecteurs : appels de procédures
- Ex : protocoles de communication/réseaux, bases de données, systèmes d'exploitation (ex : Unix)
- +
  - » niveaux croissants d'abstraction
  - » extensibilité
  - » réutilisabilité
- -
  - » pas universel
  - » pas toujours aisé de déterminer les bons niveaux d'abstraction
  - » performance

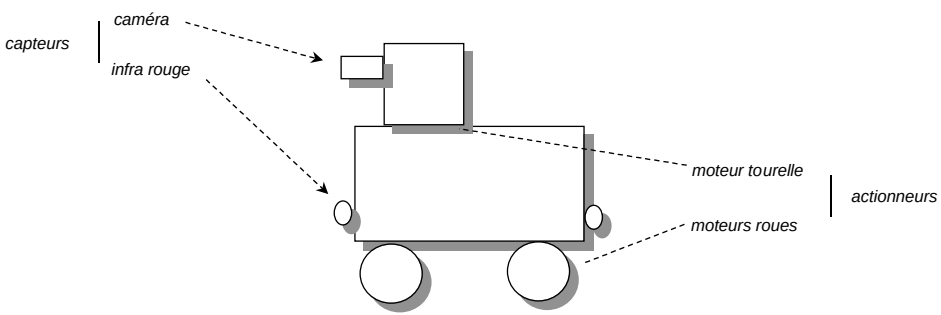
Ex4 : Repositories



- Composants :
  - structure de données centrale
  - processus
- Connecteurs : accès directs processus <-> structure
  - processus -> structure, ex : bases de données
  - structure -> processus, ex : démons, data-driven/trigger
- Ex : (Linda)Tuple space, blackboard (tableau noir)
- +
  - » partage des données
- -
  - » contrôle opportuniste

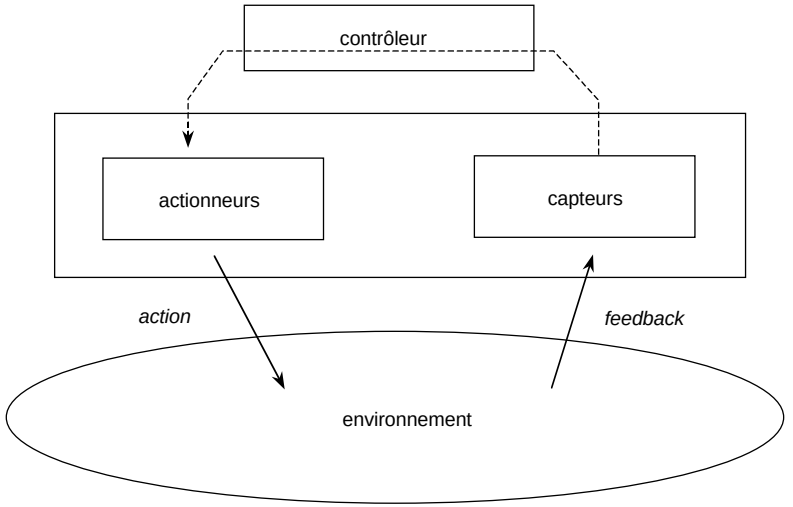
Comparaison de styles architecturaux

- Exemple d'application :
  - (architecture de contrôle d'un) robot mobile autonome

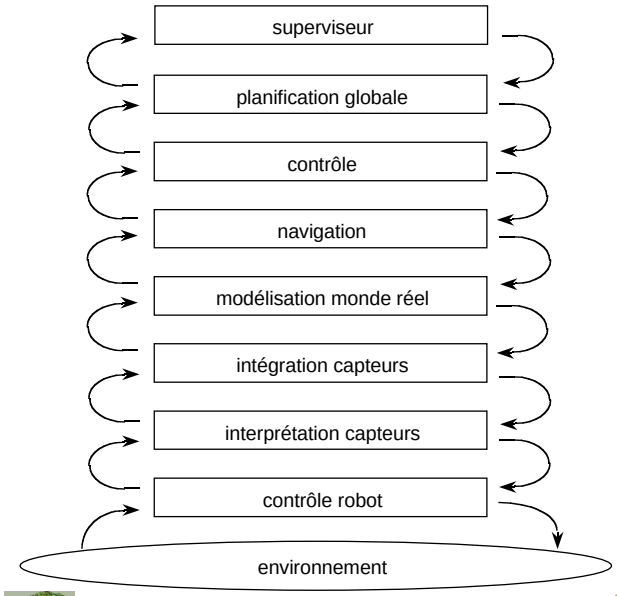


- Propriétés/caractéristiques recherchées :
  - comportement à la fois délibératif et réactif
  - perception incertaine de l'environnement
  - robustesse (résistance aux pannes et aux dangers)
  - flexibilité de conception (boucle conception/évaluation)

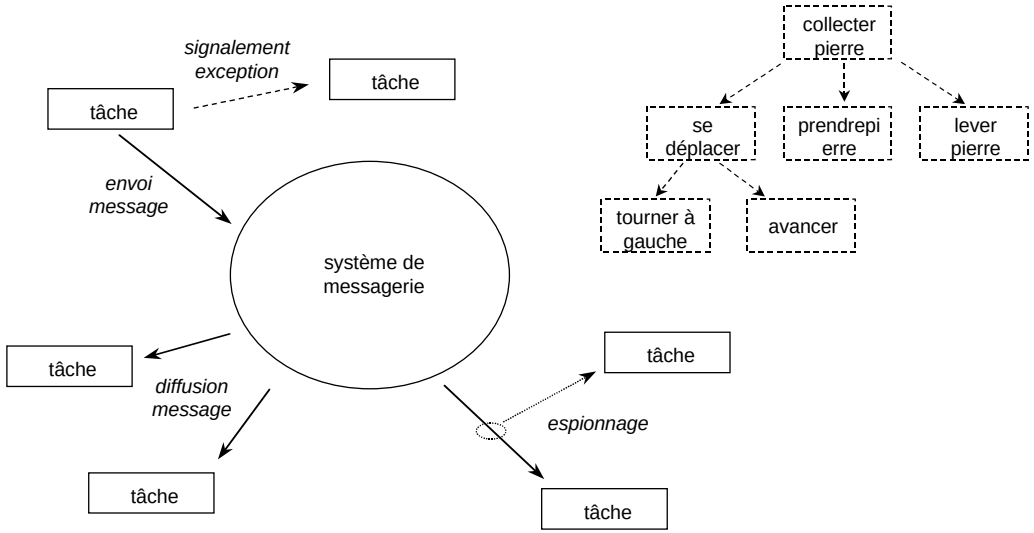
Solution 1 - boucle de contrôle



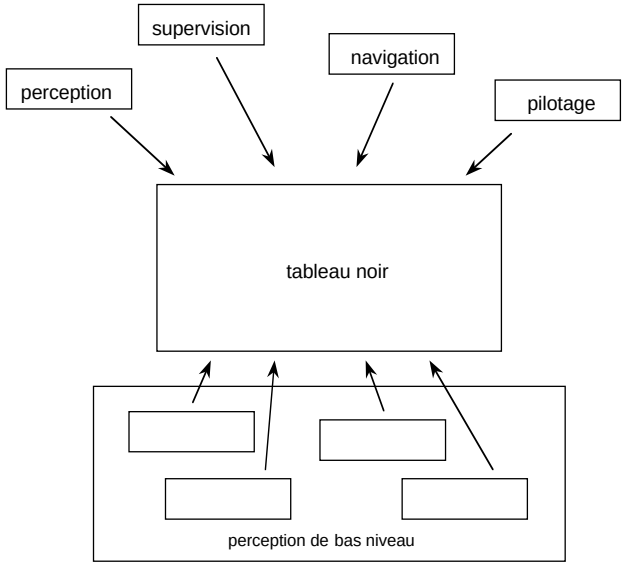
Solution 2 - couches



Solution 3 - (tâches et) événements



Solution 4 - tableau noir

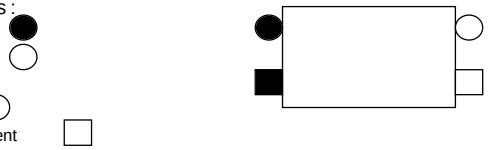


Comparaison

	Boucle de contrôle	couches	événements	tableau noir
coordination des tâches	+ -	-	+ +	+
incertain	-	+ -	+ -	+
robustesse	+ -	+ -	+ +	+
sûreté	+ -	+ -	+ +	+
performance	+ -	+ -	+ +	+
flexibilité	+ -	-	+	+

Langages de description d'architectures

- Architecture Description Languages (ADLs)
  - définition des composants
    - deux types d 'interfaces :
      - requises (in)
      - fournies (out)
    - sémantique d'appel
      - synchrone
      - asynchrone/événement



  - définition des connexions
    - connecteurs utilisés (ex : RPC)
  - vérification de la sémantique d'assemblage
    - conformité types/interfaces
    - contraintes de déploiement (OLAN)
      - ex : taille mémoire minimale, charge machines, etc.
  - Unicon [Shaw et al. 95]
  - Rapide [Lucham & Vera 95]
  - OLAN [Belissard et al. 96]

plus sur les ADLs (et le reste)

- Transparents de cours Ecole d'Eté sur la Construction d'Applications Réparties IMAG-INRIA-LIFL
- http://sirac.imag.fr/ecole/
  - 1998
  - 1999
- En particulier sur les ADLs (exemples en Unicon, OLAN, Rapide...) :
  - http://sirac.imag.fr/ecole/98/cours/composants.pdf
  - transparents de Michel Riveill
  - pages 3-4 et 27-43
  - également http://sirac.imag.fr/ecole/99/cours/99-8.pdf
  - et tous les autres transparents !
    - http://sirac.imag.fr/ecole/98/cours/
    - http://sirac.imag.fr/ecole/99/cours/

Composants

- Un composant est du code exécutable et son mode d 'emploi
  - module logiciel autonome (et persistant)
  - exporte interfaces
  - auto-description
  - « composable »
- Composants « source »
  - architectures logicielles
  - ex : Sun JavaBeans
- Composants binaires
  - ex : Microsoft COM
- « Petits » composants
  - ex : composants graphiques JavaBeans
- « Gros » composants
  - ex : MS Word, ILOG Solver...

Pourquoi les composants ? [Albert et Haren 2000]

- Analyse sur + de 2000 clients de composants (ILOG et autres)
  - 11 Critères pour l 'application développée (à base ou pas de composants) :
- flexibilité offerte *(éventail de choix ou forte rigidité)*
  - ex : fenêtres rondes rares et difficiles à intégrer
  - peut brider l'imagination des architectes
- compétences requises *(communes ou rares/pointues)*
  - conception vs utilisation
- moyens nécessaires au projet *(incluant déploiement et maintenance)*
  + coût de développement important, + composants avantageux
- vitesse de développement
  - excellente avec composants, ex : presque indispensable aux startups
  - mais adaptation composants peut être difficile
- incrémentalité du développement
  - porte sur l'extension de certains composants du prototype
- fiabilité du résultat
  - composants améliorent toujours fiabilité (capitalisation des tests)
  - mais (factorisation fait que la) criticité des composants augmente

Pourquoi les composants ? (2)

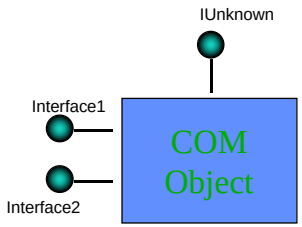
- performance du résultat final
  - performance en général inversement proportionnelle à généricité
  - mais capitalisation de l'optimisation
- facilité de déploiement *(portabilité sur différentes plates-formes)*
  - capitalisation des portages
  - utilisation quasi-générale pour les IHM
- indépendance vis-à-vis des fournisseurs *(possibilités de migrer d'un fournisseur à un autre, absorber la disparition ou rachat par compétiteur...)*
  - actuellement interfaces encore souvent propriétaires
  - pérennité du contrat avec fournisseurs de composants vs grand turnover développeurs internes
- lisibilité du code source
  - interne : découpage forcé en composants l'améliore
  - externe : API documentées facilite lisibilité du logiciel métier
- répétabilité du processus *(réutilisabilité code-source, savoir-faire, équipe...)*
  - capitalisation de l'apprentissage de l'utilisation de composants

COM / DCOM / ActiveX (d'après Peschanski&Meurisse)

- COM : ***Component Object Model***
- Définition d'un standard d'interopérabilité de **Composants binaires**
  - Indépendant du langage de programmation (i.e VB et C++ ?)
  - Modèle de composants extrêmement simple (voire vide...)
  - notion de composition de composants limité à la notion d'interface (*containment / aggregation*)
- But : fournir un modèle à base de composants le plus simple possible permettant l'adaptabilité, l'extensibilité, la transparence à la localisation (in process, local, remote) et des performances optimums...

Principes de COM (d'après Peschanski&Meurisse)

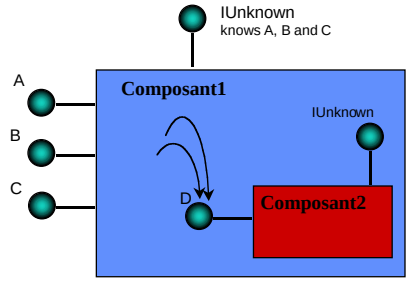
- Encapsulation "totale"
  - *Black-Box* : chaque composant est vu comme une boîte noire
  - L'interopérabilité entre composants ne se fait que via leurs *interfaces*
  - Possibilité de définir des *interfaces multiples* pour un même composant
  - QueryInterface : 'découvrir' les interfaces en cours d'exécution ***(réflexion !!)***
  - IUnkwnow : gestion du cycle de vie des composants (GC)



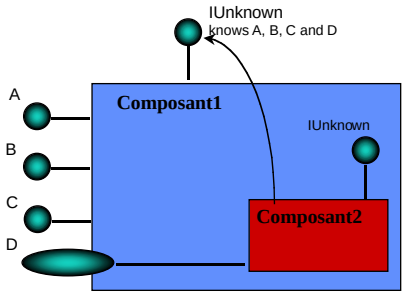
La composition dans COM (d'après Peschanski&Meurisse)

Principes de 'Réutilisabilité' [Microsoft97]

Par confinement / délégation



Par agrégation



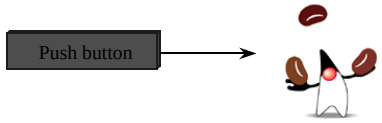
Cycle de vie des composants ('Versioning')...

JavaBeans (d'après Peschanski&Meurisse)

- **Motivations** : Composition graphique d'applications
- Définition :
  - Entité logicielle manipulable graphiquement
  - **"A Java Bean is a reusable software component that can be manipulated visually in a builder tool." [Sun Spec97]**
- "**Modèle**" inspiré des *Architectures logicielles*
- mais principalement orienté **implémentation**...

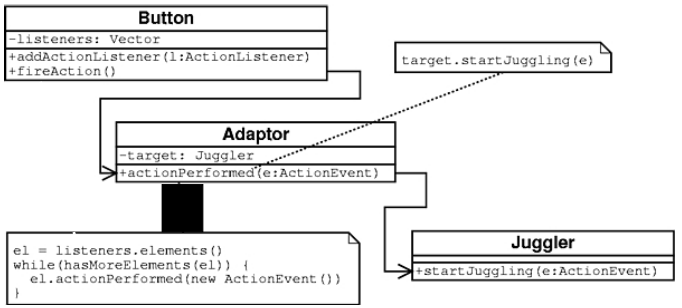
Exemple JavaBeans (d'après Peschanski&Meurisse)

Modèle



Inspiré d'un style d'A. L. : *Communication implicite* (*publish/subscribe*)

Implémentation



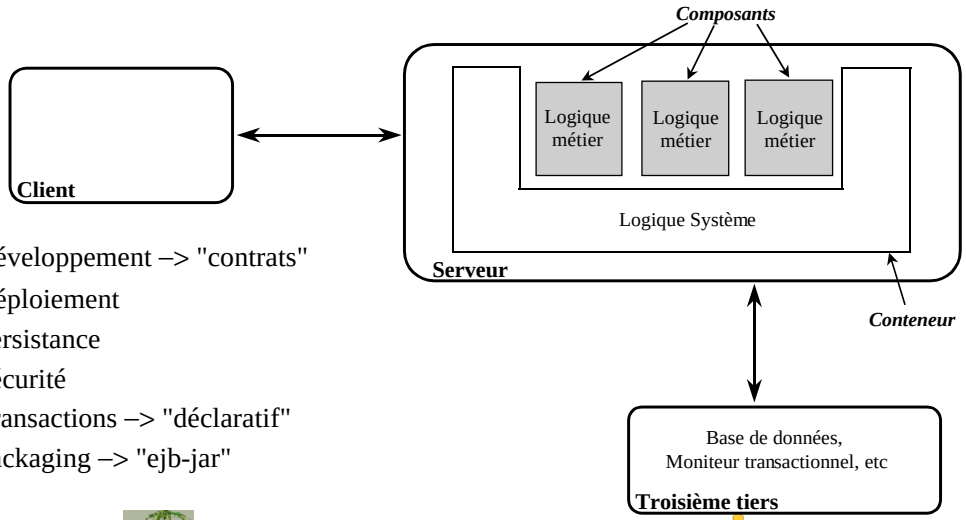
Réification des connexions par appel de méthodes

Propriétés JavaBeans (d'après Peschanski&Meurisse)

- Propriétés (méthodes *get* - *set*) - Editeurs de propriétés spécialisés (*Customizers*)
- Introspection granularité méthode/attribut
- Déploiement - Packaging (JAR)
- Support de Sérialisation Beans - Evénements
- etc.

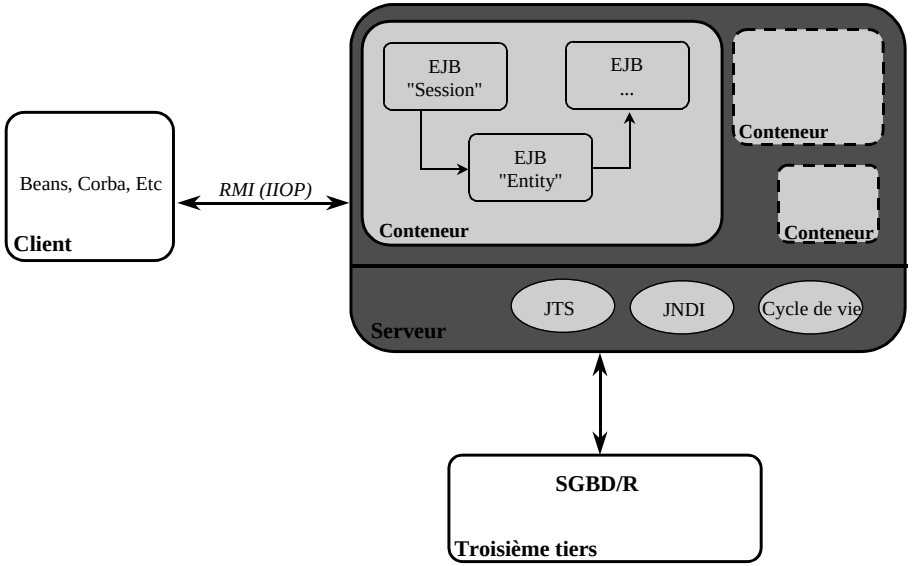
Enterprise JavaBeans (d'après Peschanski&Meurisse)

- But : **Simplifier** le développement d'architectures "3 tiers", côté serveur



- Développement –> "contrats"
- Déploiement
- Persistance
- Sécurité
- Transactions –> "déclaratif"
- Packaging –> "ejb-jar"

Exemple Entreprise JavaBeans (d'après Peschanski&Meurisse)



Composants Corba (d'après Peschanski&Meurisse)

Des objets (Corba 2.2) → **aux composants (Corba 3)**

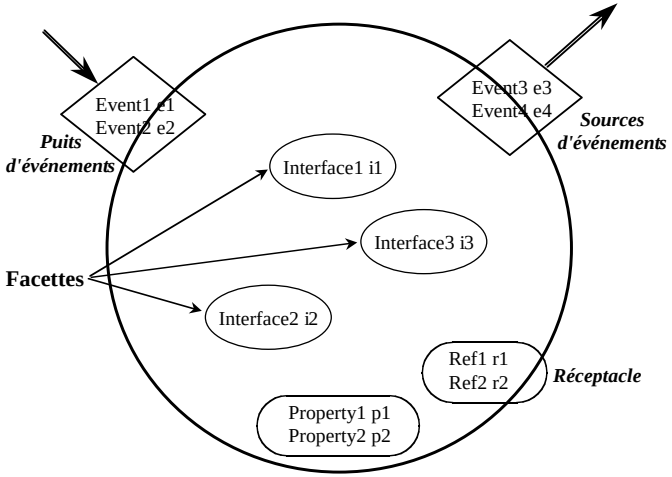
Serveurs archi client/Serveur — ***Serveurs architecture 3 tiers***

EJB + interopérabilité (y compris avec les EJBs)

- Modèle abstrait –> IDL étendu
- Modèle de programmation –> CIDL + interfaces standards (API composant - conteneur)
- modèle d'exécution –> Conteneur + structures d'accueil + interfaces
- Modèle de déploiement –> Langage OSD (DTD XML) + interface
- meta-modèle –> MOF

Composants Corba : modèle abstrait (d'après Peschanski&Meurisse)

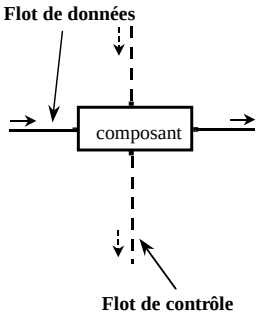
Modèle abstrait



component exemple

```
component {
  attribute Property1 p1;
  attribute Property2 p2;

  provides Interface1 i1;
  provides Interface2 i2;
  provides Interface3 i3;

  consumes Event1 e1;
  consumes Event2 e2;

  uses Ref1 r1;
  uses multiple Ref2 r2;

  emits Event3 e3;
  publishes Event4 e4;
} exemple;
```

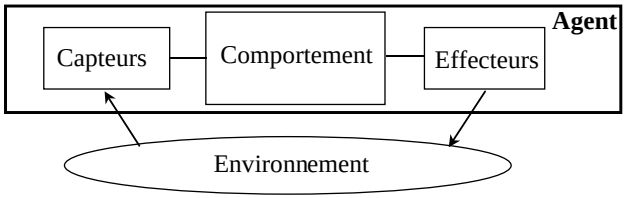

Composants Maleva [Lhuillier 98, Meurisse 99]



- **Séparation explicite des flots de contrôle et de données**
  - Permet une plus grande généricité via l'expression de différents contextes de contrôle pour des mêmes composants
- **2 types de bornes :**
  - ***Bornes de données***
    - Modification des ***variables d'instance*** du composant
  - ***Bornes de contrôle***
    - Un ***comportement encapsulé*** n'est enclenché que lors d'une activation via une borne de contrôle associée.

Exemples de Conception

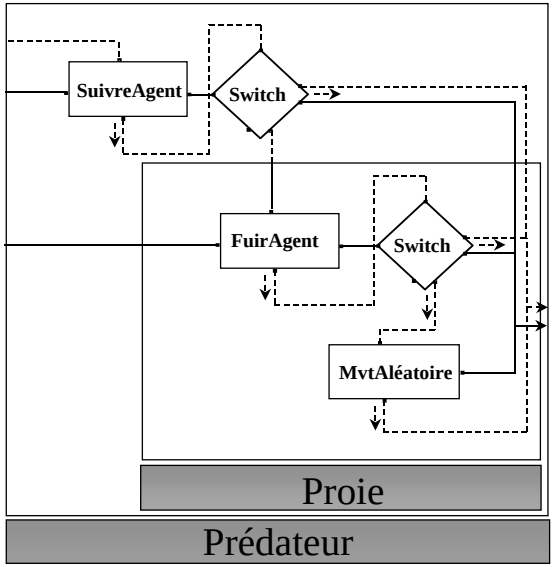
Agents situés dans un écosystème simulé



Exemple des Proies / Prédateurs

Les proies	Les prédateurs
•Fuient les prédateurs •Mouvement aléatoire	•Poursuivent les proies •Fuient les prédateurs •Mouvement aléatoire

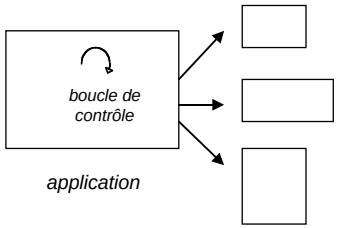
Exemples de Conception



Frameworks

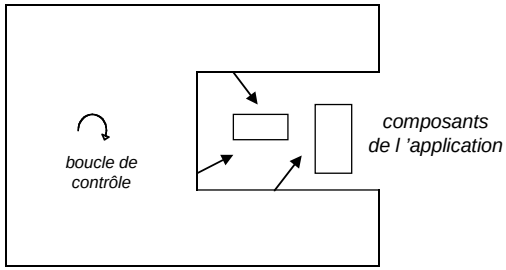
- Squelette d'application
- Ensemble de classes en collaboration
- Framework vs Boîte à outils (Toolkit)
  - inversion du contrôle
  - principe d'Hollywood

Ecrire le corps principal de l'application et appeler le code à réutiliser



bibliothèque de composants (ex : windows toolkit)

Réutiliser le corps principal et écrire le code applicatif qu'il appelle



framework

Frameworks (2)

- Un framework est une généralisation d'un ensemble d'applications
- Un framework est le résultat d'itérations
- Un framework est une unité de réutilisation
- Un framework représente la logique de collaboration d'un ensemble de composants : variables et internes/fixés

- « If it has not been tested, it does not work »

Corollaire :

- « Software that has not been reused is not reusable »
  [Ralph Johnson]

Exemples :

- Model View Controller (MVC) de Smalltalk
- Actalk

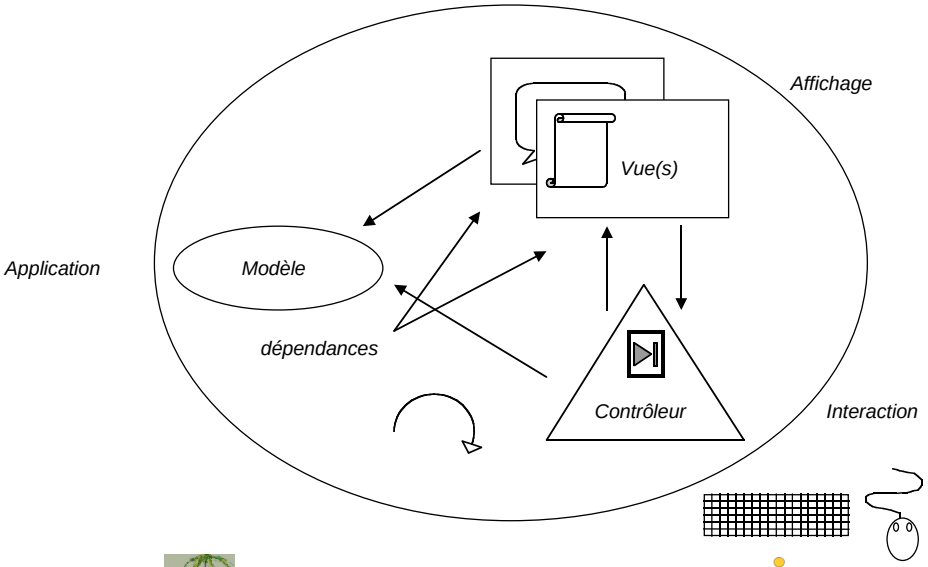
Architectures logicielles/Composants vs Frameworks

- Architectures logicielles et composants (et connecteurs)
  - Générique
  - Approches de conception
    - » descendante
      - décomposition
      - connexions
    - » ou ascendante
      - assemblage de composants existants
  - Les connexions et la coordination (« boucle de contrôle ») restent à définir, puisqu'elle est spécifique à l'application : difficile !

- Frameworks
  - Conception initiale du framework ascendante
  - Mais utilisation (spécialisation du framework) descendante
  - Les connexions et la coordination sont déjà définies (et testées) pour une classe d 'applications : plus facile !

Model View Controller (MVC)

Modèle d'interface homme-machine graphique de Smalltalk



Patrons de conception (Design Patterns)

- Idée : identifier les solutions récurrentes à des problèmes de conception
- « Patterns in solutions come from patterns in problems »
  [Ralph Johnson]

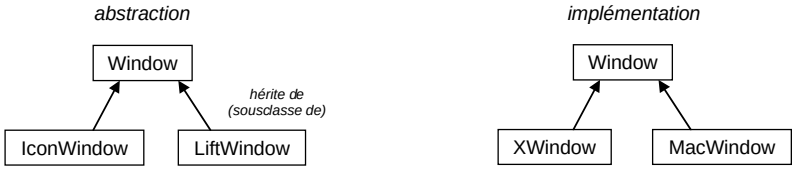
- Analogie :
  - principes d'architecture (bâtiments, cathédrales) [Christopher Alexander]
  - patrons/archétypes de romans (ex : héros tragique : Macbeth, Hamlet...)
  - cadences harmoniques : II-V-I, Anatole...

- Des architectes (C. Alexander) aux architectes logiciels
  - Design Patterns : Elements of Reusable O-O. Software
    [E. Gamma, R. Helm, R. Johnson, J. Vlissides (the « GoF »), Addison Wesley 1994]

- Les patterns ne font sens qu 'à ceux qui ont déjà rencontré le même problème (Effet « Ha ha ! »)
  - documentation vs génération

Ex : pattern Bridge

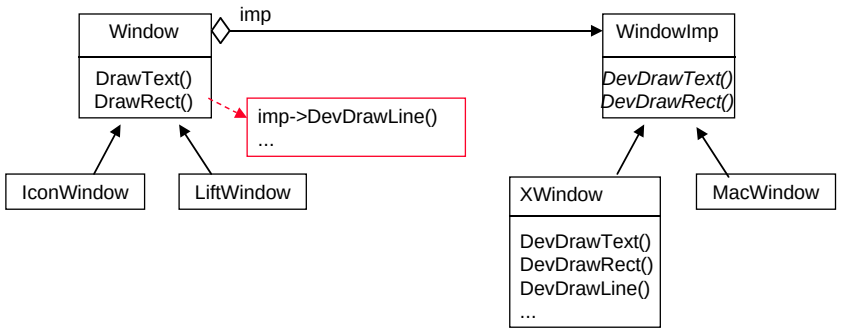
- Problème : une abstraction peut avoir différentes implémentations



- Solution naïve :
  - énumérer/nommer toutes les combinaisons
    - MacIconWindow, XIconWindow, etc.
  - problèmes :
    - » combinatoire
    - » le code du client dépend de l'implémentation

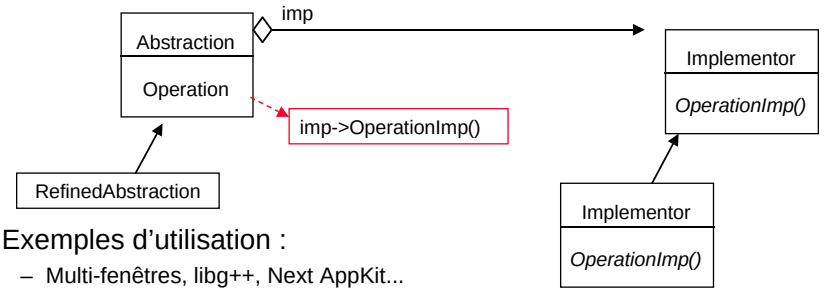
Ex : pattern Bridge (2)

- Solution :
  - séparer les 2 hiérarchies



Ex : pattern Bridge (3)

- Solution générale (le pattern Bridge)



- Exemples d'utilisation :
  - Multi-fenêtres, libg++, Next AppKit...
- Egalement :
  - Langages de patterns (Pattern Languages), ex : GoF book
  - Patterns d'analyse (Analysis Patterns)
  - Patterns seminar group (équipe de Ralph Johnson à UIUC)
  - Voir : `http://www.hillside.net/patterns/patterns.html`
  - Crise actuelle des patterns ?

Des Objets aux Acteurs

- approche intégrative
  - intégration des objets et des activités (tâches/threads)
  - intégration de l'envoi de message avec l'invocation distante
- la concurrence comme fondement
  - envoi de message asynchrone (sans attente, sans réponse)
  - la concurrence est le défaut. Dans le modèle de calcul de Gul Agha [Agha 86], la séquentialité n'est qu'une conséquence de la causalité (message arrivé après être parti)

A Generic Software Architecture/Framework: Actalk

- Objectives
  - help at analyzing and classifying various OOCP models and constructs
  - helps at evaluating them on actual programs (based on the Smalltalk-80 programming environment)
  - helps at customizing models and constructs by refinements of the library of models
- Principles
  - fully experimental approach
  - a generic software architecture, a *framework*, modular (component-based) and expressive (parameterized), designed/intended to be specialized
  - based on a standard object-oriented programming environment: Smalltalk
    - » high-level
    - » flexible
    - » environment tools
- `http://www-poleia.lip6.fr/~briot/actalk/actalk.html`

Actalk: Experiments and Applications

- Libraries
  - language models: Actors behavior replacement, POOL body concept...
  - communication models: ABCL/1 three types (synchronous, asynchronous, future) and two modes (normal, express) of message transmission...
  - synchronization schemes: enabled sets, guards, synchronization counters, generic invocations, explicit acceptance, method suspension...
- Pedagogy
  - Teaching and projects at University of Nantes
- Environment tools
  - customizing standard Smalltalk progr. environment tools for concurrency
- Experiments
  - ReActalk (S. Giroux, U. of Montréal), exception handling...
- Developments
  - various multi-agent platforms (LAFORIA, LIRMM...), software engineering process models, natural language processing (Univ. Freiburg)

Component Classes

- Three main component classes:
  - `ActiveObject`
    - » behavior (functionality / programmer intention)
  - `Activity`
    - » selection, scheduling of invocations
      - implicit acceptance of messages (reactive), explicit acceptance...
    - » synchronization
      - guards, synchronization counters, abstract states...
  - `Address`
    - » communication
      - synchronous, asynchronous, eager-reply...

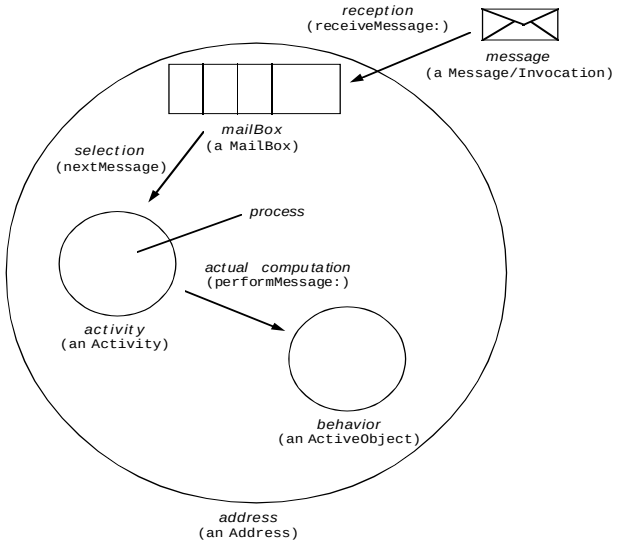
Actalk Architecture

- Three main generic components of an active object:
  - Behavior : programming constructs and user program
  - Activity : selection and activation of messages
  - Address : message passing
- Two other components:
  - Mailbox : buffering, ordering and access to messages
  - Invocation/Message : attaching further information (time stamp, priority...)
- Generic parameter methods for each main component.
  - e.g., method nextMessage for component Activity
- Generic event methods:
  - associated to : message reception, acceptance and completion

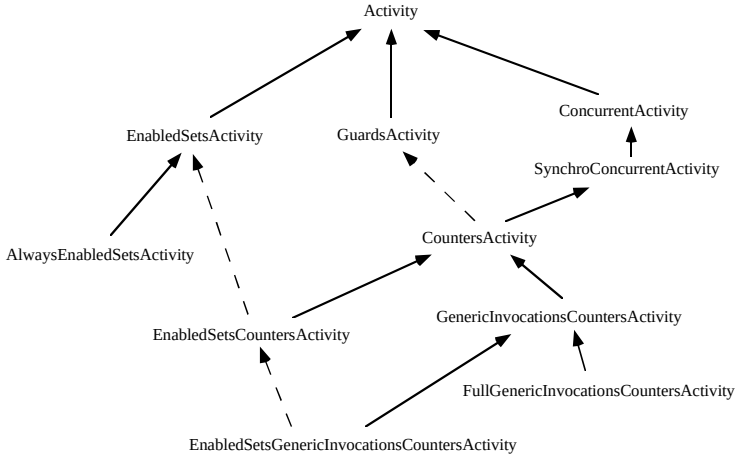
Component Classes (2)

- Generic parameter methods for each main component
  - e.g., method `nextMessage` for component `Activity`
- Generic event methods:
  - associated to : message reception, acceptance and completion
- Two complementary component classes :
  - `MailBox`
    - » message buffering/ordering
      - indexing, priorities...
  - `Invocation/Message`
    - » invocation management (attaching further info)
      - time stamps, priorities...

Relations between Components

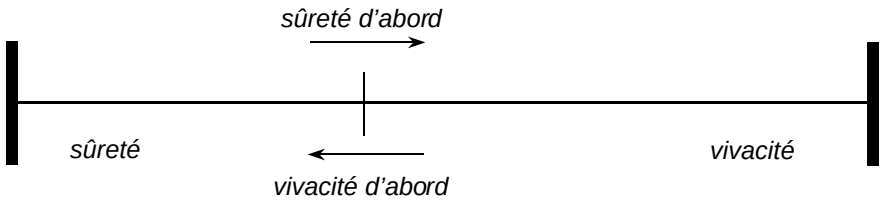


(Partial) Hierarchy of Synchronization Schemes



Programmes concurrents (Java [Lea 97])

- Propriétés à garantir :
  - sûreté (safety)
    - » pas d'états incohérents
  - vivacité (liveness)
    - » pas de terminaison prématurée, pas de deadlock, pas de livelock, équité, pas de famine
- Problème :
  - garantir la sûreté par des synchronisations peut entraîner des problèmes de vivacité
  - et vice versa

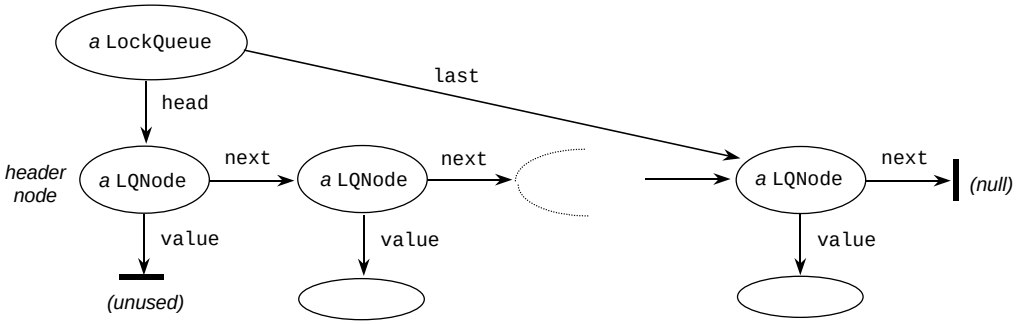


synchronisation maximale *aucune synchronisation*

Programmes concurrents (Java) (2)

- Techniques pour garantir la sûreté
  - objets immutables (sans état ou sans changement d'état)
    - » méthodes sans état
  - synchronisation (garantir de manière dynamique un accès exclusif)
  - encapsulation/privé (garantir de manière statique un accès exclusif - en évitant des variables partagées)
    - » synchronisation existe mais dans un objet contenant
    - » ressource exclusive : jeton, capacité... verrou !
- Techniques pour garantir la vivacité
  - analyse par variable d'instance (synchroniser ou pas les accès)
  - accès désynchronisé aux variables constamment mises à jour
    - » ex : `température`
  - partitionner la synchronisation (grain plus fin)
    - » partitionner une classe
    - » partitionner un verrou
      - ex : classe `TwoLockQueue`
  - introduire une asymétrie
    - » ex : classe `Cell`, méthode `swapContents`

class LockQueue




```
final class LQNode { // local node class for queue
  Object value;
  LQNode next;
  LQNode(Object x, LQNode n) { value = x; next = n; }
}
```


class LockQueue (2)

```
public class LockQueue {
  private LQNode head_;    // pointer to dummy header node
  private LQNode last_;    // pointer to last node

  public LockQueue() {
    head_ = last_ = new LQNode(null, null);
  }

  public synchronized void put(Object x) {
    LQNode node = new LQNode(x, null);
      // insert at end of list
      last_.next = node;
      last_ = node;
    } }

 public synchronized Object take() { // returns null if empty
    Object x = null; // return value
    LQNode first = head_.next; // first real node is after head
    if (first != null) {
      x = first.value;
      head_ = first; // old first becomes new head
    }
    return x;
  }
}
```


class TwoLockQueue

```
public class TwoLockQueue {
  private LQNode head_;    // pointer to dummy header node
  private LQNode last_;    // pointer to last node
  private Object lastLock_; // protect access to last

  public TwoLockQueue() {
    head_ = last_ = new LQNode(null, null);
    lastLock_ = new Object();
  }

  public void put(Object x) {
    LQNode node = new LQNode(x, null);
    synchronized (lastLock_) {  // insert at end of list
      last_.next = node;
      last_ = node;
    }
}

public synchronized Object take() { // returns null if empty
   // same as in class LockQueue
}
```


class Cell

```
public class Cell { // local node class for queue
  private int value_;

  public synchronized int getValue();
    return value_;
  }

  public synchronized void setValue(int v) {
    value_ = v;
  }

  public synchronized void swapContents(Cell other) {
    int otherValue = other.getValue();
    other.setValue(getValue());
    setValue(otherValue);
  }
}
```

class Cell (2)

```
public synchronized void swapContents(Cell other) {
  if (other == this) return; // alias check
  Cell first = this;         // ressource ordering
  Cell second = other;
  if (this.hashCode() > other.hashCode()) {
    first = other;
    second = this;
  }
  synchronized(first) {
    synchronized(second) {
      int otherValue = other.getValue();
      other.setValue(getValue());
      setValue(otherValue);
    }
  }
}
```

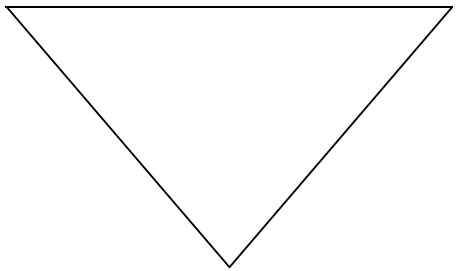

Des Objets (ou Acteurs) aux Agents

- au niveau de l'entité
  - agent non purement procédural
    - » connaissances
      - ex : états mentaux, plans, règles d'inférence des agents cognitifs
  - pro-activité
    - » pas uniquement purement réactif
- au niveau d'un ensemble d'agents
  - différents modes de communication
    - » via l'environnement, ex : colonies de fourmis
    - » messages typés, ex : KQML (inform, request, reply...)
  - coordination
    - » interactions arbitrairement complexes, pas juste client/serveur
- au niveau de la conception (vs implantation)
  - organisation
    - » structuration forte, quoique souvent dynamisque, conditionnant les interactions
  - une conception sous forme d'agents peut ensuite être réalisée sous forme d'objets ou d'acteurs, le niveau agent n'apparaissant plus explicitement dans l'implantation

OASIS = Intégration

programmation
génie logiciel
données,
procédures,
algorithmes

représentation et traitement
des données et connaissances
indexation,
connaissances du domaine,
raisonnement,
heuristiques

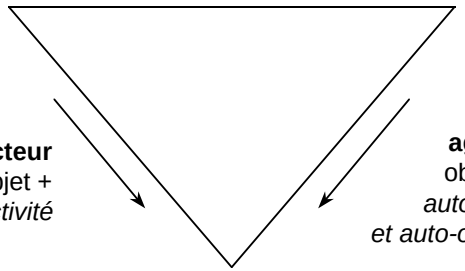


parallélisme et distribution
concurrence, communication,
coordination, répartition, persistance,
mobilité

OASIS = Intégration (2)

objet
module,
encapsulation

objet
module de persistance,
module de connaissances



acteur
objet +
activité

agent
objet +
autonomie
et auto-organisation

objet
unité de concurrence,
de répartition et de mobilité

De la Simulation Objet à la Simulation Multi-Agent

- au niveau de l'entité
  - comportement non nécessairement purement déterministe
    - » mémoire, connaissances, désirs, interactions
- au niveau d'un ensemble d'agents
  - différents modes de communication
    - » via l'environnement, ex : colonies de fourmis
  - coordination
    - » interactions arbitrairement complexes
  - simulation multi-niveau
    - » un ensemble d'agent peut être aussi considéré (émerger) comme un agent avec son comportement propre
      - ex: émergence d'un banc de poisson, d'une rivière
- au niveau de la conception (vs implantation)