Programmation, Modèles de Calcul et Architectures Logicielles
à base d'Objets, Composants, Agents

Module RAAR
Réalisation Assistée d'Applications Réparties
1/3

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

Supports de cours

Copie transparents
Et articles bibliographie en :

http://www-poleia.lip6.fr/~briot/cours/

raar04-05.1.pdf
raar04-05.2.pdf
raar04-05.3.pdf

Plan général

- I - Objets pour la Programmation Concurrente et Répartie
  - Enjeux, problèmes, approches
  - Approche applicative
  - Approche intégrée
  - Approche réflexive
- II - Composants
  - Des objets aux composants
  - Architectures Logicielles
  - Frameworks et design patterns
- III - Agents
  - Motivations
  - Des objets aux agents
  - Différents types d'agents
  - Principes et techniques
- IV - Modèles de calcul et d'implémentation
  - Concurrents
  - Répartis
  - Agents

Concevoir et réaliser des applications… concurrentes et réparties…

Modèles

Vérification

Langages de Programmation — Programmes — Bibliothèques

Modèles de calcul

Supports d'exécution

Intergiciels

Système d'exploitation

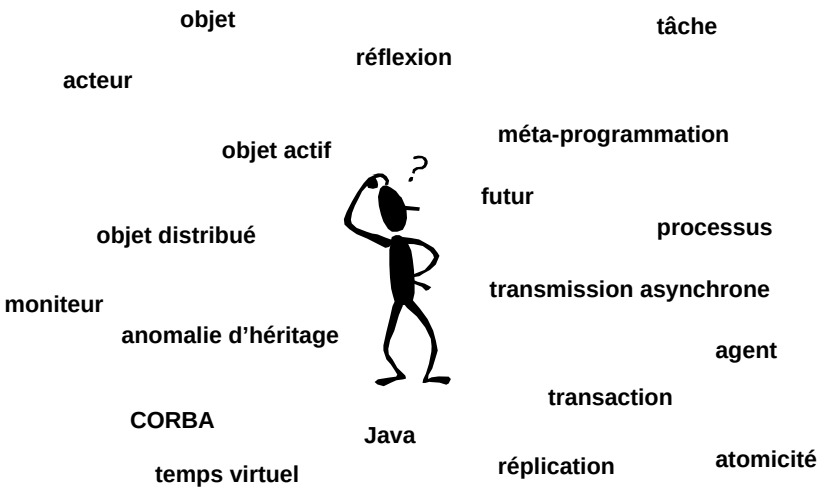
Concurrence

Répartition

Matériel (Processeurs, Mémoire, Réseau, Devices…)

I - Objets pour la Programmation Concurrente et Répartie

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,
    - » Technique et Science Informatiques (TSI),15(6):765-800, Hermès, France, juin 1996.
    - » dans «Langages et modèles à objets», édité par Amedeo Napoli et Jérôme Euzenat, Collection Didactique, INRIA, 1998.
  - «Concurrency and distribution in object-oriented programming», Jean-Pierre Briot, Rachid Guerraoui et Klaus-Peter Löhr, ACM Computing Surveys, 30(3):291-329, septembre 1998.

(sous)-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

Concurrence potentielle

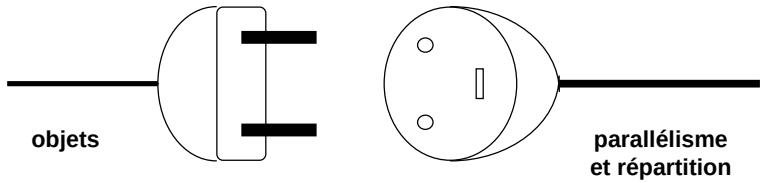
- *Simula-67* [Birtswistle et al.'73]
  - *body* d'une classe : corps de programme exécuté lors de la création d'une instance
  - *coroutines* : suspension (`detach`) et relance (`resume`)
- Objets <-> Processus [Meyer, CACM'9/93]
  - variables
  - données persistantes
  - encapsulation
  - moyens de 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

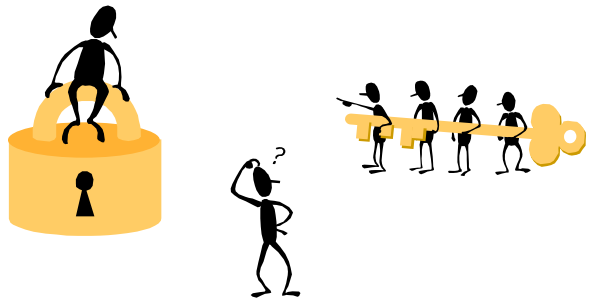
  ont développé différents types d'abstractions :
  - synchronisation
  - transactions
  - communication de groupes
  - ...

  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

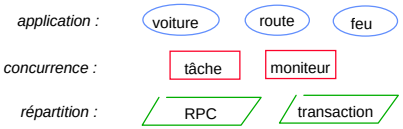
- Langage de programmation par objets minimal
- Riches bibliothèques de classes représentant :
  - constructions du langage (ex : structures de contrôle, `ifTrue:ifFalse:`)
  - ressources (messages, multi-tâche, compilateur...)
  - outils de l'environnement (ex : browser, debugger...)
- Concurrence
  - 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	chaque objet un verrou (en fait un moniteur) est associé
objet réparti	NON	OUI avec Java RMI

Objet actif

- objet = activité
  - une activité : sériel
  - 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

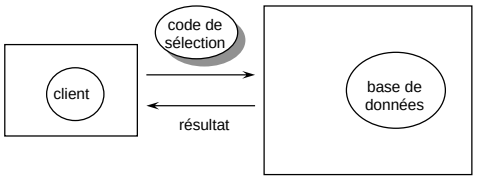
- Origines : systèmes d'exploitation, parallélisme, bases de données
- Intégration relativement aisée dans un modèle objet
  - formalismes centralisés, associés au niveau des classes
    - » *path expressions* (specif. abstraite des entrelacements possibles entre invocations)
      - ex : *Procol* [Lafra'91]
    - » *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

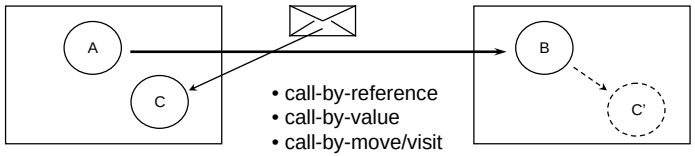
- objet : unité indépendante d'exécution
  - données, traitements, et même ressources (si objet actif)
  - transmission de messages conduit à la transparence de la localisation
  - autonomie et relative atomicité de l'objet facilite migration et duplication
- association de l'invocation distante à la 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])

Emerald

- Code mobile
  - rapprocher (code) traitement des données
  - ex : SQL



- Objet mobile
  - PostScript (code + données constantes)
  - Emerald [Black et al. IEEE TSE 87]



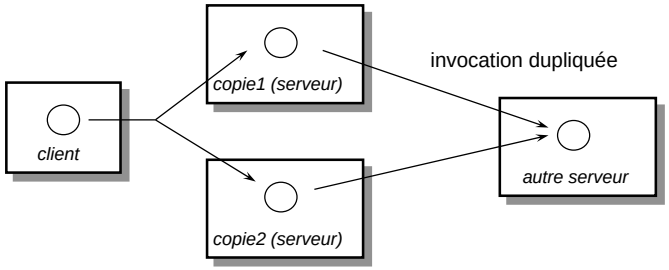
Limite 1 : Spécialisation de la synchronisation

- 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
      - solution Java : méthodes qualifiées `synchronized` en exclusion mutuelle entre elles
    - » 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

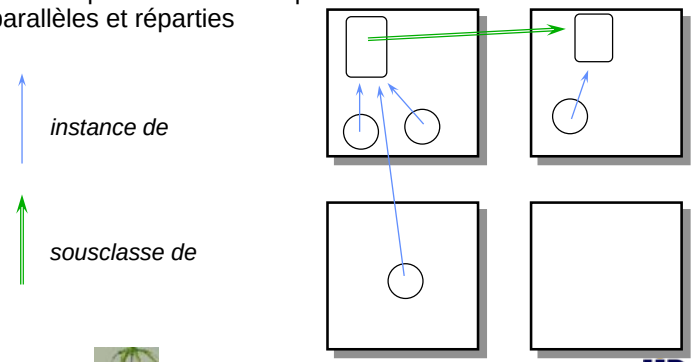


  - Solution possible : *pré-filtrage* par un coordinateur arbitrairement désigné (un des serveurs dupliqué) (en fait solution un peu plus complexe pour résistance aux pannes du coordinateur -> *post-filtrage*) [Mazouni et al. TOOLS-Europe'95]

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

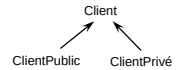


factorisation vs répartition (2)

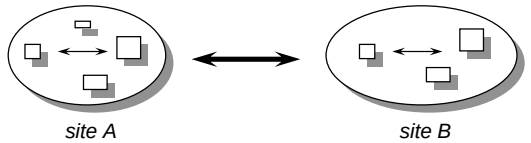
- Solution1 : dupliquer l'ensemble des classes
  - Cela suppose qu'elles sont immutables
    - » constantes de classe OK, mais pas de variables de classe
    - » problème de mise à l'échelle
- Solution2 : partitionner statiquement les classes en modules [Gransart'95]
  - Mais complexifie les possibilités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

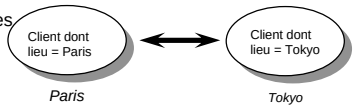
- » refactorisation «roll up» des attributs/méthodes de sous-classes dans une super-classe pour éviter que l'héritage ne «traverse» PLUSIEURS sites

Client (Privé ou/et Public)

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

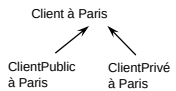
Client dont lieu = Paris
Client dont lieu = Tokyo

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

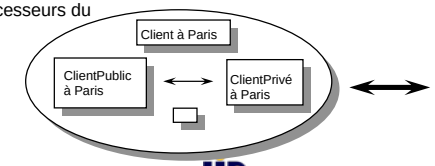


– phase 2 : répartition à l'intérieur d'un site donné

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

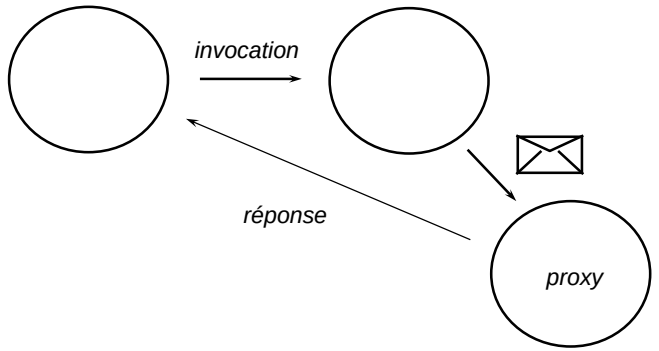


- » optimisation de l'allocation des fragments sur les processeurs du site
  - décision multi-critères :
    - adéquation du processeur
    - concurrence
    - communication inter-processeurs
    - réplication des instances



Délégation

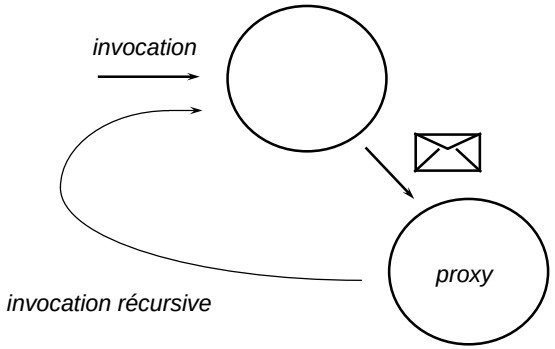
- Le mécanisme de délégation [Lieberman OOCP'87] offre une alternative a priori séduisante à l'héritage
  - Repose uniquement sur la transmission de messages, donc indépendant d'une hypothèse de mémoire centralisée



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

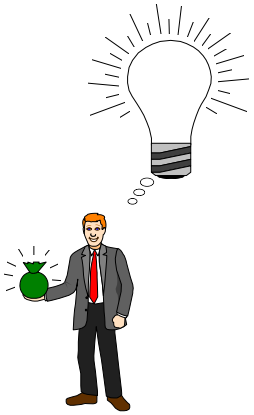
Méta-Bilan

- 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*

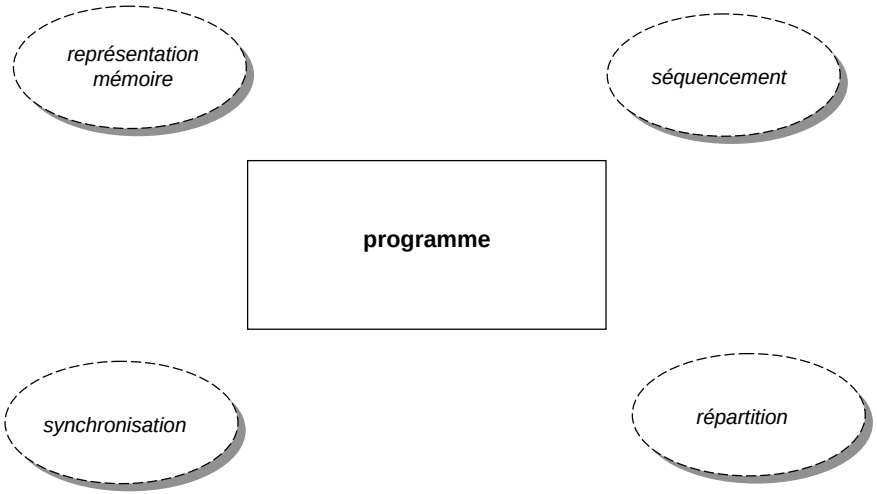


Réflexion (2)

- 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

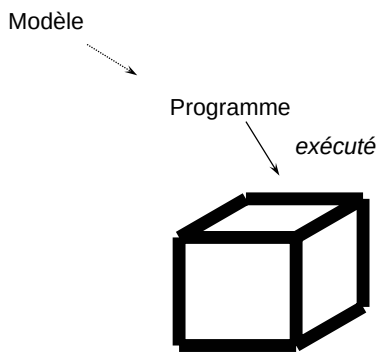
Contexte d'exécution



(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



Boîte noire

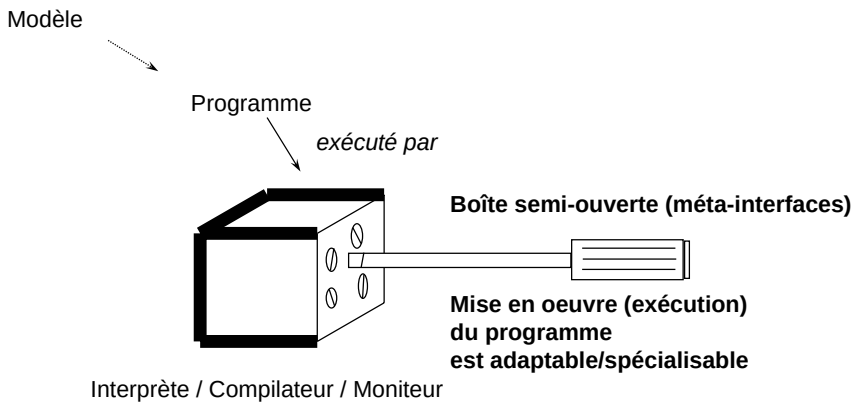
Mise en oeuvre (exécution) du programme est non modifiable

Interprète / Compilateur / Moniteur

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

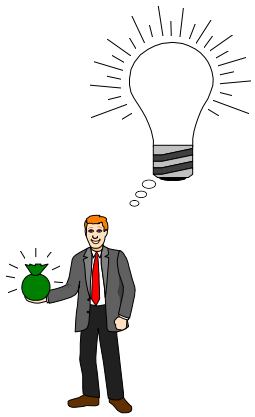
Réflexion (3)



Open Implementation [Kiczales 94]

Réflexion

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

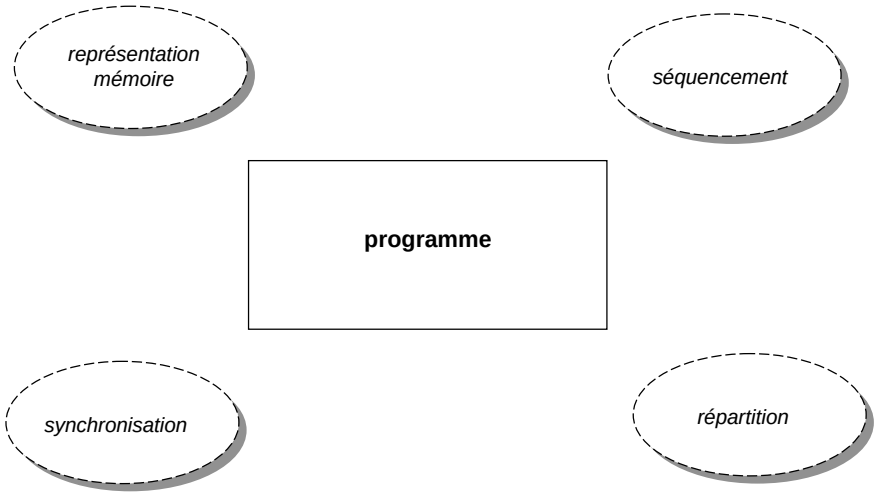


Réflexion (2)

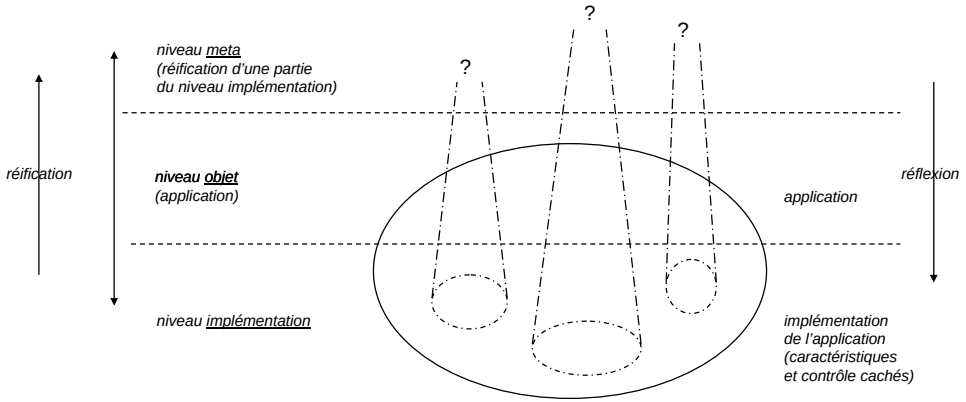
- Diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes (*réifiées*) sous la forme de *méta-programmes*.
  - Habituellement elles sont invisibles et immuables (interprète, compilateur, moniteur d'exécution...)
- La spécialisation de ces 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

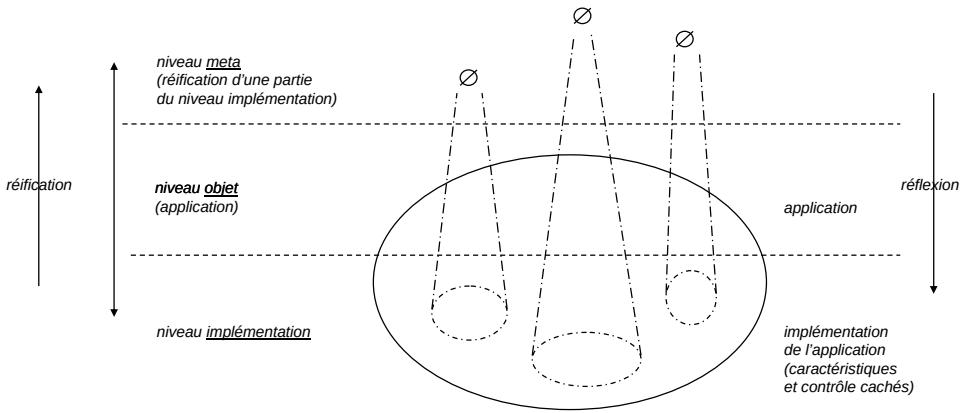


Réification/réflexion



Réification/réflexion

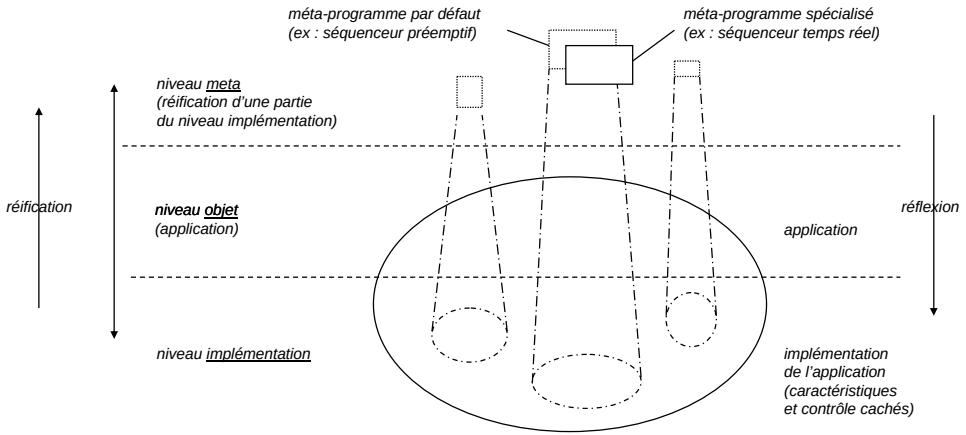
- Réification numérique (potentiomètres)
  - Ex : Options de compilation d'un compilateur
    - Efficacité vs taille du code généré



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]
  - introspection
    - » représentation d'entités du langage Java (ex : entiers, classes) sous forme d'objets Java
- 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

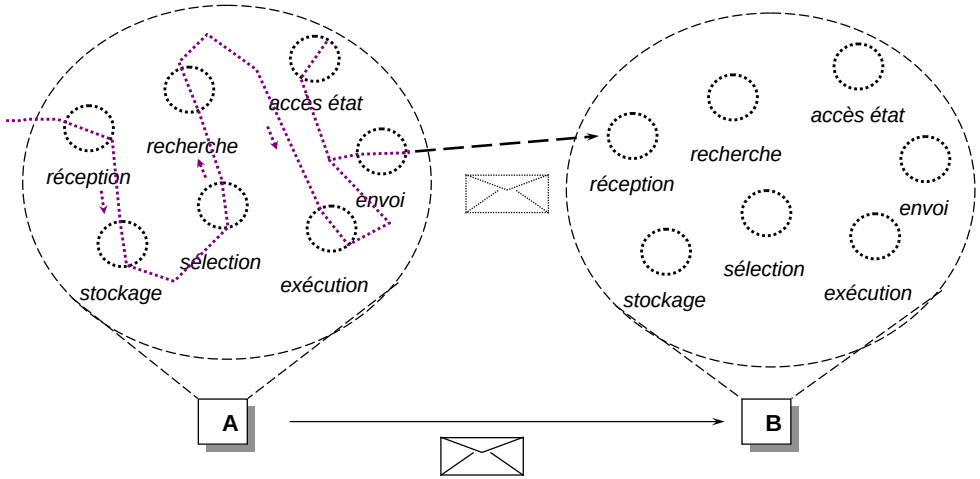
Approche réflexive

- 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

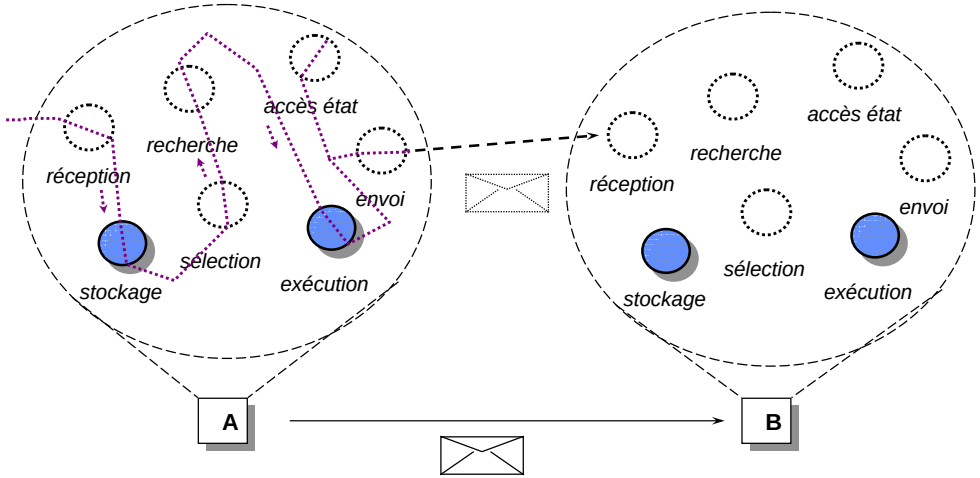
CodA



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)

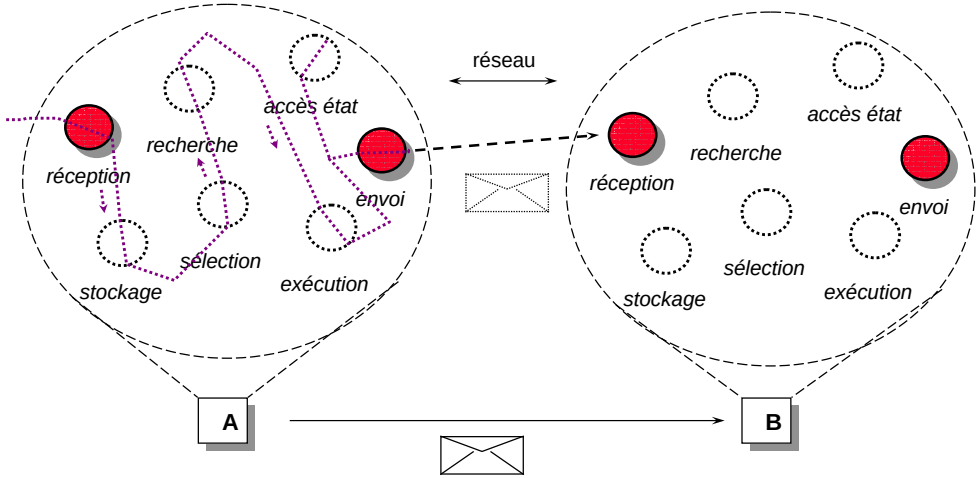


Ex : Exécution répartie

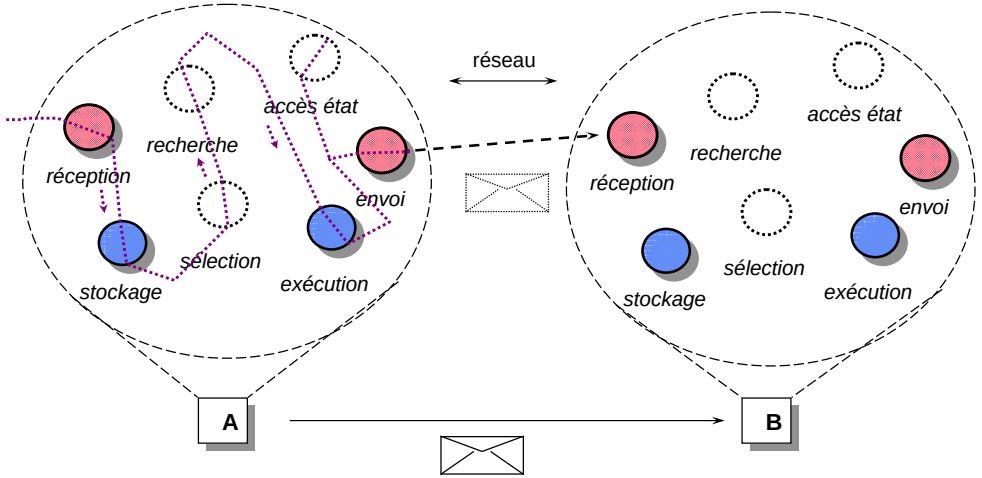
- **envoi de message**
  - » encodage des messages, envoi via le réseau
- **réception de messages**
  - » réception via le réseau, décodage des messages
- stockage des messages reçus
- sélection du premier message à traiter
- recherche de méthode correspondant au message
- exécution de la méthode
- accès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

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

Exécution répartie (2)

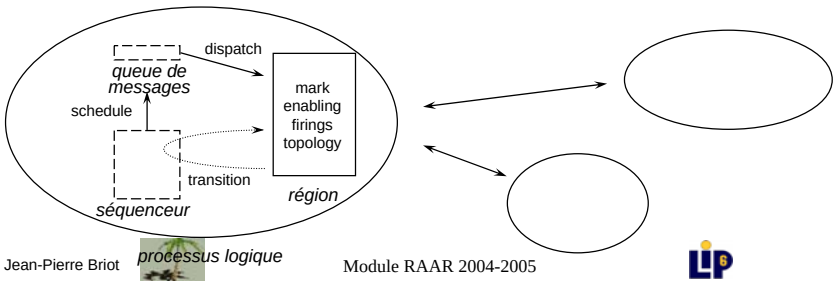


Exécution concurrente et répartie (composition)



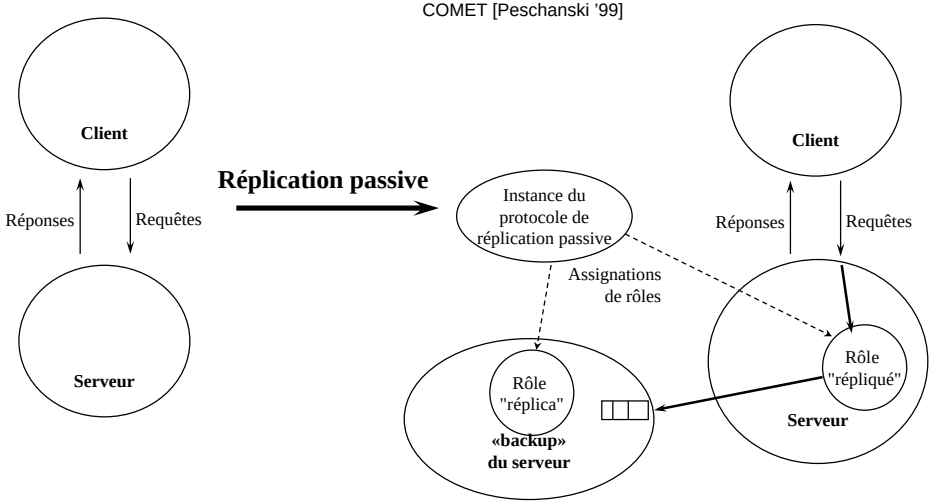
Temps réel

- 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]

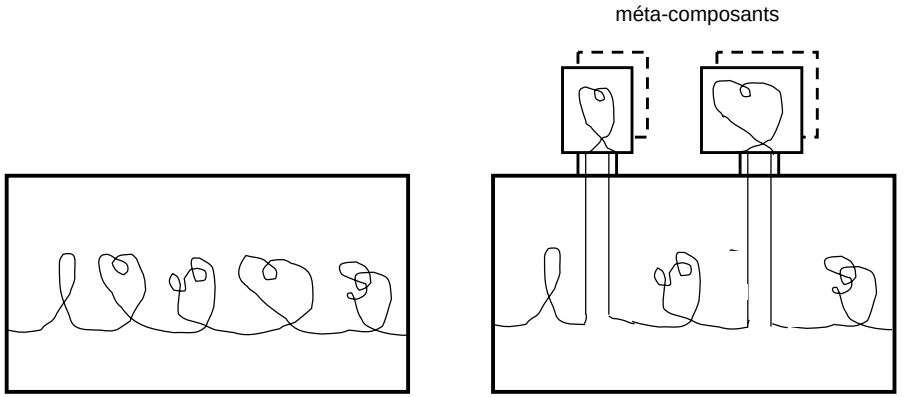


Systèmes commerciaux

- *Muse (ex-Aperios)* [Yokote OOPSLA'92]
  - spécialisation dynamique de la politique de séquencement (ex : passer au temps réel)
  - application au «video on demand» et aux robots-chiens Aibo (Sony)

- 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)

Moniteur de transaction rendu réflexif/ouvert



Exemple : CORBA

- 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

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]
    - » Javassist [Chiba, ECOOP'2000] - liaison au chargement des bytecodes
  - transformation de programmes - évaluation partielle
    - » [Masuhara et al., OOPSLA'95] [Consel et al. 2000]
- Ne dispense pas du travail nécessaire à l'identification des bonnes abstractions

Concevoir et réaliser des applications… concurrentes et réparties…

Modèles

Vérification

Langages de Programmation — Programmes — Bibliothèques

Modèles de calcul

Supports d'exécution

Intergiciels

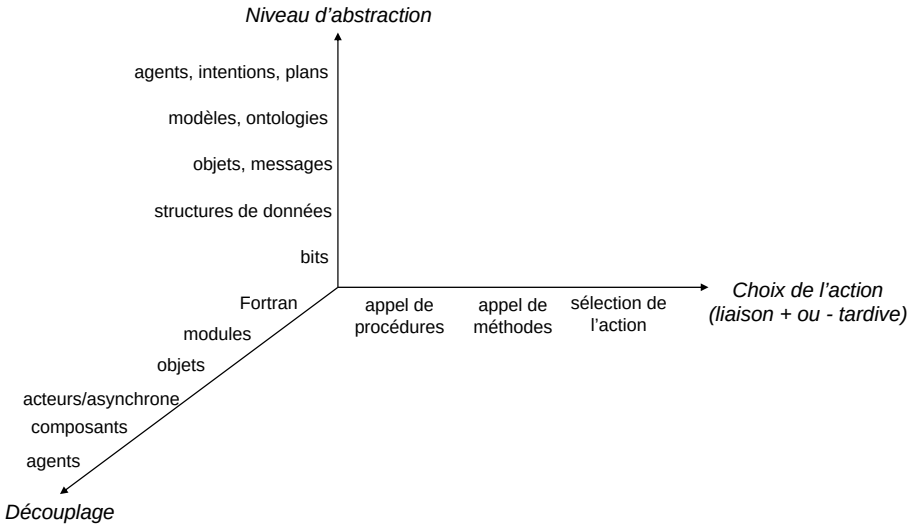
Système d'exploitation

Concurrence

Répartition

Matériel (Processeurs, Mémoire, Réseau, Devices…)

Evolution de la programmation



Evolution (2) - une autre grille [Odell, 99]

	Programmation monolithique	*Programmation modulaire*	*Programmation par objets*	*Programmation par agents*
Comportement	non modulaire	modulaire	modulaire	modulaire
Etat	externe	externe	interne	interne
Invocation (et choix)	externe	externe (appel)	externe (message)	Interne (règles, buts)