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

Module RAAR
Réalisation Assistée d'Applications Réparties
2/3

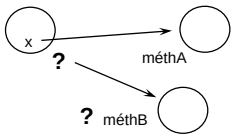
Jean-Pierre Briot

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

II - Composants

(Limites) des objets...

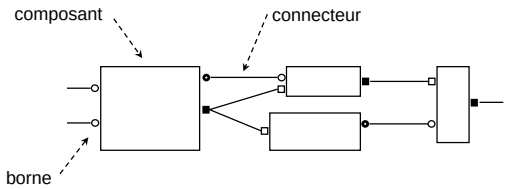
- granularité encore trop fine-moyenne
  - pas trop bien adapté à la programmation à grande échelle
- pas encore assez modulaire
  - références directes entre objets
  - donc connexion non reconfigurable sans changer l'intérieur de l'objet
    - » objet appelé
    - » nom de la méthode appelée



... aux composants

Idées :

- composants
  - plus «gros»
  - plus autonomes et encapsulés
  - symétrie retrouvée : interfaces d'entréeS mais aussi de sorties
  - plusieurs interfaces de sortie, et d'entrées : notions de "bornes"
- réification des relations/connexions entre composants
  - références hors des objets -> couplage externe (mais reste explicite)
  - notion de connecteur

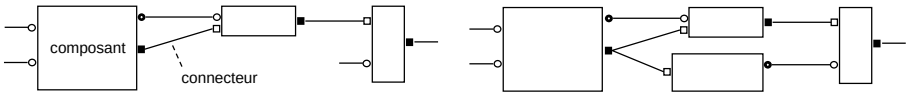


Architectures logicielles

- 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&filters...



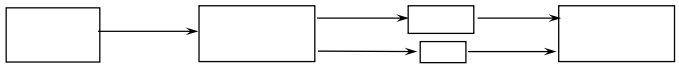
Architectures logicielles [Shaw et Garlan 96]

- différents types d'architectures (styles architecturaux)
  - pipes & 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

Ex1 : Pipes & filters



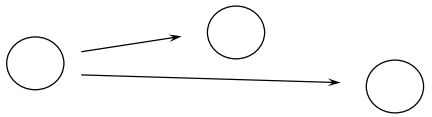
- Composants : filters
- Connecteurs : pipes

- 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



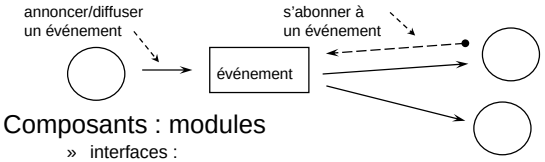
- Composants : objets
- Connecteurs : transmission de messages

- Ex : Java

- +
  - » encapsulation
  - » décomposition

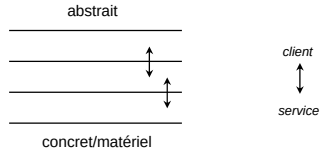
- -
  - » références directes
    - nécessité de recoder les références si reconfiguration

Ex3 : Diffusion d'événements (publish/subscribe)



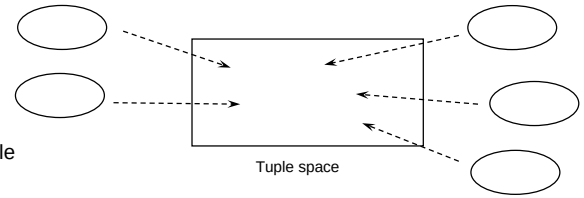
- Composants : modules
  - » interfaces :
    - procédures
    - événements
- Connecteurs : diffusion d'événements
- Ex : interfaces homme machine, bases de données (contraintes d'intégrité), Java Beans
- +
  - » réutilisation
  - » évolution
- -
  - » contrôle externe aux composants
    - difficile 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

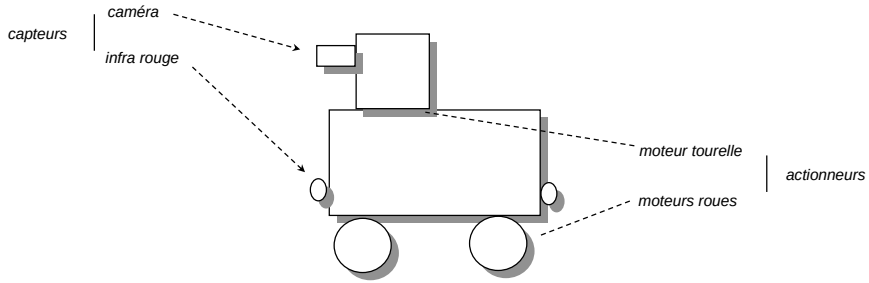
Ex4 : Repositories



- 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

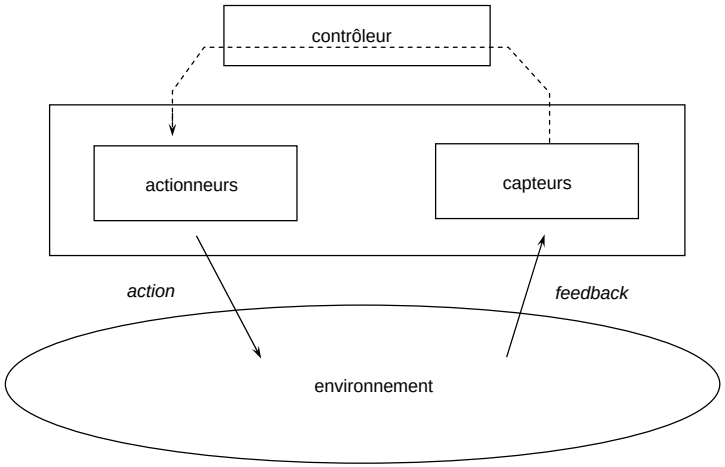
Comparaison de styles architecturaux

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

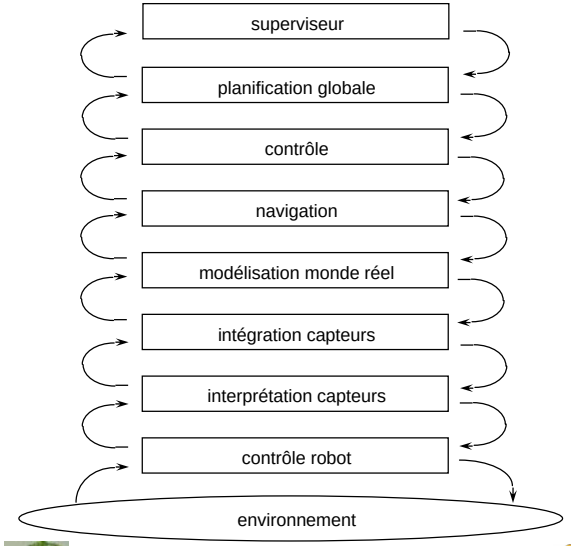


- Propriétés/caractéristiques recherchées :
  - comportement à la fois délibératif et réactif
  - perception incertaine de l'environnement
  - robustesse (résistance aux pannes et aux dangers)
  - flexibilité de conception (boucle conception/évaluation)

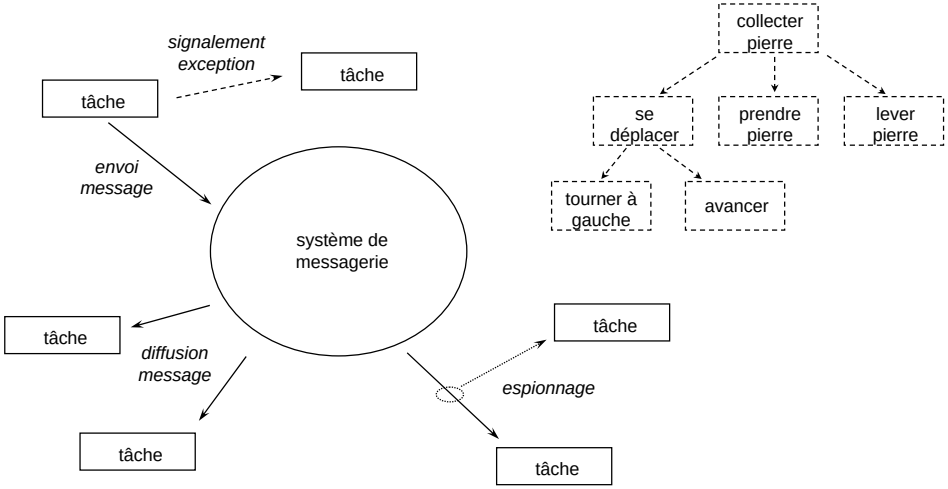
Solution 1 - boucle de contrôle



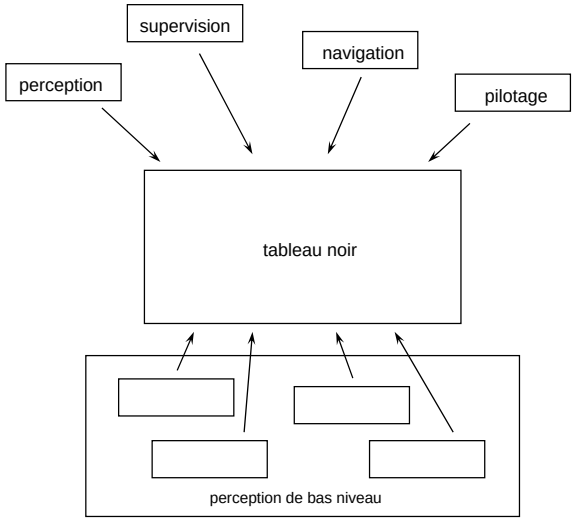
Solution 2 - couches



Solution 3 - (tâches et) événements



Solution 4 - tableau noir



Comparaison

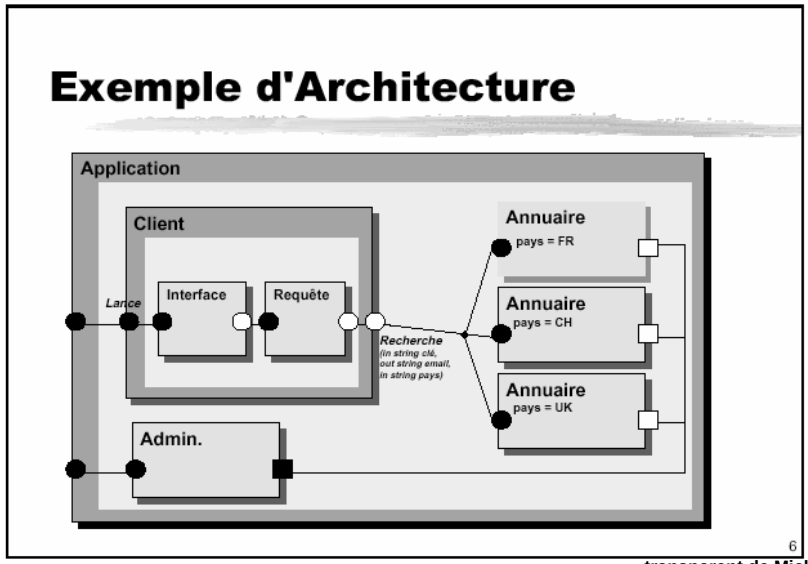
	Boucle de contr™le	couches	ˇvˇnements	tableau noir
coordination des t%hes	+ -	-	+ +	+
incertain	-	+ -	+ -	+
robustesse	+ -	+ -	+ +	+
s¸ retˇ	+ -	+ -	+ +	+
performance	+ -	+ -	+ +	+
flexibilitˇ	+ -	-	+	+

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/



transparent de Michel Riveill

UniCon

```
COMPONENT Annuaire
   INTERFACE IS
      TYPE Process
      PLAYER Lookup IS RPCDef
      SIGNATURE ("char *", "char**")
        End Lookup
   End INTERFACE

   IMPLEMENTATION IS
      VARIANT annuaire IS "annuaire.c"
      IMPLTYPE (Source)
   End IMPLEMENTATION
END Annuaire
```

```
COMPONENT Client
   INTERFACE IS
      TYPE Process
      PLAYER Lookup_Annuaire IS RPCCall
      SIGNATURE ("char *", "char **")
        End Lookup_Annuaire
   End INTERFACE

   IMPLEMENTATION IS
        VARIANT client IS "client.c"
        IMPLTYPE(source)
   END IMPLEMENTATION
End Client
```

57

UniCon

```
COMPONENT Application
   INTERFACE IS General
   // pas de typage strict utilisation du type
   générique
   END INTERFACE
   IMPLEMENTATION IS

// Définition des interfaces utilisées
    USE client1 INTERFACE Client
        PROCESSOR ("toua.inrialpes.fr")
        ENTRYPOINT (client1)
    END client1
    USE annuaire1  INTERFACE Annuaire
        PROCESSOR("dyade.inrialpes.fr)
    END annuaire1
```

```
// Définition des interactions
    ESTABLISH Remote-proc-call WITH
        client1.Lookup_Annuaire AS calle
         annuaire1.Lookup AS definer
    END Remote-proc-call
   END IMPLEMENTATION
END Application
```

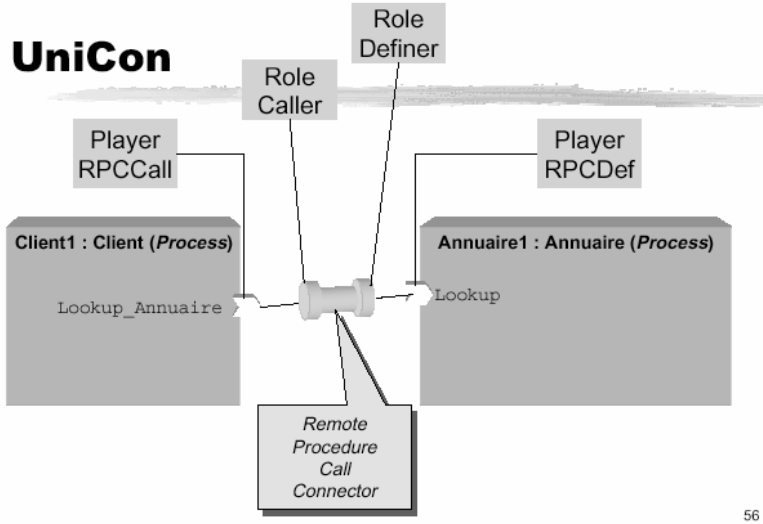
58

Unicon

```
CONNECTOR Remote-proc-call
   PROTOCOL IS
       TYPE RemoteProcCall
       ROLE definer IS definer
       ROLE caller IS caller
   END PROTOCOL
   IMPLEMENTATION IS
       BUILTIN
   END IMPLEMENTATION
END Remote-proc-call
```

59

UniCon



56

Composants

- Un composant est du code exécutable et son mode d'emploi
  - module logiciel autonome (et persistant)
  - exporte interfaces (d'entrée et de sortie)
  - 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)

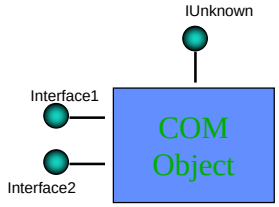
- performance du résultat final
  - performance en général inversement proportionnelle à généricité
  - mais capitalisation de l'optimisation
- facilité de déploiement *(portabilité sur différentes plates-formes)*
  - capitalisation des portages
  - utilisation quasi-générale pour les IHM
- indépendance 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)

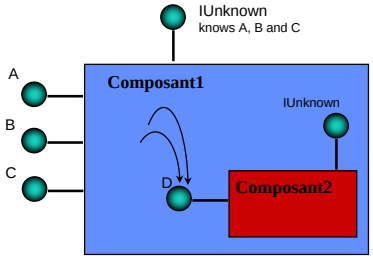
- Encapsulation "totale"
  - *Black-Box* : chaque composant est vu comme une boîte noire
  - L'interopérabilité entre composants ne se fait que via leurs *interfaces*
  - Possibilité de définir des *interfaces multiples* pour un même composant
  - QueryInterface : 'découvrir' les interfaces en cours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



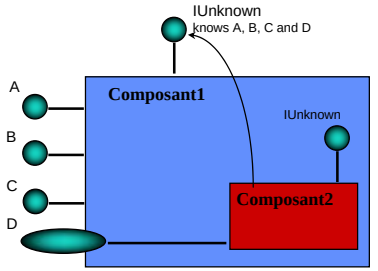
La composition dans COM (d'après Peschanski&Meurisse)

Principes de 'Réutilisabilité' [Microsoft97]

Par confinement / délégation



Par agrégation

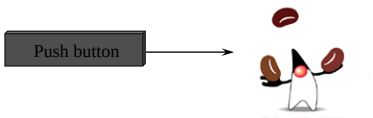


Cycle de vie des composants ('Versioning')...

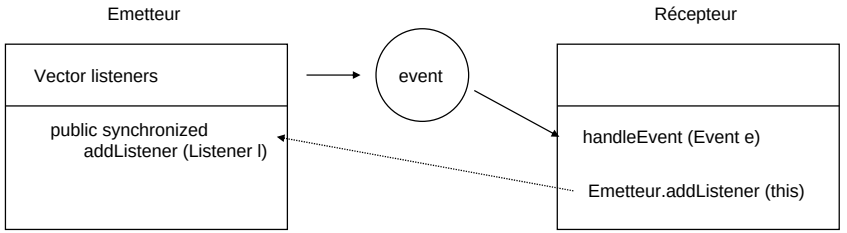
JavaBeans (d'après Peschanski&Meurisse)

- **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**...

Communication des JavaBeans



Inspiré d'un style architectural :
Communication implicite
(*publish/subscribe*)

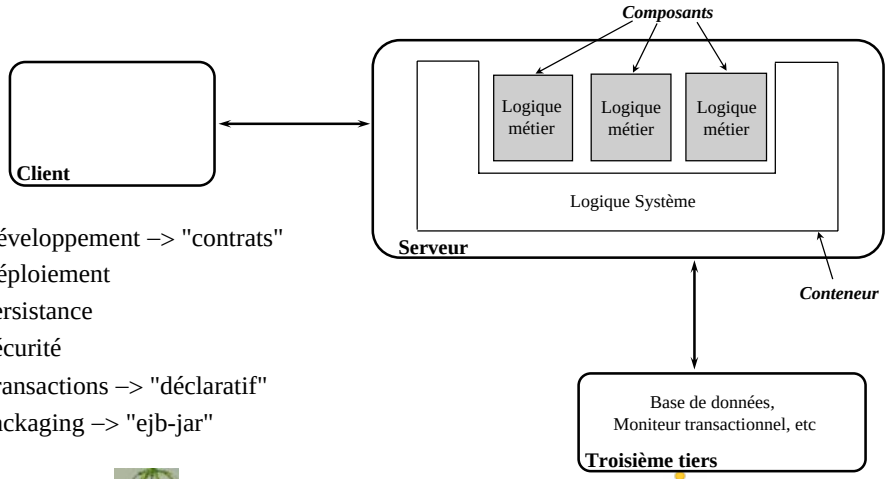


Propriétés JavaBeans (d'après Peschanski&Meurisse)

- Propriétés
  - attributs, ex : couleur, taille, position...
- peuvent être accédés de l'extérieur
  - convention de nommage des méthodes d'accès
  - `int i getX` et `setX(int i)`
- peuvent être "liées" (déclenchement d'événements notification)
- Introspection granularité méthode/attribut
- Déploiement - Packaging (JAR)
- Support de Sérialisation Beans - Evénements
- etc.

Enterprise JavaBeans (d'après Peschanski&Meurisse)

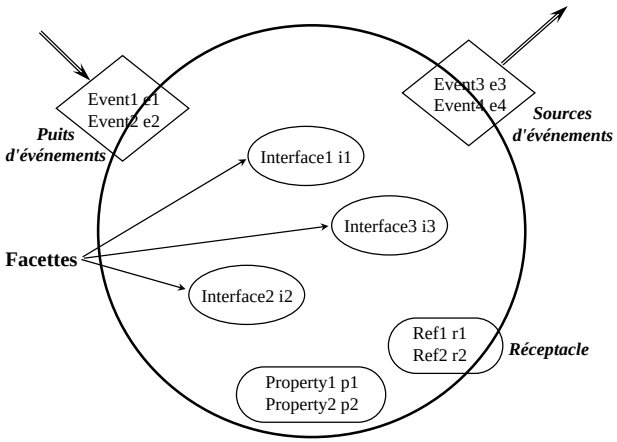
- But : **Simplifier** le développement d'architectures "3 tiers", côté serveur



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

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

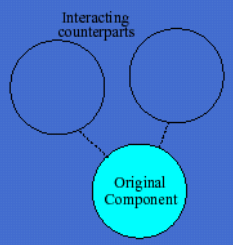
Modèle abstrait



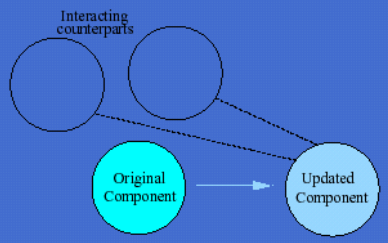

```
component {
  attribute Property1 p1;
  attribute Property2 p2;

  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;
```

component exemple

Example 1 : Component Replacements



Example 1 : Component Replacements



Issues:

➔interaction consistency

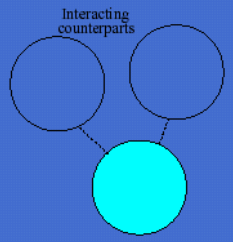
➔state preservation

➔...

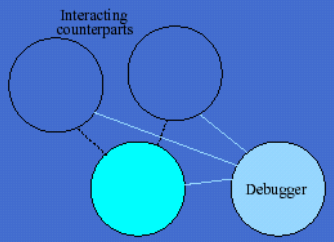
+ can be applied on existing approaches
(server object = Component [XRMI])

- What if changes cross-cut component boundaries ?

Example 2 : Beyond mechanisms



Example 2 : Beyond mechanisms

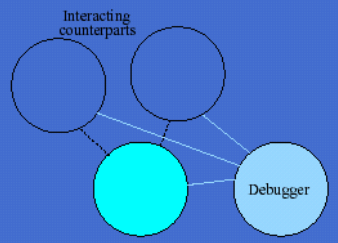


Issues:

➔interaction consistency

➔...

Example 2 : Beyond mechanisms



Issues:

➔interaction consistency

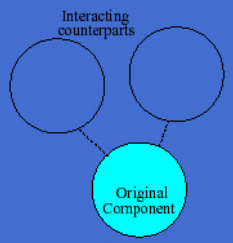
➔...

In our opinion, important prerequisites:

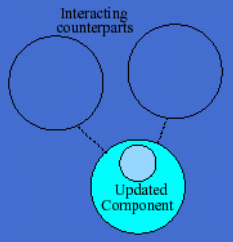
- Structural decoupling = explicit (and dynamic) links
- Control-level decoupling = asynchronism

(Strong) claim : Object-orientation is not adequate

Example 3 : Fine-grained Changes



Example 3 : Fine-grained Changes



Issues:

→impact on component internals

→security

→...

- Design abstractions?
- **Language constructs?**
- **Operational mechanisms?**

The Comet Middleware

[Frédéric Peschanski, Middleware 2003]

A (Free Software) Distributed Environment
for Dynamic Distributed Systems

- Architectural reconfigurations :
  - Adding/replacing/removing components
  - Modifying interconnexions
  - Mobility (strong migration is on the way)
- **Finer-grained changes**
  - Whithin component boundaries
  - Behavioral changes

The Comet Middleware

A (Free Software) Distributed Environment
for Dynamic Distributed Systems

- **The Scope composition language**
  - DSL constructs embedded in a mother language (Java/Scheme)
  - Source-to-source compiler
- **The Comet Middleware**
  - Efficient distributed execution of Scope programs
  - Implemented in Java (JDK >=1.2) above Sockets

Scope Example : Client/Server

Component Definitions :

```
component MMClient {
 receive MMEvent; send MMRequest;
 when(MMEvent event) { show(event); }
 void askServer(MMRequest req) { send(req); } }

component MMServer {
 receive MMRequest; send MMEvent;
 when(MMRequest request) {
  // subclasses refine this method
  doRequest(request); } } }
```


Scope Example : Client/Server

Component Definitions :

```
component MMClient {
 receive MMEvent; send MMRequest;
 when(MMEvent event) { show(event); }
 void askServer(MMRequest req) { send(req); } }

component MMServer {
 receive MMRequest; send MMEvent;
 when(MMRequest request) {
  // subclasses refine this method
  doRequest(request); } } }
```

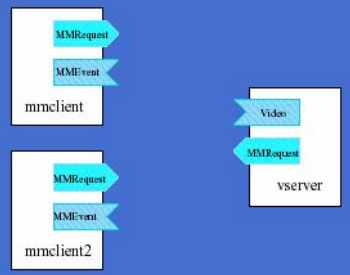
Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
```


Scope Example : Client/Server

Component Definitions :

```
component MMClient {
 receive MMEvent; send MMRequest;
 when(MMEvent event) { show(event); }
 void askServer(MMRequest req) { send(req); } }

component MMServer {
 receive MMRequest; send MMEvent;
 when(MMRequest request) {
  // subclasses refine this method
  doRequest(request); } } }
```



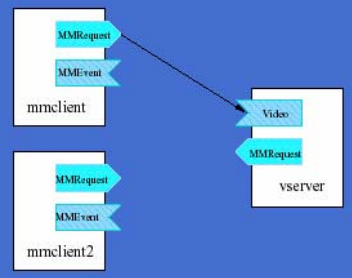
Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);
```


Scope Example : Client/Server

Component Definitions :

```
component MMClient {
 receive MMEvent; send MMRequest;
 when(MMEvent event) { show(event); }
 void askServer(MMRequest req) { send(req); } }

component MMServer {
 receive MMRequest; send MMEvent;
 when(MMRequest request) {
  // subclasses refine this method
  doRequest(request); } } }
```



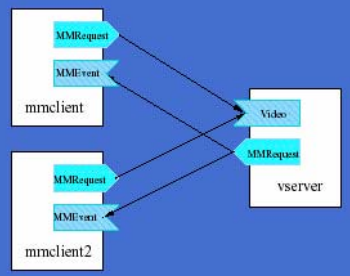
Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);
connect(mmclient, vserver, MMRequest);
```

Scope Example : Client/Server

Component Definitions :

```
component MMClient {
 receive MMEvent; send MMRequest;
 when(MMEvent event) { show(event); }
 void askServer(MMRequest req) { send(req); } }

component MMServer {
 receive MMRequest; send MMEvent;
 when(MMRequest request) {
  // subclasses refine this method
  doRequest(request); } } }        send(event);
```

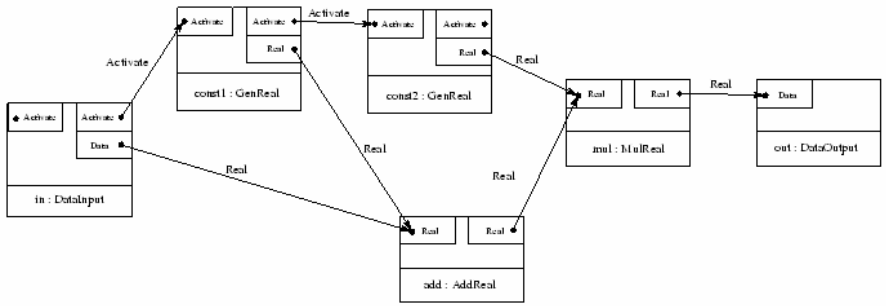


Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);
connect(mmclient, vserver, MMRequest);
connect(vserver, mmclient, MMEvent);
... // and so on
```


Scope (F. Peschanski) : Connection Type Inference

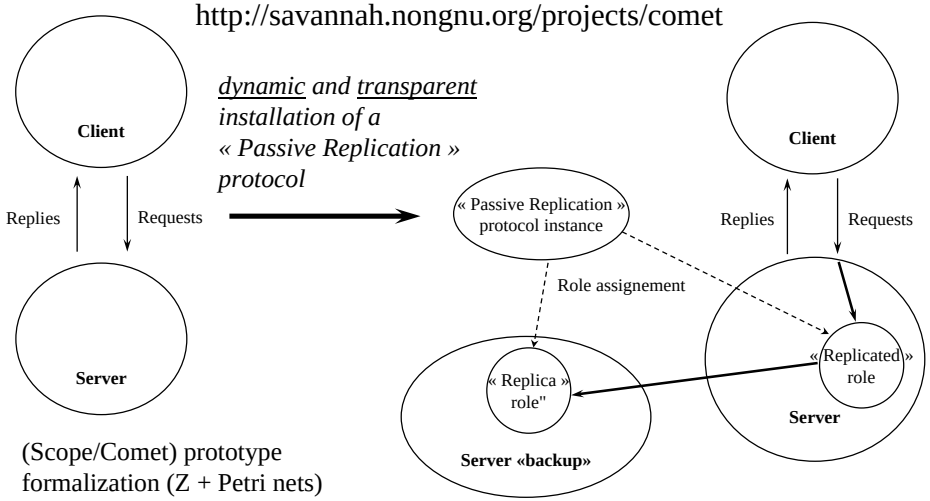
- Expressing the dynamics of the type system
  - adding/removing components and connections
- Automatic type inference of connection between components
  - connection types used for : data type checks, optimization of connections
- Formalization (state-transition semantics for dynamicity)
- Hierarchical (composite) components
- Type-based multicast (automatic discrimination based on input-types)



Ex : Passive Replication (backup) Protocol Installation (Frédéric Peschanski PhD, Middleware'2003)

http://savannah.nongnu.org/projects/comet

dynamic and transparent installation of a « Passive Replication » protocol



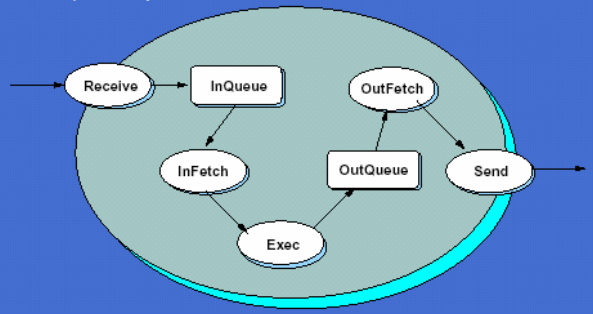
(Scope/Comet) prototype
formalization (Z + Petri nets)
Scheme/Java implementations

Replication Protocol (2)

```
role ReplicatedRole {
   prehook MMRequest ;
   send MMRequest ;
   before MMRequest(event) {
      send (event);
   } }

protocol ReplicationProtocol {
   public void createReplica(String repClass, Address machine) {
      ComponentRef ref = upload (repClass, machine);
      instantiate (ref);
   }
   public void doReplication(ComponentRef server, ComponentRef replica) {
      assign(ReplicatedRole, server);
      connect(server, replica, Event);
   } }
```

Component Internals

Within (top-level) Components : Micro-architecture of sub-components



Fine-grained Dynamic Adaptability :

- Add/change/remove sub-components
- Change interconnections

Protocols and Roles

Language-level abstractions

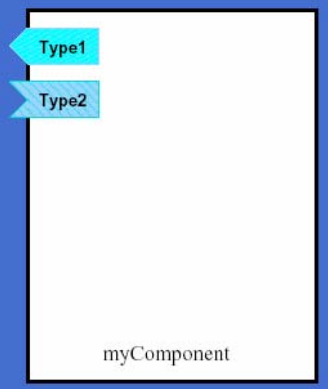
- **Protocols** : Description of Adaptive Interactions
- **Roles** : What should provide/require the involved components

Categories of adaptation:

- **Functional roles** : Add/Replace functionalities
- **Hooks** : pluggable wrappers around functionalities
- **Filters** : Type/Content-based filtering

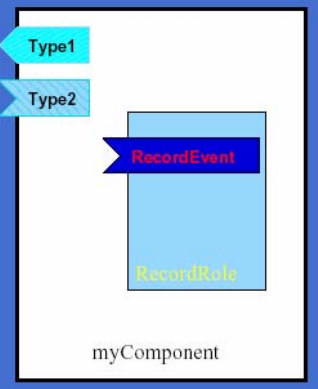
Functional Roles : an Example

```
event ConfigEvent {
  slot key type String;
  slot val type Object; }
  String getKey() { return key; }
  Object getVal() { return val; }

role RecordRole {
  receive RecordEvent;
  HashMap config = new HashMap();
  when(RecordEvent re) {
    config.put(re.getKey(), re.getVal()); }}

protocol ConfigAdapt {
  void adapt(ComponentRef comp) {

  }}
```



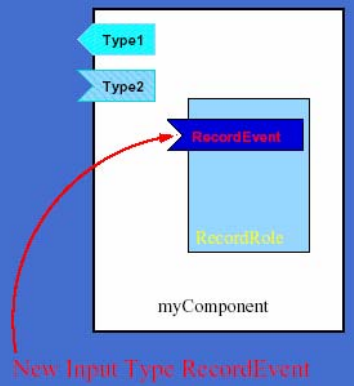
Functional Roles : an Example

```
event ConfigEvent {
  slot key type String;
  slot val type Object; }
  String getKey() { return key; }
  Object getVal() { return val; }

role RecordRole {
  receive RecordEvent;
  HashMap config = new HashMap();
  when(RecordEvent re) {
    config.put(re.getKey(), re.getVal()); }}

protocol ConfigAdapt {
  void adapt(ComponentRef comp) {
    assign(comp, RecordRole);
  }}
```



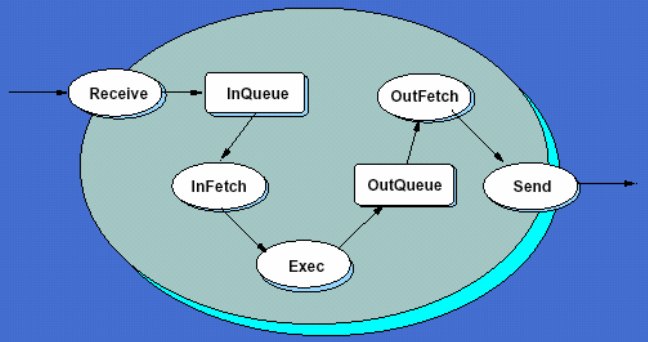
Functional Roles : an Example

```
event ConfigEvent {
 slot key type String;
 slot val type Object; }
 String getKey() { return key; }
 Object getVal() { return val; }

role RecordRole {
 receive RecordEvent;
 HashMap config = new HashMap();
 when(RecordEvent re) {
  config.put(re.getKey(), re.getVal()); }}

protocol ConfigAdapt {
 void adapt(ComponentRef comp) {
  assign(comp, RecordRole);
 }}
```



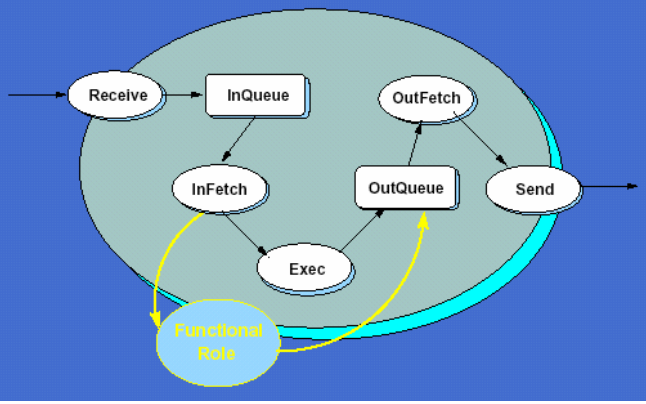
Functional Roles : Assigment

Internally : Derivation as “Exec-like” sub-component



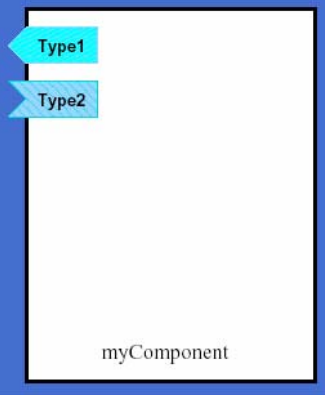
Functional Roles : Assigment

Internally : Derivation as “Exec-like” sub-component



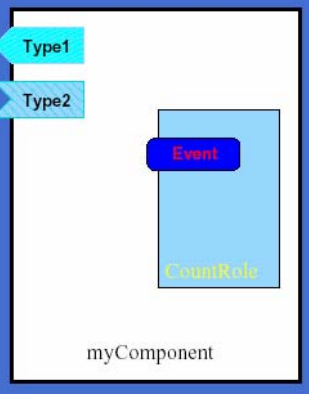
Hook Roles : an Example

```
role CountRole {
 prehook Event;
 int count = 0;
 when(Event e) {
  count++; }}

protocol CountAdapt {
 void adapt(ComponentRef comp) {

 }}
```



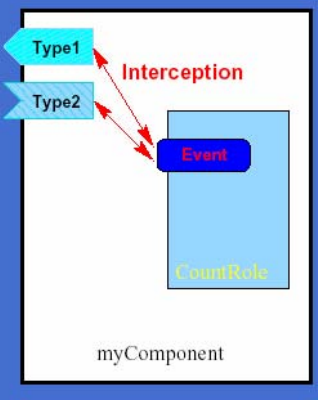
Hook Roles : an Example

```
role CountRole {
 prehook Event;
 int count = 0;
 when(Event e) {
  count++; }}

protocol CountAdapt {
 void adapt(ComponentRef comp) {
  assign(comp, CountRole);
}}
```



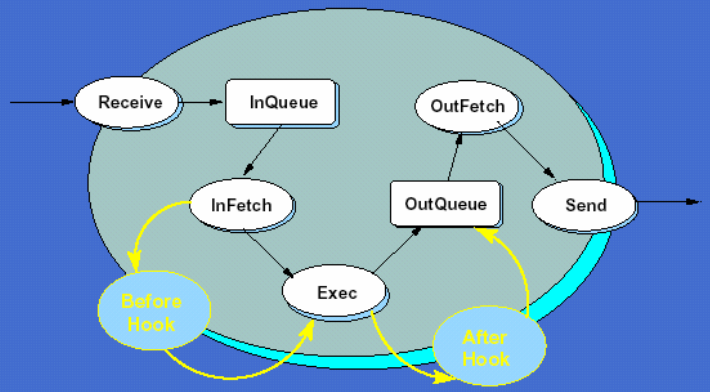
Hook Roles : an Example

```
role CountRole {
 prehook Event;
 int count = 0;
 when(Event e) {
  count++; }}

protocol CountAdapt {
 void adapt(ComponentRef comp) {
  assign(comp, CountRole);
}}
```



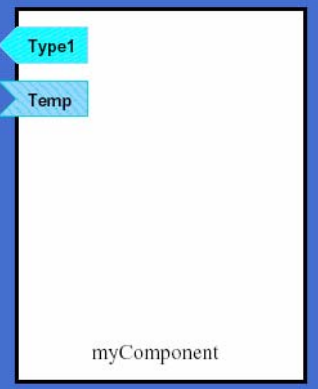
Hook Roles : Assigment

Internally : Wrappers around the Exec sub-component



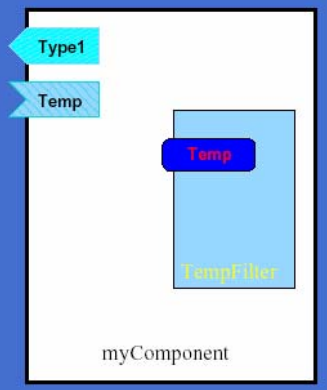
Filter Roles : an Example

```
event TempEvent {
 slot temp type float;
 float getTemp() { return temp } }

role TempFilter {
 infilter TempEvent;
 int count = 0;
 when(TempEvent t) {
  if(t.getTemp()>0.0) return t;
   else return null; }}

protocol TempAdapt {
 void adapt(ComponentRef comp) {

 }}
```



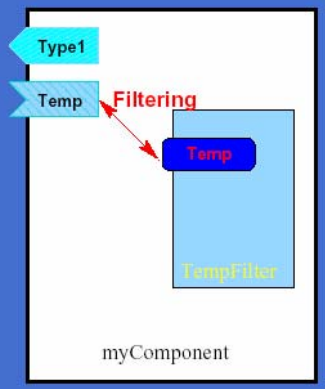
Filter Roles : an Example

```
event TempEvent {
  slot temp type float;
  float getTemp() { return temp } }

role TempFilter {
  infilter TempEvent;
  int count = 0;
  when(TempEvent t) {
    if(t.getTemp()>0.0) return t;
      else return null; }}

protocol TempAdapt {
  void adapt(ComponentRef comp) {
    assign(comp, TempFilter);
  }}
```



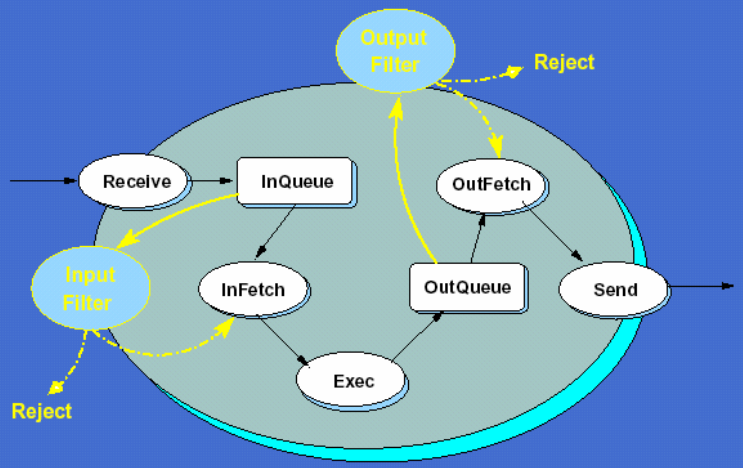
Filter Roles : an Example

```
event TempEvent {
  slot temp type float;
  float getTemp() { return temp } }

role TempFilter {
  infilter TempEvent;
  int count = 0;
  when(TempEvent t) {
    if(t.getTemp()>0.0) return t;
      else return null; }}

protocol TempAdapt {
  void adapt(ComponentRef comp) {
    assign(comp, TempFilter);
  }}
```



Filter Roles : Assigment

Internally : Pre-fetching sub-components



Role composition : an example

Issue: Asynchronous control requires special care when :

- source and destination components do not work at the same speed
- network slowdowns

Risk : event queue “explosions”

Role composition : an example

Default Event flow management algorithm :

- Send and queue event until **ceil capacity** exhausted on either source or destination
- Stop exceptions sent to senders (no propagation though)
- Restart exceptions sent when **floor capacity** reached

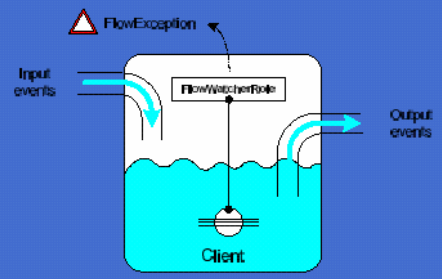
Role composition : an example

Efficient **optimistic** algorithm for most systems But some systems needs more pessimistic algorithms (perhaps temporarily)

Idea: Dynamic adaptation for pessimistic flow regulation

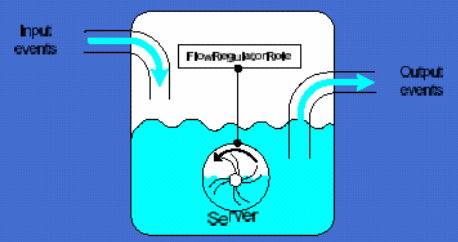
Flow Watching Role

```
role FlowWatcher {
 prehook Event; send FlowException;
 long _event_count=0; long _avg_limit=100;
 long _avg_time=0;
 before(Event event) {
  _event_count++;
  long current = System.currentTimeMillis();
  _total_time = _total_time + current;
  _avg_time = _total_time / _event_count;
  if(_avg_time > _avg_limit)
   send(new FlowException(_avg_time));
 }}
```

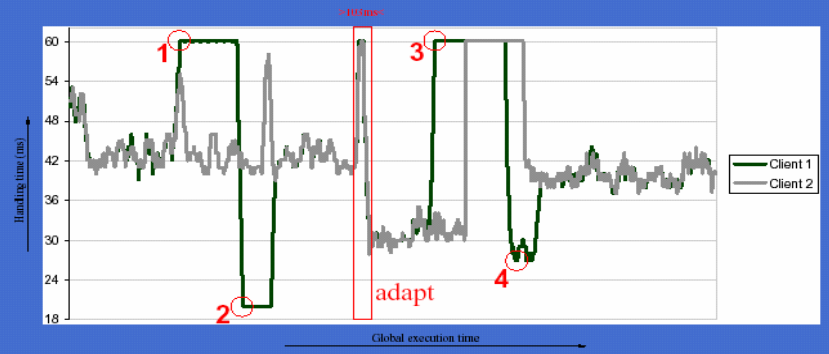


Flow Regulation Role

```
role FlowRegulator {
 prehook Event; posthook Event;
 receive FlowException;
 long _start_time; long _end_time;
 long _rate=1000;
 before(Event event) {
  _start_time = System.currentTimeMillis();
 } after(Event event) {
  _end_time = System.currentTimeMillis();
  if(_end_time - _start_time < _rate)
   Thread.sleep(min_rate - (_end_time - _start_time));
 }
 when(FlowException except) {
  _rate = except.getRate(); // Detected anomaly
 }}
```



Flow Regulation in Practice




```
Protocol FlowSyncProtocol {
   void sync(ComponentRef server ; ComponentRef client) {
      assign(client, FlowWatcher) ;
      assign(server, FlowRegulator) ;
      connect(server, client, FlowException) ;
   } }
```


Security: Verification Contracts

"Hand-shake" contracts (using XML):

- Component host verifies assigned role
- Role verifies host

Contract categories:

- **Typing contracts** : check compatibility of role and components input, output, filter and hook types
- **Access contracts** : allow/forbid references/method calls within role bodies (e.g. `outer` reference)
- **Security contracts** : authentication, encryption, certification (Java Security API)

Conclusion and Future Work

Comet/Scope = Adaptation in Practice:

- Architectural Reconfigurations (see bibliography)
- **Finer-grained dynamic adaptations**
- A Free-software Project at GNU Savannah : `http://www.non-gnu.org/comet`
- Various Applications : MAS (DIMA), Dist. CAD (SALOME), Load-balancing, Dist. Debugging, ...

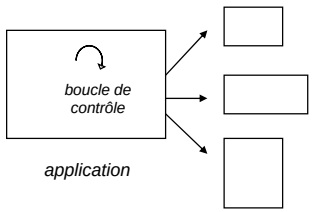
Ongoing work:

- **Mobility** : Self-stabilizing forwarding algorithm, Strong migration (integration with JavaGO)
- **Formal insights** : Essentially about Multicast Interactions
- **Real-world** : RMI/Corba Compararision/Interoperability
- **Longer term** : Low-cost mobility for large-scale reactive multi-agent systems

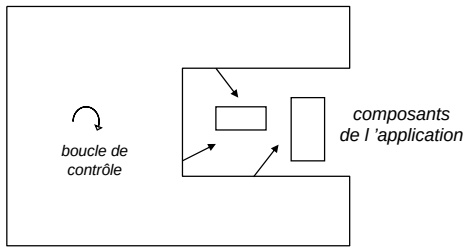
Frameworks

- 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



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



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

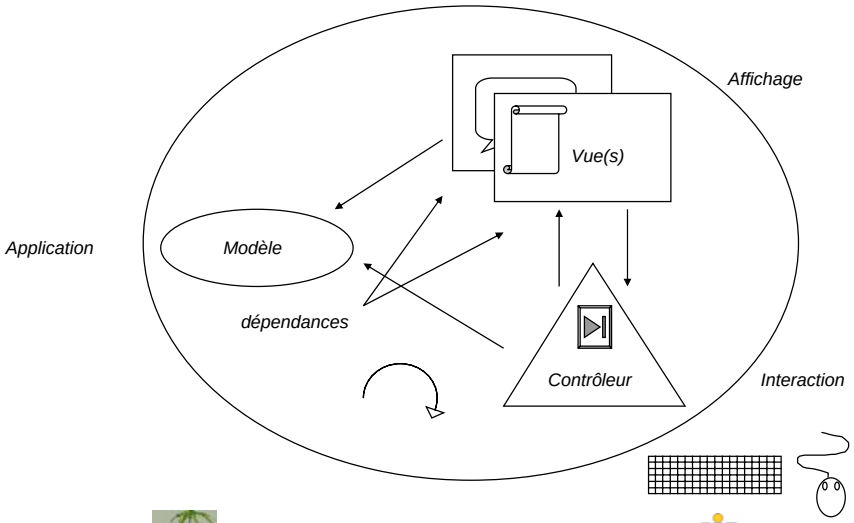
Architectures logicielles/Composants vs Frameworks

- 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

Pattern = < Problème , Solution >

- Un pattern n'est *pas* juste une solution, mais est la discussion d'un type de solution en réponse à un type de problème
- nom (ex : Bridge, Observer, Strategy, Decorator...)
- contexte
- problème
- forces
- collaboration (possible avec d'autres patterns)
- directives
- exemples
- ...

Utilisation :

- Capitalisation de connaissances
- Explication
- Génération (vers une instanciation automatique de patterns)

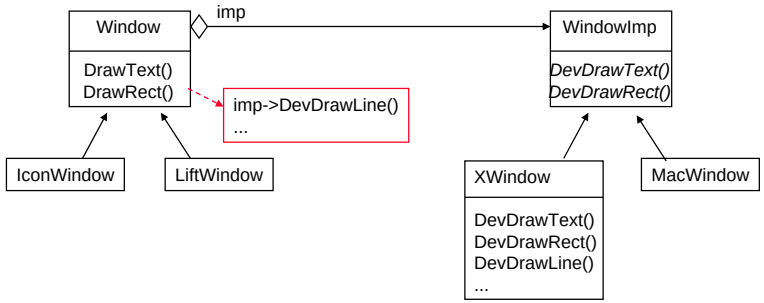
Ex : pattern Bridge

- 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

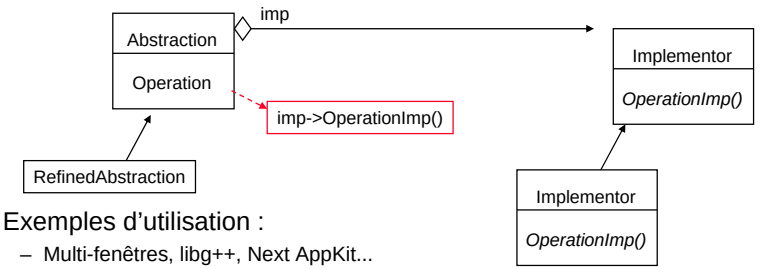
Ex : pattern Bridge (2)

- 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 ?

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