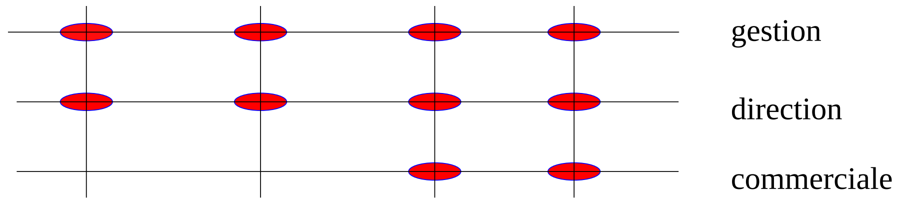
Sommaire

- √ Les objectifs de CROME
  - ⟹ conception fonctionnelle
  - ⟹ exploitation des objets "Crome"
- √ L'outil CromeDB
- √ Des "objets CROME" distribués

De la conception fonctionnelle à l'exploitation d'objets multi-contextes

Olivier.Caron@lifl.fr

Orthogonalité Objets / Fonctions



Crome : conception objet par contextes fonctionnels

√ **Objectif 1 :** conception du SI d'une entreprise

⟹ classiquement : définition des objets du monde réel

- modèle ER (SGBDR)
- modèle objet UML
- modèle ODL de l'ODMG.

⟹ Apport de Crome :

Prise en compte de la structuration en fonctions du SI **au sein de l'objet**

Crome : de la conception à l'exploitation

√ **Objectif 2 :** Préservation de la structuration fonctionnelle lors de l'exploitation

√ Les avantages :

$\Longrightarrow$ vue simplifiée des objets pertinente pour la fonction

$\Longrightarrow$ modèle de sécurité

- au sens génie logiciel à la conception
- droits d'accès à l'exploitation

$\Longrightarrow$ modèle adapté à la répartition des informations

- Conception : répartition logique
- Implantation : répartition physique

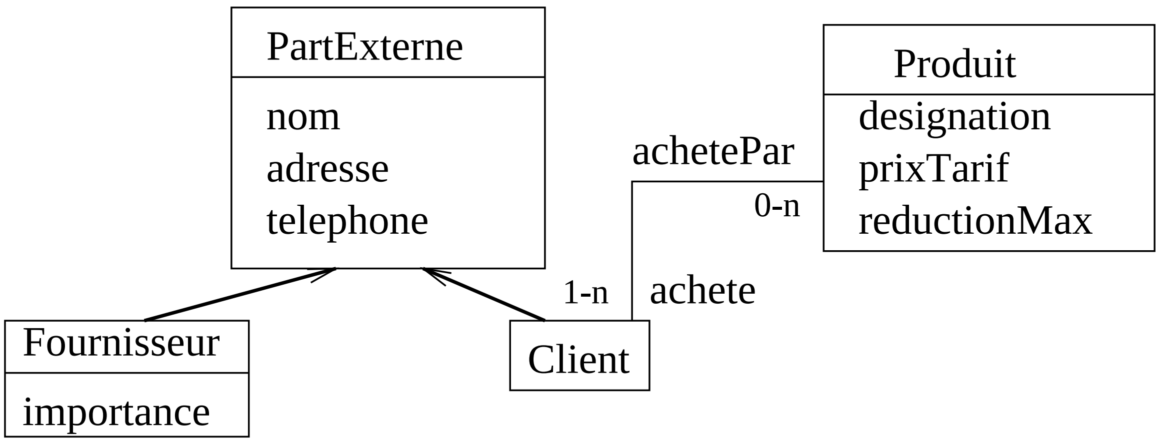
Contexte fonctionnel

√ Les fonctions déterminent les contextes d'intervention de ces objets

$\Longrightarrow$ conséquences intensionnelles

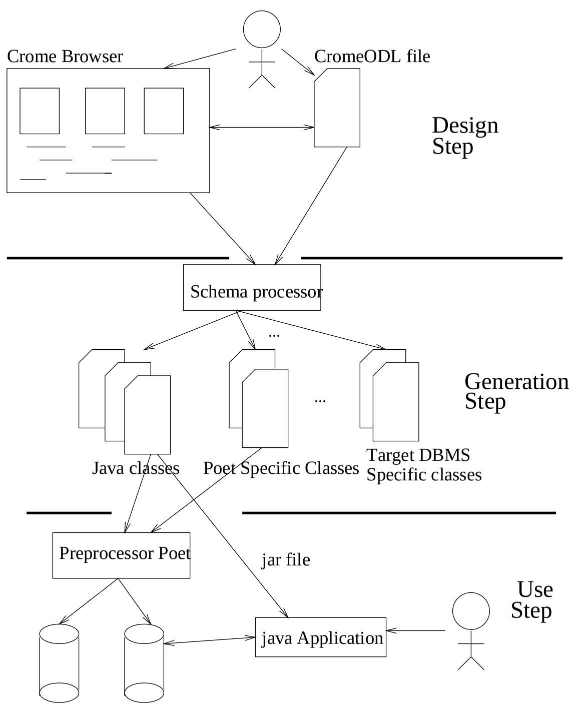
$\Longrightarrow$ conséquences extensionnelles

Définition du plan de base (1)



- √ Informations communes : le référentiel du schéma
- √ Définition de la structure

L'environnement CromeBD



Définition du plan de base (3) : Mapping Java

```
// fichier SI/Base/PartExterne.java
package SI.Base ;

public interface PartExterne {
  String getNom() throws SI.Crome.CromeException ;
  String getAdresse() throws SI.Crome.CromeException ;
  String getTelephone() throws SI.Crome.CromeException ;
}
```

```
// fichier SI/Base/Client.java
package SI.Base ;

public interface Client extends PartExterne {
  java.util.Vector  getAchete()  throws SI.Crome.CromeException ;
}
```

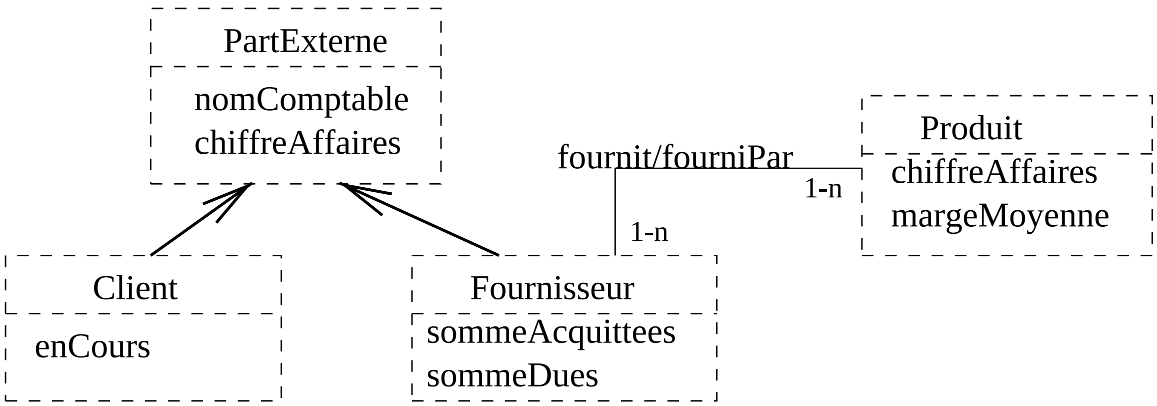

Définition du plan de base (2) : CromeODL

```
interface PartExterne {
  attribute  string  nom;
  attribute  string  adresse;
  attribute  string  telephone;
} ;

interface Fournisseur : PartExterne { attribute string importance } ;

interface Client : PartExterne {
  relationship set<Produit>[1] achete inverse Produit::achetePar ;
} ;
interface Produit {
  relationship set<Client>[1] achetePar inverse Client::achete ;
  ...
```

Définition du plan fonctionnel (2) : CromeODL

```
function gestion interface  increment PartExterne {
  write base part : {
    attribute  string  nom;
    attribute  string  adresse;
    attribute  string  telephone;
  } ;
  function part : {
    attribute  string  nomComptable;
    attribute double chiffreAffaires ;
  } ;
} ;
function gestion interface  increment Client {
  instanciation function ;
  write base part : { ...
```


Définition plan fonctionnel (1)



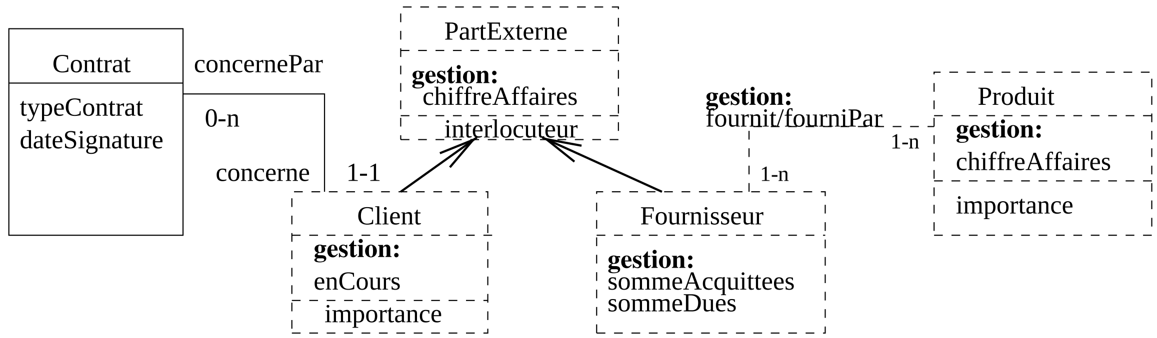
Plan fonctionnel de la fonction "gestion"

√ Préservation structure hiérarchique

√ Enrichissement implicite par héritage

$\Longrightarrow$ intension fonctionnelle

Classes locales et partage d'informations inter-contextes (1)



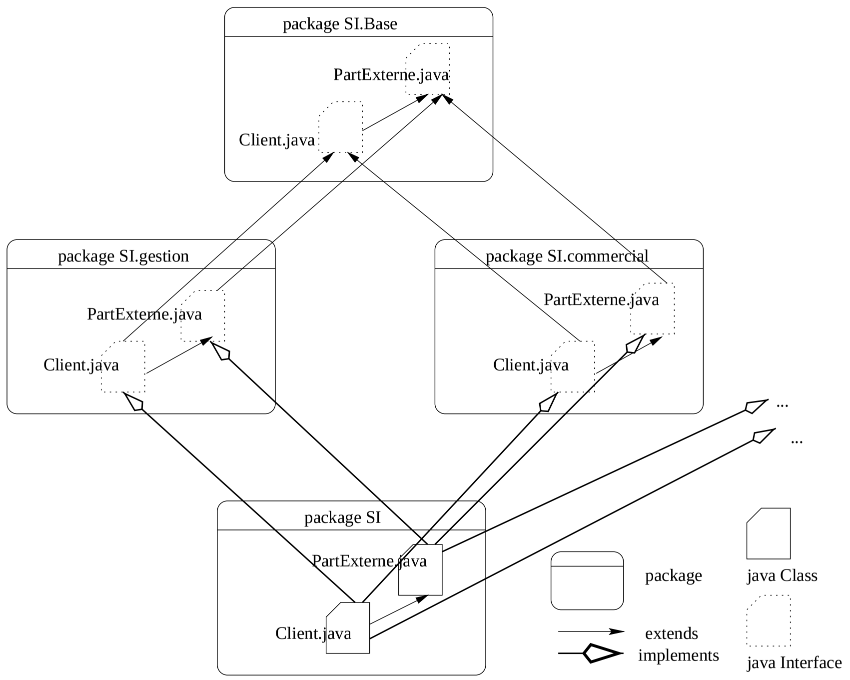
Plan fonctionnel "direction"

- √ classes locales : préservation de la structure du plan de base
- √ notion d'accès inter-plan

Définition du plan fonctionnel (3) : Mapping Java

```
// fichier SI/gestion/PartExterne.java
package SI.gestion ;
public interface PartExterne extends SI.Base.PartExterne {
 void setNom(String value) throws SI.Crome.CromeException ;
 void setAdresse(String value) throws SI.Crome.CromeException ;
 void setTelephone(String value) throws SI.Crome.CromeException ;
 String getNomComptable() throws SI.Crome.CromeException ;
 void setNomComptable(String value) throws SI.Crome.CromeException ;
 double getChiffreAffaires() throws SI.Crome.CromeException ;
 void setChiffreAffaires(double value) throws SI.Crome.CromeException ;
}
// fichier SI/gestion/Client.java
package SI.gestion;
public interface Client extends SI.Base.Client, PartExterne { ...
```

Architecture Java



package SI.Base ;
interface Client extends
PartExterne {...

package SI.gestion ;
interface Client extends
SI.Base.Client, PartExterne {...

package SI.commercial ;
interface Client extends
SI.Base.Client, PartExterne {...

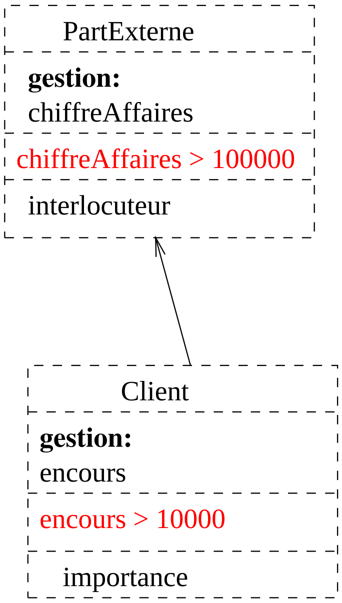
package SI ;

public class Client extends
PartExterne
implements SI.commercial.Client,
SI.gestion.Client, ... {

Classes locales et partage d'informations inter-contextes (2)

```
function direction interface Contrat {
  instanciation function ;
  ... }
function direction interface  increment Produit {
  access function gestion : {
  relationship set<Fournisseur>[1] fourniPar
                 inverse Fournisseur::fournit ;
  attribute  float chiffreAffaires;
  } ;
  function part : { attribute  string  importance; } ;
} ;
```

- √ Mapping java pour accès Inter-Plan -> définition fonction "get" pour l'interface fonctionnelle
- √ Mapping java pour classes locales : une interface Java dans le paquetage fonctionnel



fonction "direction"

Extension fonctionnelle (1)

√ prédicat de participation associé à chaque classe pour chaque fonction

- attributs issus du plan de base
- attributs provenant d'accès inter-plans

√ prise en compte pour le calcul de l'extension :

- héritage des prédicats
- contextes fournisseurs
- cardinalités association

Adéquation Concepts Java / concepts Crome

code JAVA généré	**Signification CROME**
Paquetage `nomBase`	classes persistentes
Sous-paquetage `nomBase.Base`	description du plan de base
Sous-paquetage `nomBase.`*fi*	description du plan fonctionnel *fi*
Fichier `nomBase.Base.`*Ci*	description de base de la classe *Ci*
Fichier `nomBase.`*fi*`.`*Ci*	partie fonctionnelle *fi* de *Ci*
Sous-paquetage `nomBase.Crome`	classes utilitaires

Extension fonctionnelle (3) : mapping Java

```
public boolean setParticipe() {
  String fonction=this.getFonctionCourante() ;
  participe=super.setParticipe() ;
  if (fonction.equals("direction"))
    participe = participe && (enCours > 10000.0) ;
  if (fonction.equals("commercial"))
    participe = participe && (enCours > 10000.0) ;
  return participe ;
}
```

La fonction `setParticipe` de la classe Client

Extension fonctionnelle (2) : CromeODL

```
function direction interface  increment Client {
  access function gestion : {
    attribute  float chiffreAffaires;
  } ;
  extent condition function : "enCours > 10000.0" ;
  function part : {
  attribute  string  importance;
  relationship set<Contrat>concernePar inverse Contrat::concerne ;
  attribute  string  importance;
  } ;
} ;
```

Un exemple d'exploitation : contexte "gestion"

```
import SI.gestion.* ;
import SI.Crome.* ;
...
  try {
    Client unClient = ( Client ) new SI.Client() ;
    Produit unProduit = (Produit) new SI.Produit() ;
    unClient.setNom("dupont") ;
    ...
    unClient.addAchete(unProduit) ;
  } catch (CromeException e) { ...
```


Génération des classes Java

- √ implémentent toutes les interfaces Java : méthodes d'accès
  - ⟹ mécanisme d'intégrité référentielle
  - ⟹ condition de participation
- √ mécanisme d'exceptions fonctionnelles :
  - ⟹ `InstanciationException`
  - ⟹ `ViolationException`
  - ⟹ `AssociationException`

Comparaison Vues - Crome

Les vues	Crome
. Outil général	+ Outil orienté fonction
= Définition intensionnelle	= Définition intensionnelle
= Définition extensionnelle	= Définition extensionnelle
= Sécurité	= Sécurité
− Structuration base non fonctionnelle (extraction)	+ Structuration base fonctionnelle (augmentation)
− peu courant SGBDOO du commerce	+ SGBDOO couplés Java
− Mise à jour de vues	+ Mise à jour possible

Adaptation SGBD cible couplés avec Java

√ Trois portages :

⟹ PSE/PSE pro (`http://www.odi.com`)

⟹ Poet software (`http://www.poet.com`)

⟹ smallDB (`http://www.lifl.fr/~carono/smallDB`)

√ L'adaptation consiste à filtrer les instances selon la fonction courante pour chaque requête.

```
select * from Client c
    where c.setParticipe()
```

Vers des "objets CROME" distribués

Etude du passage de la distribution logique du modèle de conception (induites par les contextes) à une distribution physique

√ Deux environnements étudiés :

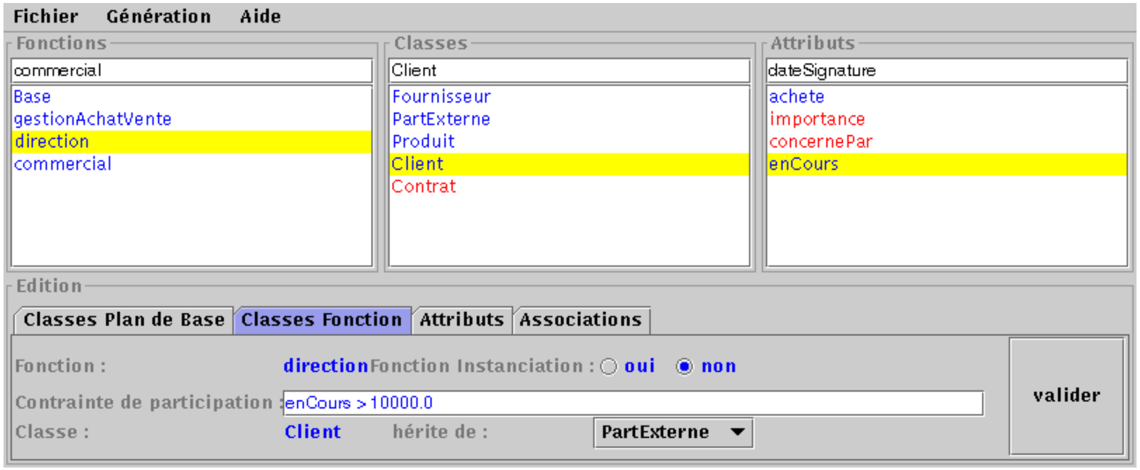
⟹ Java RMI

⟹ CORBA

Conservation de l'architecture de packages d'interfaces générée

√ Vers des composants à interfaces multiples fonctionnelles et distribuées.

L'environnement CromeDB

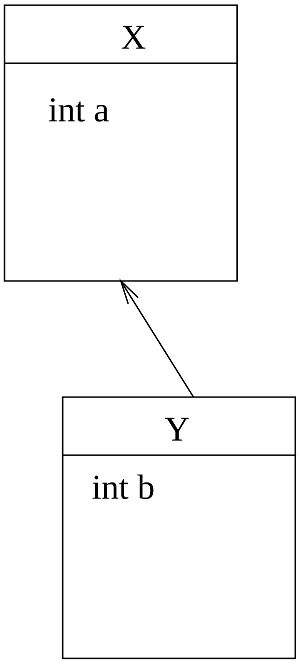


√ "100 % Pure java"

√ Classes générées conforme protocole d'accès Java Beans : *Outil CromeQuery (introspection)*

√ Téléchargeable `http://www.lifl.fr/DOWNLOAD/`

Un exemple simple : plan de base



Des "objets CROME" distribués

Etude du passage de la distribution logique du modèle de conception (induites par les contextes) à une distribution physique

√ Une première Approche :

⟹ Une fonction = un site physique distant

⟹ un objet de conception Crome correspond à un serveur d'objets unique avec accès filtré.

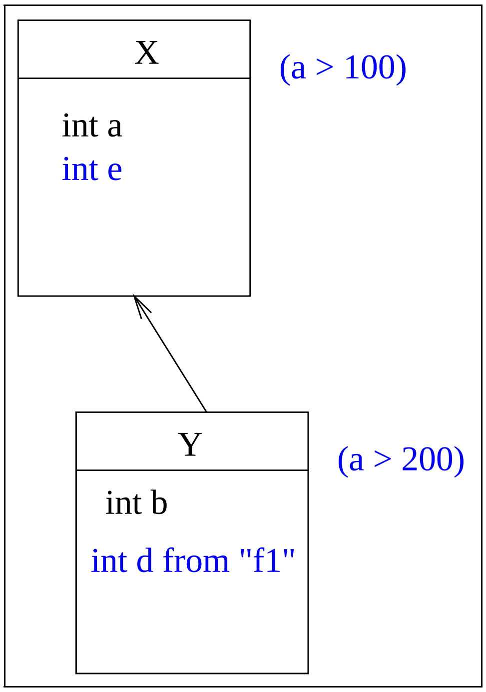
√ Deux environnements étudiés :

⟹ Java RMI

⟹ CORBA

$\boxed{\text{Conservation de l'architecture de packages d'interfaces générée}}$

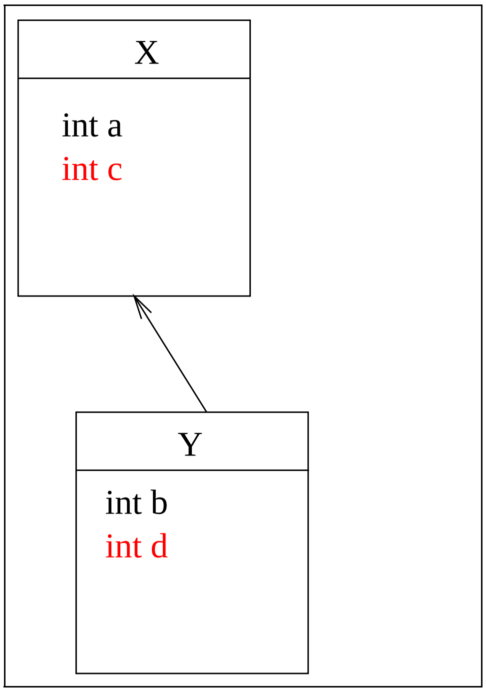
Un exemple simple : contexte f2



contexte "f2"

- √ contexte "consommateur"
- √ contrainte de participation
- √ accès inter-plan

Un exemple simple : contexte f1



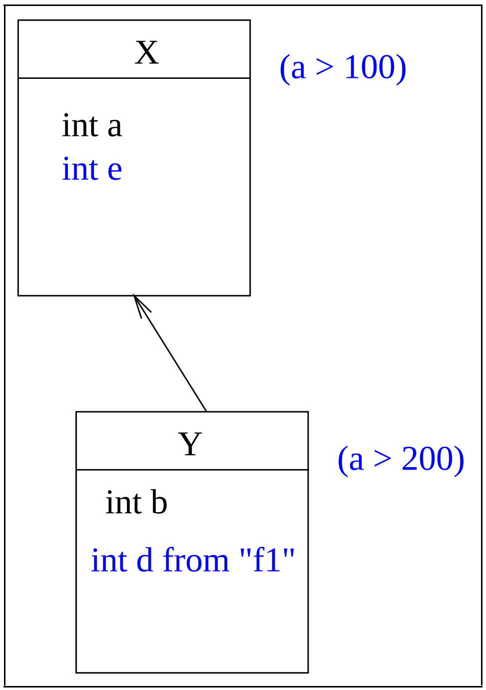
contexte "f1"

- √ contexte instanciation pour X et Y
- √ droit d'écriture pour les attributs du plan de base

Crome et CORBA se rencontrent

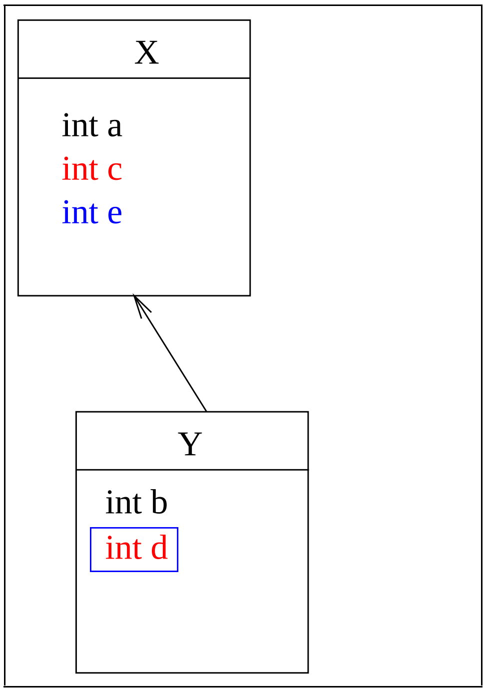
```
module Base {
  interface X {  readonly attribute long a ; } ;
  interface Y : X { readonly attribute long b ; } ;
} ;
module f1 {
  interface X : Base::X {  attribute long a ; attribute long c ; } ;
  interface Y : X , Base::Y { attribute long b ; attribute long d ; } ;
} ;
module f2 {
  interface X : Base::X { attribute long e ; } ;
  interface Y : X, Base::Y { readonly attribute long d ; } ;
} ;
module SI {
  interface X : f1::X, f2::X  {} ;
  interface Y : X, f1::Y, f2::Y {} ;
} ;
```


CROME et Java RMI




```
package SI.f2 ;

import java.rmi.* ;

public interface Y
  extends SI.Base.Y, X {

  int getD() throws
    SI.Crome.CromeException,
    RemoteException ;
}
```

Crome et CORBA du coté serveur




```
module SI {
  interface X {
    attribute long a ;
    attribute long c ;
    attribute long e ;
  } ;
  interface Y : X {
    attribute long b ;
    attribute long d ;
  } ;
} ;
```


Crome et CORBA : A chaque problème, une solution

√ Le langage IDL ne permet pas la surcharge.

⟹ une description IDL coté serveur

⟹ une description IDL pour chaque client (fonction)

√ Héritage d'implémentation ET héritage d'interface

⟹ mécanisme de délégation (tie - norme CORBA 2.X) : compatible avec l'architecture de packages d'interfaces CROME

Règles entre IDL client et IDL serveur

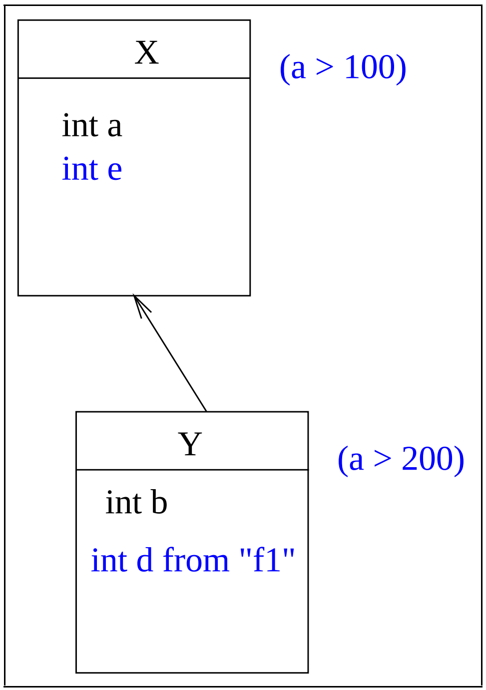
IDL Client $\subseteq$ IDL serveur

ModuleDef client	$\subseteq$	ModuleDef serveur
InterfaceDef client	$\subseteq$	InterfaceDef serveur
AttributeDef client	$\subseteq$	AttributeDef serveur

...

$\surd$ assure un comportement correct pour les "clients statiques" et les "clients dynamiques"

Crome et CORBA du coté client



contexte "f2"

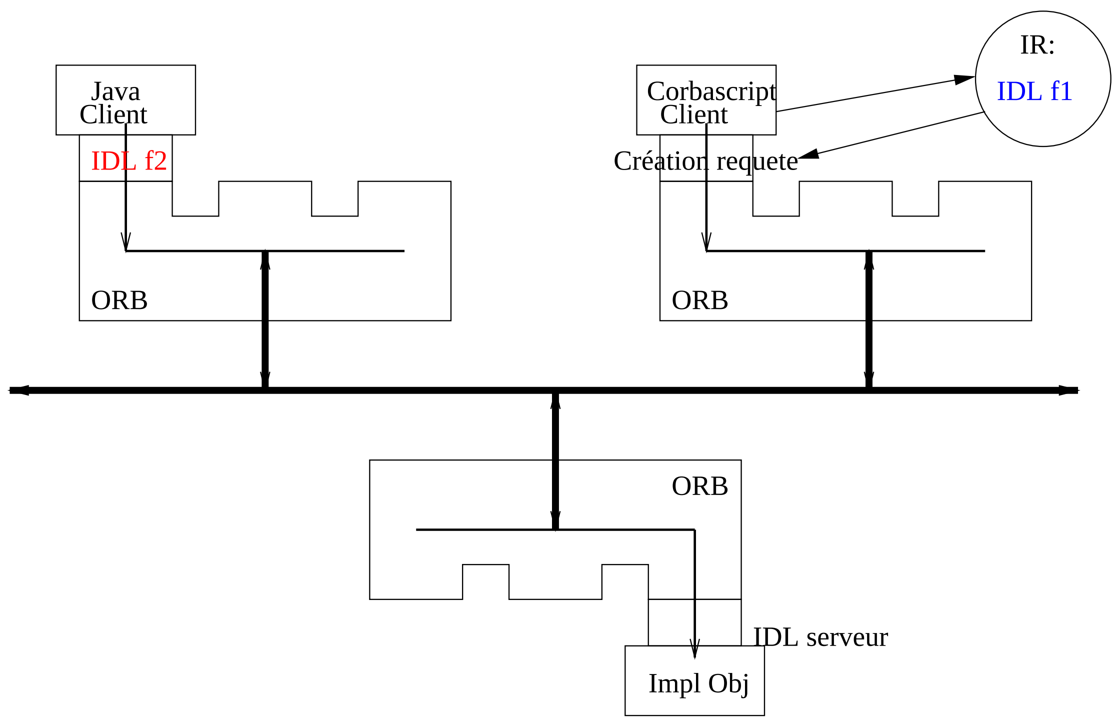
```
module SI {
  interface X {
    readonly attribute long a ;
    attribute long e ;
  } ;
  interface Y : X {
    readonly attribute long b ;
    readonly attribute long d ;
  } ;
} ;
```

Un scénario possible

√ Scénario pour un client dans un contexte :

1. Accès au serveur de noms (NS)
2. Recherche référence du serveur d'objets CROME
3. invocation méthode sur objet serveur (création, récupération d'instances,. . .)
4. invocation méthode sur les objets CROME

Crome et Interopérabilité entre bus CORBA



Un exemple en JAVA dans le contexte "f1"

```
import SI.* ; import SI.Crome.* ;
import org.omg.CORBA.* ; import org.omg.CosNaming.* ;
public class Client {
    public static void main(String args[]) {
      try {
        ORB orb = ORB.init(args, null) ;
        org.omg.CORBA.Object nsref=
        orb.resolve_initial_references("NameService") ;
        NamingContext ns = NamingContextHelper.narrow(nsref) ;
        NameComponent[] serveurSI={ new NameComponent("serveurSI","")} ;
        Serveur serveur=ServeurHelper.narrow(ns.resolve(serveurSI)) ;
        X unX= XHelper.narrow(serveur.creerObjet("f1","X")) ;
        unX.a(300) ;
        System.out.println("valeur de a:"+unX.a()) ;
      } catch(Exception ex) { ...
```


Le serveur d'objets CROME

```
module Crome {
 exception InstanciationException {} ;
 exception ViolationException {} ; ...
 interface Object { ... } ;
} ;
module SI { ...
  typedef string NomFonction ;
  typedef string NomClasse ;
  typedef sequence<Crome::Object> listeObjets ;
  interface Serveur {
    Crome::Object  creerObjet(in NomFonction nomF, NomClasse nomC)
      raises (Crome::InstanciationException) ;
    void detruireObjet(in NomFonction nom, in Crome::Object obj)
      raises (Crome::ViolationException) ;
    listeObjets getExtension(in NomFonction nomF, in NomClasse nomC) ;
  } ; } ;
```

Conclusions sur la première approche

√ Possibilités de descriptions multiples :

- Java RMI
- CORBA

√ Conservation de la structure de packages d'interfaces

√ A faire...

1. Etude des différents services CORBA sur les composants CROME
2. Transactions (société ARJUNA)
3. Sécurité (mécanisme d'authentification, gestionnaire de sécurité java)

Un exemple en CorbaScript dans le contexte "f1"

```
NS = CORBA.ORB.resolve_initial_references ("NameService")

try {
  serveur = NS.resolve ([["serveurSI", ""]])
} catch (CosNaming.NamingContext.NotFound exc) {
  println ("\nWARNING, ‘serveurSI’ non lancé")
  return
}

unX = serveur.creerObjet("f1","X") // on obtient SI.X

unX.a=300
println("valeur de a : ", unX.a)
```

Et demain...

- √ Prise en compte de tous les concepts CROME (Outil de spécification de systèmes d'information répartis) :
  1. contextes fournisseurs
  2. contextes consommateurs
  3. contextes d'instanciation

Un composant CROME sera, à l'exploitation, matérialisé par un ensemble d'objets fonctionnels distribués

- √ Vers la notion de composant Corba à multiples interfaces fonctionnelles
- √ Vers des systèmes de composants à vues multiples et distribués
- √ Vers la dynamicité des composants (ex : ajout d'une fonction)