Composants Logiciels

Christian Pérez

Le modèle de composant de CORBA

Année 2009-10

Plan

- Un rapide tour d'horizon de CORBA 2
- Introduction au modèle de composant de CORBA
- Définition de composants CORBA
  - Exemple du dîner des philosophes
- Programmation des composants CORBA coté clients
- Introduction à CIDL

Un rapide tour d'horizon de CORBA2

Qu'est ce que CORBA

- CORBA = Common Object Request Broker Architecture
  - Défini par l'Object Management Group (OMG)
- CORBA 2: Standard pour la programmation d'objets distribués
  - Bus logiciel
  - Orienté objet
  - Mécanisme d'invocation de méthode à distance
  - Indépendant des machines, des systèmes d'exploitation et des langages de programmation
  - Indépendant des vendeurs (interopérabilité)

4

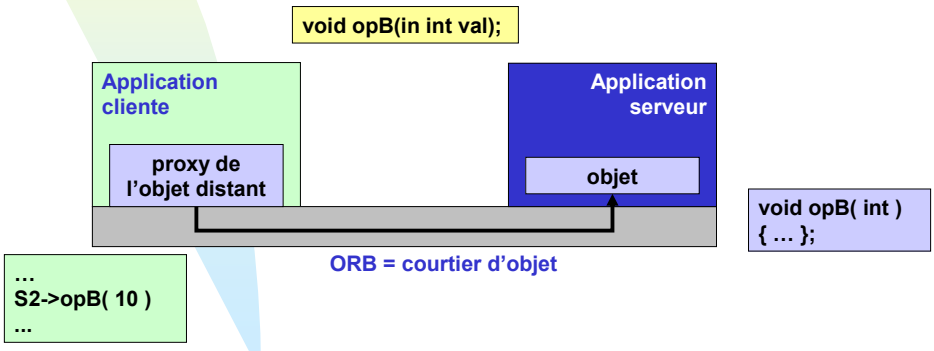
Qu'est ce que CORBA 2 (suite)

- CORBA définit
  - Un langage de spécification (IDL)
    - ~ interfaces en Java + sens des données (IN, OUT, INOUT)
  - Des interfaces pour invoquer des méthodes à distance
  - Un ensemble de services pour manipuler aisément les objets
- CORBA simplifie
  - La localisation des objets
  - L'invocation des méthodes distantes

5

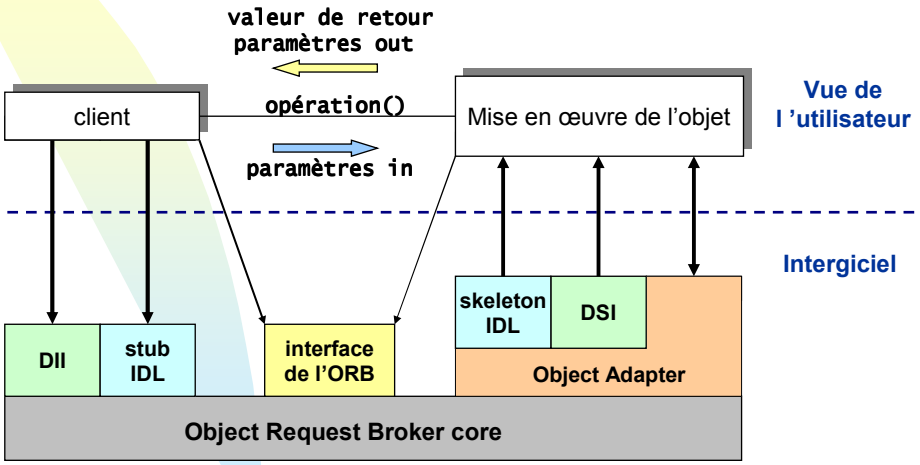
Application CORBA 2

- Ensemble d'objets et de "clients"
- Chaque composant possède une description publique
  - Opérations et attributs pouvant être accédés à distance



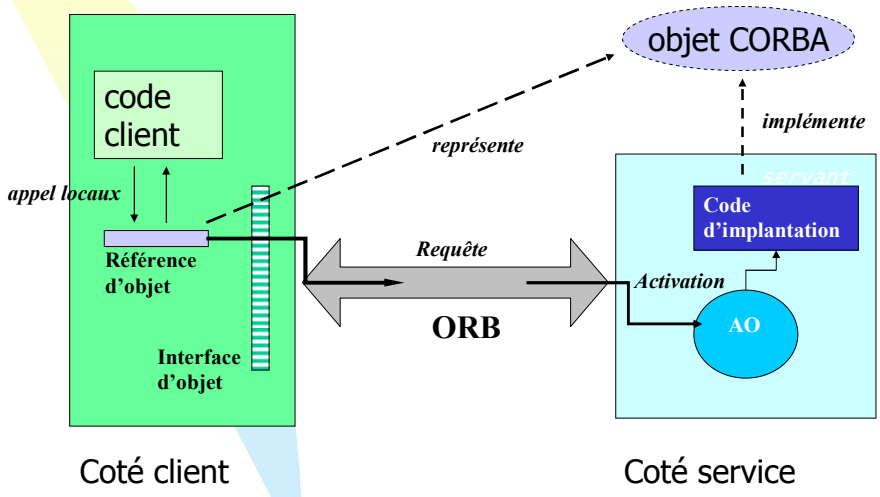
6

L'architecture de CORBA



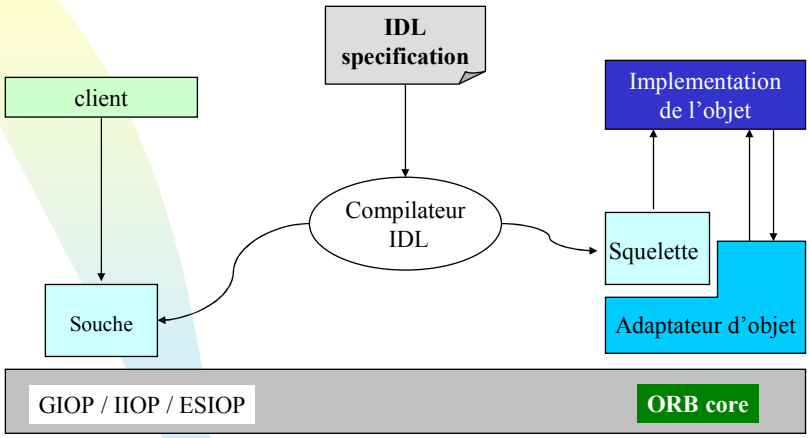
7

CORBA
un modèle client/serveur objet



8

Rôle du langage IDL



9

OMG-IDL : le langage d'interfaces

Ce langage a été défini:

- pour décrire les interfaces des objets
- comme langage pivot entre applications
- pour générer des squelettes de programme dans les langages de programmation des applications

10

OMG-IDL : évaluation

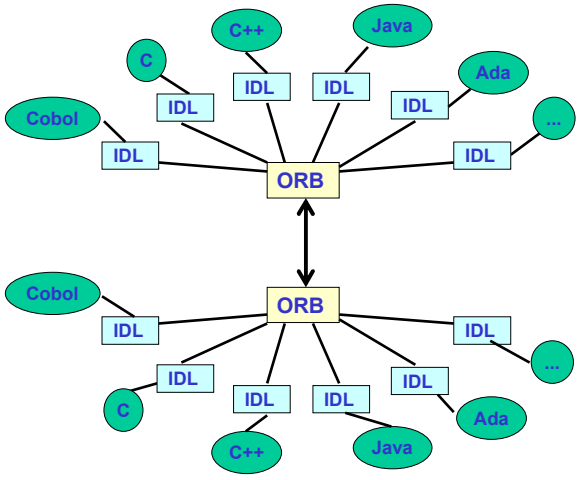
- Bon langage de spécification d'interfaces d'objets
- Facile à apprendre

- Types de base figés
- Passage d'objets par valeur possible mais non recommandé
- Pas de sémantique ni de qualité de service
- Pas de notion d'architecture logicielle

11

Interface Definition Language (IDL)

- Langage indépendant de la mise en œuvre
- Langage de spécification != langage de programmation
- Utilisé pour spécifier des interfaces contenant des opérations et des attributs accessible à distance
- Utilisé pour générer du code dans plusieurs langages de programmation



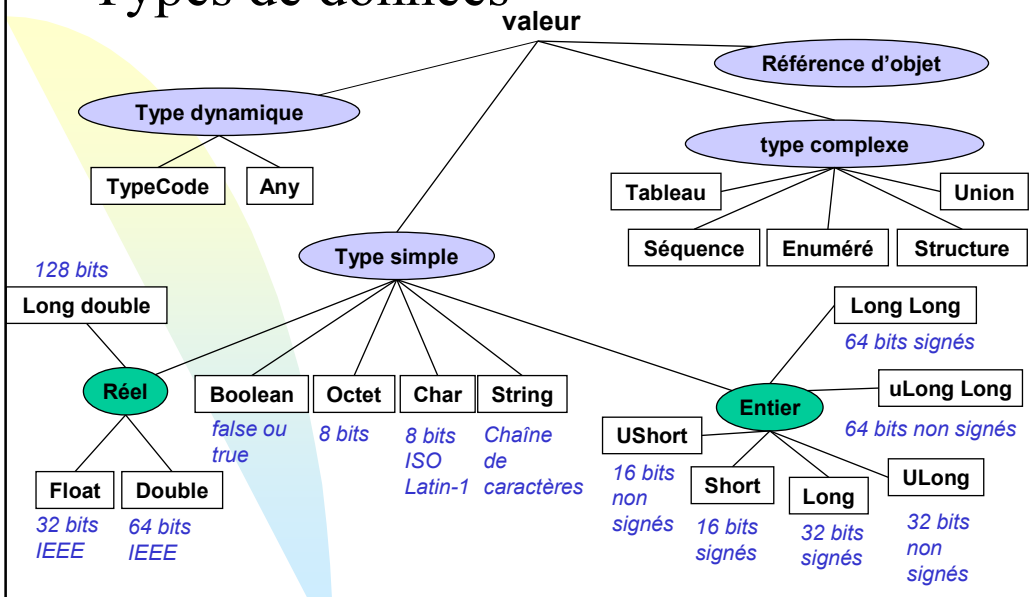
12

Exemple de description d'interface en IDL

```
module ServiceDate {

  typedef unsigned short Jour;
  enum Mois {
    Janvier, Février, Mars, Avril, Mai, Juin, Juillet,
    Août, Septembre, Octobre, Novembre, Décembre
  };
  typedef unsigned short Annee;
  struct Date {
    Jour le_jour;
    Mois le_mois;
    Annee l_annee;
  };
  typedef sequence<Date> desDates;

  interface Calendrier {
    attribute Annee annee_courante;
    boolean vérifier_une_date(in Date d);
    void le_jour_suivant(inout Date d);
  }
};
```

13

Types de données



14

Projection de type (IDL vers C++)

IDL	*C++*
short	CORBA::Short
long	CORBA::Long
long long	CORBA::LongLong
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
unsigned long long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble
char	CORBA::Char
wchar	CORBA::WChar
string	char *
wstring	CORBA::WChar *
boolean	CORBA::Boolean
octet	CORBA::Octet
any	CORBA::Any

15

Introduction au modèle de composant de CORBA

De CORBA 2 . . .

- Un modèle orienté objet distribué
  - Hétérogénéité: OMG Interface Definition Language
  - Portabilité: des projections standardisées
  - Interopérabilité: GIOP / IIOP
  - Plusieurs modèles d'invocation: SII, DII et AMI
  - Middleware: ORB, POA, etc.
- Pas de support standard pour l'empaquetage et le déploiement !!!
- Programmation explicite des propriétés non fonctionnelles!
  - cycle de vie, (de)activation, service de nomage, de courtier, de notification, de persistance, de transactions, ...
- Pas de notion d'architecture logicielle

17

...au modèle de composant de CORBA (CCM)

- Un modèle orienté composant distribué
  - Une architecture pour définir des composants et leurs interactions
  - Une technologie d'empaquetage pour déployer des binaires exécutable multi-langages
  - Un framework à conteneur pour gérer le cycle de vie, (de)activation, sécurité, transactions, persistance et les évènements
  - Interopérabilité avec Enterprise Java Beans (EJB)
- Le premier modèle de composant industriel multi-langages
  - Multi-langages, multi-OSs, multi-ORBs, multi-vendeurs, etc.
  - CORBA 3.0 – Juillet 2002

18

Contenu des spécifications CCM

- Un modèle abstrait de composants
  - Étend l'IDL et le modèle objet
- Framework d'implémentation des composants (CIF)
  - Composant Implementation Definition Language (CIDL)
- Un modèle de programmations des conteneurs de composants
  - Vues de l'implémenteur et du client
  - Intégration avec les services de sécurité, de persistance, de transactions et d'évènements

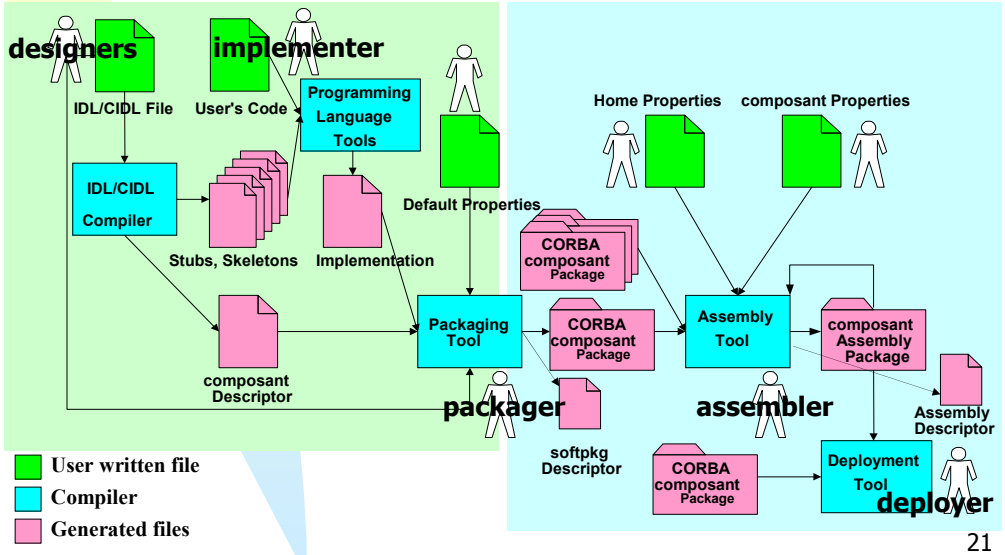
19

Contenu des spécifications (2)

- Des facilités d'empaquetage et de déploiement
  - Fichiers de descriptions en XML
- L'interopérabilité avec EJB via des passerelles
- Des méta-modèles
  - De composants
  - Une interface « Repository » et des extensions au MOF

20

Le grand schéma de CCM



21

Définition de composants CORBA

- Le modèle abstrait des composants
- Les extensions de IDL 3.0

Le modèle abstrait des composants

- Il décrit comment un composant CORBA est vu par les autres composants et par les clients
  - Ce qu'offre un composant aux autres composants (*provide*)
  - Ce qu'il demande des autres composants (*use*)
  - Le modèle de collaboration utilisé entre composant
    - Synchrone via les invocations d'opération
    - Asynchrone via la notification d'évènement
  - Quelles propriétés du composant sont configurables
  - Quelles sont les opération du cycle de vie (*home*)

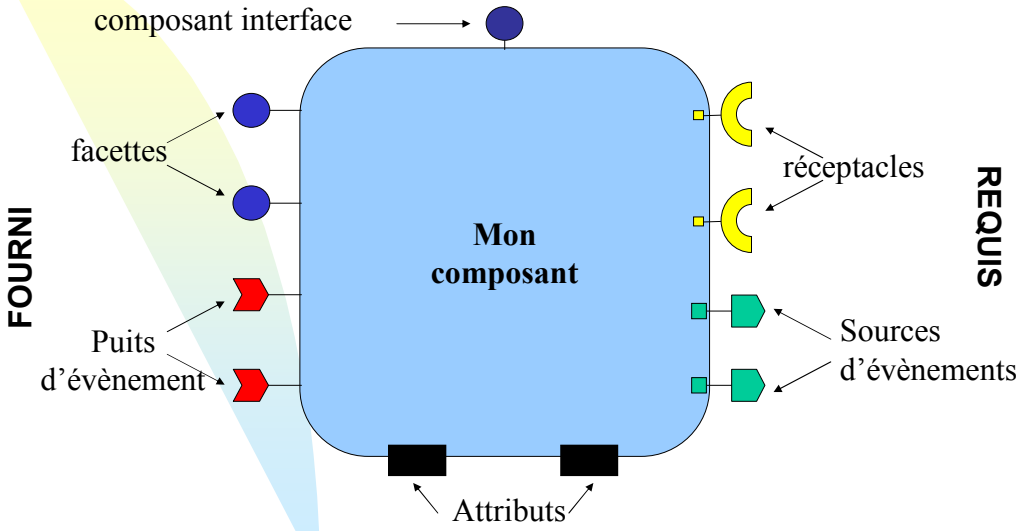
23

Ce qu'est un composant CORBA

- Un composant est un nouveau meta-type CORBA
  - Extension d'un objet (avec quelques contraintes)
  - Possède une interface et une référence d'objet
  - Une utilisation stylisée des interfaces/objets CORBA
- Fourni des caractéristiques de composant, nommées ports
  - 5 types de port
  - *Attributs* : propriétés configurables
  - *Facettes* : interfaces d'opérations fournies
  - *Réceptacles* : interfaces d'opérations requises
  - *Sources d'évènement* : produit des évènements
  - *Puits d'évènement* : consomme des évènements

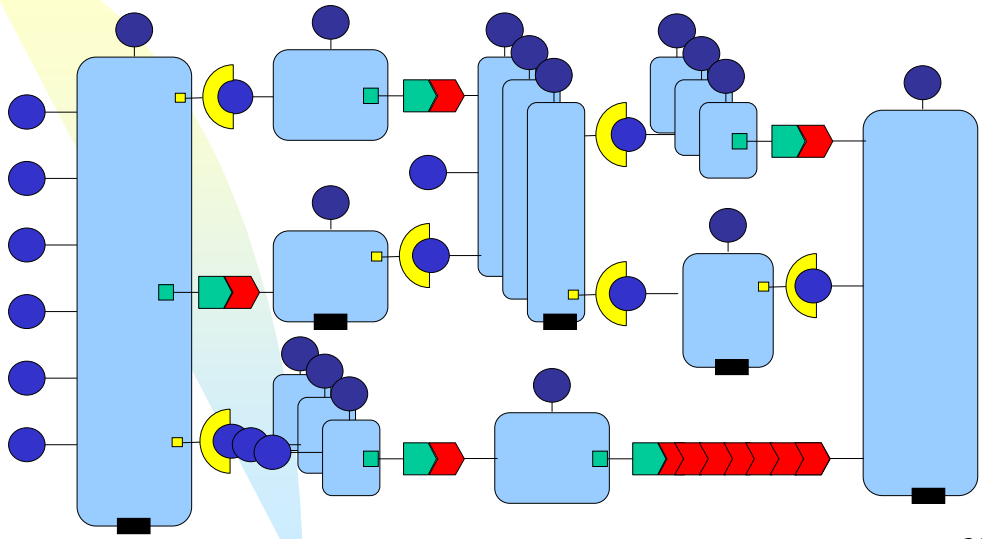
24

Un composant CORBA



25

Construction d'une application CCM = Assemblage d'instances de composants



26

Définition d'un composant

- Composant simple

```
component nom_composant { ...};
```

- Héritage simple de composant

```
component nom_composant : composant_pere { ...};
```

- Peut supporter plusieurs interfaces (*supports*)

```
component nom_composant supports interface1, interface2
{ ...};
```

27

Les attributs

```
component nom_composant {
  attribute A;
  readonly attribute B;
};
```

- Propriétés configurables nommées
  - Clé vitale pour une réutilisation effective
  - Vise la configuration d'un composant
    - p.ex., comportement optionnel, modalité, etc.
  - Peut lever des exceptions
  - Exposé via des accès en lecture / écriture
- Peuvent être configurés
  - Par des mécanismes visuels dans des environnement d'assemblage et/ou de déploiement
  - Par les maisons ou par les implémentations durant l'initialisation
  - Potentiellement non modifiable par la suite

28

Les facettes

```
component nom_composant {
  provides type nom_port;
  ...
};
```

- Interfaces distinctes nommées qui fournit les fonctionnalités du composants aux clients
- Chaque facette correspond à une vue du composant, c'est à dire à un rôle
- Une facette représente le composant lui-même, pas une chose séparée et contenue dans le composant
- Les facettes ont des références d'objet indépendantes

29

Les réceptacles

```
component nom_composant {
  uses type portA;
  uses multiple type portB;
  ...
};
```

- Points distincts et nommées de connexion pour une connectivité potentielle
  - Possibilité de spécialiser par délégation ou de composer des fonctions
  - ~ la base d'un Lego !
- Stocke une référence d'objet simple ou multiple
  - Mais n'est pas destiné à un être un service de mise en relation
- Configuration
  - Statiquement durant l'initialisation ou l'assemblage
  - Dynamiquement à l'exécution

30

Les évènements

- Modèle d'évènement publication / souscription
  - Modèle "push" seulement
  - Sources (2 types) et puits
- Les évènements sont des « value types »
  - Défini par le nouveau meta-type **`eventtype`**
  - Spécialisation de `valuetype` pour les composants

```
eventtype nom_evenement {
 public string A;
 ...
}
```

31

Les sources d'évènements

- Points de connexion nommés pour la production d'évènement
  - «Pousse» un « eventtype » spécifié
- Deux types: *publieur* & *émetteur*
  - `publishes` = plusieurs puits peuvent souscrire
  - `emits` = seulement un puit connecté
- Un puit souscrit directement à une source d'évènement
- Le conteneur gère l'accès aux canaux de notification
  - extensibilité, qualité de service, transactionnel, etc.

```
component nom_composant {
  publishes etype portE;
  emits etype portF;
  ...
};
```

32

Les puits d'évènement

- Points de connexion nommés dans lesquels des évènements de type spécifié peuvent être « poussés »

```
component nom_composant {
  consumes etype portE;
  ...
};
```

- Souscription aux sources d'évènements
  - Potentiellement plusieurs (n vers 1)
- Pas de distinction entre émetteur et publieur
  - Les deux « poussent » dans un puit d'évènement

33

Maisons de composant CORBA

- Chaque instance de composant instance est créée et gérée par une *maison* unique de composant (`home`)
- Gère un unique type de composant
  - Plus d'une maison peuvent gérer un même type de composant
  - Mais une instance de composant n'est gérée que par une seule instance de maison
- Nouveau meta-type CORBA : home
  - Définition d'une maison est distincte de celle d'un composant
  - Une maison a une interface et une référence d'objet
- Est instancié durant le déploiement

```
home maison manages nom_composant {... };
```

34

Maisons de composant CORBA

- Définition d'une maison d'un composant

```
home maison manages composant {... };
```

- Peut hérité simplement d'un autre type de maison

```
home maison2 : maison manages composant2
{... };
```

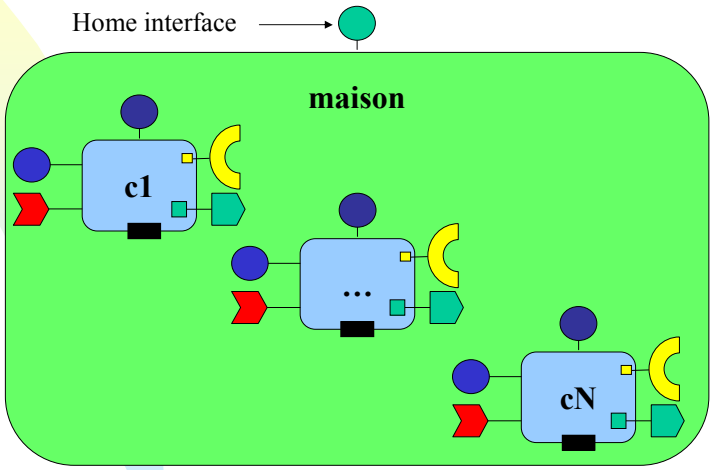
  - Contrainte sur le type du composant
    - identique ou maison2 doit hérité directement ou non de maison

- Peut supporter de multiples interfaces (*supports*)

```
home maison supports itf1, itf2 manages composant {... };
```

35

Une maison de composant CORBA



36

Caractéristiques d'une maison de composant

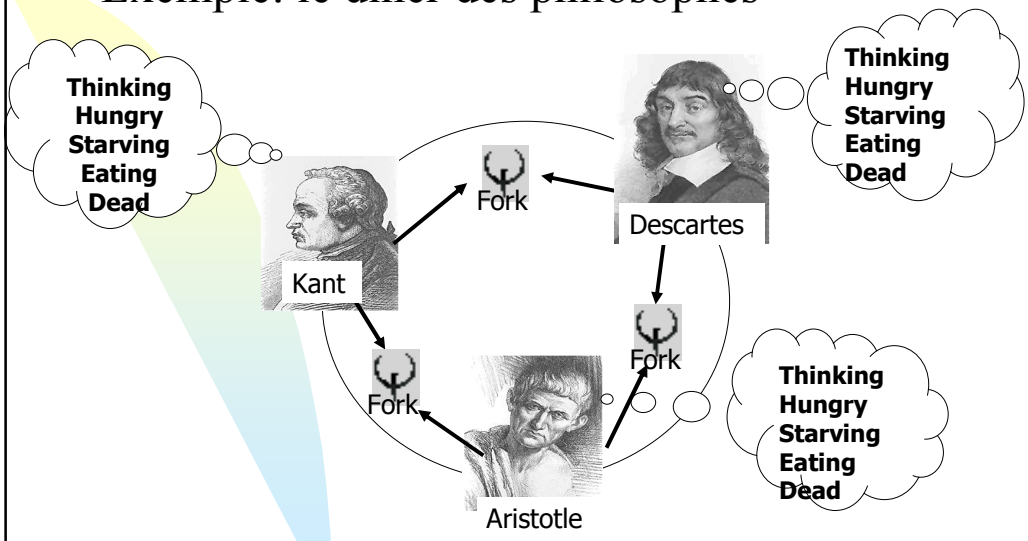
- Permet l'évolution des caractéristiques du cycle de vie ou du type de clé sans changer la définition d'un composant
- Clé primaire optionnelle (*primarykey*) comme identité d'un composant ou comme clé primaire de persistance
- Fournit des opérations standards de création (*factory*) et de recherche (*finder*)
- Extensible par des opérations de niveau utilisateur

```
home maison manages nom_composant primarykey keytype {
  factory creation(in string name) raises E;
  finder  cherche(in string name) raises F;
  ...
};
```

37

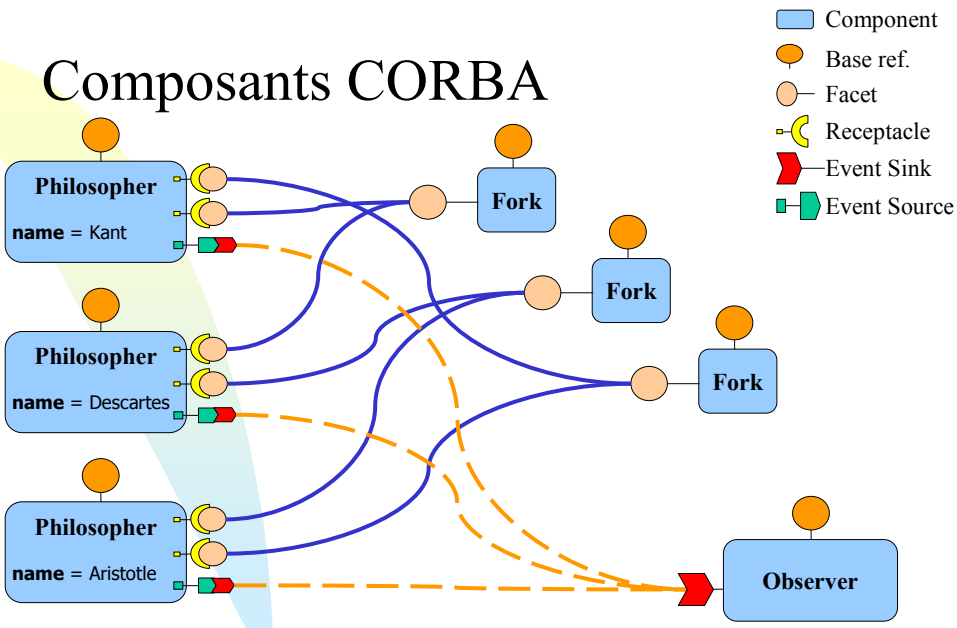
Exemple du dîner des philosophes

Exemple: le dîner des philosophes



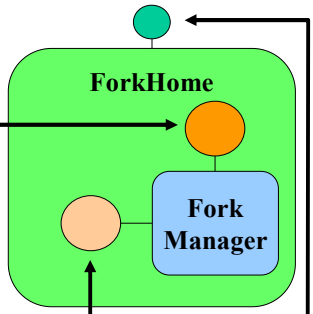
39

Composants CORBA



40

Maison du composant




```
exception InUse {};

interface Fork {
  void get() raises (InUse);
  void release();
};

// Le composant
component ForkManager {
  // Facette utilisé par les phisolphes.
  provides Fork the_fork;
};

// Maison pour instancier les composants ForkManager
home ForkHome manages ForkManager {};
```

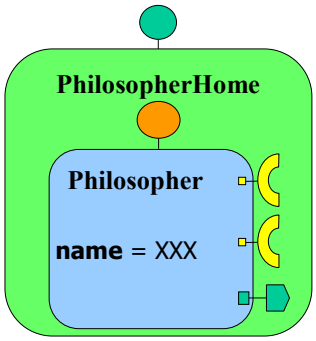
41

Les types d'état d'un philosophe

```
enum PhilosopherState
{
   EATING, THINKING, HUNGRY,
   STARVING, DEAD
};

eventtype StatusInfo
{
   public string name;
   public PhilosopherState state;
   public unsigned long ticks_since_last_meal;
   public boolean has_left_fork;
   public boolean has_right_fork;
};
```

42

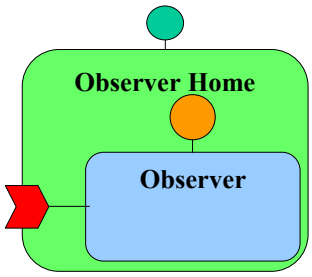
Le composant philosophe




```
component Philosopher {
   attribute string name;

   // The left fork receptacle.
   uses Fork left;

   // The right fork receptacle.
   uses Fork right;

   // The status info event source.
   publishes StatusInfo info;
};

home PhilosopherHome manages Philosopher {
    factory new(in string name);
};
```

43

Le composant observateur




```
component Observer
{
  // The status info sink port.
  consumes StatusInfo info;
};

// Home for instantiating observers.
home ObserverHome manages Observer {};
```

44

Programmation des composants CORBA coté

- Le modèle de programmation coté client
- Projection coté client de l'IDL

Le modèle de programmation coté client

- Clients avertis ou non de la notion de composant
- Clients voient deux patrons de conception
  - Fabrique : recherche d'une maison et utilisation pour créer une instance d'un composant
  - Recherche : recherche d'un composant existant via les services de nomage, de courtage ou les opérations de recherche des maisons
- Support des transactions et de mécanisme de sécurité
- Invocation d'opérations sur les instances de composant
  - Celles définies par la projection coté client

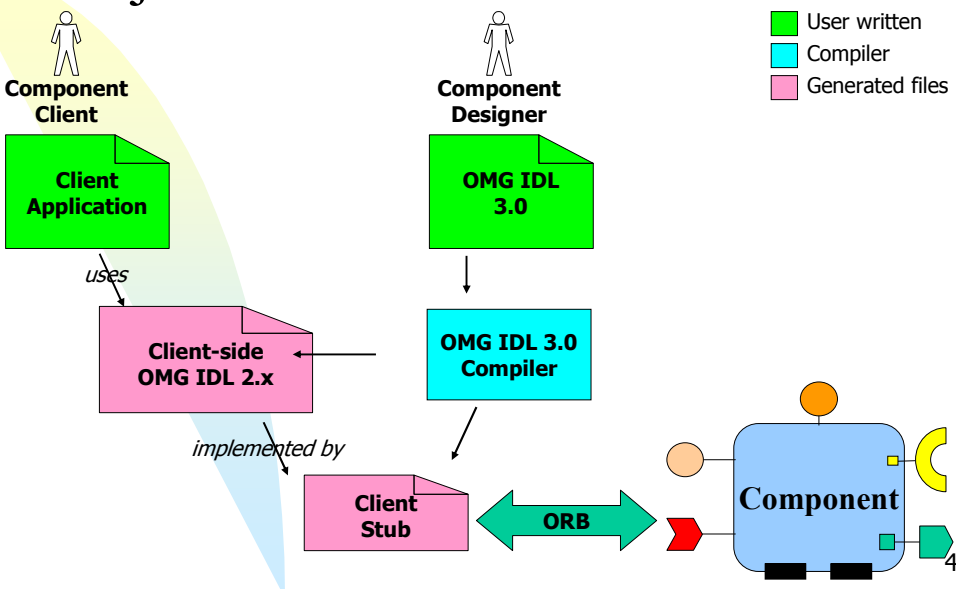
46

Projection coté client de l'IDL

- Chaque construction IDL 3.0 à un équivalent en terme d'IDL 2
- Le composant et sa maison sont vus coté-client via les projections coté client
- Permet de ne pas changer le langage de projection coté client
  - Les clients peuvent toujours utiliser leurs outils favoris
- Les clients NE sont PAS obligés d'être "component-aware"
  - Ils invoquent juste des opérations sur des interfaces

47

Projection IDL coté client



48

Projection des composants

- Exprimé via des extensions à l' IDL 3.0
  - Construction syntaxique pour des patrons de conceptions bien connus
  - Projeté sur les interfaces IDL pour les clients et les implémenteurs
- Exemple

```
component nom_composant { ... };
```

↓ projeté en

```
interface nom_composant : Components::CCMObject { ... };
```

49

Projection des opérations

- Facettes

```
provides type portA;
```

↓ projeté en

```
type provide_portA ();
```

- Réceptacles

```
uses type portB;
```

↓ projeté en

```
void connect_portB ( in type conxn )
  raises ( Components::AlreadyConnected, Components::InvalidConnection );

type disconnect_portB ( ) raises ( Components::NoConnection );

type get_connection_portB ( );
```

Projection des sources d'évènements (1)

```
component composant {
  emits etype source;
};
```

↓ projeté en

```
module composantEventConsumers {
   interface etypeConsumer: Components::EventConsumerBase {
      void push (in etype evt);
   };
};

interface composant : Components::CCMObject {

   void connect_source (
            in composantEventConsumers::etypeConsumer consumer )
            raises (Components::AlreadyConnected);

   composantEventConsumers::etypeConsumer disconnect_source()
            raises (Components::NoConnection);
};
```

51

Projection des sources d'évènements (2)

```
component composant {
  publishes etype source;
};
```

↓ projeté en

```
module composantEventConsumers {
   interface etypeConsumer: Components::EventConsumerBase {
      void push (in etype evt);
   };
};

interface composant : Components::CCMObject {

   Components::Cookie subscribe_source (
        in composantEventConsumers::etypeConsumer consumer)
        raises (Components::ExceededConnectionLimit);

   composantEventConsumers::etypeConsumer unsubscribe_source (
        in Components::Cookie ck)
        raises (Components::InvalidConnection);
};
```

52

Projection des puits d'évènements

```
component composant {
  consumes etype puit;
};
```

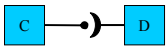
projeté en

```
module composantEventConsumers {
   interface etypeConsumer : Components::EventConsumerBase {
      void push (in etype evt);
   };
};

interface composant : Components::CCMObject {
   composantEventConsumers::etypeConsumer get_consumer_puit();
};
```


Connexion de composants

- Facette/réceptacle

```
type_var a = C->provide_portA ();
D->connect_portB(a);
```



- Source/puit d'évènement

```
etypeConsumer_var p = E->get_consumer_puit();
F->connect_source(p);
```


```
etypeConsumer_var p = G->get_consumer_puit();
H->subscribe_source(p);
```

Projection des maisons

```
home maison manages composant {
   factory fabrique(...) raises (...);
   finder  cherche (...) raises (...);
};
```

projeté en

```
interface maisonExplicit : Components::CCMHome {
   composant fabrique ( ... ) raises (Components::CreateFailure, ... );
   composant cherche  ( ... ) raises (Components::FinderFailure, ... );
};

interface maisonImplicit : Components::KeylessCCMHome {
   composant create() raises(CreateFailure);
};

interface maison : maisonExplicit, maisonImplicit { };
```

55

Introduction à CIDL

- Description du problème
- CIDL
- Exemple

Quel est le problème ?

- L'interface abstraite du composant définie en IDL3
- Question: comment on l'implémente ?

57

Implémentation de composants - 1

- Première solution : implémentation dépendant
- Exemple : les webservices ☺
- Avantage
  - Très grande liberté d'implémentation
  - Contrôle des dépendances (bibliothèques)
- Inconvénient
  - Pas portabilité du code écrit
    - Quid de la génération de code ?
  - Pas de gestion des services systèmes

58

Implémentation de composants - 2

- Deuxième solution : définir un environnement d'implémentation
- Exemple : CCM
- Avantage
  - Portabilité du code écrit
  - Gestion automatique de service système
  - Gain de temps
- Inconvénient
  - Il faut maîtriser le modèle
  - Liberté d'implémentation *a priori* restreinte
    - Non restreinte car CIDL non obligatoire

59

Implémentation de composants CCM

- Deux modèles de CCM sont principalement utilisés
- Framework d'implémentation des composants (CIF)
  - Composant Implementation Definition Language (CIDL)
- Un modèle de programmations des conteneurs de composants
  - Vues de l'implémenteur et du client
  - Intégration avec les services de sécurité, de persistance, de transactions et d'évènements

60

Component Implementation Definition Language (CIDL)

- Décrit une composition de composant
  - Entité agrégé qui décrit tous les artéfacts requis pour implémenté un composant et sa maison
- Gère la persistance d'un composant
  - Basé sur le OMG Persistent State Definition Language (PSDL)
  - Liens entre des types de stockage et les exécuteurs segmentés
- Génère des squelettes d'exécuteur fournissant
  - La segmentation des exécuteurs des composants
  - Une implémentation par défaut des opérations de callback
  - La persistance de l'état d'un composant

61

Notion de Composition CCM

- Entité agrégée décrivant les artéfacts requis pour implémenter un composant et sa maison
  - Nom de la composition
  - Catégories de cycle de vie des composants
    - Service, session, process, entity
  - Un type de maison de composant (1)
    - Le type du composant à implémenter est défini implicitement
  - Une définition d'exécuteur de la maison (2)
  - Une définition d'exécuteur du composant (3)
  - *Une définition de délégation*
  - *Une définition pour un proxy de la maison*
  - *Une association à un espace de stockage abstrait*

```
composition <categorie> nom_composition {
   home executor nom_executeur_maison {          // (2)
      implements nom_type_maison;                // (1)
      manages nom_executeur;                     // (3)
   }
};
```

62

Notion d'exécuteur

- Artéfact de programmation implémentant une abstraction.
- Dans les langages objets, un exécuteur = un objet

63

Notion d'exécuteur de maison

- Le contenu de la définition d'un exécuteur de maison définit les relations entre
  - la maison d'exécuteur
  - et les autres éléments de la composition
- Il détermine également des caractéristiques des squelettes générés.

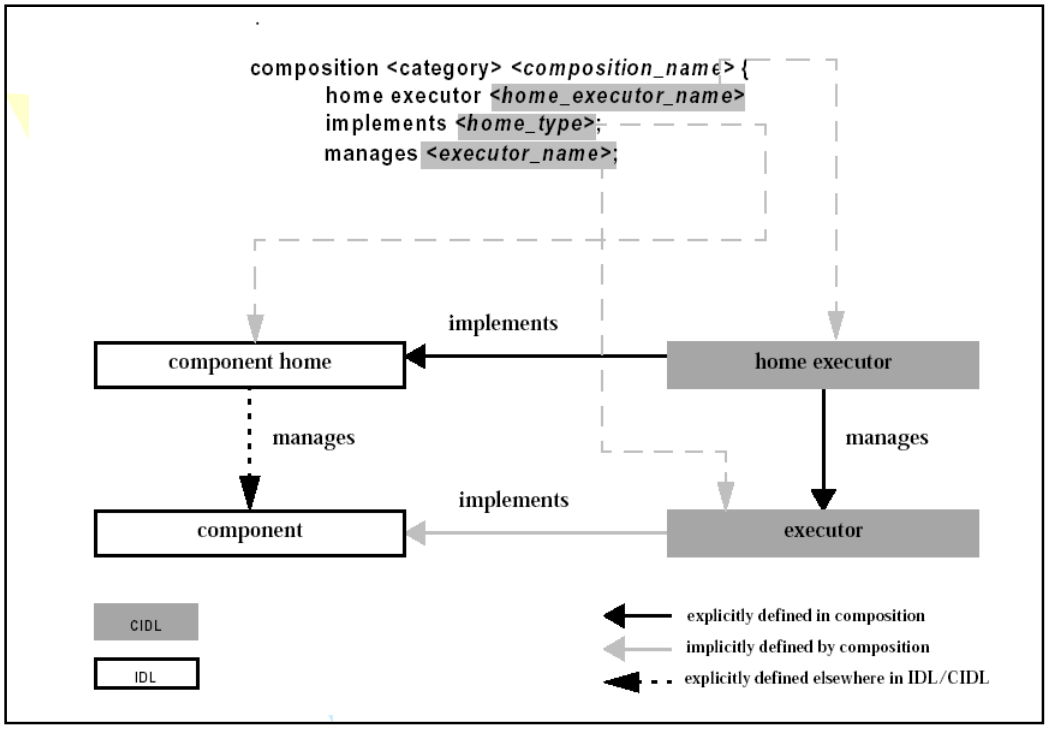
```
composition <categorie> nom_composition {
   home executor nom_executeur_maison {
      implements nom_type_maison ;
      manages nom_executeur;
   }
};
```

64

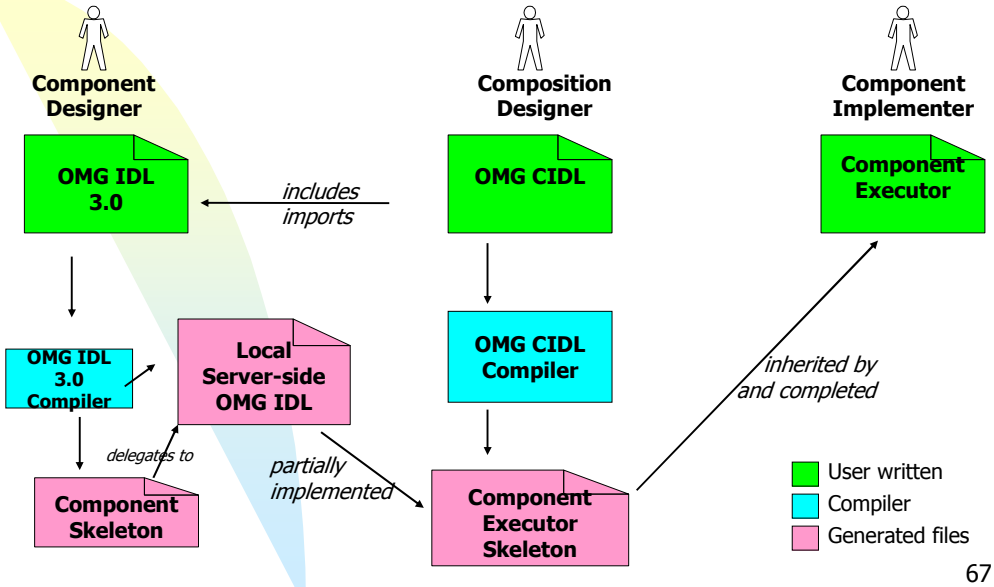
Notion d'exécuteur de composant

- Le contenu de la définition d'un exécuteur de composant contient optionnellement
  - des segments d'exécuteur
  - une projection (délégation) d'un certain nombre de caractéristiques (par exemple les attributs) à des membres de stockage
- Segments d'exécuteur
  - Partition physique de l'exécuteur de composant
  - Encapsule indépendamment son état
  - Peuvent être indépendamment activé
- On se restreindra au cas d'exécuteur monolithique

65

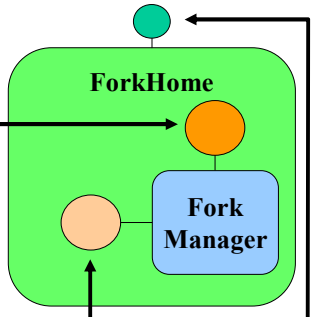


OMG CIDL Compilation Process



67

Maison du composant




```
exception InUse {};

interface Fork {
  void get() raises (InUse);
  void release();
};

// Le composant
component ForkManager {
  // Facette utilisé par les phisolphes.
  provides Fork the_fork;
};

// Maison pour instancier les composants ForkManager
home ForkHome manages ForkManager {};
```

68

CIDL Composition for Observer Component

```
#include <philo.idl>
// or import DiningPhilosophers;

composition session ForkManagerSessionComposition
{
  home executor ForHomeImpl
  {
    implements DiningPhilosophers::ForkHome;
    manages ForkManagerImpl;
  };
};
```

69

Implémentation de la maison

```
package DiningPhilosophers.monolithic;
import DiningPhilosophers.*;
public class ForkHomeImpl extends org.omg.CORBA.LocalObject
                          implements CCM_ForkHome {
  public  ForkHomeImpl(){}
  public org.omg.Components.EnterpriseComponent create()
  {
      return new ForkManagerImpl();
  }
  public static org.omg.Components.HomeExecutorBase create_home()
  {
      return new ForkHomeImpl();
  }
}
```

70

Implémentation du composant

```
package DiningPhilosophers.monolithic;
import DiningPhilosophers.*;
public class ForkManagerImpl extends org.omg.CORBA.LocalObject
                        implements CCM_ForkManager, CCM_Fork,
                                   org.omg.Components.SessionComponent
{
  private boolean available_;
  public CCM_Fork get_the_fork() { // From CCM_ForkManager
       return this; // Returns an implementation of the Fork facet
  }
  public void get() throws InUse { // From CCM_Fork
       if (! available_) throw new InUse();
       available_ = false;
  }
  public void release() {          // From CCM_Fork
       if (available_) return;
       available_ = true;
  }
}
```

71