
# CCM : CORBA Component Model

Alexandre Denis – Alexandre.Denis@inria.fr

Inria Bordeaux – Sud-Ouest
France

ENSEIRB PG306

# La composition

- La notion de composant est très largement répandue...
  - Composants électroniques
  - Composants d'une voiture
  - Composants d'un meuble
- C'est un ancien rêve de l'informatique
  - Les premiers travaux datent des années 60
- Rôle principal d'un composant : être **composé** !
  - Les composants prônent :
    - la préfabrication
    - la réutilisation

# Définition par les propriétés

- Un composant est défini par ses propriétés:
  - 1. c'est une unité de déploiement indépendant
  - 2. c'est une unité de composition par une tierce entité
  - 3. un composant n'a pas d'état persistant
- Remarques
  - -> bien séparé par rapport à l'environnement
  - -> pas de déploiement partiel
  - -> pas d'accès aux détails
  - -> notion de copie non définie
    - Soit le composant est disponible soit il ne l'est pas

# 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**

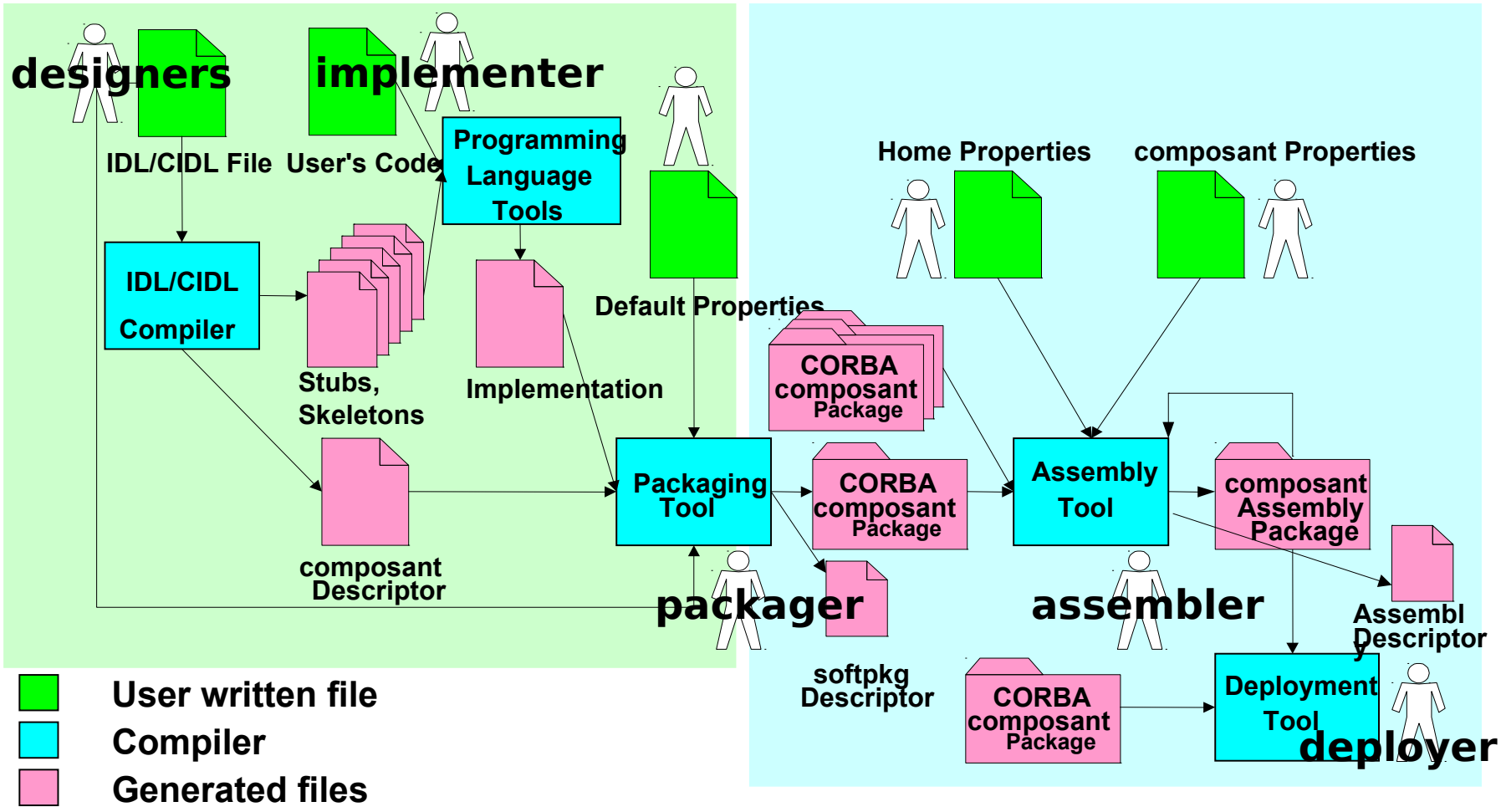
# ...au modèle de composants CORBA 3

- 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

# 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
- Des facilités d'empaquetage et de déploiement
  - Fichiers de descriptions en XML
- L'interopérabilité avec EJB via des passerelles
- Des méta-modèles

# CCM big picture



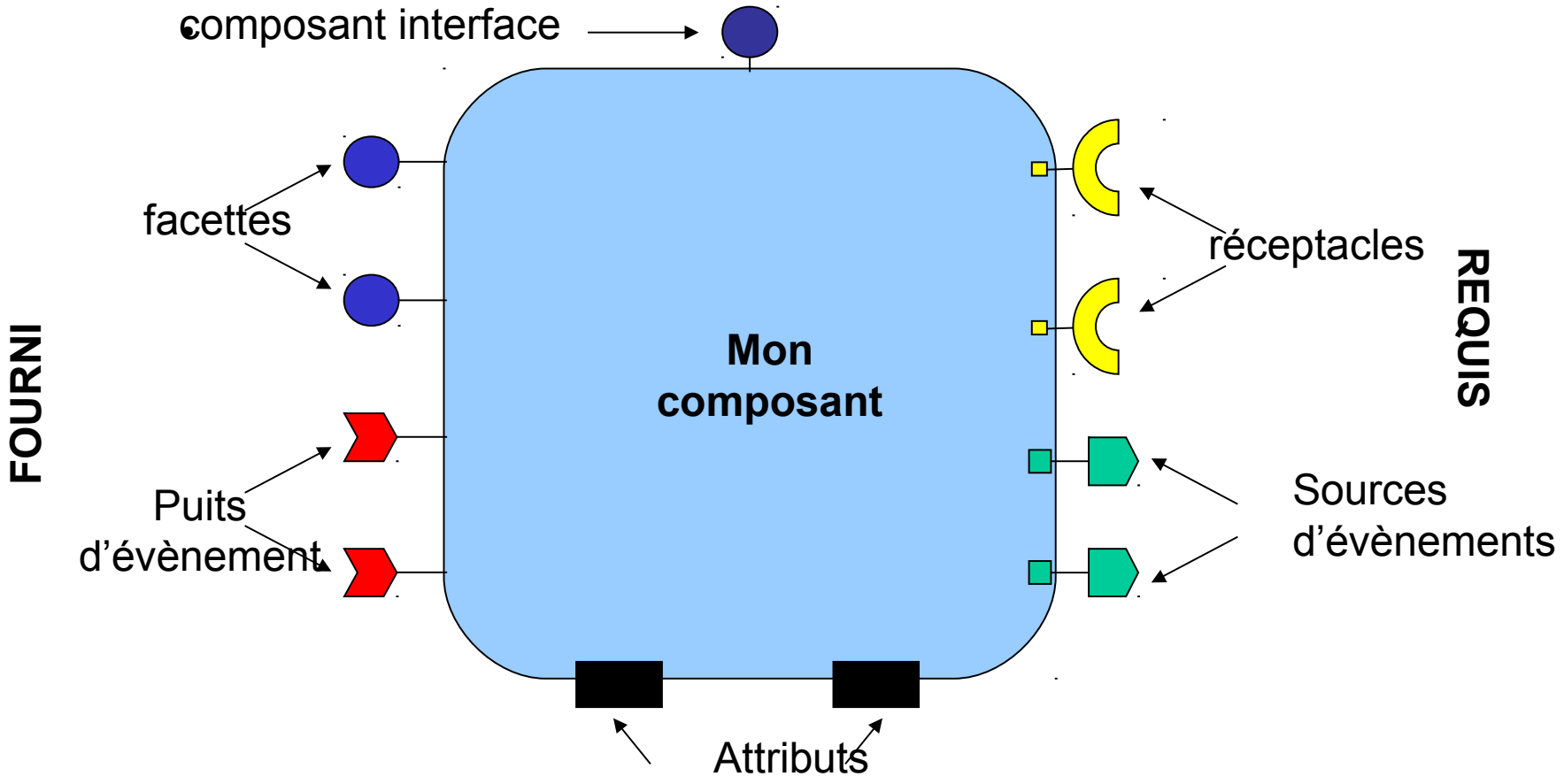
# Modèle abstrait des composants

- 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 opérations du cycle de vie (*home*)

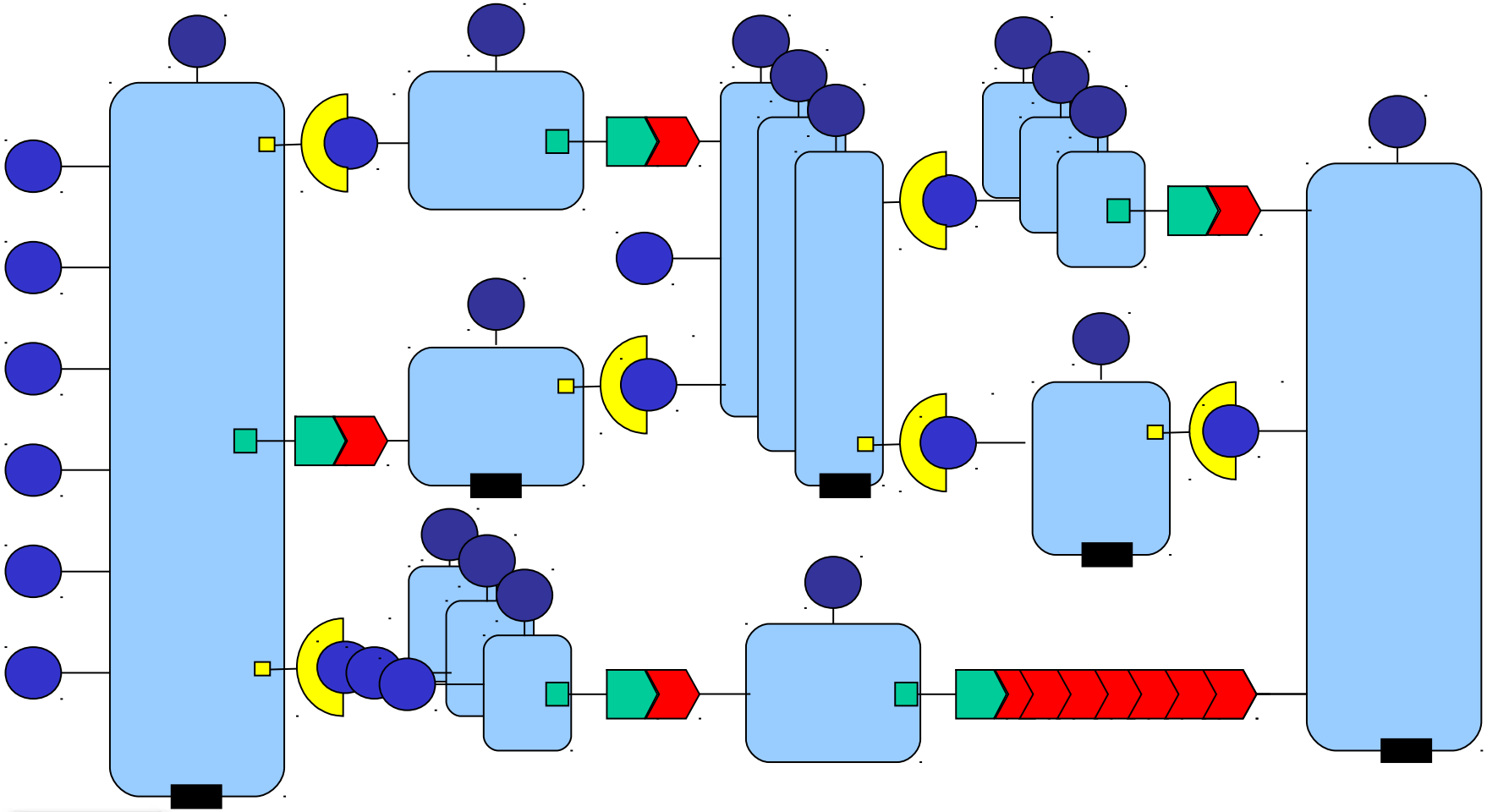
# Un composant CORBA

- 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
- Fournit 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

# Un composant CORBA



# Construction d'une application = assemblage



# 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
{ ...};
```

# 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
    - ex.: comportement optionnel, modalité, etc.
  - Peut lever des exceptions
  - Exposées 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

# Les facettes

- Interfaces distinctes nommées qui fournissent les fonctionnalités du composant 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

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

# Les réceptacles

- 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

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

# Les évènements

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

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

# Les sources d'évènements

- Points de connexion nommés pour la production d'évènement
  - Pousse un `eventtype` spécifié
- Deux types: publieur & émetteur
  - `publishes` = plusieurs clients peuvent souscrire
  - `emits` = seulement un client connecté
- Un client 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;
  ...
};
```

# Les puits d'évènements

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

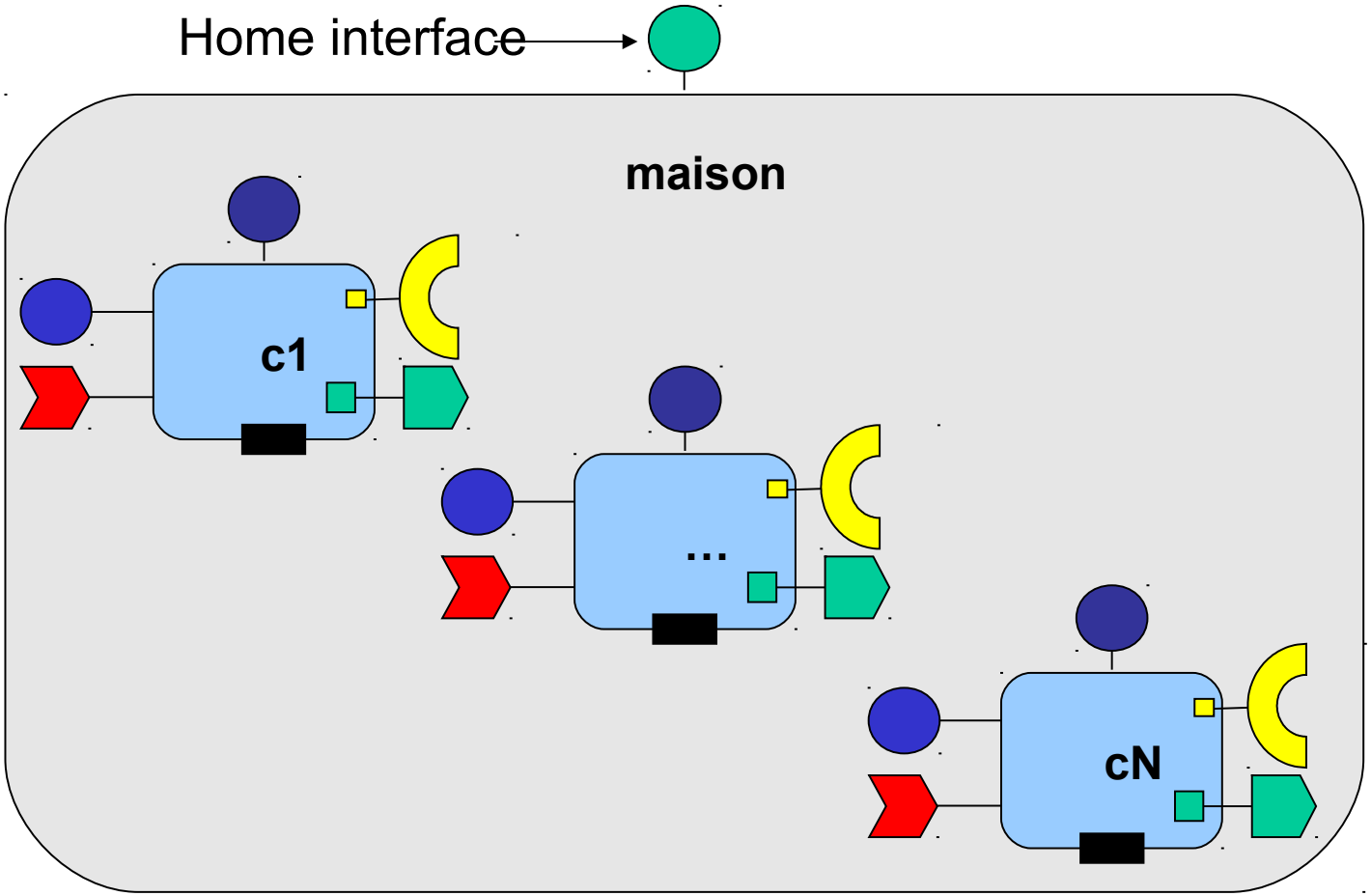
```
component nom_composant
{
  consumes etype portE;
  ...
};
```

# Maison de composant CORBA

- Chaque instance de composant 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 instanciée durant le déploiement

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

# Maison de composant CORBA

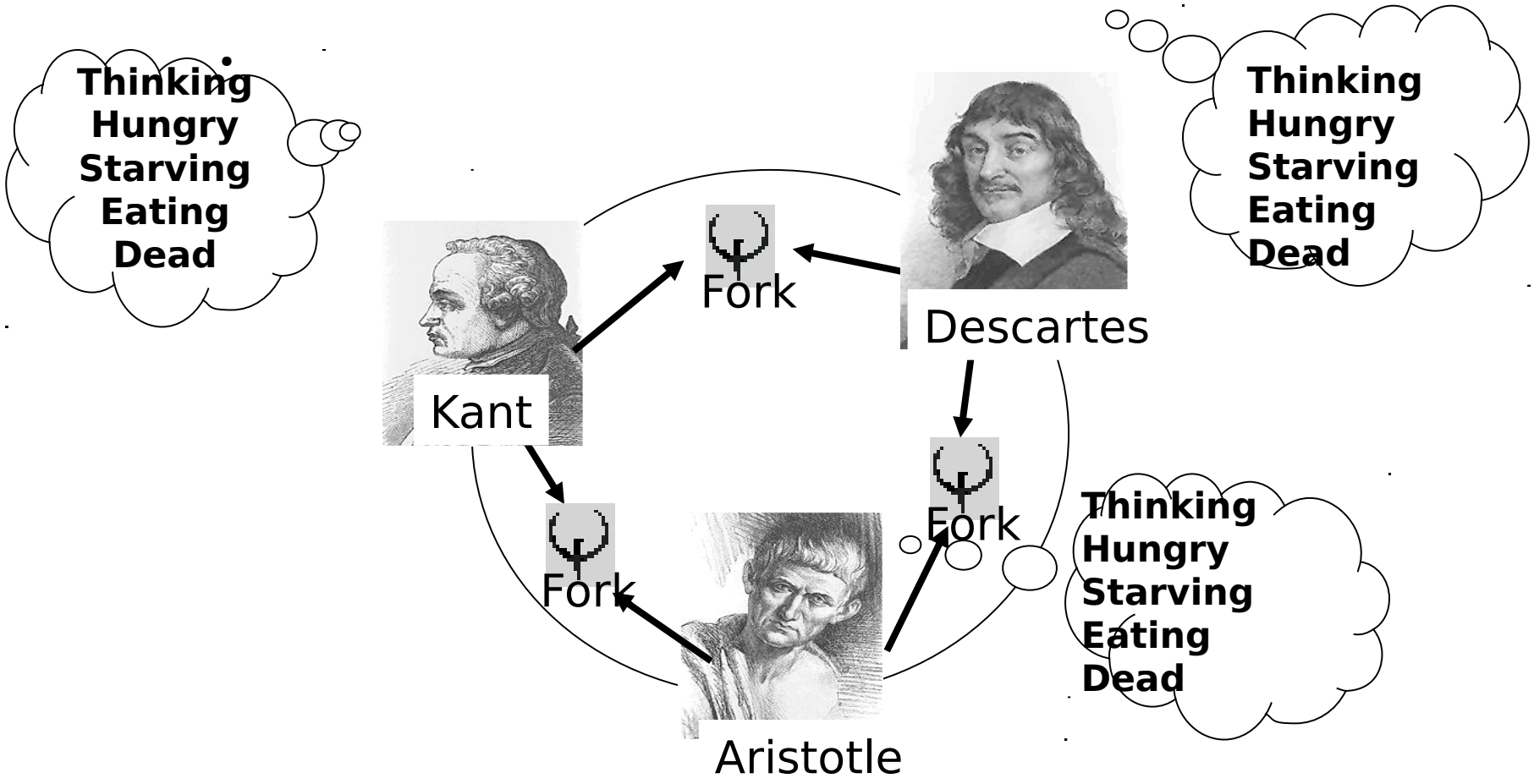


# Propriétés des maisons

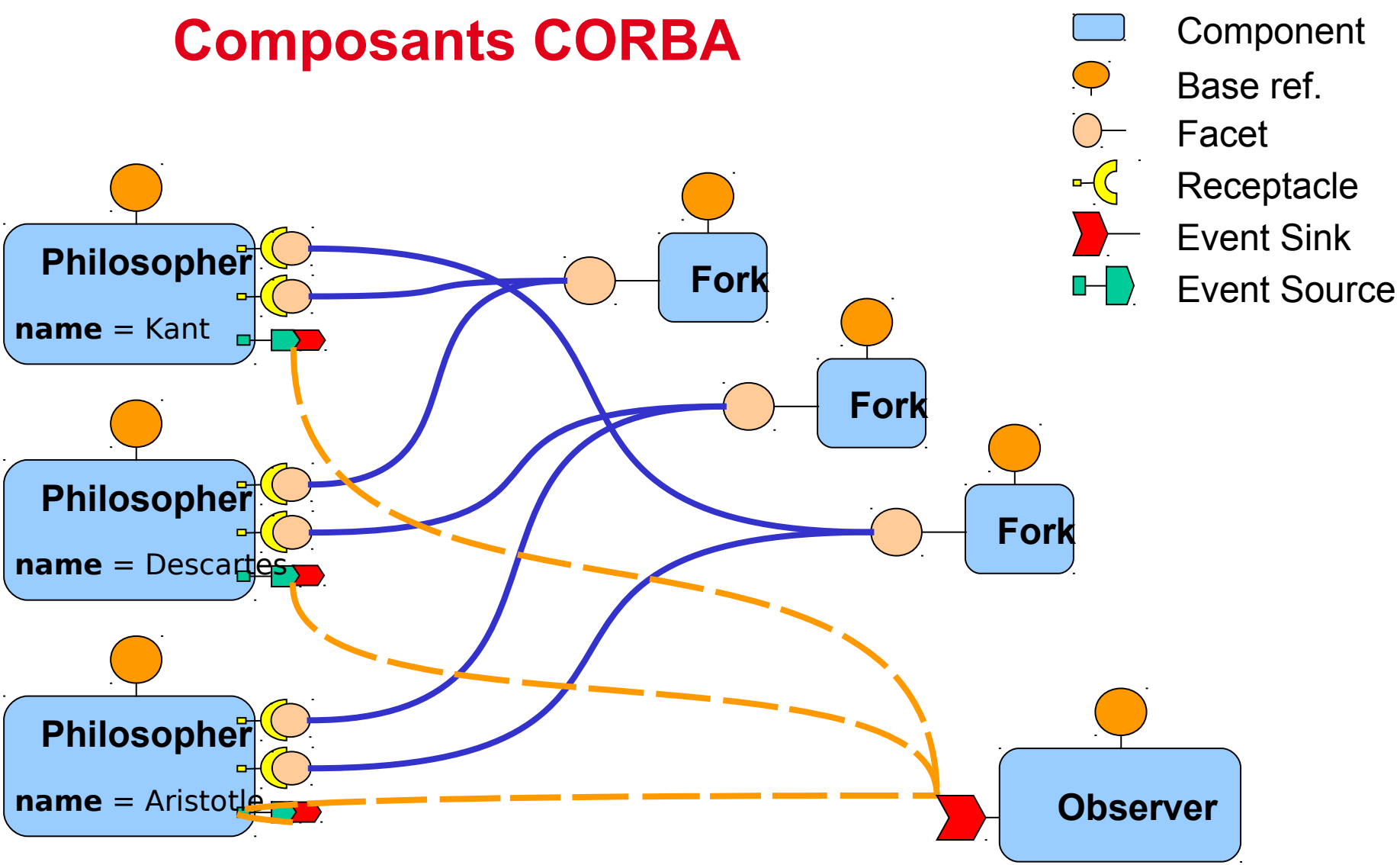
- Permet l'évolution des caractéristiques du cycle de vie ou du type de clé sans changer la définition d'un composant
- Clé primaire optionnelle (`primarykey`) comme identité d'un composant ou comme clé primaire de persistance
- Fournit des opérations standards de création (`factory`) et de recherche (`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;
  ...
};
```

# Exemple : le dîner des philosophes

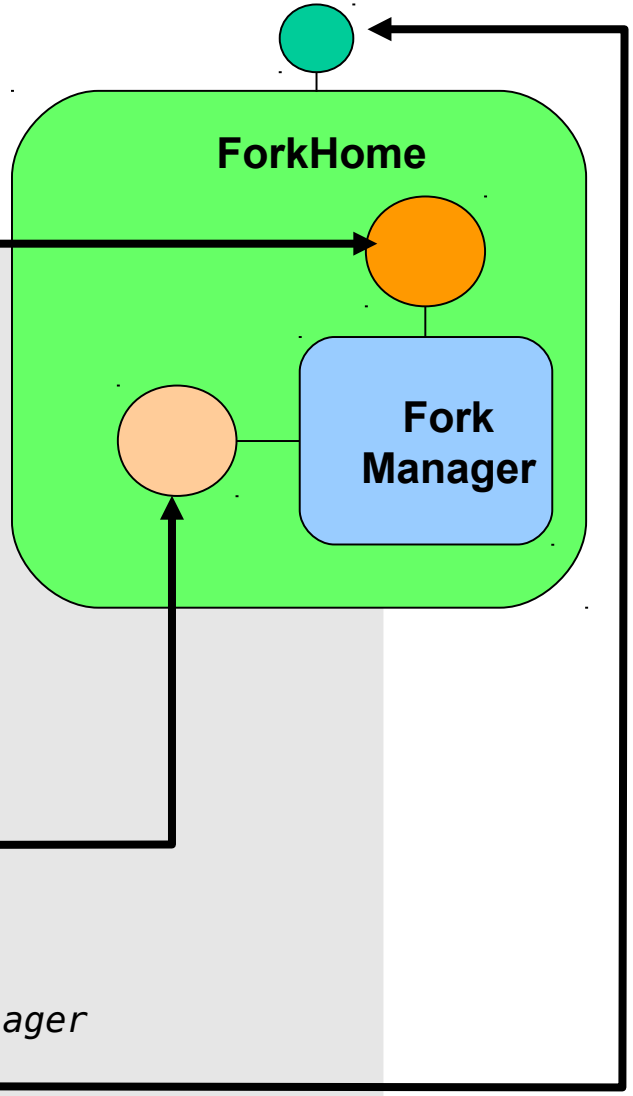


# Composants CORBA



# Maison du composant

```
exception InUse {};

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

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

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

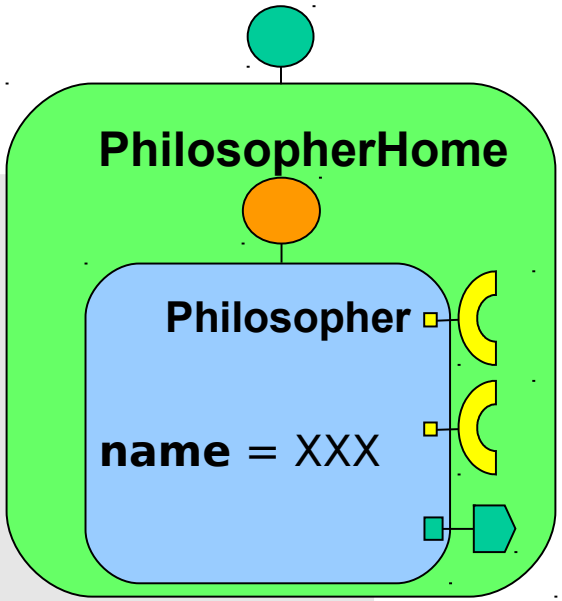


# État d'un philosophe

```
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;
};
```

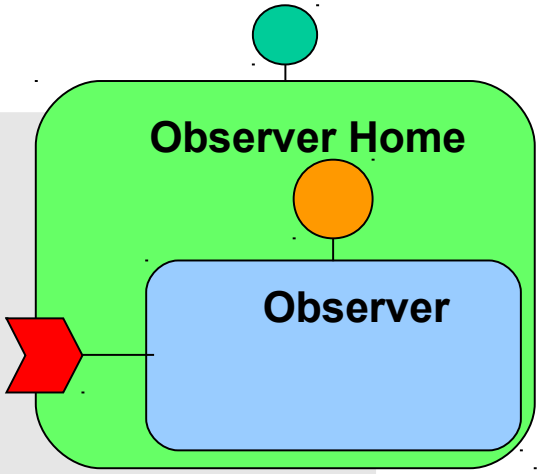
# Le composant philosophe

```
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);
};
```



# Le composant observateur

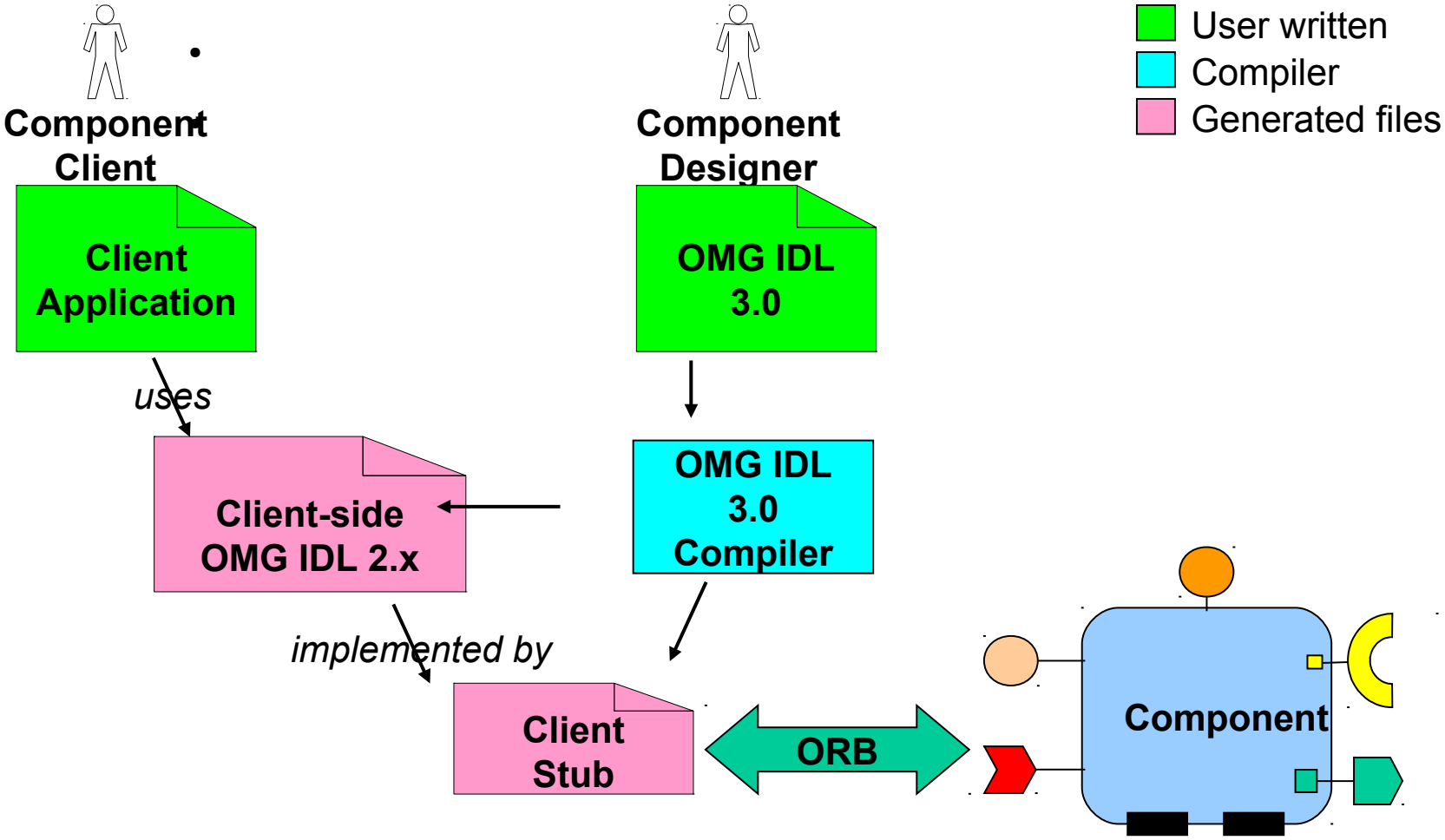
```
component Observer
{
  // The status info sink port.
  consumes StatusInfo info;
};

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



# Projection côté client

- 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
- Chaque construction IDL 3.0 à un équivalent en terme d'IDL 2
- Le composant et sa maison sont vus côté client via les projections
- 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

# Projection IDL côté client



# 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 { ... };
```

# Projection des opérations

- Facettes

```
provides type portA;
```

projeté en

```
type provide_portA ();
```

- Réceptacles

```
uses type portB;
```

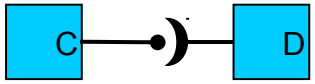
projeté en

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

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

type get_connection_portB ( );
```

# Connexion de composants

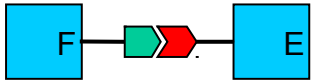
- Facette/réceptacle

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

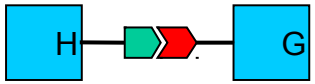



- Source/puit d'évènement

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




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



# CIDL – Component Implementation Definition Language

- Décrit une composition de composant
  - Entité agrégé qui décrit tous les artefacts requis pour implémenter 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
-

# Composition CCM

- Entité agrégée décrivant les artefacts 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

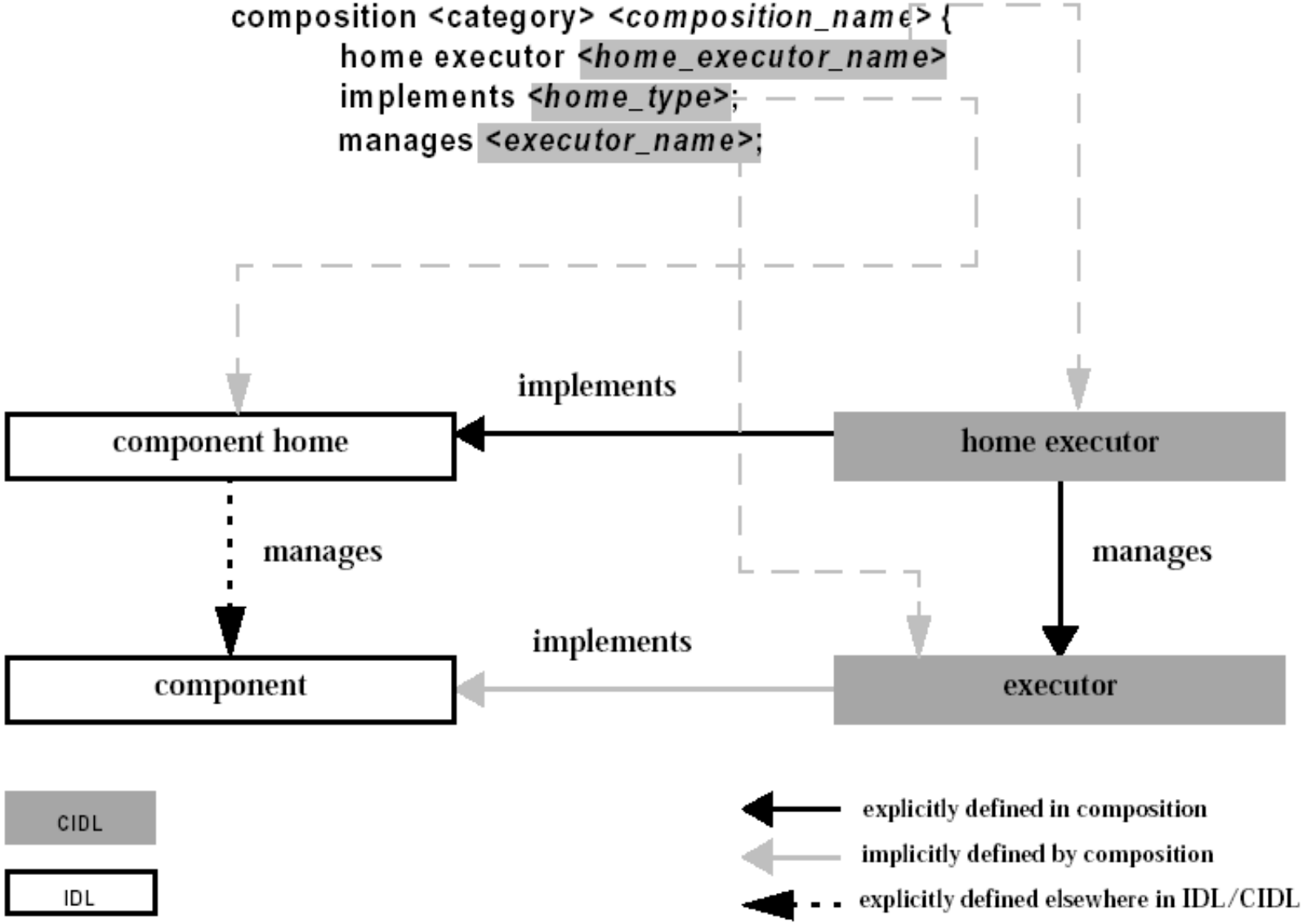
```
composition <categorie> nom_composition {
   home executor nom_executeur_maison {          // (2)
      implements nom_type_maison;                // (1)
      manages nom_executeur;                     // (3)
   }
};
```

# Exécuteur

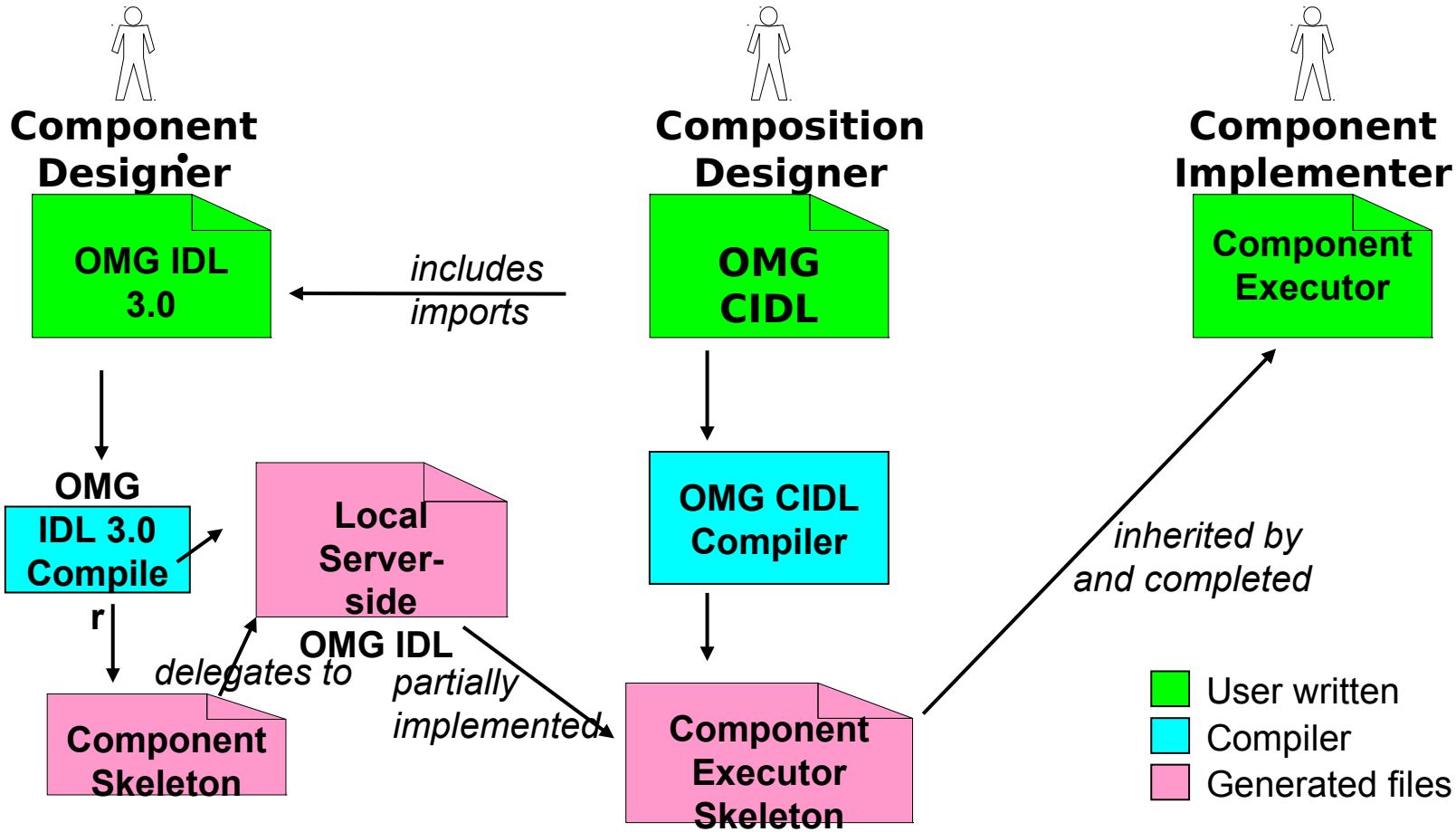
- Artefact de programmation implémentant une abstraction.
- Dans les langages objets, un exécuteur = un objet
- Exécuteur de maison, exécuteur de composant

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

```
composition <category> <composition_name> {
        home executor <home_executor_name>
        implements <home_type>;
        manages <executor_name>;
```



# Compilation CIDL



# CIDL pour l'observateur

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

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

# 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();
  }
}
```

# 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;
  }
}
```


http://dept-info.labri.fr/~denis/