Projet RNRT 98

Configuration et Exécution de Services
pour les Usagers mobiles des Réseaux Etendus

D3

Vers un modèle de composants pour CESURE

Le CORBA Component Model

Raphaël MARVIE & Philippe MERLE
{marvie,merle}@lifl.fr

Novembre 2000

Table des matières

1 Introduction
Vers un modèle de composants pour CESURE **1**

2 Vision d'ensemble des composants CORBA **3**
2.1 Quels types d'applications ? Quels concepts ? 3
2.2 Le modèle de composants CORBA 4
2.3 De la production au déploiement des applications 5

3 Le modèle abstrait de composants **7**
3.1 Introduction 7
3.2 Type de composant 7
3.2.1 Modèle 7
3.2.2 Définition en IDL3 8
3.2.3 Projection en IDL2 9
3.2.4 L'interface `Components::CCMObject` 9
3.2.5 Critique 10
3.3 Facettes 10
3.3.1 Modèle 10
3.3.2 Déclaration en IDL3 11
3.3.3 Projection en IDL2 11
3.3.4 Utilisation 11
3.3.5 L'interface `Components::Navigation` 12
3.3.6 Critique 12
3.4 Réceptacles 13
3.4.1 Modèle 13
3.4.2 Déclaration IDL3 13
3.4.3 Projection en IDL2 13
3.4.4 Utilisation 14
3.4.5 L'interface `Components::Receptacles` 15
3.4.6 Critique 15
3.5 Les événements 16
3.5.1 Modèle 16
3.5.2 Déclarations en IDL3 17
3.5.3 Projection en IDL2 17
3.5.4 Utilisation 17
3.5.5 L'interface `Components::Events` 19
3.5.6 Critique 19
3.6 Les maisons de composants 20
3.6.1 Modèle 20

3.6.2 Déclaration en IDL3 20
3.6.3 Projection en IDL2 20
3.6.4 Utilisation 21
3.6.5 Les interfaces `Components::CCMHome` et `Components::KeylessCCMHome` . 22
3.6.6 Critique 22
3.7 Configuration de composants 23
3.7.1 Phase de configuration 23
3.7.2 Objets de configuration 23
3.7.3 Configuration basée sur les fabriques 23
3.7.4 Critique 24
3.8 Critique 24

4 Le modèle de programmation **27**
4.1 Implantation d'un composant 27
4.1.1 Framework 27
4.1.2 CIDL 27
4.1.3 Projection 28
4.1.4 Critique 29
4.2 Modèle de programmation des containers 29
4.2.1 Vue d'ensemble 29
4.2.2 Types externes 30
4.2.3 Types de containers 30
4.2.4 Types d'implantations de container 30
4.2.5 Les catégories de composants 30
4.3 Environnement de programmation serveur 31
4.3.1 Fabrique 31
4.3.2 Activation 31
4.3.3 Gestion de la durée de vie des servants 31
4.3.4 Interfaces de programmation du serveur 32
4.4 Modèle de programmation client 33
4.4.1 Type de client 33
4.4.2 Patterns 34
4.5 Critique 34

5 Packaging et déploiement **35**
5.1 Introduction 35
5.2 Packages de déploiement 35
5.2.1 Package d'entité logicielle 35
5.2.2 Descripteur de package de composants 35
5.2.3 Assemblage de composants 36
5.3 Déploiement 36
5.3.1 Processus de déploiement 37

6 Le modèle d'exécution **41**
6.1 Caractérisation des containers 41
6.1.1 Définition 41
6.1.2 Framework 41

6.2 L'architecture des containers . . . 41
6.2.1 Le Portable Object Adapter . . . 42
6.2.2 Les services de CORBA . . . 42
6.2.3 Les catégories de container . . . 43
6.3 Critique . . . 44

7 Méta-modèle de composants 45

8 Synthèse 47
8.1 A propos du CCM . . . 47
8.1.1 Les apports . . . 47
8.1.2 Les points faibles . . . 47
8.2 Les travaux en cours au LIFL . . . 48

9 Conclusion 49

A Problème dans la projection IDL3 vers IDL2 des types d'événements 51

B Les descripteurs OSD du distributeur de boissons 55
B.1 Descripteur de package logiciel . . . 55
B.2 Descripteur de composant . . . 56
B.3 Descripteur d'assemblage . . . 57

C Interfaces OMG IDL3 du distributeur de boissons 59

D Interfaces OMG IDL du distributeur de boissons 63

E Les interfaces OMG IDL du module `Components` 67

Table des figures

1.1 L'exemple du distributeur de boissons 2

2.1 Une application distribuée type 3
2.2 De la production au déploiement 6

3.1 De la définition IDL3 à son utilisation en IDL2 7
3.2 Le modèle abstrait de composant CORBA, le distributeur de boissons. 8
3.3 La déclaration d'un composant CORBA. 8
3.4 La déclaration d'un composant CORBA avec une relation d'héritage. 9
3.5 La projection en IDL2 de la déclaration de composants CORBA. 9
3.6 L'interface `Components::CCMObject`. 10
3.7 Définition de facettes en IDL3. 11
3.8 Projection en IDL2 d'une définition de facettes. 11
3.9 L'interface `Components::Navigation` 12
3.10 Définition de réceptacles en IDL3. 13
3.11 Projection en IDL2 de la définition de réceptacles. 14
3.12 L'interface `Components::Receptacles` 15
3.13 Définition de deux sources et d'un puit d'événements en IDL3. 17
3.14 Projection en IDL2 de la définition de sources d'événements (emitter puis publisher). 18
3.15 Projection en IDL2 de la définition d'un puit d'événements. 18
3.16 L'interface `Components::Events` 19
3.17 Définition d'une maison de composants avec clé primaire. 21
3.18 Projection en IDL2 de la définition d'une maison de composants avec clé primaire. 21
3.19 Les interfaces `Components::CCMHome` et `Components::KeylessCCMHome` 22
3.20 Interface OMG IDL de `Components::Configurator`. 23
3.21 Interface OMG IDL de `Components::StandardConfigurator`. 24
3.22 Interface OMG IDL de `Components::HomeConfiguration`. 24

4.1 Exemple de déclaration CIDL simple. 27
4.2 Relations entre type de maison, type de composants et exécuteurs. 28
4.3 La génération des squelettes de composants. 28
4.4 L'architecture du modèle de programmation des containers. 29
4.5 Exemple d'interface locale : extrait de `Components::Basic::CCMContext`. . . . 32

5.1 L'architecture du processus déploiement. 37
5.2 Points d'entrées pour la création des maisons de composants en Java et en C++. 39

6.1 L'architecture d'un serveur de composants. 42

7.1 Méta-modèle de composants : les extensions à IDL2. 45

Chapitre 1

Introduction
Vers un modèle de composants pour CESURE

Dans le cadre du projet CESURE, les applications visées sont les applications à base de composants distribués. Dès lors, le choix de définir un nouveau modèle de composants ou de réutiliser un modèle existant se pose. Ces deux options ont à la fois des avantages et des inconvénients. Réutiliser un modèle de composants existants permet de profiter des outils disponibles pour développer et exécuter des composants et potentiellement de réutiliser des composants existants. Par contre, il est fort probable que le modèle en question ne réponde pas exactement aux besoins : manque de fonctionnalités, choix technologiques inadaptés au contexte particulier du projet, lourdeur du modèle dont on ne peut s'abstraire, etc. Par opposition, définir complètement un nouveau modèle permet de coller parfaitement avec les besoins particuliers du projet et de ne pas devoir entrer dans un moule de pensée (implicite avec tout modèle de composants). Mais, le gros handicap c'est qu'il est à la charge du projet de produire tous les outils nécessaires à la production de composants ainsi que leur support d'exécution.

Dans le cadre d'un projet purement académique, visant à étudier finement un ensemble réduit de concepts, le choix de définir un nouveau modèle est pleinement justifié. Cependant, dans le contexte de CESURE, les besoins ne sont pas de cet ordre. Bien que l'objectif n'est pas, à ce jour, de fournir une plate-forme industrielle, il est projetté de disposer à la fin du projet d'un prototype de plate-forme utilisable incluant aussi bien la partie développement que la partie exécution. Nous sommes donc arrivé à la conclusion, suite à l'étude de modèles de composants existants [MP00], qu'un choix pertinant d'un modèle de composants comme base de travail était le plus adéquat. Comme les différents modèles étudiés ne répondaient pas parfaitement à nos besoins, nous avons choisi de retenir le modèle le plus complet et le plus récent, quitte à devoir l'adapter sur certains points pour notre utilisation particulière, pour un usage orienté industriel.

Du fait de l'hétérogénéité évidente de l'environnement d'exécution de notre projet, nous avons besoin d'un modèle utilisable sur différents devices, donc sur différents systèmes et potentiellement avec plusieurs langages. De plus, le projet CESURE vise, entre autre, à fournir une nouvelle approche du déploiement des applications à base de composants logiciels. Pour cette raison, le modèle de composants DCOM a été écarté car étant principalement destiné aux plates-formes Microsoft de type PC et n'offrant aucune fonctionnalité particulière pour le déploiement. Bien qu'offrant des éléments de réponse au problème du déploiement, et au vu de la disponibilité de machines virtuelles Java sur un grand nombre de devices, le choix des EJB n'a pas été retenu. En effet, le déploiement des EJB 1.1 est essentiellement mono-site, ce qui répond essentiellement aux besoins des applications de type commerce électronique s'exécutant sur des sites marchands, mais pas aux applications réellement distribuées. De plus, la composition entre EJBs n'est pas explicite, mais gérée au sein même des composants, ce qui ne permet pas de réaliser le déploiement comme nous l'entendons dans [BMPT00].

Nous avons donc finalement opté pour le modèle de composants de la future norme CORBA 3. Bien que cette spécification n'est pas (à l'heure actuelle) achevée, elle annonce un modèle de composants à la fois complet et souple, réellement destiné à concevoir, développer et supporter des applications à base de composants distribués. Cette spécification n'est pas exempte de défauts, sous-spécifications et incohérences dues à sa jeunesse, mais nous considérons que le modèle de composants CORBA est celui au potentiel le plus élevé et donc nous avons pris le parti de miser dessus. Bien que nos développements initiaux sont essentiellement mono-langage (Java), il sera possible par la suite d'utiliser tout autre langage disposant de règles de projections pour l'OMG IDL.

Ce document représente une synthèse, version longue, de différents articles de recherche publiés dans des conférences internationales [MMGG99, MMG00a, MMG00b] et dans des revues [MMGG00].

Exemple fil rouge

Tout au long de ce document, une application « jouet », un « distributeur de boissons », va être utilisée pour illustrer le propos. Tout comme le distributeur de boissons disponible au fond du couloir, cette application met en jeu différents éléments : une prise de courant, une prise d'eau, une sonde de température et le distributeur. A cela viennent s'ajouter deux éléments simulant un consommateur et un fournisseur de boissons. La figure 1.1 donne une vue d'ensemble de cette application.

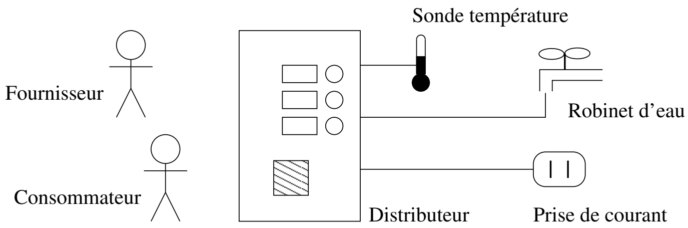

FIG. 1.1 – *L'exemple du distributeur de boissons*

Chapitre 2

Vision d'ensemble des composants CORBA

Comme nous l'avons déjà discuté dans [MP00], les modèles à objets ont progressivement montré leurs limites. Certains industriels, comme Microsoft, Sun et l'OMG, ont fait évoluer leurs modèles vers les composants, dans le but de simplifier le développement d'applications. La réponse de l'OMG est le *CORBA Component Model* (CCM) modèle de base de la future norme CORBA 3. Dans un but de compatibilité ascendante, l'OMG a défini son modèle de composants comme une extension du modèle objet de CORBA 2. Les composants CORBA représentent donc une spécialisation des objets CORBA tels que nous les connaissons aujourd'hui.

2.1 Quels types d'applications ? Quels concepts ?

Les applications visées par les composants CORBA sont des applications réalisées à partir de composants hétérogènes distribués. Les composants formant une application ne s'exécuteront donc pas tous sur un site unique, le serveur d'application, comme c'est el cas avec les EJB. La figure 2.1 présente une vue d'ensemble de ce que peut être une application à base de composants CORBA.

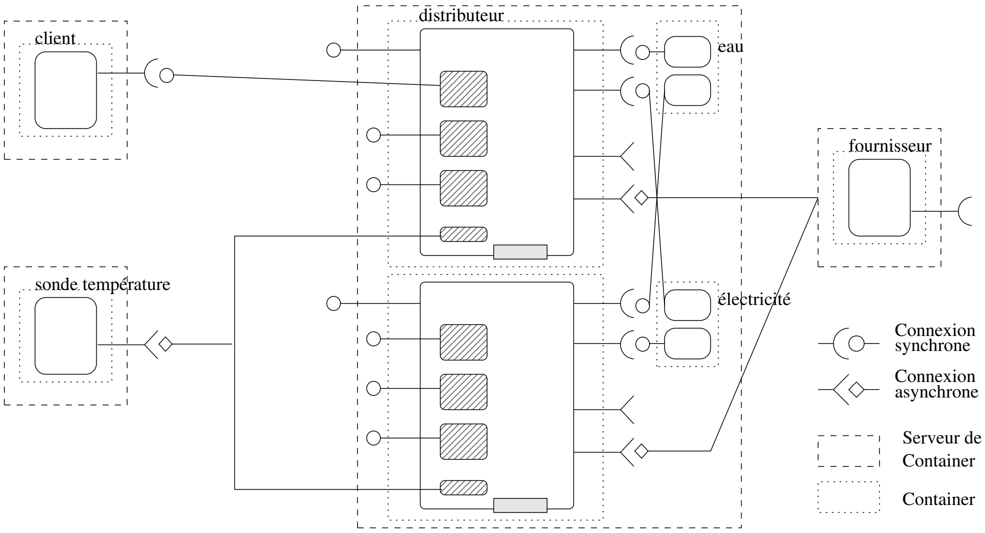

FIG. 2.1 – *Une application distribuée type*

Cette application représente une instance possible de l'application « jouet » distributeur de boissons, fil rouge de ce document. Cette instance de l'application s'exécute sur quatre sites distincts (dans le sens machine) et distants les uns des autres. Tout d'abord, un site central à notre application, le site qui héberge deux instances de composants distributeurs, ce qui est souvent le cas dans le fond des couloirs, un distributeur pour les boissons froides et un pour les boissons chaudes. Pour que ces deux instances s'exécutent, elles nécessitent un support d'exécution. Celui-ci est fourni par un container. Un container est un support d'exécution générique qui peut accueillir différents types de composants, mais ayant toutefois des caractéristiques systèmes communes (en terme de persistance par exemple).

Pour ne pas réduire les capacités d'un site d'exécution à un ensemble de containers statiquement définis et pour ne pas avoir à instancier tous les containers possibles sur un site, les containers reposent sur des serveurs de containers. Ces serveurs sont des processus, au sens Unix du terme, qui permettent d'activer des containers à la demande et selon les besoins. Au sein du serveur de containers principal, il y a donc quatre containers qui permettent l'exécution de six instances de composants : deux instances de distributeurs, deux instances de prises de courants et deux instances de prises d'eau.

Comme un container vient d'être défini comme un support d'exécution générique, pour éviter de réduire les possibilités disponibles lors de l'instanciation de composants, une entité appelée *maison de composants* (*home*) est utilisée. Une maison est un gestionnaire d'instances de composants, s'occupant principalement du cycle de vie. Elle est aux composants ce que l'opérateur `new` est aux objets. Les maisons de composants contiennent le code réifiant l'instanciation des composants, ce qui apporte une certaine souplesse. En plus de permettre l'utilisation d'un même type de containers avec différents types de composants, cette approche permet d'automatiser le déploiement des applications.

En effet, il est possible de choisir des sites d'exécution à partir d'un ensemble connu, puis d'installer une application sur ces sites à partir d'un poste d'administration. Pour cela, en plus de fournir un environnement d'exécution (un serveur de containers), un site doit offrir un service de chargement de code, il est ainsi possible de charger une implantation de composant sur un site distant (ainsi que l'implantation de sa maison associée). Une fois les containers instanciés, les implantations de composants et de maisons chargés, les maisons et les composants instanciés, il ne reste plus qu'à interconnecter les instances de composants entre elles pour former l'application.

La connexion entre deux instances de composants se fait au travers de *ports*, les points de connexion. Pour que deux types de composants puissent être mis en relation, il est requis que ces deux types aient un port fourni et un port requis compatibles. La figure 2.1 présente deux modes de connexion entre les types de composants : un mode synchrone, l'invocation de méthodes, et un mode asynchrone, l'utilisation d'événements. Le modèle de composants CORBA offre une réponse à tous ces besoins.

2.2 Le modèle de composants CORBA

Le CCM est un *framework* de composants de type serveurs, mais qui, semble-t-il, peut facilement être utilisé du côté client des applicatifs. Le CCM se décompose en deux niveaux : un niveau basique, qui permet essentiellement de « componentiser » des objets CORBA, et un niveau étendu, beaucoup plus riche et intéressant que nous allons présenter dans ce document. Tout comme les EJB de Sun, auxquels correspond la version basique du CCM en Java, la technologie CCM repose sur l'utilisation de containers pour héberger les instances de composants et faciliter leur déploiement. La spécification du CCM [OMG99], qui représente plus de 1000 pages, est découpée en quatre modèles et un méta-modèle (Chapitre 7)

Le modèle abstrait offre aux concepteurs le moyen d'exprimer les interfaces (fournies et utilisées) et les propriétés d'un type de composant. Pour cela, le langage OMG IDL a été étendu pour prendre en compte les nouveaux concepts introduits dans le CCM, essentiellement la prise en compte des interfaces multiples : les *ports*. Le modèle abstrait permet aussi de définir les gestionnaires d'instances de composants. C'est essentiellement sur ce première modèle que va porter ce document. (Chapitre 3)

Le modèle de programmation spécifie le langage CIDL (*Component Implementation Definition Language*) à utiliser pour définir la structure de l'implantation d'un type de composant, ainsi que certains de ses aspects non-fonctionnels (persistance, transactions, sécurité). L'utilisation de ce langage est associée à un *framework*, le CIF (*Component Implementation Framework*), qui définit comment les parties fonctionnelles (programmées) et non-fonctionnelles (décrites en IDL / CIDL et générées) doivent coopérer. Il inclut aussi la manière dont l'implantation d'un composant interagit avec le container. (Chapitre 4)

Le modèle de déploiement définit un processus qui permet d'installer une application sur différents sites d'exécution de manière simple et automatique. Ce modèle s'appuie sur l'utilisation de packages de composants, ainsi que de descripteurs OSD (*Open Software Description*, un vocabulaire XML), les packages étant déployables et composables. (Chapitre 5)

Le modèle d'exécution définit l'environnement d'exécution des instances de composants. Le rôle principal des containers est de masquer et prendre en charge les aspects non-fonctionnels des composants qu'il n'est alors plus nécessaire de programmer. (Chapitre 6)

2.3 De la production au déploiement des applications

La figure 2.2 présente l'ensemble du cycle de développement et de déploiement des applications à base de composants CORBA. Elle souligne le découpage du processus de production en différents rôles qui trouvent chacun un ensemble de moyens fournis par le CCM. Ces moyens permettent à chaque acteur du processus de coopérer avec les autres et donc d'avoir une approche multi-acteurs qui soit cohérente. Ce cycle de production des applications à base de composants CORBA est discutée tout au long de ce document, en présentant les différents moyens fournis par le CCM.

Un concepteur définit les types de composants à l'aide du langage IDL étendu. Ces définitions sont projetées en IDL compatible CORBA 2, pour permettre l'implantation des composants par les développeurs. La phase de projection produit aussi un descripteur de composants qui doit être complété par le développeur. Une fois implanté un type de composant est packagé, avec son descripteur et une configuration par défaut, au sein d'une archive pour devenir diffusable. Ce package de composant est par la suite déployé et utilisé tel quel, ou utilisé par un architecte pour être inclus dans un assemblage et produire des applications. Pour être diffusable, un assemblage de composants est packagé, regroupant dans une archive les implantations, le descripteur de l'assemblage et la configuration des composants pour ce contexte précis. Les packages d'assemblage sont ensuite diffusables ou réutilisables pour participer à une nouvelle composition. Enfin, les packages de composants et d'assemblages sont déployés pour instancier les applications sur leurs sites d'exécutions.

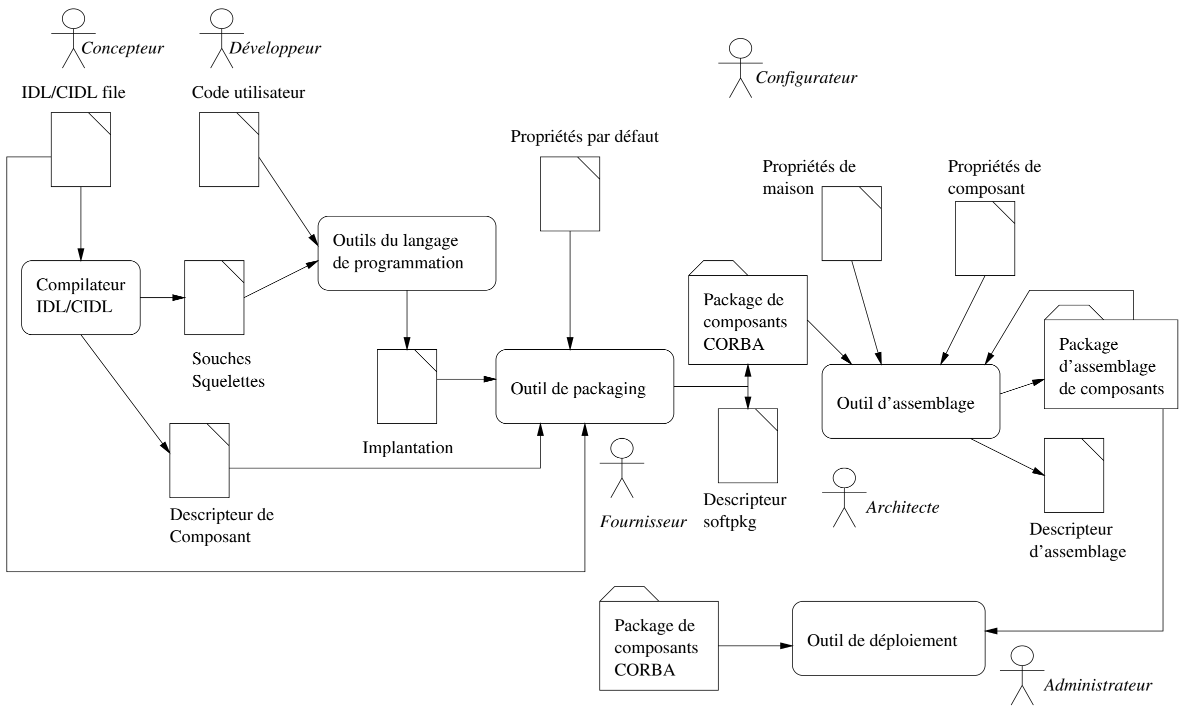

FIG. 2.2 – *De la production au déploiement*

Chapitre 3

Le modèle abstrait de composants

3.1 Introduction

Le modèle abstrait permet de caractériser les interfaces fonctionnelles d'un composant CORBA. Pour définir des types de composants, ce modèle introduit une version enrichie du langage OMG IDL. Dans la suite de ce document, nous utilisons le terme d'IDL3 pour désigner cette nouvelle version du langage et IDL2 pour désigner la version associée à CORBA 2.

Un concepteur de composants va définir un type de composant à l'aide du langage IDL3. Cette construction ne sert que pour la définition du type de composant. Pour implanter un type de composant, la définition IDL3 doit être projettée en IDL2. À chaque construction IDL3 est associée une règle de projection en IDL2. C'est à partir de cet IDL2 que le développeur va pouvoir générer le squelette de l'implantation de composant et programmer sa partie fonctionnelle.

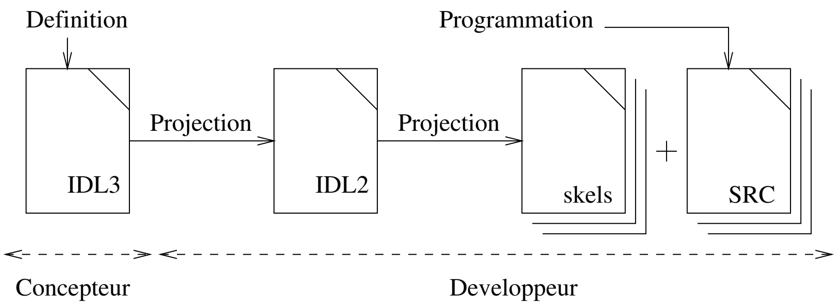

FIG. 3.1 – *De la définition IDL3 à son utilisation en IDL2*

Tout au long de ce chapitre, nous allons suivre la démarche présentée ici, et résumée dans la figure 3.1, pour présenter les différents concepts du modèle abstrait. Chaque concept est donc présenté sous la forme : modèle, expression en IDL3, règle de projection en IDL2, sens de cet IDL2 et API associée, ainsi qu'une rapide critique.

3.2 Type de composant

3.2.1 Modèle

Un type de composant regroupe la définition d'attributs et de *ports*. Les attributs représentent les propriétés configurables du type de composant. Une particularité des attributs dans le cadre du CCM est de pouvoir lever des exceptions lorsqu'ils sont mal utilisés. Un *port* représente une interface (au sens CORBA 2) soit fournie, soit utilisée, par le type de composant. Quatre types de *ports* sont définis (voir figure 3.2) dans le contexte du CCM.

- Une *facette* est une interface fournie par un type de composant et qui est utilisée par des clients en mode synchrone.
- Un *réceptacle* est une interface utilisée par un type de composant en mode synchrone.
- Un *puit d'événement* est une interface fournie par un type de composant et utilisée par ses clients en mode asynchrone.
- Une *source d'événement* est une interface utilisée par un type de composant en mode asynchrone.

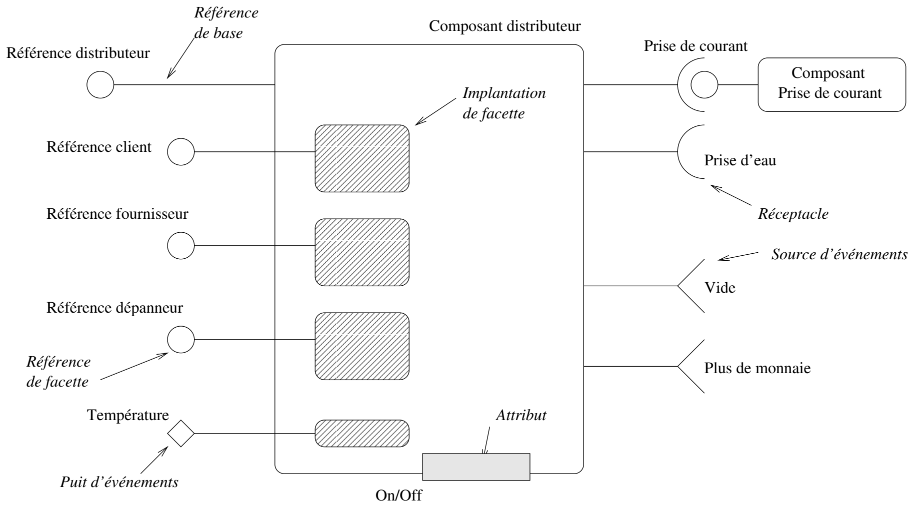

FIG. 3.2 – *Le modèle abstrait de composant CORBA, le distributeur de boissons.*

3.2.2 Définition en IDL3

La déclaration d'un type de composant se fait à l'aide du nouveau mot-clé `component` (voir Figure 3.3). Celui-ci représente un nouveau méta-type, spécialisation du méta-type `interface`. La définition d'un type de composant reste donc très proche de la définition d'une interface.

```
component Distributeur {
    attribute boolean on ;
};
```

FIG. 3.3 – *La déclaration d'un composant CORBA.*

Tout comme les interfaces, les types de composants peuvent être définis en utilisant une relation d'héritage. Toutefois, au contraire des interfaces, les types de composants ne peuvent utiliser qu'un héritage simple. Cet héritage simple est toutefois étendu « à la » Java : un type de composant peut supporter une ou plusieurs interfaces comme présenté dans la figure 3.4. Le `DistributeurCarte` va donc offrir les mêmes ports que le distributeur standard, mais il va en plus offrir les opérations de l'interface `LecteurCarte` et les ports éventuellement définis dans le corps de sa propre déclaration.

```
interface LecteurCarte { ... };
component DistributeurCarte : Distributeur supports LecteurCarte {
    // ...
};
```

FIG. 3.4 – *La déclaration d'un composant CORBA avec une relation d'héritage.*

3.2.3 Projection en IDL2

La définition d'un composant introduit implicitement une interface qui supporte les caractéristiques déclarées dans la définition du corps du composant, ainsi que toutes les opérations de base sur un composant définies dans l'interface `Components::CCMObject`. La figure 3.5 présente la projection d'une déclaration de composant sans et avec héritage. On peut remarquer qu'une fois projetté, l'héritage revient à un héritage multiple d'interfaces standard en CORBA 2.

```
interface Distributeur : Components::CCMObject {
    attribute boolean on ;
};

interface DistributeurCarte : Distributeur, LecteurCarte {
    // ...
};
```

FIG. 3.5 – *La projection en IDL2 de la déclaration de composants CORBA.*

L'interface IDL2 générée est utilisée d'un côté, par les développeurs comme contrat d'implantation du type de composant et, d'un autre côté, par les utilisateurs d'instances de ce type de composant à l'exécution.

3.2.4 L'interface `Components::CCMObject`

L'interface `Components::CCMObject` représente l'interface de base de tout type de composant, implicitement héritée par celui-ci. Cette interface hérite des interfaces `Components::Navigation`, `Components::Receptacles`, et `Components::Events` présentée par la suite. Les opérations définies dans cette interface permettent de retrouver le gestionnaire d'instance d'une instance de composant (`get_ccm_home()`), de valider sa phase de configuration (`configuration_complete()`), et donc de passer en phase d'utilisation du composant (voir section 3.7), et enfin de détruire une instance de composant `destroy()`. L'ensemble des interfaces non décrites du module `Components` concernant le modèle abstrait est présenté dans l'annexe E.

Tout comme les objets CORBA 2, les instances de composants sont caractérisées par leur(s) interface(s) et par une référence, dite « de base » dans le contexte du CCM. Cette référence permet d'utiliser les services offerts par l'instance de composant. Nous verrons dans la section 3.3 qu'il existe d'autres types de références dans le cas des composants.

```
module Components {
    interface CCMObject : Navigation, Receptacles, Events {
        CORBA::IRObject get_component_def () ;
        CCMHome get_ccm_home () ;
        PrimaryKeyBase get_primary_key ()
            raises (NoKeyAvailable) ;
        void configuration_complete ()
            raises (InvalidConfiguration) ;
        void destroy () ;
    };
};
```

FIG. 3.6 – *L'interface* `Components::CCMObject`*.*

3.2.5 Critique

Dans le cadre des composants, les attributs représentent vraiment des propriétés. Le fait que leur mauvaise utilisation soit associée à des cas d'exceptions représente aussi un progrès en terme de qualité du logiciel. Pour ce qui est des relations d'héritage, il est à regretter que les composants ne supportent que l'héritage simple, ce qui n'est pas la règle en CORBA 2. Tout comme pour les interfaces un type de composant est un contrat, il n'y a donc *a priori* pas de risque de disposer de l'héritage multiple puisqu'il ne s'agit pas d'héritage d'implantations. Enfin, puisque `component` est une extension d'`interface`, il est surprenant de ne pouvoir définir des opérations indépendemment d'une interface sans passer par l'utilisation de la clause `supports`. Le langage IDL3 ne représente qu'une solution technologique à la définition d'un type de composant. Ce langage n'impose pas de méthodologie qui doit être définie en amont et utilisée pour aboutir au résultat qu'est la définition du type en IDL3.

3.3 Facettes

3.3.1 Modèle

Une facette représente un point de vue sur un type de composants. Dans le contexte de la norme CORBA 2, un objet dispose d'une interface OMG IDL unique, donc qui regroupe toutes les opérations disponibles sur cet objet. Même dans le cas d'une relation d'héritage où une référence peut être conforme à plusieurs interfaces, le type le plus dérivé regroupe toutes les opérations. Il n'était donc pas possible de regrouper logiquement les opérations d'un objet. Les composants représentent un progrès sur ce point. Il est désormais possible de définir une facette par rôle joué par un type de composant. Si on regarde un distributeur de boissons, plusieurs interfaces sont disponibles : une interface pour le consommateur (payer, sélectionner une boisson, prendre la boisson), une interface pour le fournisseur (ouvrir distributeur, mettre des boissons, mettre de la monnaie, vider la caisse, fermer le distributeur) et une interface pour le dépanneur (ouvrir le capot arrière, fermer le capot arrière, purger la pompe). Ce qui justifie d'avoir une facette par utilisation pour ne pas que l'usager lambda soit tenté d'intervenir sur le moteur du distributeur.

Comme tout élément d'une application CORBA, une facette dispose d'une interface OMG IDL. Cette interface est le seul élément disponible pour un client du composant. Similairement aux objets, les composants reposent sur le concept d'encapsulation. En cela, une implantation de facette est totalement masquée et enfouie au sein d'une implantation de composant. Il en résulte que les facettes d'un composant ne sont pas dissociables du composant et donc leur cycle de vie y est intimement lié. En plus de la référence de base d'une instance de composant, chaque facette dispose de sa propre référence, appelée *référence de facette*. Il peut exister à un moment donné, plusieurs copies d'une référence de facette pour une même facette (gestion possible du cas de plusieurs usagers connectés).

3.3.2 Déclaration en IDL3

Comme le présente la figure 3.7, la déclaration d'une facette se fait à l'aide de la clause `provides`. La facette déclarée doit nécessairement correspondre à une interface OMG IDL définie. Pour chaque facette d'un type de composant, il est nécessaire de déclarer une clause `provides`. L'arité d'une telle clause est nécessairement un. Il est toutefois possible de définir plusieurs ports du même type. Chaque facette est identifiée par un nom unique.

```
interface FacadeClient { ... } ;
interface FacadeFournisseur { ... } ;
interface FacadeDepanneur { ... } ;
component Distributeur {
    provides FacadeClient      client ;
    provides FacadeFournisseur fournisseur ;
    provides FacadeDepanneur   depanneur ;
};
```

FIG. 3.7 – *Définition de facettes en IDL3.*

3.3.3 Projection en IDL2

La projection en IDL2 est obtenue en définissant pour chaque clause `provides` une opération du nom de la facette déclarée en IDL3, préfixée par `provide_` et retournant une référence du type de la facette. Une fois projettée en IDL2, la déclaration d'une facette se résume à un *design pattern* syntaxique de nommage des opérations de récupération des facettes.

```
interface Distributeur : Components::CCMObject {
    FacadeClient       provide_client () ;
    FacadeFournisseur  provide_fournisseur () ;
    FacadeDepanneur    provide_depanneur () ;
};
```

FIG. 3.8 – *Projection en IDL2 d'une définition de facettes.*

3.3.4 Utilisation

La prise en charge des opérations de connectique se fait au sein du squelette de composant qui est généré à partir de l'IDL2. Cette prise en charge se fait à deux niveaux : d'une part, les opérations génériques de l'interface `Components::Navigation`, et, d'autre part, les opérations spécifiques au type de composant qui proviennent de la projection de la déclaration IDL3 du type de composant. L'utilisation des méthodes génériques est toujours possible, utile pour faire de l'introspection dynamique. Elle est donc souple et puissante, mais contrôlée uniquement à l'exécution. L'utilisation des opérations spécifiques et quant à elle statique, donc figée mais fiable. Une bonne solution de production des opérations spécifiques est donc de s'appuyer sur les opérations génériques. En effet, le nom de facette fournie par une opération spécifique est nécessairement correct, et donc l'opération générique n'échouera pas à l'exécution. Il ne reste au développeur que la partie fonctionnelle de l'implantation à programmer.

3.3.5 L'interface `Components::Navigation`

En plus d'offrir la possibilité de proposer plusieurs facettes, un composant offre un moyen de naviguer entre celles-ci. Un client peut donc changer son point de vue sur le composant au cours du temps, par exemple si ses besoins changent : le fournisseur a soif. La navigation dans les facettes est centralisée, elle gravite autour de la référence de base du composant. Pour permettre la navigation, une interface de base est offerte : `Components::Navigation` (héritée par `Components::CCMObject`).

L'interface `Components::Navigation` offre des opérations génériques disponibles et utilisables sur tout type de composants. Cette interface est implicitement héritée par tout type de composant. Les opérations principales de `Components::Navigation` sont des opérations d'introspection (`describe_facets()`), de récupération des facettes (`provide_all_facets()`, `provide_named_facets()` et `provide_facet()`). Lorsque l'on recherche des facettes de manière ciblée, les noms des facettes recherchées sont utilisées (le nom de déclaration de la facette dans la clause `provides` du composant). Pour terminer, une opération permet de tester si deux facettes appartiennent au même composant (`same_component()`).

```
module Components {
    interface Navigation {
        Object provide_facet (in FeatureName name)
            raises (InvalidName);
        FacetDescriptions describe_facets () ;
        Facets provide_all_facets () ;
        Facets provide_named_facets (in NameList names)
            raises (InvalidName) ;
        boolean same_component (in Object ref) ;
    };
};
```

FIG. 3.9 – *L'interface `Components::Navigation`*

Dans le contexte du CCM, toute entité logicielle devient composant, le seul cas d'utilisation d'interfaces est dans la définition de facettes. Il est donc nécessaire de pouvoir retrouver le composant offrant une facette pour laquelle on dispose d'une référence. L'interface `CORBA::Object` a été enrichie d'une opération : `get_component()`. Cette opération permet de savoir à quel instance de composant se rapporte une facette.

3.3.6 Critique

Les facettes représentent un apport important de la spécification des composants. En effet, elles permettent d'éviter la définition d'interfaces « obèses » regroupant en vrac tous les services offerts par un type d'objets. Ensuite, les capacités d'introspection associées aux facettes permettent une découverte simple de l'ensemble des services fournis par un composant, ainsi que de leur utilisation.

Toutefois, l'arité (un) et la co-localisation des facettes peuvent être vus comme des limitations du modèle. D'autre part, il est dommage que la déclaration des facettes soit uniquement statique et qu'il n'est pas possible d'en déclarer à l'exécution.

3.4 Réceptacles

3.4.1 Modèle

Un réceptacle permet à un type de composant d'accepter une référence d'objet (c'est-à-dire une référence de facette, de composant ou d'objet au sens CORBA 2). Cette relation est appelée connexion. Le réceptacle est un point de connexion conceptuel. C'est une abstraction qui se manifeste concrètement dans un composant par un jeu d'opérations pour établir et gérer les connexions. Un réceptacle permet donc d'assembler des instances composants et potentiellement des objets. La composition permet de concevoir plus facilement des applications. Elle permet aussi aux instances de composants de faire de la délégation de traitement. Les réceptacles peuvent être de deux types : des réceptacles simples (une seule référence), ou des réceptacles multiples (plusieurs références d'objets peuvent être fournies).

Dans notre exemple, le réceptacle `PriseEau` est simple, alors que le réceptacle `PriseCourant` est multiple. L'utilisation d'un réceptacle simple signifie que l'on a au plus une instance de composant connectée à un moment donné. Dans le cas des réceptacles multiples, une limite pour le nombre maximum de connexions simultanées peut être fixée dans l'implantation du composant.

3.4.2 Déclaration IDL3

Pour les deux types de réceptacles, la déclaration se fait au moyen du mot-clé `uses`, tel que présenté dans la Figure 3.10. Dans le cas de réceptacle multiple, le mot-clé `multiple` s'ajoute à l'utilisation du mot-clé `uses`. Chaue réceptacle est nommé à l'aide d'un identifiant unique.

```
interface PriseCourant { ... } ;
interface PriseEau { ... } ;
component Distributeur {
    uses PriseEau              prise_eau ;
    uses multiple PriseCourant prise_courant ;
};
```

FIG. 3.10 – *Définition de réceptacles en IDL3.*

3.4.3 Projection en IDL2

La projection en IDL2 de la déclaration de réceptacles est plus complexe que la projection des facettes. Dans le cas de réceptacles multiples, une structure est déclarée pour permettre la gestion de plusieurs connexions. À chaque connexion est associé un *cookie* lors de son établissement. Ce cookie identifie de manière unique la connexion et représente l'information vitale pour pouvoir agir sur une connexion (comme la supprimer). Pour les deux types de réceptacles, des opérations de connexion et de déconnexion sont définies. Cette projection est présentée dans la figure 3.11.

```
interface Distributeur : Components::CCMObject {
    void connect_prise_eau (in PriseEau cnx)
        raises (Components::AlreadyConnected,
                Components::InvalidConnection) ;
    PriseEau disconnect_prise_eau ()
        raises (Components::NoConnection) ;
    PriseEau get_connection_prise_eau () ;

    struct prise_courantConnection {
        PriseCourant        objref ;
        Components::Cookie  ck ;
    };
    typedef sequence<prise_courantConnection> prise_courantConnections;

    Components::Cookie connect_prise_courant (in PriseCourant cnx)
        raises (Components::ExceededConnectionLimit,
                Components::InvalidConnection) ;
    PriseCourant disconnect_prise_courant (in Components::Cookie ck)
        raises (Components::InvalidConnection) ;
    prise_courantConnections get_connections_prise_courant () ;
};
```

FIG. 3.11 – *Projection en IDL2 de la définition de réceptacles.*

3.4.4 Utilisation

Pour utiliser des connexions, l'implantation fonctionnelle d'un composant dispose d'opérations à invoquer sur le réceptacle pour récupérer une référence d'objet. La manière dont les réceptacles sont gérés peut être variable d'une implantation à une autre, c'est pourquoi un développeur doit utiliser l'opération `get_connections()`. Lorsque le composant demande les connexions par rapport au nom du réceptacle, la valeur retournée est une liste de structures contenant le cookie et la référence d'objet de la connexion. Dans le cas d'un réceptacle simple, seule la référence de l'objet est retournée.

Une fois une connexion établie entre un réceptacle et un objet, le composant peut invoquer des opérations sur l'objet à tout moment et sans restriction. La manière dont l'invocation sera effectuée est totalement, et uniquement, dépendante de l'implantation du composant.

Connexion

L'opération de connexion est en partie générée et, potentiellement, en partie implantée par le développeur du composant. Une fois l'opération de connexion effectuée, le réceptacle conserve la référence jusqu'à ce qu'il y ait une déconnexion explicite.

Dans le cas de réceptacles simples, une seule connexion peut exister à la fois (sinon l'exception `AlreadyConnected` est levée). Dans le cas de connexions multiples, le nombre de connexions peut être borné (si ce nombre est dépassé, l'exception `ExceededConnectionLimit` est levée). Dans les deux cas, l'implantation peut refuser une connexion pour des raisons arbitraires, elle lève alors l'exception `InvalidConnection`.

Pour les réceptacles multiples, une opération de connexion retourne un cookie qui sert à identifier la connexion. Ce cookie est produit par le réceptacle (partie générée) et utilisé par le client pour exploiter une connexion et pour la supprimer [1]. Des cookies sont nécessaires parce que des références d'objets CORBA ne peuvent être comparées sémantiquement.

Déconnexion

L'opération de déconnexion supprime la relation qui existe entre le composant et l'objet. Dans le cas d'un réceptable simple, l'opération de déconnexion retourne la référence de l'objet qui était connecté (ou lève l'exception `NoConnection`). Dans le cas d'un réceptacle multiple, il est nécessaire de spécifier le cookie de la connexion à terminer. Si le cookie n'est pas correct, l'exception `InvalidConnection` est levée.

3.4.5 L'interface `Components::Receptacles`

L'interface `Components::Receptacles` offre des opérations génériques pour (dé)connecter des composants entre eux (voir figure 3.12). Les opérations s'appliquent sur le port dont le nom est passé en paramètre. Dans le cas d'une mauvaise utilisation ou d'un nom de port inconnu, les opérations lèvent une exception. L'opération `get_connections()` retourne quant à elle une liste des descriptions des facettes connectées à un réceptacle [multiple].

```
module Components {
    interface Receptacles {
        Cookie connect (in FeatureName name,
                        in Object connection)
            raises (InvalidName,
                    InvalidConnection,
                    AlreadyConnected,
                    ExceededConnectionLimit) ;
        void disconnect (in FeatureName name,
                         in Cookie ck)
            raises (InvalidName,
                    InvalidConnection,
                    CookieRequired,
                    NoConnection) ;
        ConnectedDescriptions get_connections (in FeatureName name)
            raises (InvalidName);
    };
};
```

FIG. 3.12 – *L'interface `Components::Receptacles`*

3.4.6 Critique

Les réceptacles représentent le moyen de faire de la composition. Ils représentent un progrès par rapport aux objets CORBA dans le sens où les services utilisés par un composant sont

1. Le client doit conserver le cookie, qui ne peut être récupéré une deuxième fois.

explicites et non uniquement des références enfouies au fond de l'implantation. Il est donc d'une part, possible de connaître les services à fournir à un composant pour son exécution et, d'autre part, simple de modifier les connexions tout au long du cycle de vie de l'instance de composant.

La sémantique des réceptacles multiples n'est pas claire dans la spécification. Une utilisation possible est la tolérance aux pannes : dans le cas des prises de courant, si un composant prise de courant ne marche plus (par exemple une prise sur batterie avec une batterie vide) alors le composant distributeur va utiliser une autre prise de courant, elle aussi connectée. L'utilisation de réceptacles multiples peut aussi trouver un sens dans la mise en œuvre d'interfaces de *callback* entre un type de composant et d'autres types utilisant ses facettes. Ce choix va être dépendant de l'implantation. Il serait donc bon de pouvoir préciser si toutes les facettes connectées à un réceptacle multiple recoivent les messages ou si les facettes les reçoivent à tour de rôle.

Ensuite, il est à regretter l'absence d'introspection structurelle des réceptacles, c'est-à-dire pourvoir découvrir dynamiquement les réceptacles disponibles sur un type de composant avec une opération fournie par le type de composant. L'introspection est possible par rapport au référentiel des interfaces, mais une opération similaire à `provide_all_facets()`, `get_receptacles()` par exemple, serait un atout pour une utilisation dynamique ou automatique des instances de composants.

Enfin, l'interface `Components::Receptacles` semble mieux adresser les réceptacles multiples que les réceptacles simples. En effet, l'opération `get_connections()` semble, de par son nom et son utilisation (type de la valeur de retour), viser les réceptacles multiples plutôt que les réceptacles simples. Les opérations de (dé)connexion sont quant à elles clairement conçues pour les réceptacles multiples du fait de la gestion d'un cookie absent des opérations relatives aux réceptacles simples. Il est toutefois possible de les utiliser pour implanter la projection des opérations pour les ports de type réceptacle simple, mais cela semble plus de l'adaptation que du « prévu pour ».

3.5 Les événements

3.5.1 Modèle

Le modèle d'événements est du type producteur/consommateur. Ce modèle est compatible avec les services d'événements et de notification de CORBA. Pour recevoir des événements, un client doit souscrire à ce type d'événements. Ensuite, seul le mode *push* est utilisé. Les mécanismes sont définis par les interfaces de consommateurs. Des services pour les événements sont fournis par l'implantation de la structure d'accueil. Il y a deux types de ports qui traitent des événements : les sources et les puits d'événements.

Les sources d'événements sont de deux ordres. *Emitter* est la catégorie de source qui n'accepte qu'un seul consommateur (un vers un). Dans ce cas, il n'y a pas de médiation par un canal d'événements. La connexion producteur-consommateur est directe. Cette connexion se fait à l'initialisation du composant. *Publisher* est la catégorie de source d'événements qui accepte plusieurs consommateurs (un vers n). Dans ce cas, l'abonnement d'un consommateur à un type d'événements est délégué à un canal d'événements fourni par la structure d'accueil. La seule source pour ce canal d'événements est l'instance de composant pour qui il a été créé.

Un puit d'événements permet à un composant de recevoir des événements d'un certain type. C'est une facette dont le type est un consommateur d'événements (interface de base :

`Components::EventConsumerBase`). Il n'y a pas, ici, de différentiation entre connexion et abonnement. Le composant ne contrôle pas ce récepteur, il le rend publique pour recevoir des événements. Le puit peut donc recevoir des événements en provenance de plusieurs producteurs.

3.5.2 Déclarations en IDL3

La déclaration de sources d'événements (présentée dans la figure 3.13) se fait selon le cas avec le mot clé `emits` (un vers un) ou `publishes` (un vers n). Pour les puits d'événements, la déclaration se fait à l'aide du mot clé `consumes`. Le type de chacun de ces ports est un *Object by value* représentant le type d'événement émis ou consommé, type qui hérite du type de base pour les événements : `Components::EventBase`.

```
valuetype PlusDeMonnaieEvt    : Components::EventBase { ... } ;
valuetype DistributeurVideEvt : Components::EventBase { ... } ;
valuetype TemperatureEvt      : Components::EventBase { ... } ;
component Distributeur {
    emits     PlusDeMonnaieEvt     pdm ;
    publishes DistributeurVideEvt  dvide ;
    consumes  TemperatureEvt       temp ;
};
```

FIG. 3.13 – *Définition de deux sources et d'un puit d'événements en IDL3.*

3.5.3 Projection en IDL2

Tout comme les projections de réceptacles, les projections relatives aux événements sont assez complexes (voir figure 3.14). Dans le cas des sources d'événements, une interface de consommateur d'événements (comme `PlusDeMonnaieEvtConsumer`) est définie pour le client de cette source[2]. Ensuite, des opérations de (dé)connexion (`connect_pdm()` et `disconnect_pdm()`) ou de (dés)abonnement (`subscribe_dvide()` et `unsubscribe_dvide()`) sont ajoutées à l'interface du composant. L'ensemble des projections des déclarations de sources de la figure 3.13 est décrit dans la figure 3.14.

La projection d'un puit d'événements (figure 3.15) est très similaire à celle d'une source. Néanmoins, l'interface consommateur (`TemperatureEvtConsumer`) n'est plus destinée au client, mais au composant. L'opération `get_consumer_temp()` permet de récupérer la référence du puit d'événements pour la connecter à une source fournie par une autre instance de composant.

3.5.4 Utilisation

Deux opérations sont générées par le compilateur IDL3 pour la définition d'un *emitter*. Ces deux opérations permettent la connexion et la déconnexion d'un consommateur. Lors d'une déconnexion, la référence du consommateur est retournée.

Les deux opérations générées pour un port de type *publisher* sont les opérations d'abonnement et de résiliation d'abonnement d'un consommateur aux événements produits par ce port.

2. Cette règle de projection pose un problème puisqu'elle est inutilisable. Se reporter à la section 3.5.6 et à l'annexe A pour une proposition de résolution soumise à l'OMG.

```
module DistributeurEventConsumers {
    interface PlusDeMonnaieEvtConsumer : Components::EventConsumerBase {
        void push (in PlusDeMonnaieEvt evt) ;
    };
    interface DistributeurVideEvtConsumer : Components::EventConsumerBase {
        void push (in DistributeurVideEvt evt) ;
    };
};

interface Distributeur : Components::CCMObject {
    void connect_pdm
        (in DistributeurEventConsumers::PlusDeMonnaieEvtConsumer consumer)
        raises (Components::AlreadyConnected) ;
    DistributeurEventConsumers::PlusDeMonnaieEvtConsumer
    disconnect_pdm () raises (Components::NoConnection) ;

    Components::Cookie subscribe_dvide
        (in DistributeurEventConsumers::DistributeurVideEvtConsumer consumer)
        raises (Components::ExceededConnectionLimit) ;
    DistributeurEventConsumers::DistributeurVideEvtConsumer
    unsubscribe_dvide (in Components::Cookie ck)
        raises (Components::InvalidConnection) ;
};
```

FIG. 3.14 – *Projection en IDL2 de la définition de sources d'événements (emitter puis publisher).*

```
module DistributeurEventConsumers {
    interface TemperatureEvtConsumer : Components::EventConsumerBase {
        void push (in TemperatureEvt evt) ;
    };
};
interface Distributeur : Components::CCMObject {
    DistributeurEventConsumers::TemperatureEvtConsumer get_consumer_temp () ;
};
```

FIG. 3.15 – *Projection en IDL2 de la définition d'un puit d'événements.*

La source du composant ne publiera que dans ce canal. Lors de l'abonnement, un cookie est généré. Il représente la souscription du consommateur. Ce cookie est nécessaire pour terminer un abonnement.

L'opération disponible sur un puit d'événement retourne une référence sur le consommateur qui correspond au type d'événement déclaré.

3.5.5 L'interface `Components::Events`

L'interface `Components::Events` (figure 3.16) offre un accès générique aux sources / récepteurs d'événements (`Components::CCMObject` dérive de `Components::Events`). Les différentes opérations permettent de (dé)connecter ou bien de se (dés)abonner à une source d'événements, ainsi que de récupérer un consommateur d'événements.

```
module Components {
    interface Events {
        EventConsumerBase get_consumer (in FeatureName sink_name)
            raises (InvalidName) ;
        Cookie subscribe (in FeatureName publisher_name,
                          in EventConsumerBase subscriber)
            raises (InvalidName) ;
        void unsubscribe (in FeatureName publisher_name,
                          in Cookie ck)
            raises (InvalidName, InvalidConnection) ;
        void connect_consumer (in FeatureName emitter_name,
                               in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected) ;
        EventConsumerBase
        disconnect_consumer (in FeatureName source_name)
            raises (InvalidName, NoConnection) ;

    };
};
```

FIG. 3.16 – *L'interface `Components::Events`*

3.5.6 Critique

En premier lieu, la projection des événements contient une erreur qui implique une impossibilité d'utiliser deux types de composants prévus pour être connectés ensemble au travers d'un port de type événements. En effet, les interfaces de consommation d'événements générées lors de la projection en IDL2 ne sont pas compatibles, alors que définies en IDL3 pour être utilisées ensemble. Pour résoudre ce problème, nous recommandons l'introduction d'un nouveau mot-clé (`eventtype`) pour définir les types d'événements dans le cadre du CCM, ce mot-clé servirait alors de base pour définir les interfaces des consommateurs d'événements. Ces derniers seraient alors définis de manière identique pour les consommateurs requis (interface d'une source d'événement) et pour les consommateurs fournis (interface d'un puit d'événement). Un exemple de projection incorrecte et l'illustration de notre proposition est présentée dans l'annexe A.

Ensuite, ici encore, l'absence d'introspection structurelle sur les ports de type événements représente un point faible. Il n'est pas possible de découvrir les ports asynchrones sans passer par le référentiel des interfaces et par la complexité associée à cette approche.

3.6 Les maisons de composants

Jusqu'ici, le modèle de composant a été présenté. Les différents concepts ont été discutés et il est maintenant possible de définir un type de composant dans son intégralité. Cette section présente les gestionnaires de composants qui permettent à l'exécution de créer des instances pour les types de composants définis à l'aide des concepts présentés dans les sections précédentes. L'objectif de cette section est de présenter la réification de l'instanciation, c'est-à-dire l'équivalent de l'opérateur `new` dans les modèles à objets.

3.6.1 Modèle

Tout comme `component`, `home` est un nouveau méta-type défini dans le cadre du CCM. Une maison de composants est un gestionaire pour des instances d'un même type de composant. Elle gère le cycle de vie des instances, éventuellement en leur associant des clés primaires dans le cas des instances de composants persistantes. Pour cela, elle offre une fabrique d'instances de composants et des opérations de recherche utilisant ces clés. Les interfaces de maisons supportent l'héritage simple. Le type de base est `Components::CCMHome`, et les attributs et opérations classiques sont supportés par les interfaces de maison de composant.

Un type de composant est défini de manière indépendante des types de maison. Par contre, un type de maison doit spécifier le type de composant qu'elle va gérer. Plusieurs types de maison peuvent gérer un même type de composant, mais pas les mêmes instances. À l'exécution, une instance de composant est gérée par une unique instance de maison.

Les clés primaires sont des identifiants uniques d'instances de types de composants. Ces clés permettent de gérer des instances de composants persistantes. Les clés sont des `valuetype`s qui héritent de `Components::PrimaryKeyBase`. Une clé primaire doit être un type concret avec au moins un membre public et sans membre privé. Un type clé primaire ne peut pas contenir de référence d'interfaces CORBA (et récursivement dans le cas où la *valuetype* contient des structures ou unions).

3.6.2 Déclaration en IDL3

La déclaration d'une maison de composant (voir figure 3.17) définit un type de maison pour le type de composant `Distributeur` avec utilisation d'une clé primaire, `IdentifiantDistributeur`. En plus des opérations de base implicitement définies et générées, le type `MaisonDeDistributeurs` inclu deux opérations déclarées par le concepteur. Ces deux opérations représentent les *design patterns* « fabrique », utilisation du mot-clé `factory`, et « recherche », utilisation du mot-clé `finder`.

3.6.3 Projection en IDL2

La projection de la déclaration d'une maison se décompose en trois interfaces. L'interface explicite regroupe les opérations déclarées par le concepteur (`creer_distributeur()` et

```
valuetype IdentifiantDistributeur : Components::PrimaryKeyBase {
    public long identifiant ;
};
home MaisonDeDistributeurs manages Distributeur
                           primaryKey IdentifiantDistributeur {
    factory creer_distributeur (in long id)
        raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
    finder  rechercher_distributeur (in long id)
        raises (Components::UnknownKeyValue,Components::InvalidKey) ;
};
```

FIG. 3.17 – *Définition d'une maison de composants avec clé primaire.*

`rechercher_distributeur()` qui sont des opérations applicatives pour ne pas avoir à manipuler les opérations génériques purement techniques). L'interface implicite regroupe les opérations génériques des maisons de composants typés pour ce type de maison. L'interface effective de la maison hérite des deux précédentes, comme le présente la figure 3.18.

```
interface MaisonDeDistributeursExplicit : Components::CCMHome {
    Distributeur creer_distributeur (in long id)
        raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
    Distributeur rechercher_distributeur (in long id)
        raises (Components::UnknownKeyValue,Components::InvalidKey) ;
};
interface MaisonDeDistributeursImplicit {
    Distributeur create (in IdentifiantDistributeur key)
        raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
    Distributeur find_by_primary_key (in IdentifiantDistributeur key)
        raises (Components::UnknownKeyValue,Components::InvalidKey) ;
    void remove (in IdentifiantDistributeur key)
        raises (Components::UnknownKeyValue,Components::InvalidKey) ;
    IdentifiantDistributeur get_primary_key (in Distributeur compo) ;
};
interface MaisonDeDistributeurs :
    MaisonDeDistributeursExplicit, MaisonDeDistributeursImplicit {
};
```

FIG. 3.18 – *Projection en IDL2 de la définition d'une maison de composants avec clé primaire.*

3.6.4 Utilisation

Lorsque les définitions de maisons sont compilées, différentes opérations sont systématiquement générées pour le type de composant géré par la maison. Dans le cas d'une maison sans clé, la seule opération générée est `create`. Dans le cas d'une maison avec clé primaire les opérations `create`, `find` et `destroy` sont générées. Enfin, les opération de l'interface explicite du type de composant sont à implanter manuellement par un développeur.

Les éléments de recherche (Home Finders) sont des interfaces de recherche simplifiées analogues à l'interface `CosLifeCycle::FactoryFinder`. Ils permettent de trouver des maisons avec des critères tels qu'un type de composants particulier, un type de maison particulier, ou par rapport à la façon dont les maisons sont nommées dans le service de nommage (utilisation d'un pattern de nommage). Un élément de recherche de maisons est accessible via un appel à `CORBA::ORB::resolve_initial_references` pour le service nommé `ComponentHomeFinder`.

3.6.5 Les interfaces `Components::CCMHome` et `Components::KeylessCCMHome`

L'interface `Components::CCMHome` représente l'interface de base de toute maison de composant. Toute projection, que ce soit pour une maison de composant avec ou sans clés, va hériter de cette interface au travers de son interface explicite. Dans le cas des maisons de composants sans clé, l'interface `Components::KeylessCCMHome` est hérité (en plus) au niveau de l'interface implicite. Dans un contexte d'utilisation dynamique, les opérations `get_component_def()` et `get_home_def()` permettent d'accéder à la définition de la maison de composant, et du type des instances de composants qu'elle peut créer, contenues dans le référentiel des interfaces. Ces définitions permettent l'introspection du type de la maison de composant et ainsi de connaître le nom de l'opération qu'il faut utiliser pour créer des instances et quels paramètres doivent être fournis pour cette instanciation (voir section 3.6.3). Dans le cas des maisons de composants sans clé, l'introspection n'est pas nécessaire car l'opération `create_component()` est toujours disponible.

```
module Components {
    interface CCMHome {
        CORBA::IRObject get_component_def () ;
        CORBA::IRObject get_home_def () ;
        void remove_component (in CCMObject comp) ;
    };

    interface KeylessCCMHome {
        CCMObject create_component () ;
    };
};
```

FIG. 3.19 – *Les interfaces `Components::CCMHome` et `Components::KeylessCCMHome`*

3.6.6 Critique

L'approche des maisons de composants, reprise aux EJB, représente un atout pour l'instanciation de composants dans un contexte générique et sans avoir à écrire un programme pour chaque composant. La définition des maisons résume bien la philosophie du CCM : extraire de la programmation CORBA 2 des patterns qui pourront être mis en œuvre de manière systématique et en s'appuyant sur la génération automatique de code. Il est toutefois à noter l'incohérence de la déclaration de `Components::KeylessCCMHome` qui n'hérite pas de `Components::CCMHome` et que cette dernière ne propose pas l'opération `create_component()` réduisant les capacités de traitement générique dans un cadre d'utilisation statique. Dans un cadre générique, il est de toute façon toujours possible d'utiliser l'interface la plus dérivée. Enfin, il est impossible de définir des maisons de composants qui soient des composants, de même qu'il est imopssible de définir des composants qui soient aussi des maisons.

3.7 Configuration de composants

3.7.1 Phase de configuration

Le cycle de vie d'un composant se décompose en deux phases qui doivent être en exclusion mutuelle : la phase de configuration et la phase fonctionnelle. Il est recommandé d'avoir des interfaces distinctes pour chacune de ces phases. Néanmoins, cette distinction n'est pas clairement énoncée dans la norme CORBA. C'est donc au concepteur de prévoir cette distinction. Il doit implanter explicitement la phase de configuration d'une instance de composant, ou fournir les moyens de la réaliser. La configuration d'un composant repose principalement sur le positionnement des valeurs de ses attributs.

3.7.2 Objets de configuration

Un objet de configuration est un objet qui encapsule la configuration de certains attributs et qui peut être utilisé pour la configuration de toute instance d'un type de composant. Un objet de configuration peut utiliser toute méthode du composant disponible pendant la phase de configuration. Un objet de configuration hérite du type de base `Components::Configurator`, présenté dans la figure 3.20. Il offre une méthode, `configure()`, qui prend comme paramètre le composant à configurer.

```
module Components {
    interface Configurator {
        void configure (in CCMObject comp)
            raises (WrongComponentType) ;
    };
};
```

FIG. 3.20 – *Interface OMG IDL de* `Components::Configurator`.

Il existe une interface qui définit un objet de configuration standard : `Components::StandardConfigurator`. Cette interface offre une opération `set_configuration()` qui prend comme paramètre une séquence de couples (nom d'attribut, valeur) définis par une `valuetype` : `Components::ConfigValue`. L'utilisation de l'opération `configure()` (de l'objet de configuration) applique les valeurs contenues dans la séquence sur les attributs. Cette partie de l'interface est utilisée lors de la phase de déploiement pour initialiser les objets de configuration (voir section 5.2.3).

3.7.3 Configuration basée sur les fabriques

Les opérations des fabriques de composants présentes dans les maisons peuvent participer au processus de configuration des composants. Pour cela, une opération de fabrique peut :

- être implantée pour une configuration particulière,
- appeler un objet de configuration sur toute nouvelle instance d'un composant,
- appeler explicitement `configuration_complete()` et mettre ainsi fin à la phase de configuration d'un composant.

L'implantation d'une maison de composant peut éventuellement supporter l'interface `Components::HomeConfiguration`, qui dérive de `Components::CCMHome` (cf figure 3.22). Cette interface est principalement destinée aux agents utilisés pour déployer une instance de composant

```
module Components {
    valuetype ConfigValue {
        public FeatureName name ;
        public any value ;
    };
    typedef sequence<ConfigValue> ConfigValues ;

    interface StandardConfigurator : Configurator {
        void set_configuration (in ConfigValues descr) ;
    };
};
```

FIG. 3.21 – *Interface OMG IDL de* `Components::StandardConfigurator`.

dans un structure d'accueil. Cette interface permet à l'appelant de spécifier un objet de configuration ou une séquence de `Components::ConfigValues`. Un utilisateur de `Components::HomeConfiguration` peut aussi supprimer la possibilité d'utilisations futures de la configuration automatique par la maison de composants.

```
module Components {
    interface HomeConfiguration : CCMHome {
        void set_configurator (in Configurator cfg) ;
        void set_configuration_values (in ConfigValues config) ;
        void complete_component_configuration (in boolean b) ;
        void disable_home_configuration () ;
    };
};
```

FIG. 3.22 – *Interface OMG IDL de* `Components::HomeConfiguration`.

3.7.4 Critique

Le fait que la phase de configuration et la phase d'utilisation d'une instance de composant soient en exclusion mutuelle est réellement important. La démarche proposée ici est donc une évolution positive. Cependant, il est à regretter que ce discours ne soit que mentionné dans la spécification et que le contenu soit plus du « il serait bon de faire de cette manière » plutôt que de l'imposer. Libre à chacun de faire une interface de configuration ou non, c'est peut être un tort de laisser tant de liberté sur un aspect aussi sensible. En effet, commencer l'utilisation d'une instance de composant non ou mal configurée revient à risquer un comportement non souhaité voir dangereux pour le système (plantage).

3.8 Critique

Le modèle abstrait annonce une forte évolution du modèle objet de CORBA 2. Il représente aujourd'hui la partie la plus aboutie de la spécification. Son pouvoir d'expression est beaucoup plus riche que le modèle objet de CORBA 2 et il permet de caractériser un type de composant dans son intégralité, c'est-à-dire pas uniquement les services fournis, mais aussi les services requis et les propriétés configurables.

Pour les propriétés configurables, l'association d'exceptions à une mauvaise utilisation des attributs accroît la possibilité de produire du logiciel fiable. Cependant, il n'y a pas de règle de projection des attributs IDL3 en opérations IDL2 pour les utiliser et gérer les exceptions, ce qui semble une erreur. En effet, seuls les ORB de type CORBA 3 pourront prendre en charge la gestion des attributs, dont les contraintes d'utilisation devront être implantées manuellement dans le code fonctionnel (l'ORB ne faisant que transmettre l'exception). Cette remarque pose aussi un problème de compatibilité : Comment est ce que les ORB de type CORBA 2 pourront interopérer avec les ORB de type CORBA 3 dès lors que des attributs seront manipulés ?

Pour ce qui est des services fournis par un type de composant, les facettes offrent au concepteur le moyen de définir des vues sur le type de composant et donc de correspondre au mieux aux besoins des divers clients de ce type. Les capacités d'introspection permettent de plus de découvrir les services offerts par un type de composant et donc de choisir quel est le meilleur point de vue par rapport à l'utilisation désirée.

Lorsqu'un « client » d'un type de composant est lui même un type de composant, les réceptacles offrent de manière satisfaisante le moyen de composer ces deux types entre eux. Ils permettent aussi de faire évoluer les connexions entre instances de composants durant l'exécution de celles-ci. Il est toutefois à regretter que la sémantique des réceptacles multiples ne soit pas mieux détaillée dans le modèle.

L'aspect communication asynchrone décrite dans le CCM ne peut être évaluée dans l'état actuel des choses. En effet, l'impossibilité de faire communiquer deux types de composants en mode asynchrone, du fait d'une erreur dans la règle de projection, ne permet pas d'expérimenter ce mode. Toutefois, d'un point de vue modèle, une fois que notre proposition (cf. annexe A) de définition des événements est mise en œuvre, le modèle événementiel s'annonce intéressant.

Du côté des points faibles du modèle abstrait, il est à regretter que l'on ne dispose pas de moyens d'introspection structurelle sur les réceptacles et les sources / puits d'événements autre que via le référentiel des interfaces. Une API d'introspection structurelle fournie par le type de composant offrirait une simplicité d'utilisation non négligeable. Une autre lacune de l'introspection est la possibilité de retrouver l'environnement d'exécution d'une instance de composant (container ou serveur). Cela permettrait de faire de la reconfiguration de l'environnement à l'exécution ou du placement à proximité d'instances de composants existantes.

Le second aspect négatif, qui n'est pas uniquement propre à ce modèle (puisque transversal au modèle abstrait et au modèle d'exécution), est le faible soutien du dogme « Configurer et utiliser sont deux actions à exécuter en exclusion mutuelle sur une instance de composants, *a fortiori* à l'aide d'interfaces différentes. » Le modèle abstrait énonce la nécessité de ce principe sans toutefois l'imposer en spécifiant, par exemple, une interface de base pour la configuration sur les instances de composants. Le modèle d'exécution aurait alors à définir l'utilisation de cette interface de manière fiable et systématique.

Enfin, il est regrettable que les ports d'un composant soit co-localisés. Si l'on reste au niveau d'un composant, cette remarque peut paraître exagérée. Cependant, si l'on se place au niveau des applications on retrouve cet aspect dans le fait que le modèle abstrait est plat (à un seul niveau). Ceci veut dire qu'il n'est pas possible de concevoir des « composites » (agrégation de composants) qui soient eux-mêmes des composants et donc manipulables de manière similaire. Si les ports pouvaient être dé-localisés, alors il serait possible de faire de l'agrégation et donc des « composites ».

Chapitre 4

Le modèle de programmation

4.1 Implantation d'un composant

4.1.1 Framework

Le modèle de programmation offre aux fournisseurs de composants logiciels le moyen de produire l'implantation d'un composant. L'objectif principal est ici de décrire les aspects non fonctionnels pour une production automatique et de ne coder que la partie fonctionnelle de l'implantation. Pour que ces deux modes de production de l'implantation fonctionnent en harmonie, le CCM inclut un *framework* : le CIF (*Component Implementation Framework*). Ce *framework* décrit les interactions entre les parties fonctionnelles et non fonctionnelles de l'implantation d'un composant, i.e. comment l'implantation fonctionnelle du composant s'intègre dans la partie non fonctionnelle.

Pour générer les squelettes de composants, squelettes qui sont étendus par rapport aux squelettes de CORBA 2, le CIF se base sur l'utilisation du CIDL (*Component Implementation Definition Language*). Les squelettes de composants automatisent la prise en charge des comportements de base des composants tels que la navigation entre facettes, l'activation des différents constituants du composant, la gestion du cycle de vie, etc. En plus du langage CIDL, le CIF repose sur le *framework* défini pour le POA (*Portable Object Adaptor*) tout en masquant la complexité de celui-ci.

```
composition entity DistributeurImpl {
    home executor MaisonDeDistributeursImpl {
        implements corbar::MaisonDeDistributeurs ;
        manages    DistributeurImpl ;
    };
};
```

Fig. 4.1 – *Exemple de déclaration CIDL simple.*

4.1.2 CIDL

Le langage CIDL (*Component Implementation Definition Language*) permet de décrire la structure de l'implantation d'un composant ainsi que son état persistant. Pour cela, un composant est considéré ici comme un ensemble d'éléments comportementaux fournis par des *exécuteurs*. Deux types d'*exécuteurs* sont définis : les *exécuteurs* de composants, qui implantent les types de composant, et les *exécuteurs* de maisons, qui implantent les types de maison de composant. L'implantation d'un composant est constituée d'un agrégat d'entités ayant des comportements et relations spécifiques. Le langage CIDL permet de décrire cet agrégat. Cet agrégat est qualifié de *composition*. En CIDL, `composition` est un méta-type qui correspond à une définition d'implantation.

Une composition, telle que présentée dans la figure 4.1, spécifie un type de maison (défini en IDL3), et donc implicitement un type de composant (puisqu'un type de maison est associé avec un unique type de composant). Associé à ce type de maison, un type abstrait de stockage est potentiellement défini (i.e. une version persistante du type de composant). Ensuite, une définition d'*exécuteur* est associée au type de maison. Celle-ci spécifie la relation entre l'*exécuteur* de maison et les autres éléments de la composition. Enfin, une composition spécifie une définition d'*exécuteur* de composant. Ce type d'*exécuteurs* peut potentiellement être segmenté, c'est-à-dire que l'implantation de l'*exécuteur* sera décomposée en plusieurs classes avec des caractéristiques (comme la politique d'activation) éventuellement différentes. Les relations entre le type de maison, le type de composant et les exécuteurs sont décrites dans la figure 4.2.

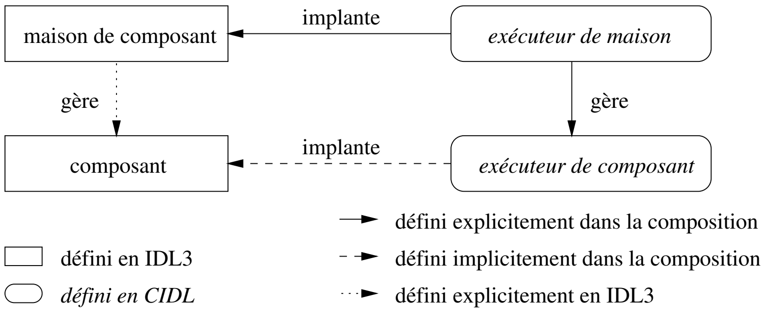

FIG. 4.2 – *Relations entre type de maison, type de composants et exécuteurs.*

4.1.3 Projection

La compilation des descriptions IDL3 et CIDL (voir figure 4.3) décrivant un composant fournit plusieurs éléments. Tout d'abord, les interfaces clientes des composants sont exprimées en IDL2. Un descripteur XML regroupant les contraintes techniques à respecter au moment du déploiement ainsi que le squelette prenant en charge la partie non fonctionnelle d'un composant sont générés.

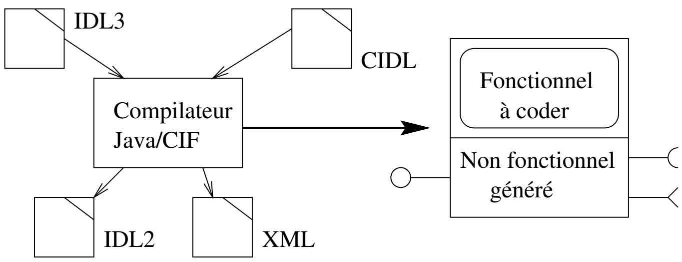

FIG. 4.3 – *La génération des squelettes de composants.*

Le squelette d'un composant implante la projection en IDL2 de la déclaration d'un type de composant. Il offre l'introspection structurelle, c'est-à-dire la découverte des ports, ainsi que la gestion de la connectique entre composants : connexion et déconnexion d'un réceptacle, abonnement et désabonnement à une source d'événements, etc. Le squelette prend aussi en charge le cycle de vie d'une instance de composant, son identité, la persistance, etc.

4.1.4 Critique

En l'état actuel, le CCM ne contient pas la spécification de l'interface entre la partie fonctionnelle d'un composant et sa partie non-fonctionnelle. Des exemples illustrent le propos, mais ne donnent un ensemble d'interfaces à fournir / utiliser. Cette lacune ne réduit pas *a priori* la possibilité de produire une implantation du CCM. Cependant si chaque fournisseur définit sa propre API, alors les développeur auront du mal à changer de produit. De plus, une implantation fonctionnelle ne pourra être utilisée avec deux projections venant de fournisseurs différents. Les implantations de composants ne sont alors pas portables.

4.2 Modèle de programmation des containers

4.2.1 Vue d'ensemble

La figure 4.4 présente les différents éléments du *framework* de containers. Les différentes interfaces sont présentées plus en détail dans la suite de ce document.

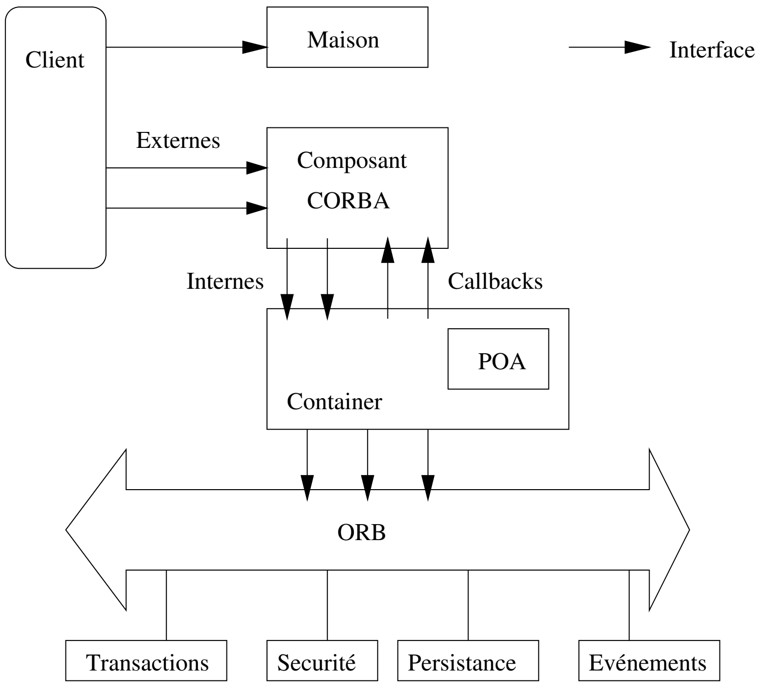

FIG. 4.4 – *L'architecture du modèle de programmation des containers.*

4.2.2 Types externes

Les types externes d'un composant représentent le contrat entre le développeur et le client d'un composant. Ces interfaces sont de deux types : les interfaces de maisons et les interfaces d'applications (voir chapitre 3). Les types externes sont définis à partir de la projection en IDL2 de l'interface IDL3 du composant. Ces interfaces IDL2 peuvent aussi être stockées dans le référentiel des interfaces et ainsi être exploitables par le client.

4.2.3 Types de containers

Le type de container définit l'API du *framework* entre le composant et le container. Il existe deux types de base. Le type transient qui correspond à l'API de base. Le type persistent qui est une extension de l'API.

4.2.4 Types d'implantations de container

Le type d'implantation du container spécifie les patterns d'interaction requises entre le container, le POA et les services CORBA. Trois patterns d'interaction sont définies :

- Les références **sans état** sont volatiles et le POA supporte tout ObjectId.
- Les références **conversationnelles** sont volatiles et le POA est dédié à un ObjectId.
- Les références **durables** sont persistantes et le POA est dédié à un ObjectId.

Le type d'implantation de container est spécifié en CIDL. Il est utilisé pour créer ou sélectionner un container de composant lors du déploiement de composant.

4.2.5 Les catégories de composants

Quatre catégories de composants sont actuellement définies. Ces quatre catégories sont présentées dans la table 4.1.

Composant	Container (impl)	Container (type)	Type externe
Service	Sans état	Volatile	Sans clé
Session	Conversationnel	Volatile	Sans clé
Processus	Durable	Persistant	Sans clé
Entité	Durable	Persistant	Avec clé

TAB. 4.1 – *Définition des catégories de composants.*

Service

Les composants « service » sont des composants sans état et sans identité. Leur durée de vie correspond à la durée de traitement requis par l'invocation d'une opération. Le design pattern utilisé par le client est la « fabrique ». Ce genre de composant peut être créé à la demande, ou être géré sous forme de pool de composants.

Session

Les composants « session » sont des composants avec un état volatile, et une identité qui n'est pas persistante. La durée de vie d'un tel composant est une interaction (suite d'invocation d'opérations) de la part du client. Ce type de composant est idéal pour implanter des itérateurs, des IHM ou tout objet similaire. Le design pattern utilisé est aussi la « fabrique ».

Processus

Les composants « processus » sont des composants avec un état persistant géré par le composant ou par le container. Le fait que cet état soit persistant n'est pas visible pour le client. L'identité du composant est persistante et gérée par le composant. Elle est rendue visible au client via des opérations définies par l'utilisateur. Le pattern utilisé est toujours la « fabrique ». Ce type de composant est utilisé pour modéliser les objets qui représentent des processus métiers, plutôt que des entités.

Entité

Les composants « entité » sont des composants semblables aux composants processus pour la persistance. Néanmoins, cette persistance est visible pour le client. Ces composants ont une identité qui est visible via une clé primaire (déclarée dans la définition de la maison de composants). Les patterns utilisés dans le cadre des composants entités sont la « fabrique » et la « recherche ».

4.3 Environnement de programmation serveur

4.3.1 Fabrique

Les fabriques d'instances de composants sont, comme nous l'avons déjà vu, les maisons de composants. Une fabrique par défaut est implantée automatiquement. Son implantation peut être générée par des outils en utilisant les informations de la définition IDL3 du composant. Les fabriques spécialisées doivent être implantées par le développeur du composant. Les opérations de fabriques sont principalement utilisées par les clients, mais peuvent l'être aussi par l'implantation du composant (pour localiser sa fabrique par exemple).

4.3.2 Activation

Les composants CORBA s'appuient sur les mécanismes d'activation automatique du POA pour façonner le comportement du composant en exploitant l'information contenue dans son descripteur de déploiement. Une fois les reférences rendues publiques, les invocations sont routées par l'ORB vers le POA qui a créé la référence, puis du POA vers le composant. Cela permet au container de contrôler l'activation et la désactivation d'un composant, d'appliquer les politiques définies dans le descripteur et d'invoquer les interfaces de callback du composant si nécessaire.

4.3.3 Gestion de la durée de vie des servants

Les servants sont des objets (de niveau langage de programmation) que le POA utilise pour dispatcher les requêtes basées sur l'`ObjectId` contenu dans la clé de la référence. Le modèle de programmation des composants CORBA permet de gérer efficacement la mémoire associée à ces isntances de composants. Pour cela le programmeur doit faire certains choix de conception :

- Choisir le type de container.
- Choisir le type d'implantation de container.
- Sélectionner une politique de durée de vie du servant. Il en existe quatre :
  méthode Le composant est activé à chaque requête, et désactivé aussitôt après.
  transaction Le composant est activé pour la première requête de la transaction et désactivé à la fin de la transaction.
  composant Le composant est activé à la première requête et ne sera désactivé que si son implantation le demande (requête au container)
  container Le composant est activé à la première requête et ne sera désactivé que lorsque le container déterminera qu'il doit l'être.
- Implanter l'interface de callback associée à ces choix si nécessaire.

4.3.4 Interfaces de programmation du serveur

Les interfaces de programmation du serveur (section 4.2) sont un ensemble d'interfaces utilisées, ou fournies, localement par le développeur de serveurs. La spécification des composants CORBA introduit pour cela un nouveau type d'interfaces : les interfaces *locales*. Ce nouveau méta-type est utilisé pour définir des interfaces de programmation d'objets ayant des contraintes d'utilisation liées à la localité.

La syntaxe des interfaces de ces objets est similaire aux autres interfaces d'objets CORBA, mais le type résultat ne peut être ni enpaqueté, ni invoqué à distance. Le méta-type `local` est prévu pour définir de telles interfaces en faisant abstraction du langage de programmation utilisé pour l'implantation et en conservant les avantages de l'uniformité de CORBA (voir figure 4.5). Le type de base `localBase` est le `any` des interfaces classiques. Tout comme il existe une méthode pour retrouver les références des objets notoires sur un bus, la méthode `ORB::resolve_local` est utilisée pour retrouver des références d'objets locaux.

```
local interface CCMContext {
    Principal get_caller_principal () ;
    CCMHome get_CCM_home () ;
    // ...
};
```

FIG. 4.5 – *Exemple d'interface locale : extrait de* `Components::Basic::CCMContext`.

Interfaces de composant

Tout composant dispose de trois types d'interfaces locales.

- Les interfaces de type **internal** sont utilisées par le développeur et fournies par le container pour aider à l'implantation du comportement du composant. Un ensemble de base de ces interfaces est défini par le type de container. Cet ensemble de base est ensuite augmenté de manière spécifique pour la catégorie de composant développée. Ces interfaces sont des interfaces locales.

– Les interfaces de type **external** sont utilisées par le client et fournies par le développeur de composant.
– Les interfaces de type **callback** sont utilisées par le container et implantées par le composant (par génération ou directement) pour le déploiement du composant au sein du container. Tout composant implante une interface de ce type. Elle est déterminée par la catégorie de composant. Cette interface est en général générée par le compilateur, mais peut nécessiter des modifications du développeur.

Interfaces de container

Comme nous l'avons déjà vu, il existe deux types de containers (volatiles et persistants). Pour chaque type de container un ensemble d'interfaces est défini. Il existe des interfaces communes aux deux types de containers et des interfaces propres à chacun d'eux. Les interfaces communes sont des interfaces de base qui sont étendues par les interfaces propres à chaque type de composant. Elles permettent de supporter tout type de composant.

4.4 Modèle de programmation client

Un client interagit avec un composant CORBA au travers des deux formes d'interfaces externes : une interface de maison et une ou plusieurs interfaces d'application. Les interfaces de maison permettent au client d'avoir accès à des références d'interface d'application implantées par le composant. Le client peut utiliser les maisons avec deux design patterns : les fabriques et les éléments de recherche. Le choix se fait par rapport à la présence de clé primaire dans la définition de la maison de composant. Si une clé est définie, les deux patterns sont disponibles, sinon seule une fabrique peut être utilisée.

4.4.1 Type de client

Il existe deux types de clients pour les composants : les clients conscients d'utiliser des composants et les autres (un composant s'utilise comme un objet CORBA).

Clients orientés composants

Ce sont des clients définis en utilisant les extensions du langage IDL présentés précédemment. Ces clients sont capables de profiter des fonctionnalités des composants qui ne sont pas supportées par les objets CORBA ordinaires. Un tel client peut interagir avec le composant au travers de plusieurs interfaces :
– l'interface associée avec la définition du composant en IDL3,
– les interfaces « supportées » par la définition du composant (clause `supports`),
– les interfaces offertes par le composant (facettes),
– l'interface de maison qui supporte les opérations de fabrique et de recherche.

Clients classiques

Les composants CORBA peuvent aussi être utilisés par des clients « standards ». Ces clients ne peuvent utiliser qu'une interface du composant (l'interface supportée) et ne sont pas capable de naviguer entre les différentes facettes.

4.4.2 Patterns

Fabrique

Le pattern fabrique permet de créer une nouvelle instance de composant. Elle est différente selon le type de client.

Conscient La maison de composant offre une opération de création d'instance. Il en existe toujours une par défaut qui retourne une référence de type Components::CCMObject. Le développeur peut aussi offrir des opérations spécifiques. Pour trouver la fabrique, le client peut utiliser un objet de type `Components::HomeFinder`.

Inconscient Le pattern fabrique n'est utilisable que si les interfaces supportées disposent d'opération applicatives. Pour trouver la fabrique, la maison ou un objet de recherche de fabrique (`Components::FactoryFinder`) doit être enregistrée dans le service de nommage ou auprès d'un trader. Ensuite l'opération de création peut être appelée.

Recherche

Le pattern de recherche ne peut s'utiliser que pour des instances de composant de type entité. Pour les deux types de clients, la recherche peut s'effectuer à partir du service de nommage si l'instance a été enregistrée. Dans le cas d'un client conscient des composants, le client peut utiliser un objet de recherche de type `Components::HomeFinder`.

4.5 Critique

Le modèle de programmation répond bien au souci de produire rapidement des implantations de composants : décrire des besoins est plus simple qu'écrire du code. En second lieu, il répond bien aussi au problème de produire des composants logiciels de qualité : du code produit automatiquement réduit le risque de bugs. Le CIF définit un cadre strict pour la production de composants, offrant ainsi un atout dans la transition d'une production artisanale vers une production industrielle de composants.

Dans l'état actuelle des choses, le CIDL n'est pas encore totalement spécifié. Ensuite, les règles de projection du CIDL vers les langages de programmation ne sont pas encore définies. Enfin, l'absence de plates-formes permet uniquement d'estimer le CIDL comme une bonne approche, mais ne permet pas encore de valider les choix qui ont été faits.

Chapitre 5

Packaging et déploiement

5.1 Introduction

Le modèle de déploiement est le second apport majeur des composants CORBA. Il offre des moyens pour automatiser la diffusion et la mise en place d'une application distribuée. Il contribut à accroître la réutilisabilité d'entités logicielles en facilitant l'utilisation et l'intégration de composants existants.

Plusieurs types de packages de déploiement ont été abordés dans la figure 2.2 page 6. Ces différents packages sont présentés tout au long de ce chapitre. Dans le cadre des composants CORBA, un package est une entité regroupant un ensemble d'implantation de composants ainsi qu'un (ou plusieurs) descripteurs, éventuellement regroupés dans un fichier archive. Les descripteurs de packages sont écrits en OSD (Open Software Descriptor) qui est un langage de description basé sur XML. Ce langage a tout d'abord été introduit au W3C par Marimba et Microsoft, puis il a été repris par l'OMG et étendu pour les besoins propres aux composants CORBA. Des exemples de descripteurs sont présentés dans l'annexe B.

5.2 Packages de déploiement

5.2.1 Package d'entité logicielle

Un package d'entité logicielle est représenté par un descripteur et un ensemble de fichiers regroupant l'implantation de l'entité. Ces différents éléments sont regroupés dans un fichier au format « ZIP ». Ce type de package peut être utilisé pour déployer tout package de logiciel. Dans le contexte courant, ce package regroupe les implantations de composants CORBA.

Un descripteur décrit le contenu du package. Il offre des informations d'ordre général relatives à l'entité logicielle (auteur, description, interface OMG IDL, etc) ainsi qu'une description des implantations (nom, système d'exploitation cible, langage d'implantation, dépôt de code, etc). Ce descripteur regroupe aussi les dépendances du composant par rapport à l'environnement.

5.2.2 Descripteur de package de composants

Ce descripteur décrit un composant. Il spécifie les caractéristiques de ce composant spécifiées pendant la phase de conception et la phase de développement. Ce descripteur est en partie généré par un compilateur CIDL, et en partie modifié par un outil de packaging (pour les notions de persistance, transaction, threading,. . .).

Description des fonctionnalités

Ce descripteur est utilisable par un outil de conception pour afficher les informations sur le(s) composant(s). Il décrit la structure d'un composant : héritage, interfaces supportées, ports,... Les différentes interfaces du composant sont décrites et référencées par leur RepositoryId. Ce descripteur permet à un outil de connecter les composants entre eux lors de la phase de déploiement.

Informations de déploiement

Lors du déploiement, ce descripteur est aussi utilisé pour déterminer le type de structure d'accueil nécessaire au composant, et pour fournir aux structures d'accueil des informations sur le composant. Il indique les besoins transactionnels du composant, ainsi que la qualité de service des ports. Des informations concernant les capacités multithread du composant et leur mode d'utilisation, ainsi que la manière d'achever la phase de configuration sont incluses dans ce descripteur.

5.2.3 Assemblage de composants

Un package de composant permet de déployer un composant « seul ». Un package d'assemblage permet de déployer de manière simple des composants dépendants les uns des autres. Il offre un patron (ou template) pour instancier un ensemble de composants et les connecter les uns aux autres. Un tel package regroupe un descripteur, un ensemble de packages de composants et un ensemble de fichiers de propriétés. Le fichier d'assemblage est une archive (ZIP) regroupant le descripteur, les archives de composants et les fichiers propriétés.

Descripteur d'assemblage de composants

Il décrit l'assemblage des composants, c'est-à-dire les éléments de description des composants, des connexions et du partitionnement. Les instances de composants sont connectées au travers des déclarations de provides / uses et emits / consumes. Le descripteur référence les fichiers de chaque composant et décrit les connexions entre composants avec des assertions telles que `connectinterface` ou `connectevent`. Ce descripteur définit des regroupements logiques pour les composants, regroupements qui seront projettés au déploiement sur des sites physiques. Ce descripteur va servir de base au processus de déploiement.

Descripteur de propriétés

Il décrit le paramétrage des composants et des maisons de composants. Il est utilisé pour configurer un composant ou une maison : positionnement des attributs. Il fournit des propriétés par défaut qui peuvent être modifiées par l'utilisateur. Les informations contenues dans ce type de descripteur seront utilisées par les objets de configurations (voir section 3.7.2).

5.3 Déploiement

Le déploiement se fait à l'aide d'un outil de déploiement fourni par le fournisseur de l'ORB, ou d'outils. Cet outil déploie les composants et assemblages sur les sites cibles. Cette application de déploiement est une application cliente qui utilise les services d'objets présents sur les sites d'installation pour installer, instancier, puis configurer les composants et les connexions. Les quatre étapes de base pour le déploiement sont :

1. Définir et choisir les sites d'installation.
2. Installer les implantations (si pas déjà présentes) où cela est nécessaire.
3. Instancier les maisons puis les composants.
4. Connecter les composants (dans le cas des assemblages).

Ces opérations de déploiement sont mises en œuvre au sein de plusieurs objets définis[1] dans la spécification.

5.3.1 Processus de déploiement

Le processus de déploiement présenté dans la spécification se découpe en treize points (de deux ordres). Ces différents points sont illustrés dans la figure 5.1 et discutés ci-dessous.

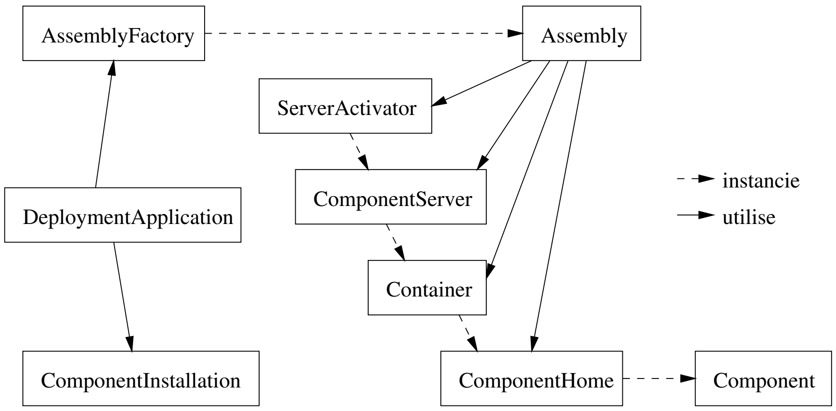

FIG. 5.1 – *L'architecture du processus déploiement.*

1. L'application de déploiement, qui dialogue avec les objets de type AssemblyFactory et ComponentInstallation, interroge l'utilisateur pour connaître les sites cibles du déploiement. Une fois les sites connus, elle génère un descripteur d'assemblage spécialisé à partir du descripteur de l'application et des informations fournies par l'usager. Ce descripteur sera par la suite utilisé par l'instance d'Assembly relatif à ce déploiement.
2. Les implantations de composants sont installées sous forme d'archives sur les sites où elles sont requises. L'outil de déploiement utilise pour cela les instances de ComponentInstallation disponibles sur les sites concernés.
3. Une fabrique d'assemblage est utilisée pour créer une instance d'Assembly sur un site (une seule pour toute l'application). Lors de la création, l'URL du descripteur d'assemblage spécialisé est fournie pour que l'instance d'Assembly puisse l'utiliser.
4. Le descripteur d'assemblage est utilisé comme une « recette » pour déployer l'application. En se basant sur ce descripteur, l'instance d'Assembly crée les maisons et les instances de composants sur leurs sites respectifs.

1. Ou plus précisément esquissés.

Pour créer une instance de composant, l'instance d'Assembly doit créer au préalable un serveur de containers, puis un container dans ce serveur, puis une maison de composant au sein du container, puis enfin l'instance de composant.

(a) Tout d'abord, l'instance de ServerActivator présente sur le site cible est utilisée pour créer un serveur de container.

(b) Ensuite, l'instance de serveur de container est utilisée pour instancier un container. Lors de cette instanciation, la configuration du container est fournie : identifiant du container et la liste des propriétés du container.

(c) L'interface du container fournit une opération pour installer une maison. Une fois l'implantation de la maison installée, le container en crée une instance et retourne sa référence. Pour créer cette maison, le container va devoir charger en mémoire la librairie dynamique contenant son implantation. Pour cela, il va interroger l'instance de ComponentInstallation afin de connaître quelle librairie utiliser, i.e. quelle est la librairie implantant le type de composant.

(d) L'instance d'Assembly va utiliser la maison de composants pour créer une instance du composant. Cette opération va retourner la référence de base de l'instance de composants, référence qui sera du type `Components::CCMObject`.

(e) Si c'est nécessaire / prévu, un objet de configuration est appliqué sur la nouvelle instance de composant.

5. Une fois que tous les composants sont installés, l'instance d'Assembly connecte les différentes instances de composants relatives à l'assemblage en se basant sur l'information de la partie *connection* du descripteur d'assemblage. Cette action est effectuée en utilisant l'opération `connect()` sur les réceptacles appropriés[2].
6. Si toutes les étapes précédentes se sont bien déroulées, alors l'instance d'Assembly invoque sur chaque instance de composant faisant parti de l'assemblage, l'opération `configuration_complete()` pour notifier aux instances de composants que leurs connexions initiales ont été effectuées.

Points d'entrée de composants

Chaque package contient (au moins) une implantation de composant. Le code de cette implantation est un module dynamiquement chargeable. Pour instancier un composant, il faut disposer d'une maison. La maison de composant est créée en utilisant un point d'entrée de l'implantation du composant. Le nom de cette méthode et sa signature sont standard (donc toujours valides). Par contre, le nom de la classe fournissant cette opération varie d'un package à l'autre. Le nom complet de la classe est donc contenu dans le descripteur d'assemblage pour permettre à l'instance d'Assembly de connaître le nom exact du point d'entrée. L'implantation des points d'entrée est spécifique à chaque langage et dispose donc de règles pré-définies. La figure 5.2 présente les deux règles de nommage pour les langages Java et C++. Le point d'entrée sera `fr.lifl.cim.corbar.MaisonDistributeurFactory.create` dans le cas de Java et tout simplement `createDistributeurHome` dans le cas de C++.

Le type `HomeExecutorBase` représente le type de base de l'implantation fonctionnelle des maisons de composants, c'est-à-dire l'interface de l'*exécuteur* de maison tel que définie dans la section 4.1.2. Ce choix pose deux problèmes. Premièrement, l'interface `HomeExecutorBase`

2. Ceci répond à la connexion des ports en mode synchrone, mais la specification reste floue sur la connexion des ports de type asynchrones.

```
// Java
package fr.lifl.cim.corbar ;
public class MaisonDistributeurFactory {
    public HomeExecutorBase create () {
        return new MaisonDistributeurImpl () ;
    }

// C++
extern "C" {
    HomeExecutorBase* createMaisonDistributeur ()
    {
        return new MaisonDistributeurImpl () ;
    };
};
```

FIG. 5.2 – *Points d'entrées pour la création des maisons de composants en Java et en C++.*

n'est pas définie dans la spécification. Deuxièmement, le choix d'avoir un point d'entrée qui retourne autre chose qu'un `Components::CCMHome` tend à rendre l'utilisation des maisons de composants non standard. En effet, doit on faire un *narrow* sur la référence retournée pour la transformer en `Components::CCMHome` ou bien, doit on « encapsuler » la référence retournée avec le mécanisme de *tie* offert par CORBA ? Cet aspect fait douter de la réelle portabilité des maisons de composants entre containers, ce qui est pourtant souhaitable afin de permettre une utilisation de composants / containers provenant de fournisseurs différents.

Chapitre 6

Le modèle d'exécution

Les informations présentées ici sont plus à destination des concepteurs de structure d'accueil qu'à destination des utilisateurs.

6.1 Caractérisation des containers

6.1.1 Définition

Un container est un environnement d'exécution pour une instance de composant CORBA. Cet environnement est implanté comme un serveur d'applications et/ou comme une plateforme de production de telles applications. Toutes les instances de composants, quel que soit leur type, sont créées et gérées par un container. Une instance de composant ne peut pas exister sans être supportée par un container. D'autre part, un type de container ne peut accueillir qu'un unique type de composant pour lequel il a été conçu. Deux types de containers ont été définis : des volatiles et des persistants.

6.1.2 Framework

Un container offre un ensemble standard de services aux instances de composants qu'il héberge. Les composants et les maisons de composants sont déployés dans des containers grâce à l'utilisation d'outils spécifiques. Ces outils permettent, entre autre, de générer les extensions utiles au container pour les accueillir. Une API de programmation destinée à faciliter le développement d'applications CORBA est spécifié dans le cadre du *framework* de containers (voir section 4.2).

6.2 L'architecture des containers

Un serveur de composants est un processus qui contient un nombre arbitraire de containers de composants. Chaque type de container est associé à un type d'implantation de container qui implique un mode d'interaction avec le POA et l'ORB. Chaque type de container inclut un POA spécialisé géré par un manager de container. Ce manager de container est responsable de la création et de la destruction de containers. Ce sont des fabriques de containers qui sont créées lors de la phase d'installation et de déploiement. La création du container, du POA, la configuration des politiques, la liaison avec les services CORBA,... sont définis à partir du descripteur de déploiement.

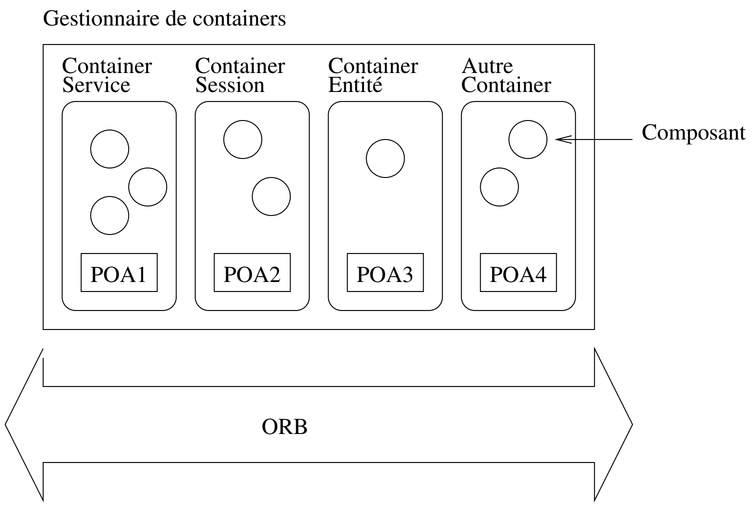

FIG. 6.1 – *L'architecture d'un serveur de composants.*

6.2.1 Le Portable Object Adapter

Un POA est utilisé pour créer des références qui seront exportées vers des clients. Il est aussi utilisé pour gérer l'activation d'instances de composants lorsque des requêtes sont reçues. Le fait de créer un container implique, en général[1], la création d'un POA que ce container utilisera. Cette création se fait en quatre étapes :

1. Création de la politique pour le POA définie par le type d'implantation du container.
2. Création du POA pour la politique définie.
3. Assignation d'un ServantManager au POA.
4. Activation du POA.

Pour l'installation d'un composant, une phase d'initialisation (handshake) est nécessaire entre le composant et le container. Cette phase permet l'échange de références entre l'implantation du composant et le container.

6.2.2 Les services de CORBA

Le type de container définit les services CORBA nécessaires au container. Les types de containers de la spécification peuvent utiliser les services de sécurité, de transaction, de persistance, de notification et de nommage. Lors de la phase de création d'un container, l'accessibilité à ces services doit être établie. Cela veut dire deux choses : d'une part trouver le service sur le bus, et d'autre part assurer la configuration initiale requise par un service.

Les types de containers définis dans la spécification offrent des *frameworks* pour le déploiement de composants CORBA. Chaque *framework* gère les interactions avec l'ORB, le POA et les services de CORBA selon la demande du composant. Cela permet au développeur de se concentrer sur la logique de l'application et d'oublier un peu les problèmes techniques.

1. L'absence de création de POA n'est pas normalisée.

6.2.3 Les catégories de container

Comme nous l'avons déjà vu dans la section 4.2.5, il existe quatre catégories de composants. C'est pourquoi quatre catégories de container spécialisées sont définies (plus un container vide) pour une utilisation dans le contexte du CCM. D'autre part, pour des raisons de compatibilité, les deux types de containers EJB (session et persistant) sont aussi inclus dans la spécification CCM.

Empty

Le container vide n'offre aucun *framework* de développement. Il offre directement au développeur les fonctionnalités de CORBA, ainsi que les fonctionnalités des composants (interfaces multiples, packaging, déploiement).

Service

Le container service implante l'environnement d'exécution des composants services. Un container de ce type n'utilise pas d'`ObjectId` (pas de notion d'entité). Il crée une référence de composant selon la demande du client. Dans le cas de tel container, il n'y a pas de contrainte de co-localité composant/maison de composants. Les références de composant et d'interfaces sont créées par le POA.

Session

Le container session implante l'environnement d'exécution des composants session. Un container session gère des ObjectIds sans intervention du développeur. L'implantation du container doit juste garantir l'unicité des ObjectIds. Les références d'objets et les instances de composants sont créées à la demande. Un tel type de container exige que les instances de composants et les fabriques soient colocalisées.

Processus

Le container processus implante l'environnement d'exécution des composants processus. Il crée et gère des ObjectIds uniques qui définissent la localisation persistante de l'état externe. Soit l'état est déclaré et géré par le container, soit l'état est géré par l'application. Les maisons créent des références et les exportent vers les clients. Les instances sont créées à la demande lorsqu'une référence est utilisée pour invoquer une opération. Les fabriques peuvent être utilisées par le client et par l'implantation d'interfaces de composants. Les références d'objets pour les interfaces du composant sont créées par le POA qui supporte le container de ce composant.

Entity

Le container entité implante l'environnement d'exécution des composants entité. Il offre des opérations qui associent une clé à un identifiant persistant (pid). A une instance de composant est associée une clé unique. Le container traduit les pids en ObjectIds. La création d'une référence ou d'une instance implique la création d'une entrée dans une « base de donnée ». Les fabriques peuvent être utilisées par les clients et par l'implantation d'interfaces fournies par le composant.

6.3 Critique

Comme critique principale il est essentiellement possible de souligner que certains aspects ne sont que partiellement abordés. En effet, aucune implantation du modèle d'exécution n'est disponible donc il n'a pas été possible de le tester. Pour ce qui est des points qui semblent sous-spécifiés, la majorité des interfaces relatives aux containers ne sont que mentionnées dans la spécification, aucune n'est réellement définie donc il n'y a pas d'interfaces standardisées. Ceci pose un gros problème : comment peut on déployer un composant provenant de chez X dans un container provenant de chez Y ? Découlant de ce point faible, la phase de handshake (au moment du déploiement et de l'instanciation) entre un composant et un container n'est pas, non plus, standardisée.

Enfin, l'idée que l'on peut se faire aujourd'hui des containers, c'est qu'ils seront statiques et figés. En effet, le nombre de containers définis est figé et l'ensemble des propriétés non fonctionnelles supportées par un container est statique et non extensible. Disposer de containers adaptatifs serait cependant un atout majeur. Il serait possible de fournir simplement des nouveaux types de containers ou d'ajouter des services de base à un container. De plus, si les containers étaient définis comme des assemblages de composants il serait alors possible de déployer des containers dans les serveurs de containers en fonction des besoins des applications.

Chapitre 7

Méta-modèle de composants

La spécification du CCM définit un méta-modèle, le méta-modèle de composants, basé sur UML et qui est fourni avec des projections vers le MOF (Meta-Object Facility). L'ensemble du méta-modèle est assez volumineux, c'est pourquoi seule la partie concernant les extensions d'IDL est succintement présentée dans la figure 7.1.

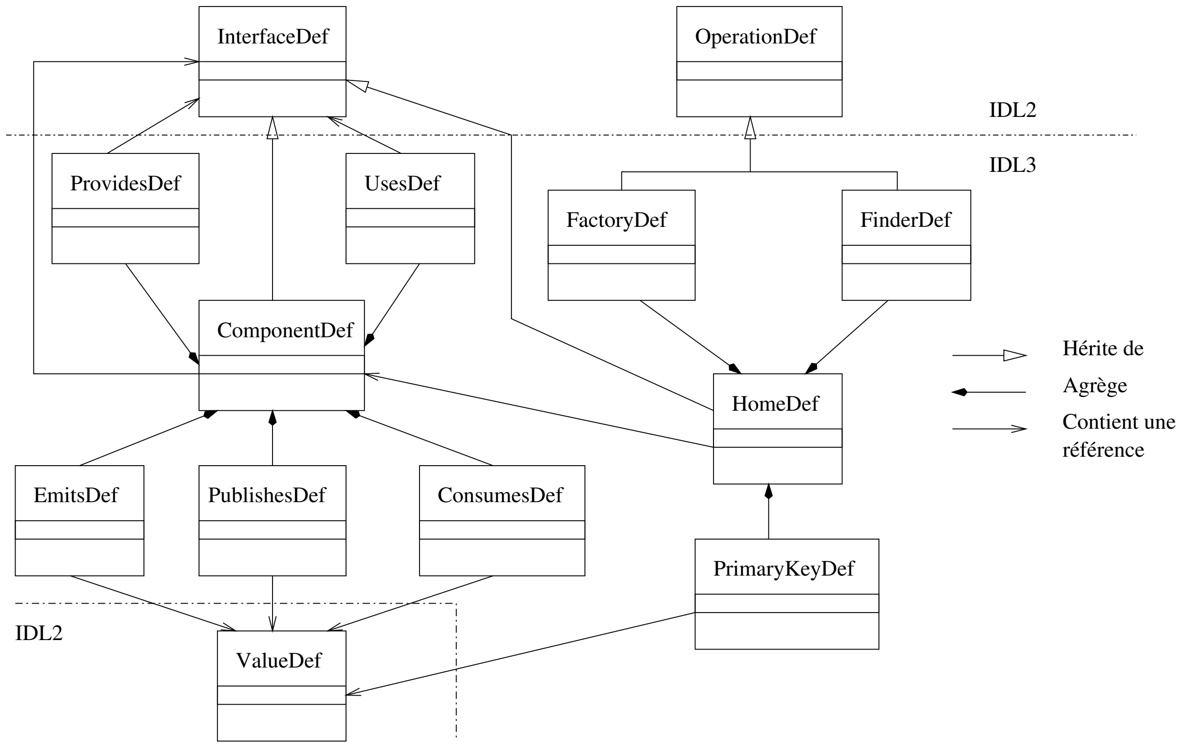

FIG. 7.1 – *Méta-modèle de composants : les extensions à IDL2.*

Cette partie du méta-modèle définit les extensions à IDL2 propres au CCM. Ces extensions regroupent la définition du méta-type `component` ainsi que les différents types de ports (facettes, réceptacles, sources et puits d'événements). Toutes ces informations seront désormais contenues dans le référentiel des interfaces (compatible avec le CCM). L'intérêt de ces extensions dans le référentiel des interfaces est la possibilité de manipuler les concepts du CCM lors de l'instrospection d'une instance de composant et non pas de manipuler uniquement leur projection en IDL2. Sans ces extensions, l'introspection se résumerait à exploiter des *patterns* de nommage comme dans le cas des JavaBeans [Ham97].

Dans un contexte dynamique, par exemple en utilisant un langage de script tel OMG IDLscript [OL00], ces nouvelles fontionnalités sont donc importantes pour manipuler correctement des composants. D'autre part, le déploiement d'applications étant très dynamique (recherche par type sans nécessairement connaître les implantations) dans le cadre de CESURE, ces fonctionnalités sont essentielles pour nous permettre de déployer et connecter des instances de composants dont la connaissance est *a priori* limitée.

Chapitre 8

Synthèse

8.1 A propos du CCM

Comme c'est toujours le cas à l'OMG, le CCM représente une réflexion menée en commun par différents acteurs du paysage informatique mondial (fournisseurs et utilisateurs) pour un modèle de composants distribués multi-fournisseurs et potentiellement hétérogènes. Globalement, le CCM représente un progrès par rapport aux autres modèles de composants industriels tels que COM/DCOM [Gri97] et EJB [MH99], cependant il n'est pas encore parfait.

8.1.1 Les apports

Les apports de la spécification des composants CORBA sont indéniables. Tout d'abord, ce modèle offre une vue globale du processus de production des applications réparties. Pour chaque étape du cycle, le CCM apporte des réponses : IDL3 pour les concepteurs, CIDL et CIF pour les développeurs, OSD pour les architectes et pour le déploiement, ainsi qu'une première version de l'infrastructure d'exécution.

Ensuite, le CCM fournit une multitude de moyens pour décrire. Ces moyens de description s'appliquent aussi bien à la définition des services fournis, comme c'était déjà le cas dans les modèles à objets, mais aussi et surtout la spécification des pré-requis aussi bien fonctionnels que techniques. Une fois les différents éléments spécifiés, l'information générée servira par la suite lors de l'utilisation des moyens d'introspection sur les composants.

Enfin, le CCM permet de se focaliser sur les aspects fonctionnels des composants logiciels. La complexité du système est masquée derrière des interfaces simplifiées et prise en charge d'une part, par les containers, et d'autre part, par les parties du composant qui sont automatiquement générées. Cette génération est possible précisément grâce aux moyens de description sus-cités, qui permettent de ne pas programmer les aspects non-fonctionnels, mais de les décrire et donc de les générer.

8.1.2 Les points faibles

Le plus gros défaut à l'heure actuelle est que cette spécification est très récente. Cela veut dire d'une part qu'il est difficile de prendre du recul par rapport aux propositions, et d'autre part qu'il n'y a pas encore d'expérimentation des composants CORBA. De plus, l'objectif de la spécification est ambitieuse et on ne sait pas encore si toutes les propositions sont réalisables. Pour certains points, nous sommes déjà certains que les propositions ne sont pas bonnes et essayons de contribuer lorsque cela est possible.

D'un point de vue plus « académique », les composants CORBA sont monolithiques, les ports sont co-localisés et ne peuvent être répartis dans différents containers. Par opposition, nous regrettons l'absence de possibilité d'agrégation des composants : un assemblage de composants n'est pas un composant, et ne peut donc être manipulé comme tel. La définition des ports est statique et aucun port ne peut être ajouté dynamiquement sur un composant. Cette remarque recoupe un peu l'absence de capacité d'agrégation sur les composants, elle insiste plus spécifiquement sur l'absence d'agrégation dynamique.

Ensuite, le CIF est un *framework* uniquement technique. Le CCM ne fournit pas de *framework* de conception des composants. Ce manque soulève quelques questions : Est ce facile de développer un composant ? Dans quel cas est ce que je dois utiliser un composant, dans quel cas est ce que je continue à utiliser un objet ? Est ce que tout doit devenir composant ? Est ce que tout **peut** devenir composant ?

Enfin, le CCM semble bien définir comment concevoir des composants logiciels pour les applications distribuées. Cependant, la spécification ne donne aucune information sur comment produire, à l'aide de ces composants, des applications distribuées. Aucune règle ou méthodologie n'est esquissée.

Malgré ces quelques points noirs, le CCM s'annonce aujourd'hui comme un des modèle les plus complet et fera certainement partie des modèles de composants industriels qui compterons dans les années à venir. Il ne sera pas la solution clé en main des deux années à venir, mais il est souhaitable que ce modèle et ses outils associés soient prêt dans quelques années lorsque les acteurs de l'industrie du logicielles se rendront compte des limites inhérentes aux composants de type COM et EJB, que l'on pourrait qualifier de composants de première génération.

8.2 Les travaux en cours au LIFL

Notre objectif au sein de l'équipe GOAL n'est pas d'implanter dans son intégralité la spécification du CCM pour en faire une plate-forme complète et industrielle. Notre volonté consiste à implanter les aspects du CCM qui nous intéressent comme support pour nos travaux de recherche afin de mener à bien nos expérimentations. Les points d'actualités dans notre démarche sont les suivants :

- le référentiel des interfaces pour la nouvelle version du langage OMG IDL (IDL3) (une première implantation est déjà réalisée) ;
- un compilateur IDL3 pour charger l'IDL3 dans le référentiel des interfaces, produire l'IDL2 associé, ainsi que pour générer les squelettes de composants pour les langages Java et C++ (une première version est déjà réalisée) ;
- un outil de packaging de composants CORBA (en cours de réalisation) ;
- un langage de description des applications à base de composants ainsi qu'une démarche de conception pour construire des applications (en cours de conception) ;
- une machine de déploiement prenant en charge le processus défini par l'OMG, mais permettant d'étendre ce dernier pour répondre au mieux à un plus grand nombre de besoins et se basant sur l'utilisation du langage sus-cité (en cours de conception) ;
- la définition de containers qui se présentent comme des composites, permettant de composer et déployer les containers selon les besoins des applications (en cours de conception).

Chapitre 9

Conclusion

Par rapport aux objectifs de CESURE, le CCM représentait au moment de faire le choix d'un modèle de composants, la meilleure réponse pour ne pas avoir à définir un nouveau modèle spécifique à notre contexte. Toutefois, le CCM ne répond pas à tous les besoins de CESURE.

Pour ce qui est de la production, du développement et de l'utilisation de packages logiciels, le CCM semble un bon choix. Il nous sera toutefois peut être nécessaire de spécialiser la génération automatique des parties non-fonctionnelles pour notre contexte, spécialement dans le cas des petits devices avec peu de mémoire tels que les PDA et téléphones mobiles. Dans le même état d'esprit, nos containers ne seront pas exhaustifs et il est probable que nous utilisions essentiellement le container de type *Empty* pour redéfinir nos propres containers CESURE plus « légers » que les quatre types de containers du CCM.

Les descripteurs OSD offrent un bon moyen de décrire les différentes packages (*software*, *component*, et *assembly*). Cependant, comparés à un ADL (*Architecture Definition Language*), ils représentent un outil au faible potentiel. C'est pourquoi, dans le cadre de CESURE, nous comptons utiliser un ADL (voir [PH00]) qui sera par la suite projeté vers les descripteurs du CCM pour leur utilisation lors de la phase de déploiement. Il sera ainsi possible de bénéficier de la puissance d'un ADL tout en utilisant à moindre coût des archives compatibles avec les packages du CCM.

Le CCM a en partie été choisi pour sa prise en compte du processus de déploiement, bien que cet aspect de la spécification fasse partie des points les moins bien spécifiés à l'heure actuelle. D'autre part, le processus proposé semble relativement générique pour être utilisé dans des contextes différents. Ce que nous projetons de faire est de réaliser une implantation de ce processus qui suit les grandes lignes énoncées dans la spécification, mais qui inclut les aspects propres à notre approche du déploiement, i.e. dirigé par l'utilisateur et basé sur une fonction de courtage (voir [BMPT00]).

Annexe A

Problème dans la projection IDL3 vers IDL2 des types d'événements

```
Issue 3746: Pbl: Improper mapping rule from IDL3 to IDL2 when dealing with events.
Source: LIFL (Mr. Raphael Marvie, marvie@lifl.fr)
Nature: Uncategorized Issue
Severity:
Summary: Dealing with the following IDL3 definition, a problem arises when
generating the IDL2 definitions (complete IDL2 mapping is enclosed at
the end of this mail).

    module example {
      valuetype AnEvent : Components::EventBase {
        public string value;
      };

      component Producer {
        publishes AnEvent output;
      };

      component Consumer {
        consumes AnEvent input;
      };
    };

According to the chapter 5 of the CCM specification, the publishes
definition of the Producer component is mapped to the following
definition (excerpt).

    interface Producer : Components::CCMObject {
      Components::Cookie subscribe_output
        (in ::example::ProducerEventConsumers::AnEventConsumer consumer)
        raises (Components::ExceededConnectionLimit);
    };

In the meantime, the consumes definition of the Consumer component is
mapped to the following definition.

    interface Consumer : Components::CCMObject {
      ::example::ConsumerEventConsumers::AnEventConsumer
      get_consumer_input();
    };
```

We can see that two versions of the "AnEventConsumer" interface have been defined in two distincts modules. Thus the following Java lines are not correct:

```
    example.Producer p = ...;
    example.Consumer c = ...;

    p.subscribe_output(c.get_consumer_input());
```

The Java compiler will refuse to compile the last one, producing an error like "BadTypeCoerce". However, in the IDL3 definitions, both components have been defined in order to be used together. (sic!)

Thus, we think that the mapping for a Consumer should not be based on the component definitions, but on the event definition itself. It would then avoid multiple (incompatible) definitions of the consumers. This mapping could be defined in two distinct ways.

First proposal (the bad one):

The following IDL3 definition:

```
    valuetype <EventType> : Components::EventBase {
      ...
    };
```

is used to generate

```
    interface <EventType>Consumer : Components::EventConsumerBase {
      void push(in <EventType> event);
    };
```

which would be defined in the same module as the Event. However, this raises the problem of parsing the IDL3 definition. The EventType can be discovered as it inherits the Components::EventBase interface. However, it may not inherit *directly* this interface. Thus, how to find that it is an event declaration?

Second proposal (the good one):

A new keyword,for example eventtype, is introduced. This keyword has the same relationship with 'valuetype' as 'component' is related to 'interface'. Thus, the following IDL3 definition avoids explicit inheritance declaration (like component and interface), and is easier to read. Moreover, it introduces light semantic.

```
    module example {

      eventtype <EventType> {
        ...
      };
```

Then, its IDL2 mapping becomes:

```
module example {

  valuetype <EventType> : Components::EventBase {
    ...
  };

  interface <EventType>Consumer : Components::EventConsumerBase {
    void push(in <EventType> event);
  };

};
```

Thus, our two components may communicate together. The mapping of a publishes definition would then use the event type as parameter of the subscribe operation (excerpt):

```
module example {

  interface Producer : Components::CCMObject {
    Components::Cookie
    subscribe_output (in ::example::AnEventConsumer c)
      raises (Components::ExceededConnectionLimit);
  };

};
```

What do you think of this problem and of its solution? You may not be very positive regarding the definition of a new keyword. However, it seems to be the best solution.

Resolution:
Revised Text:
Actions taken:
July 18, 2000: received issue
Discussion:

Annexe B

Les descripteurs OSD du distributeur de boissons

Les différents descripteurs présentés dans cette section sont simplifiés, par rapport aux DTD XML présentées dans la spécification, pour illustrer les points importants du discours tenu dans ce document.

B.1 Descripteur de package logiciel

```
<softpkg name="DistributeurBoisson" version="1.0.0">
  <pkgtype>CORBAComponent</pkgtype>
  <title>Distributeur de Boissons</title>
  <author>
    <company>L'équipe GOODE du LIFL</company>
    <webpage href="http://corbaweb.lifl.fr" />
  </author>
  <description>
     Un distributeur de boissons chaudes ou fraîches pour illustrer
     la spécification des composants CORBA.
  </description>
  <licence href="http://corbaweb.lifl.fr/Composants/licence.html" />
  <idl id="IDL:corbar:1.0">
    <link href="http://corbaweb.lifl.fr/Composants/Distributeur.idl" />
  </idl>
  <propertyfile>
    <fileinarchive name="distributeurProps.cpf" />
  </propertyfile>
  <implementation id="ABC:1234567890">
    <os name="Linux">
    <os name="Solaris">
    <programminglanguage name="Java" />
    <compiler name="jdk1.1.7" />
    <dependancy type="ORB">
      <name>ORBacus</name>
    </dependency>
    <code type="JAR">
      <fileinarchive name="distributeur.jar" />
      <entrypoint>instancieMaisonDistributeurs</entrypoint>
    </code>
  </implementation>
</softpkg>
```

B.2 Descripteur de composant

```
<corbacomponent>
  <corbaversion>3.0</corbaversion>
  <repository id="IDL:corbar/Distributeur:1.0"
  <componentkind>
    <entity>
      <servant lifetime="process" />
      <persistence responsability="component">
      </persistence>
    </entity>
  </componentkind>
  <eventpolicy emit="normal" />
  <threading policy="serialized" />
  <configurationcomplete set="true" />
  <componentfeatures name="Distributeur"
                   repid="IDL:corbar/Distributeur:1.0">
    <port>
      <provides
        providesname="f_client"
        repid="IDL:corbar/FacadeClient:1.0" />
      <provides
        providesname="f_fournisseur"
        repid="IDL:corbar/FacadeFournisseur:1.0" />
      <provides
        providesname="f_depanneur"
        repid="IDL:corbar/FacadeDepanneur:1.0" />
      <uses
        usesname="prise_courant"
        repid="IDL:corbar/PriseCourant:1.0" />
      <uses
        usesname="prise_eau"
        repid="IDL:corbar/PriseEau:1.0" />
      <emits
        emitsname="pdm"
        eventtype="PlusDeMonnaieEvt" />
      <publishes
        publishesname="dvide"
        eventtype="DistributeurVideEvt" />
      <consumes
        consumesname="temp"
        eventype="TemperatueEvt" />
    </port>
  </componentfeatures>
  <interface name="FacadeClient"
           repid="IDL:corbar/FacadeClient:1.0" />
  <interface name="FacadeFournisseur"
           repid="IDL:corbar/FacadeFournisseur:1.0" />
  <interface name="FacadeDepanneur"
           repid="IDL:corbar/FacadeDepanneur:1.0" />
  <interface name="PriseCourant"
           repid="IDL:corbar/PriseCourant:1.0" />
  <interface name="PriseEau"
```

```
            repid="IDL:corbar/PriseEau:1.0" />
</corbacomponent>
```

B.3 Descripteur d'assemblage

```
<componentassembly id="XYZ:0987654321">
  <description>
    Exemple de descripteur d'assemblage de composants
  </description>
  <componentfiles>
    <componentfile id="A">
      <fileinarchive name="distributeur.jar" />
    </componentfile>
    <componentfile id="B">
      <fileinarchive name="sondetemperature.jar" />
    </componentfile>
  </componentfiles>
  <partitionning>
    <homeplacement id="AaHome">
      <componentfileref idref="A" />
      <componentinstanciation id="Aa" />
    </<homeplacement>
    <homeplacement id="BbHome">
      <componentfileref idref="B" />
      <componentinstanciation id="Bb" />
    </<homeplacement>
    <componentplacement>
  </partitionning>
  <connection>
    <connectevent>
      <consumesport>
        <consumesidentifier>evt1</consumesidentifier>
        <componentinstanciationref idref="Aa">
      </consumesport>
      <publishesport>
        <publishesidentifier>evt1</publishesidentifier>
        <componentinstanciationref idref="Bb">
      </publishesport>
    </connectevent>
  </connection>
</componentassembly>
```

Annexe C

Interfaces OMG IDL3 du distributeur de boissons

La définition des types de composants prend ici en compte notre proposition de déclaration des événements telle que soumise à l'OMG.

```
module corbar {

  enum Boissons {
    Coca,
    Pepsi,
    Orangina,
    Fanta,
    Limonade,
    The,
    Cafe,
    Chocolat
  };

  typedef double Francs;

  typedef double Centigrades;

  exception BoissonNonDisponible {
  };

  exception SommeInsuffisante {
  };

  interface FacadeClient {
    void payer (in Francs f) ;
    void selectionnerBoisson (in Boissons b)
      raises (BoissonNonDisponible,SommeInsuffisante) ;
    Francs demanderPrix (in Boissons b)
      raises (BoissonNonDisponible) ;
    Boissons prendreBoisson () ;
    Francs prendreMonnaie () ;
    void rendreMonnaie () ;
  };

  interface FacadeFournisseur {
    void ouvrirFacade () ;
```

```
  void fermerFacade () ;
  void ajouterBoissons (in Boissons type, in long nombre,
                        in Francs prix) ;
  Francs viderCaisse () ;
};

interface FacadeDepanneur {
  void ouvrirArriere () ;
  void purgerPompe () ;
  void bidouiller () ;
  void coupDePied () ;
  void fermerFacadeArriere () ;
  boolean testMarche () ;
};

interface PriseCourant {
  void allumer () ;
  void eteindre () ;
  long fournirElectricite () ;
};

interface PriseEau {
  void ouvrir () ;
  void fermer () ;
  long fournirEau () ;
};

eventtype PlusDeMonnaieEvt {
  public long id ;
};

eventtype DistributeurVideEvt {
  public long id ;
};

eventype TemperatureEvt {
  public long degres ;
};

component Distributeur {
    provides FacadeClient          client ;
    provides FacadeFournisseur     fournisseur ;
    provides FacadeDepanneur       depanneur ;

    uses multiple PriseCourant     prise_courant ;
    uses PriseEau                  prise_eau ;

    emits PlusDeMonnaieEvt         pdm ;
    publishes DistributeurVideEvt  dvide ;

    consumes TemperatureEvt        temp ;

    attribute boolean              on ;
```

```
  };

  valuetype IdentifiantDistributeur : Components::PrimaryKeyBase {
      public long identifiant ;
  };

  home MaisonDeDistributeurs manages Distributeur
                             primaryKey IdentifiantDistributeur {
      factory creer_distributeur (in long id)
          raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
      finder  rechercher_distributeur (in long id)
          raises (Components::UnknownKeyValue,Components::InvalidKey) ;
  };

  component SondeTemperature {
      publishes TemperatureEvt temp ;
  };

  home MaisonDeSondesTemperature manages SondeTemperature {
  };

};
```

Annexe D

Interfaces OMG IDL du distributeur de boissons

La projection présentée ici est le résultat de la compilation de la déclaration des types de composants présentés dans l'annexe C. Une remarque, la projection des types d'événements qui définit les consommateurs pourrait regrouper ces interfaces au sein d'un module dans le module `corbar`. Ce module pourrait regrouper toutes les interfaces de consommateurs définies dans le contexte du module `corbar`, par exemple `Consumers`.

```
module corbar {

  enum Boissons {
    Coca,
    Pepsi,
    Orangina,
    Fanta,
    Limonade,
    The,
    Cafe,
    Chocolat
  };

  typedef double Francs;

  typedef double Centigrades;

  exception BoissonNonDisponible {
  };

  exception SommeInsuffisante {
  };

  interface FacadeClient {
    void payer (in Francs f) ;
    void selectionnerBoisson (in Boissons b)
      raises (BoissonNonDisponible,SommeInsuffisante) ;
    Francs demanderPrix (in Boissons b)
      raises (BoissonNonDisponible) ;
    Boissons prendreBoisson () ;
    Francs prendreMonnaie () ;
    void rendreMonnaie () ;
  };
```

```
interface FacadeFournisseur {
  void ouvrirFacade () ;
  void fermerFacade () ;
  void ajouterBoissons (in Boissons type, in long nombre,
                        in Francs prix) ;
  Francs viderCaisse () ;

};

interface FacadeDepanneur {
  void ouvrirArriere () ;
  void purgerPopmpre () ;
  void bidouiller () ;
  void coupDePied () ;
  void fermerFacadeArriere () ;
  boolean testMarche () ;
};

interface PriseCourant {
  void allumer () ;
  void eteindre () ;
  long fournirElectricite () ;
};

interface PriseEau {
  void ouvrir () ;
  void fermer () ;
  long fournirEau () ;
};

/*
  projection des consommateurs d'événements
 */

valuetype PlusDeMonnaieEvt : Components::EventBase {
  public long id ;
};

valuetype DistributeurVideEvt : Components::EventBase {
  public long id ;
};

valuetype TemperatureEvt : Components::EventBase {
  public long degres ;
};

interface PlusDeMonnaieEvtConsumer : Components::EventConsumerBase {
  void push (in PlusDeMonnaieEvt evt) ;
};

interface DistributeurVideEvtConsumer : Components::EventConsumerBase {
  void push (in DistributeurVideEvt evt) ;
```

```
};

interface TemperatureEvtConsumer : Components::EventConsumerBase {
  void push (in TemperatureEvt evt) ;
};

interface TemperatureEvtConsumer : Components::EventConsumerBase {
  void push(in TemperatureEvt evt);
};

/*
  projection des types de composants
 */

interface Distributeur : Components::CCMObject {
  attribute boolean on ;

  FacadeClient provide_client () ;
  FacadeFournisseur provide_fournisseur () ;
  FacadeDepanneur provide_depanneur () ;

  void connect_prise_eau (in PriseEau cnx)
    raises (Components::AlreadyConnected,
            Components::InvalidConnection)  ;
  PriseEau disconnect_prise_eau ()
    raises (Components::NoConnection) ;
  PriseEau get_connection_prise_eau () ;

  struct prise_courantConnection {
    PriseCourant       objref;
    Components::Cookie ck;
  };

  typedef sequence<prise_courantConnection> prise_courantConnections;

  Components::Cookie connect_prise_courant (in PriseCourant cnx)
    raises (Components::ExceededConnectionLimit,Components::InvalidConnection) ;
  PriseCourant disconnect_prise_courant (in Components::Cookie ck)
    raises (Components::InvalidConnection) ;
  priseCourantConnections get_connections_prise_courant () ;

  void connect_pdm (in DistributeurEventConsumers::PlusDeMonnaieEvtConsumer consumer)
    raises (Components::AlreadyConnected) ;
  PlusDeMonnaieEvtConsumer disconnect_pdm ()
    raises (Components::NoConnection) ;

  Components::Cookie subscribe_dvide (in DistributeurVideEvtConsumer consumer)
    raises (Components::ExceededConnectionLimit) ;
  DistributeurVideEvtConsumer unsubscribe_dvide (in Components::Cookie ck)
    raises (Components::InvalidConnection) ;

  TemperatureEvtConsumer get_consumer_temp();
};
```

```
  interface SondeTemperature : Components::CCMObject {
    Components::Cookie subscribe_temp (in TemperatureEvtConsumer consumer)
      raises (Components::ExceededConnectionLimit) ;
    TemperatureEvtConsumer unsubscribe_temp (in Components::Cookie ck)
      raises (Components::InvalidConnection) ;
  };


  /*
    projection des types de maisons de composants
   */

  interface MaisonDeDistributeursExplicit : Components::CCMHome {
    Distributeur creer_distributeur (in long id)
      raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
    Distributeur rechercher_distributeur (in long id)
      raises (Components::UnknownKeyValue,Components::InvalidKey) ;
  };

  interface MaisonDeDistributeursImplicit {
    Distributeur create (in IdentifiantDistributeur key)
      raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
    Distributeur find_by_primary_key (in IdentifiantDistributeur key)
      raises (Components::UnknownKeyValue,Components::InvalidKey) ;
    void remove (in IdentifiantDistributeur key)
      raises (Components::UnknownKeyValue,Components::InvalidKey) ;
    IdentifiantDistributeur get_primary_key (in Distributeur compo) ;
  };

  interface MaisonDeDistributeurs :
    MaisonDeDistributeursExplicit, MaisonDeDistributeursImplicit {
  };

  interface MaisonDeSondesTemperatureExplicit : Components::CCMHome {
  };

  interface MaisonDeSondesTemperatureImplicit : Components::KeylessCCMHome {
    SondeTemperature create () ;
  };

  interface MaisonDeSondesTemperature :
    MaisonDeSondesTemperatureExplicit, MaisonDeSondesTemperatureImplicit {
  };


};
```

Annexe E

Les interfaces OMG IDL du module `Components`

Nous en présentons ici que les interface du module `Components` (module de base) et du module `Deployment` contenu dans celui-ci. Le module `Components` contient aussi des sous modules relatifs aux transactions, à la notification et aux interfaces fournies par les containers. Comme ces derniers n'ont pas été présentés dans le document, nous ne les reportons pas ici.

```
import ::CORBA
import ::SecurityLevel2
import ::CosPersistentState
import ::PortableServer
import ::CosNotification
import ::CosNotifyChannelAdmin

module Components {
  typePrefix Components "omg.org"
  typedef string FeatureName;
  typedef sequence<FeatureName> NameList;
  valuetype Cookie {
    private sequence<octet> cookieValue;
  };

  valuetype FacetDescription {
  public CORBA::RepositoryId InterfaceID;
    public FeatureName Name;
  };

  valuetype Facet : FacetDescription {
    public Object Ref;
  };

  typedef sequence<Facet> Facets;
  typedef sequence<FacetDescription>
  FacetDescriptions;

  exception InvalidName { };
  exception InvalidConnection { };
  exception ExceededConnectionLimit { };
  exception AlreadyConnected { };
  exception NoConnection { };
  exception CookieRequired { };
```

```
exception DuplicateKeyValue { };
exception NoKeyAvailable { };
exception InvalidKey { };
exception UnknownKeyValue { };
exception BadEventType {
  CORBA::RepositoryId expected_event_type;
};
exception HomeNotFound { };
exception WrongComponentType { };
exception InvalidConfiguration { };

interface Navigation {
  Object provide_facet (in FeatureName name)
    raises (InvalidName);
  FacetDescriptions describe_facets();
  Facets provide_all_facets();
  Facets provide_named_facets (in NameList names)
    raises (InvalidName);
  boolean same_component (in Object ref);
};

valuetype ConnectionDescription {
  public Cookie ck;
  public Object objref;
};

typedef sequence<ConnectionDescription> ConnectedDescriptions;

interface Receptacles {
  Cookie connect (in FeatureName name,
                  in Object connection )
    raises (InvalidName,
            InvalidConnection,
            AlreadyConnected,
            ExceededConnectionLimit);
  void disconnect (in FeatureName name,
                   in Cookie ck)
    raises (InvalidName,
            InvalidConnection,
            CookieRequired,
            NoConnection);
  ConnectedDescriptions get_connections (in FeatureName name)
    raises (InvalidName);
};

abstract valuetype EventBase {};

interface EventConsumerBase {
  void push_event (in EventBase evt) raises (BadEventType);
};

interface Events {
  EventConsumerBase get_consumer (in FeatureName sink_name)
```

```
    raises (InvalidName);
  Cookie subscribe (in FeatureName publisher_name,
                    in EventConsumerBase subscriber)
    raises (InvalidName);
  void unsubscribe (in FeatureName publisher_name,
                    in Cookie ck)
    raises (InvalidName, InvalidConnection);
  void connect_consumer (in FeatureName emitter_name,
                         in EventConsumerBase consumer)
    raises (InvalidName, AlreadyConnected);
  EventConsumerBase
  disconnect_consumer (in FeatureName source_name)
    raises (InvalidName, NoConnection);

};

abstract valuetype PrimaryKeyBase { };

interface CCMObject; //forward reference

interface CCMHome {
  CORBA::IRObject get_component_def ();
  CORBA::IRObject get_home_def ();
  void remove_component ( in CCMObject comp);
};

interface KeylessCCMHome {
  CCMObject create_component();
};

interface HomeFinder {
  CCMHome find_home_by_component_type (in CORBA::RepositoryId comp_repid)
    raises (HomeNotFound);
  CCMHome find_home_by_home_type (in CORBA::RepositoryId home_repid)
    raises (HomeNotFound);
  CCMHome find_home_by_name (in string home_name)
    raises (HomeNotFound);
};

interface Configurator {
  void configure (in CCMObject comp)
    raises (WrongComponentType);
};

valuetype ConfigValue {
  public FeatureName name;
  public any value;
};

typedef sequence<ConfigValue> ConfigValues;

interface StandardConfigurator : Configurator {
  void set_configuration (in ConfigValues descr);
```

```
};

interface HomeConfiguration : CCMHome {
  void set_configurator (in Configurator cfg);
  void set_configuration_values (in ConfigValues config);
  void complete_component_configuration (in boolean b);
  void disable_home_configuration();
};

interface CCMObject
  : Navigation, Receptacles, Events {
  CORBA::IRObject get_component_def ( );
  CCMHome get_ccm_home( );
  PrimaryKeyBase get_primary_key( )
    raises (NoKeyAvailable);
  void configuration_complete( )
    raises (InvalidConfiguration);
  void remove();
};

interface Enumeration {
  boolean has_more_elements();
  CCMObject next_element();
};

module Deployment {

  enum AssemblyState {INACTIVE, INSERVICE};
  exception UnknownImplId { };
  exception InvalidLocation { };
  exception InvalidAssembly { };

  interface ComponentInstallation {
    boolean install(
                    in string implUUID,
                    in string component_loc)
      raises (InvalidLocation);
    boolean replace(
                    in string implUUID,
                    in string component_loc)
      raises (InvalidLocation);
    boolean remove(in string implUUID)
      raises (UnknownImplId);
  };

  interface Assembly {
    boolean build();
    boolean tear_down();
    AssemblyState get_state();
  };

  interface AssemblyFactory {
    Cookie create(in string assembly_loc)
```

```
        raises (InvalidLocation);
      Assembly lookup(in Cookie c)
        raises (InvalidAssembly);
      boolean destroy(in Cookie c)
        raises (InvalidAssembly);
    };
  };
};
```

Bibliographie

[BMPT00] G. Bernard, R. Marvie, E. Putrycz, and C. Taconet. Infrastructure système. Technical Report RNRT 98-5, CESURE, mai 2000.

[Gri97] R. Grimes. *Professional DCOM Programming.* Wrox Press ltd., Birmingham, Canada, 1997.

[Ham97] G. Hamilton. *Java Beans Specification v1.01.* Sun Microsystems, July 1997.

[MH99] V. Matena and M. Hapner. *Enterprise Java Beans Specification v1.1 - Public Draft.* Sun Microsystems, Mai 1999.

[MMG00a] R. Marvie, P. Merle, and J.-M. Geib. A Dynamic Platform for CORBA Component Based Applications. In *First International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD'00)*, Reims, France, Mai 2000.

[MMG00b] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Anvers, Belgique, Septembre 2000. IEEE.

[MMGG99] R. Marvie, P. Merle, J.-M. Geib, and C. Gransart. Des objets aux composants CORBA, première étude et expérimentation avec CorbaScript. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, Décembre 1999.

[MMGG00] R. Marvie, P. Merle, J.-M. Geib, and C. Gransart. CCM + IDLscript = applications distribuées. *Evolution des plate-formes orientées objets répartis, numéro spécial de Calculateurs Parallèles et Réseau*, 2000. Ed. Hermès.

[MP00] R. Marvie and M.-C. Pellegrini. Etat de l'art des modèles de composants. Technical Report RNRT 98-2, CESURE, mai 2000.

[OL00] OOC and LIFL. *IDLscript Specification.* Object Management Group, Août 2000.

[OMG99] OMG. *CORBA Components: Joint Revised Submission.* Object Management Group, Août 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.

[PH00] N. De Palma and D. Hagimont. Description de l'architecture. Technical Report RNRT 98-4, CESURE, novembre 2000.