REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE ABOUBEKR BELKAID – TLEMCEN

FACULTE DES SCIENCES DE L'INGENIEUR

DEPARTEMENT D'INFORMATIQUE

MEMOIRE DE FIN D'ETUDE

Pour l'obtention du diplôme d'ingénieur d'état en informatique

Option : Système d'information avancé

Université Aboubakr Belkaïd
Faculté des Sciences
Chef de Département
d'Informatique

Thème

DÉPLOIEMENT D'UNE APPLICATION À BASE DE COMPOSANTS SOUS FRACTAL

Réalisé par :
M^r KOÏTA Aboubakar
M^r GUEYE Abdoulaye

Soutenu en Juin 2008
Devant la commission d'examen

M^r CHIKH Mohammed El Amine	**Président**
M^r ABDERAHIM Mohammed El Amine	**Examinateur**
M^r BENAMAR Abdelkrim	**Encadreur**

Bibliothèque sciences
BFST5175

Table des matières

Avant-propos

Le domaine de l'intergiciel (*middleware*), apparu dans les années 1990, a pris une place centrale dans le développement des applications informatiques réparties. L'intergiciel joue aujourd'hui, pour celles-ci, un rôle analogue à celui d'un système d'exploitation pour les applications centralisées : il dissimule la complexité de l'infrastructure sous-jacente, il présente une interface commode aux développeurs d'applications, il fournit un ensemble de services communs.
Les architectures des systèmes intergiciels ont connu différentes évolutions, notamment les organisations multi-étages. L'une des évolutions ayant le plus attitré l'attention fût la naissance des intergiciels à objets répartis dans la première moitié des années 1990, dont les plus représentatifs sont CORBA de L'OMG et RMI de SUN. En effet, les technologies intergicielles ayant pour but le support d'un modèle de développement de haut niveau focalisé sur des questions métiers dans le cadre d'une exécution des applications en environnement réparti, et le modèle de développement de la conception et de la programmation orientée objet occupant à l'époque une place importante dans le domaine du génie logiciel, les intergiciels à objets répartis furent accueillis avec enthousiasme.

Les objets cependant proposés comme première approche pour la mise en place d'une industrie du logiciel où les applications seraient conçues par assemblages de composants ont montré leurs limites en termes d'unité de composition tant sur le plan métier que sur le plan de la répartition principalement à cause de la faible application de la *séparation des préoccupations* et du manque de support pour la description explicite de l'architecture des applications. Ces limites entraînèrent ainsi l'émergence du concept de composant. A l'instar des objets répartis nés du portage des concepts de l'objet en environnement réparti, les composants aussi donnèrent naissance aux composants répartis pour les environnements répartis.

Les approches à base de composants apparaissent de plus en plus incontournables pour le développement de systèmes et d'applications répartis. Il s'agit de faire face à la complexité sans cesse croissante de ces logiciels et de répondre aux grands défis de l'ingénierie des

AVANT-PROPOS

systèmes : passage à grande échelle, administration, autonomie. Après donc les objets dans la première moitié des années 1990, les composants se sont imposés comme le paradigme clé de l'ingénierie des intergiciels et de leurs applications dans la seconde moitié des années 1990. L'intérêt de la communauté industrielle et académique s'est d'abord porté sur les modèles de composants pour les applications comme EJB, CCM. A partir du début des années 2000, le champ d'application des composants s'est étendu aux couches inférieures : systèmes et intergiciels. Il s'agit toujours, comme pour les applications, d'obtenir des entités logicielles composables aux interfaces spécifiées contractuellement, déployables et configurables; mais il s'agit également d'avoir des plates-formes à composants suffisamment performantes et légères pour ne pas pénaliser les performances du système.

Ce mémoire présente un modèle de composants innovant issu de la recherche, Fractal, qui remplit ces conditions. Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (c.-à-d. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation. Le modèle de composants Fractal a été défini par France Telecom R&D et l'INRIA. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET.

Fractal est organisé comme un projet du consortium ObjectWeb pour le middleware open source. Les premières discussions autour du modèle Fractal, initiées dès 2000 à France Telecom R&D ont abouti en juin 2002 avec la première version officielle de la spécification et la première version de Julia, qui est l'implémentation de référence de cette spécification. La spécification a évolué pour aboutir en septembre 2003 à une deuxième version comportant un certain nombre de changements au niveau de l'API.

Fractal se distingue de nombres de modèles de composants principalement par son caractère *extensible*, c'est-à-dire la possibilité que le modèle offre aux développeurs de personnaliser les propriétés *extra-fonctionnelles* des composants. Les modèles de composants comme EJB ou CCM fournissent des environnements dans lesquels les composants sont hébergés par des *conteneurs* fournissant des services techniques. Par exemple, les spécifications EJB définissent des services de sécurité, persistance, transaction et de gestion de cycle de vie. La plupart du temps, cet ensemble de services est fermé et codé en dur dans les conteneurs. Fractal, contrairement à ces modèles de composants, définit une spécification ouverte à l'extension.

Le mémoire est organisé comme suit.

Le premier chapitre, **Introduction à l'intergiciel**, présente la notion d'intergiciel. Il commence par mettre en évidence les contraintes des environnements répartis qui ont motivé le développement de ces logiciels. L'étude des systèmes intergiciels est ensuite abordée à travers les principaux concepts utilisés dans ces systèmes : services, interfaces, modes de communication et architectures applicatives. Le chapitre se termine en soulignant les limites du modèle objet qui ont conduit à l'émergence du paradigme composant.

AVANT-PROPOS

Le deuxième chapitre, **Composants logiciels**, est justement dédié à l'étude des composants. Après un tour d'horizon des concepts de base du paradigme, le chapitre présente les différentes techniques mises en œuvre dans les modèles de composants pour assurer le support de ce qui est considéré comme le cœur des développements centrés sur la notion de composant : séparation des préoccupations entre code fonctionnel, ou logique métier, et code non fonctionnel, ou logique système. Le chapitre continue en présentant une notion gagnant de jour en jour de l'importance dans la programmation orientée composants, celle d'architecture logicielle. Finalement nous concluons en recensant les points essentiels à tenir en compte dans le développement d'un modèle de composant.

Le troisième chapitre, **Le modèle de composants Fractal**, consacré à l'étude du modèle de composants Fractal commence par l'étude de la spécification proprement dite. Il présente le langage de description d'architecture associé au modèle. Deux des principales plateformes du modèle sont étudiées et les autres sont survolées. Des bibliothèques disponibles pour le développement d'applications Fractal sont présentées. Le modèle Fractal est comparé à nombre d'autres modèles de composants et le chapitre se termine par une conclusion.

Le quatrième chapitre, **Exemple d'application Fractal**, permet d'illustrer à traves une simple application l'utilisation concrète de l'API Fractal. Il est également l'occasion de découvrir un outil d'administration d'applications Fractal.

Le mémoire se termine finalement par une **conclusion générale**.

Chapitre 1

Introduction à l'intergiciel

1.1 Introduction

Les systèmes répartis sont dans leur définition la plus simpliste, des logiciels dont l'exécution se déroule sur un ensemble d'ordinateurs distribués géographiquement et reliés entre eux par des interconnexions réseau.

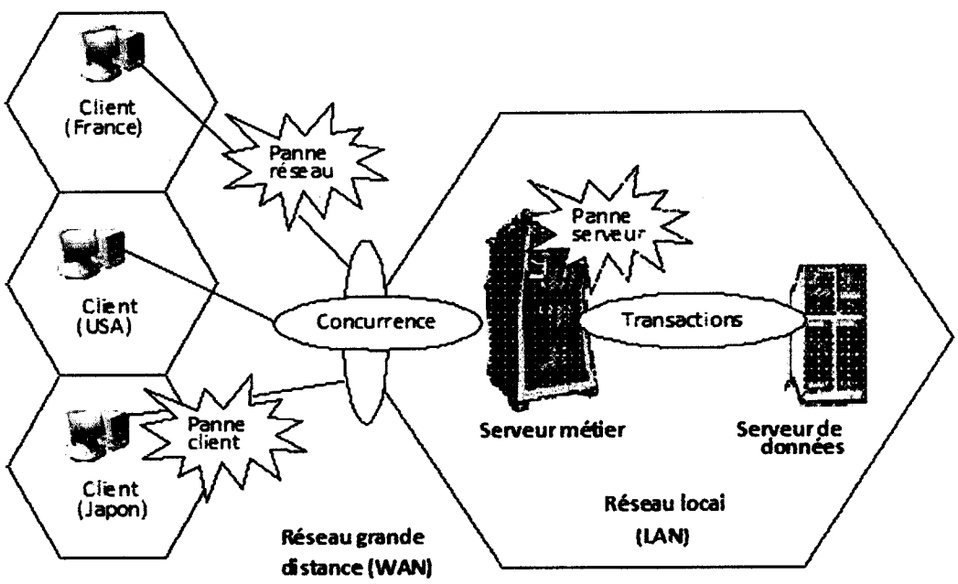


Figure 1.1-Exemple de système réparti [1]

Les applications disponibles sur Internet forment une catégorie hautement stratégique de systèmes répartis. La figure 1.1 représente une architecture typique de services Internet, mettant en jeu un serveur métier (par exemple, un serveur web, un serveur de base de données et un certain nombre de clients répartis géographiquement). Le principe de fonctionnement d'une telle application est relativement simple : les clients effectuent des requêtes vers le

serveur (comme afficher une page HTML) qui en retour répond en accédant potentiellement à des informations gérées par le serveur de données.

On le voit sur cet exemple, un système peut être réparti pour des raisons « physiques », comme ici la séparation géographique naturelle entre les clients et les serveurs. De nombreux systèmes sont ainsi naturellement répartis : systèmes bancaires dont les agences sont disséminées géographiquement, systèmes de contrôle de processus industriels répartis sur plusieurs sites, etc. La miniaturisation des ordinateurs et l'émergence des technologies de réseaux sans fil à haut débit introduit un nombre sans cesse croissant d'applications de nature intrinsèquement répartie.

D'autres considérations, de nature plus « logique », interviennent. Sur notre exemple, le choix de séparer le serveur métier du serveur de données peut notamment s'expliquer par un souci d'augmentation des performances du système. Deux machines (ordinateurs complets ou simples processeurs) physiques fonctionnent généralement plus vite qu'une seule machine comparable, c'est le postulat de base du parallélisme.

Si la machine serveur tombe en panne, l'intégrité du serveur de données ne sera pas compromise, seules les transactions en cours devront être annulées. La répartition permet donc d'augmenter la fiabilité d'un système.

Du point de vue de l'organisation fournissant le service (Figure 1.1), le serveur métier est en contact avec le monde extérieur. Il est donc soumis aux éventuelles attaques de « clients » malintentionnés. Le serveur de données, en revanche est en quelque sorte « protégé » par le serveur métier ; il n'est pas directement en contact avec l'extérieur. La séparation, ici d'ordre physique, permet de profiter de matériels spécifiques pour, la sécurisation du système ; la protection matérielle étant en général plus efficace qu'une protection purement logicielle. La répartition est donc également un outil important pour renforcer la sécurité des systèmes.

Les spécificités propres aux systèmes distribués sont [1]:

1. **Large échelle :** On distingue en général deux catégories de système réparti : les systèmes limités en taille et les systèmes à large échelle. Les développements actuels se concentrent, dans leur écrasante majorité, sur les problèmes spécifiques de la répartition à large échelle. L'Internet, qui relie des millions de machines entre elles par l'intermédiaire de réseaux de très grande complexité, représente l'archétype des systèmes à large échelle.

2. **Hétérogénéité :** Les systèmes répartis, sauf cas très spécifiques, ne sont pas l'apanage d'un constructeur donné. Ce sont des systèmes hautement hétérogènes, mettant en œuvre des plateformes matérielles et logicielles variées et souvent incompatibles entre elles. Gérer cette hétérogénéité passe par le développement de technologies standard et ouvertes permettant l'*interopérabilité* entre systèmes différents.

3. **Communication asynchrone :** Déplacer de l'information prend du temps, bien connu en sciences physiques, ce phénomène prend en informatique tout son sens dans le cadre de systèmes géographiquement répartis, et notamment si les distances entre les différents éléments répartis du système ne peuvent être prévus de façon précise et absolue, on dit alors que le système fonctionne de façon *asynchrone*. Il s'agit d'une différence fondamentale avec les environnements centralisés dont le fonctionnement synchrone est orchestré par une horloge globale unique.

4. **Contrôle concurrent :** Chaque machine ou processus d'un système réparti possède, de par son fonctionnement asynchrone, un contrôle autonome dit *concurrent*. Par exemple sur la figure 1.1, Les clients du système accèdent en même temps, et potentiellement en très grand nombre aux services proposés par le serveur métier. Ce dernier doit donc « entremêler » les requêtes des clients en mettant en œuvre une politique de *synchronisation* pour gérer les éventuels conflits entre requêtes concurrentes. Les techniques de contrôle de concurrence sont parmi les aspects les plus sensibles et les plus complexes dans la conception des applications réparties. Les moniteurs transactionnels forment une catégorie hautement stratégique d'outils pour le contrôle de concurrence.

5. **Pannes partielles :** Les systèmes informatiques, comme probablement tous les systèmes physiques, peuvent être victimes de défaillances. Ainsi, un ordinateur peut cesser de fonctionner subitement, et avec lui toutes les applications qu'il exécutait (panne d'une machine cliente de la figure 1.1). Un processus logiciel peut également cesser de fonctionner, par exemple à cause d'un bogue ou par manque de ressources. Les réseaux de communication ont également leurs pannes spécifiques : pannes de canaux de communication et pannes de message (disparition, réordonnancement ou pire, altération des messages communiqués). Ces pannes dites *partielles* sont omniprésentes dans les systèmes répartis. Théoriquement, le problème de décider si un processus est en panne ou alors simplement ralenti est indécidable. On peut au mieux « soupçonner » une machine d'être en panne quand elle est inactive pendant un temps assez long. Pour la plus part des systèmes en ligne, la disponibilité des services est essentielle, toute panne doit être « réparée » de façon la plus rapide et la plus transparente possible.

6. **Sécurité :** La sécurisation d'un système centralisé est relativement simple. En effet, le bon déroulement des applications peut être supervisé assez facilement puisque la vision instantanée et complète de l'ensemble des ressources employées par le système est possible. Dans un système réparti de grande taille, la sécurisation est un problème crucial et autrement plus difficile. Il est par exemple nécessaire de protéger certaines communications sensibles, généralement par cryptage des informations transmises. Dans l'exemple de la figure 1.1, nous pouvons supposer que des clients transmettent au serveur certaines informations sécrètes comme des numéros de carte bancaire. Il faut aussi s'assurer des accès aux domaines protégés des systèmes. Dans les services

web, il est par exemple d'usage de distinguer les clients invités (ou anonymes) accédant à la version minimale des services et les clients authentifiés possédant eux des droits d'accès étendus. De par leur *complexité* et leur *ouverture* au monde extérieur, les systèmes répartis déployés sur l'internet ne peuvent être totalement infaillibles. Tout est question d'équilibre entre la robustesse des protections proposées face à l'ingéniosité (sans limite) des attaquants potentiels.

En plus des problèmes récurrents dans les systèmes répartis que nous venons de citer, les développeurs sont confrontés dans leur pratique quotidienne à des problèmes plus concrets. Les brèves études de cas qui suivent illustrent quelques situations typiques, et mettent en évidence les principaux problèmes et les solutions proposées.

Exemple 1 : réutiliser le logiciel patrimonial. Les entreprises et les organisations construisent maintenant des systèmes d'information globaux intégrant des applications jusqu'ici indépendantes, ainsi que des développements nouveaux. Ce processus d'intégration doit tenir compte des applications dites *patrimoniales* (en anglais : *legacy*), qui ont été développées avant l'avènement des standards ouverts actuels, utilisent des outils « propriétaires », et nécessitent des environnements spécifiques.

Une application patrimoniale ne peut être utilisée qu'à travers une interface spécifiée, et ne peut être modifiée. La plupart du temps, l'application doit être reprise telle quelle car le coût de sa récriture serait prohibitif.

Le principe des solutions actuelles est d'adopter une norme commune, non liée à un langage particulier, pour interconnecter différentes applications. La norme spécifie des interfaces et des protocoles d'échange pour la communication entre applications. Les protocoles sont réalisés par une couche logicielle qui fonctionne comme un bus d''echanges entre applications, également appelé courtier (en anglais *broker*). Pour intégrer une application patrimoniale, il faut développer une *enveloppe* (en anglais *wrapper*), c'est-à-dire une couche logicielle qui fait le pont entre l'interface originelle de l'application et une nouvelle interface conforme à la norme choisie.

Une application patrimoniale ainsi « enveloppée » peut maintenant être intégrée avec d'autres applications du même type et avec des composants nouveaux, en utilisant les protocoles normalisés du courtier. Des exemples de courtiers sont CORBA, les files de messages, les systèmes à publication et abonnement (en anglais *publish-subscribe*).

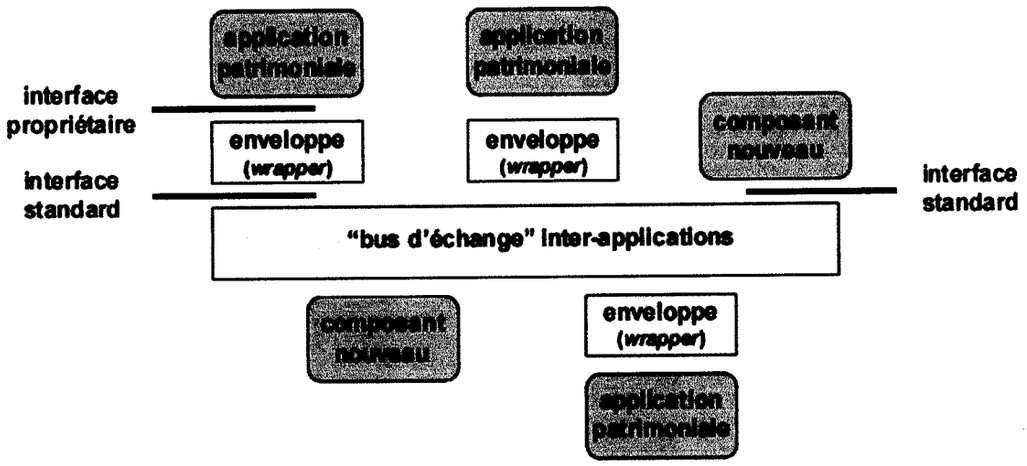


Figure 1.2-Intégration d'applications patrimoniales [2]

Exemple 2 : systèmes de médiation [2]. Un nombre croissant de systèmes prend la forme d'une collection d'équipements divers (capteurs ou effecteurs) connectés entre eux par un réseau. Chacun de ces dispositifs remplit une fonction locale d'interaction avec le monde extérieur, et interagit à distance avec d'autres capteurs ou effecteurs. Des applications construites sur ce modèle se rencontrent dans divers domaines : réseaux d'ordinateurs, systèmes de télécommunications, équipements d'alimentation électrique permanente, systèmes décentralisés de production.

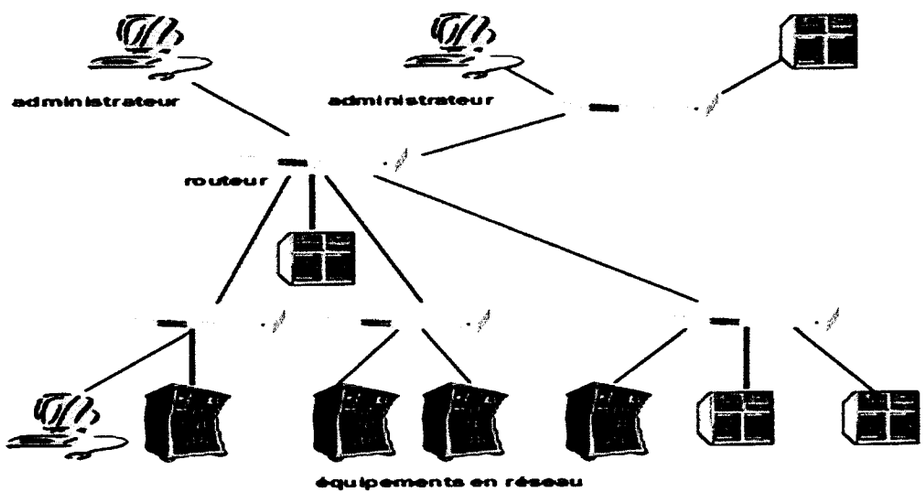


Figure 1.3-Surveillance et commande d'équipements en réseau [2]

La gestion de tels systèmes comporte des tâches telles que la mesure continue des performances, l'élaboration de statistiques d'utilisation, l'enregistrement d'un journal, le traitement des signaux d'alarme, la collecte d'informations pour la facturation, la maintenance à distance, le téléchargement et l'installation de nouveaux services.

L'exécution de ces tâches nécessite l'accès à distance au matériel, la collecte et l'agrégation de données, et la réaction à des événements critiques. Les systèmes réalisant ces tâches s'appellent *systèmes de médiation*. L'infrastructure interne de communication d'un système de médiation doit réaliser la collecte de données et leur acheminement depuis ou vers les capteurs et effecteurs. La communication est souvent déclenchée par un évènement externe, comme la détection d'un signal d'alarme ou le franchissement d'un seuil critique par une grandeur observée.

Un système de communication adapté à ces exigences est un *bus à messages*, c'est-à-dire un canal commun auquel sont raccordées les diverses entités (Figure 1.3). La communication est asynchrone et peut mettre en jeu un nombre variable de participants. Les destinataires d'un message peuvent être les membres d'un groupe prédéfini, ou les «abonnés » à un sujet spécifié.

Exemple 3 : adaptation de clients par des mandataires. Les utilisateurs accèdent aux applications sur l'Internet via des équipements dont les caractéristiques et les performances couvrent un spectre de plus en plus large. Entre un PC de haut de gamme, un téléphone portable et un assistant personnel, les écarts de bande passante, de capacités locales de traitement, de capacités de visualisation, sont très importants. On ne peut pas attendre d'un serveur qu'il adapte les services qu'il fournit aux capacités locales de chaque point d'accès. On ne peut pas non plus imposer un format uniforme aux clients, car cela reviendrait à les aligner sur le niveau de service le plus bas (texte seul, écran noir et blanc, etc.).

La solution préférée est d'interposer une couche d'adaptation, appelée *mandataire* (en anglais *proxy*) entre les clients et les serveurs. Un mandataire différent peut être réalisé pour chaque classe de point d'accès côté client (téléphone, assistant, etc.). La fonction du mandataire est d'adapter les caractéristiques du flot de communication depuis ou vers le client aux capacités de son point d'accès et aux conditions courantes du réseau. A cet effet, le mandataire utilise ses propres ressources de stockage et de traitement. Les mandataires peuvent être hébergés sur des équipements dédiés (Figure 1.4), ou sur des serveurs communs.

Des exemples d'adaptation sont la compression des données pour réagir à des variations de bande passante du réseau, la réduction de la qualité des images pour prendre en compte des capacités réduites d'affichage, le passage de la couleur aux niveaux de gris.

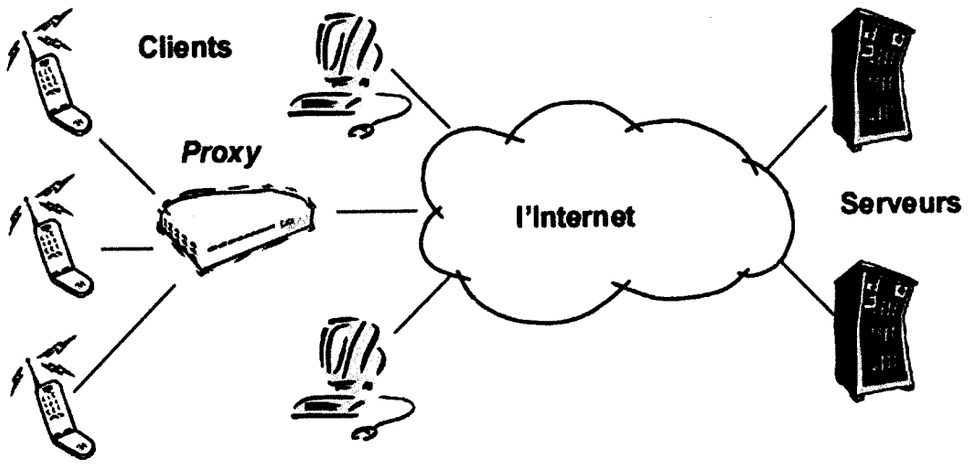


Figure 1.4-Adaptation des communications aux ressources des clients par des mandataires [2]

Les quatre études de cas qui précèdent ont une caractéristique commune : dans chacun des systèmes présentés, des applications utilisent des logiciels de niveau intermédiaire, installés au-dessus des systèmes d'exploitation et des protocoles de communication, qui réalisent les fonctions suivantes :

1. cacher la *répartition*, c'est-à-dire le fait qu'une application est constituée de parties interconnectées s'exécutant à des emplacements géographiquement répartis;
2. cacher l'*hétérogénéité* des composants matériels, des systèmes d'exploitation et des protocoles de communication utilisés par les différentes parties d'une application;
3. fournir des *interfaces* uniformes, normalisées, et de haut niveau aux équipes de développement et d'intégration, pour faciliter la construction, la réutilisation, le portage et l'interopérabilité des applications;
4. fournir un ensemble de *services* communs réalisant des fonctions d'intérêt général, pour éviter la duplication des efforts et faciliter la coopération entre applications.

Cette couche intermédiaire de logiciel, schématisée sur la figure 1.5, est désignée par le terme générique d'*intergiciel* (en anglais *middleware*). Le domaine de l'intergiciel, apparu dans les années 1990, a pris une place centrale dans le développement des applications informatiques réparties. L'intergiciel joue aujourd'hui, pour celles-ci, un rôle analogue à celui d'un système d'exploitation pour les applications centralisées : il dissimule la complexité de l'infrastructure sous-jacente, il présente une interface commode aux développeurs d'applications, il fournit un ensemble de services communs, il permet de réutiliser l'expérience et parfois le code et il facilite l'évolution des applications.

En conséquence, on peut espérer réduire le coût et la durée de développement des applications (l'effort étant concentré sur les problèmes spécifiques et non sur l'intendance), et améliorer leur portabilité et leur interopérabilité.

Les acteurs de l'intergiciel sont nombreux et divers : les organismes de normalisation, les industriels du logiciel et des services, les utilisateurs d'applications. Les consortiums et organisations diverses qui développent et promeuvent l'usage du logiciel libre tiennent une place importante dans le monde de l'intergiciel.

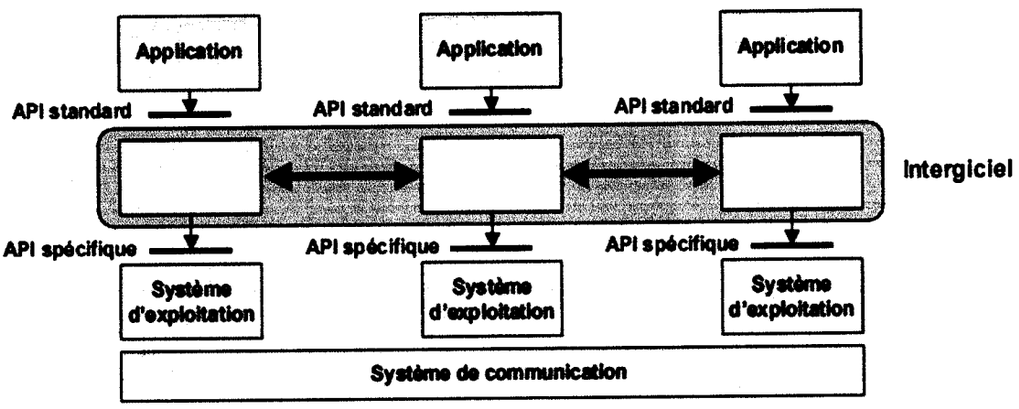


Figure 1.5-Organisation de l'intergiciel [2]

Le domaine de l'intergiciel est en évolution permanente. Pour répondre à des besoins de plus en plus exigeants, les produits doivent s'adapter et s'améliorer. De nouveaux domaines s'ouvrent, comme l'informatique ubiquitaire, les systèmes embarqués, les systèmes mobiles. Le cycle de cette évolution est rapide : chaque année apporte une transformation significative du paysage des normes et des produits de l'intergiciel. Malgré la rapidité de ces changements, il est possible de dégager quelques principes architecturaux qui guident la conception et la construction de l'intergiciel et des applications qui l'utilisent. Ces principes sont eux-mêmes développés et enrichis grâce aux résultats de la recherche et à l'expérience acquise, mais présentent une certaine stabilité.

Un inconvénient potentiel est la perte de performances liée à la traversée de couches supplémentaires de logiciel. L'utilisation de techniques intergicielles implique par ailleurs de prévoir la formation des équipes de développement.

1.2 Classes d'intergiciel

Les systèmes intergiciels peuvent être classés selon différents critères, prenant en compte les propriétés de l'infrastructure de communication, les interfaces de fourniture de services, l'architecture d'ensemble des applications.

1.2.1 Caractéristiques de la communication.

L'infrastructure de communication sous-jacente à un intergiciel est caractérisée par plusieurs propriétés qui permettent une première classification.

1. *Topologie fixe ou variable.* Dans un système de communication fixe, les entités communicantes résident à des emplacements fixes, et la configuration du réseau ne

change pas (ou bien ces changements sont des opérations peu fréquentes, prévues à l'avance). Dans un système de communication mobile, tout ou partie des entités communicantes peuvent changer de place, et se connecter ou se déconnecter dynamiquement, y compris pendant le déroulement des applications.

2. *Caractéristiques prévisibles ou imprévisibles*. Dans certains systèmes de communication, on peut assurer des bornes pour des facteurs de performance tels que la gigue ou la latence. Néanmoins, dans beaucoup de situations pratiques, ces bornes ne peuvent être garanties, car les facteurs de performance dépendent de la charge de dispositifs partagés tels que les routeurs ou les voies de transmission. Un système de communication est dit synchrone si on peut garantir une borne supérieure pour le temps de transmission d'un message; si cette borne ne peut être établie, le système est dit asynchrone.

Ces caractéristiques se combinent comme suit :

- Fixe, imprévisible. C'est le cas le plus courant, aussi bien pour les réseaux locaux que pour les réseaux à grande distance (comme l'Internet). Bien que l'on puisse souvent estimer une moyenne pour la durée de transmission, il n'est pas possible de lui garantir une borne supérieure.

- Fixe, prévisible. Cette combinaison s'applique aux environnements spécialement développés pour des applications ayant des contraintes particulières, comme les applications critiques en temps réel. Le protocole de communication garantit alors une borne supérieure pour le temps de transfert, en utilisant la réservation préalable de ressources.

- Variable, imprévisible. C'est le cas de systèmes de communication qui comprennent des appareils mobiles (dits aussi nomades) tels que les téléphones mobiles ou les assistants personnels. Ces appareils utilisent la communication sans fil, qui est sujette à des variations imprévisibles de performances. Les environnements dits ubiquitaires, ou omniprésents, permettant la connexion et la déconnexion dynamique de dispositifs très variés, appartiennent à cette catégorie.

Avec les techniques actuelles de communication, la classe (variable, prévisible) est vide. L'imprévisibilité des performances du système de communication rend difficile la tâche de garantir aux applications des niveaux de qualité spécifiés. L'adaptabilité, c'est-à-dire la capacité à réagir à des variations des performances des communications, est la qualité principale requise de l'intergiciel dans de telles situations.

1.2.2 Services et interfaces

Un système matériel et/ou logiciel est organisé comme un ensemble de parties, ou composants[1].

Le système entier, et chacun de ses composants, remplit une fonction qui peut être décrite comme la fourniture d'un service. Selon une définition de Bieber, G. et Carpenter, J. [3] « un service est un comportement défini par contrat, qui peut être réalisé et fourni par tout composant pour être utilisé par tout composant, sur la base unique du contrat ».

Pour fournir ses services, un composant repose généralement sur des services qu'il demande à d'autres composants. Par souci d'uniformité, le système entier peut être considéré comme un composant, qui interagit avec un environnement externe spécifié; le service fourni par le système repose sur des hypothèses sur les services que ce dernier reçoit de son environnement.

La fourniture de services peut être considérée à différents niveaux d'abstraction. Un service fourni est généralement matérialisé par un ensemble d'interfaces, dont chacune représente un aspect du service. L'utilisation de ces interfaces repose sur des patrons élémentaires d'interaction entre les composants du système. Dans la section 1.2.2.1, nous passons brièvement en revue ces patrons d'interaction. Les interfaces sont discutées dans la section 1.2.2.2.

1.2.2.1 Mécanismes d'interaction de base

Les composants interagissent via un système de communication sous-jacent. Nous supposons acquises les notions de base sur la communication et nous examinons quelques patrons d'interaction utilisés pour la fourniture de services.

La forme la plus simple de communication est un événement transitoire asynchrone (Figure 1.6a). Un composant A (plus précisément, un *thread* s'exécutant dans le composant A) produit un événement (c'est-à-dire envoie un message élémentaire à un ensemble spécifié de destinataires), et poursuit son exécution. Le message peut être un simple signal, ou peut porter une valeur. L'attribut «transitoire » signifie que le message est perdu s'il n'est pas attendu. La réception de l'événement par le composant B déclenche une réaction, c'est-à-dire lance l'exécution d'un programme (le traitant) associé à cet événement. Ce mécanisme peut être utilisé par A pour demander un service à B, lorsqu'aucun résultat n'est attendu en retour; ou il peut être utilisé par B pour observer ou surveiller l'activité de A.

[1] Dans cette section, nous utilisons le terme de composant dans un sens non-technique, pour désigner une unité de décomposition d'un système.

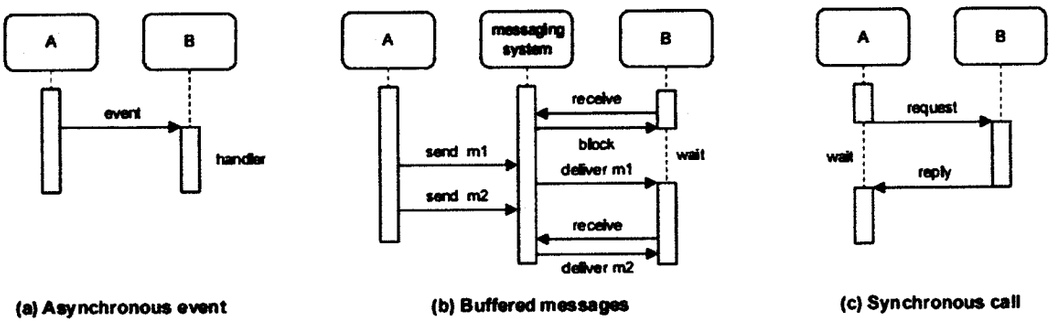


Figure 1.6 – Quelques mécanismes de base pour l'interaction [4]

Une forme de communication plus élaborée est le passage asynchrone de messages persistants (1.6b). Un message est un bloc d'information qui est transmis d'un émetteur à un récepteur. L'attribut « persistant » signifie que le système de communication assure un rôle de tampon : si le récepteur attend le message, le système de communication le lui délivre; sinon, le message reste disponible pour une lecture ultérieure.

Un autre mécanisme courant est l'appel synchrone (1.6c), dans lequel A (le client d'un service fourni par B) envoie un message de requête à B et attend une réponse. Ce patron est utilisé dans le RPC.

Les interactions synchrone et asynchrone peuvent être combinées, par exemple dans diverses formes de « RPC asynchrone ». Le but est de permettre au demandeur d'un service de continuer son exécution après l'envoi de sa requête. Le problème est alors pour le demandeur de récupérer les résultats, ce qui peut être fait de plusieurs manières. Par exemple, le fournisseur peut informer le demandeur, par un événement asynchrone, que les résultats sont disponibles; ou le demandeur peut appeler le fournisseur à un moment ultérieur pour connaître l'état de l'exécution.

Il peut arriver que la fourniture d'un service par B à A repose sur l'utilisation par B d'un service fourni par A (le contrat entre fournisseur et client du service engage les deux parties). Par exemple, dans la figure 1.7a, l'exécution de l'appel depuis A vers B repose sur un *rappel* (en anglais *callback*) depuis B à une fonction fournie par A. Sur cet exemple, le rappel est exécuté par un nouveau thread, tandis que le thread initial continue d'attendre la terminaison de son appel.

Les exceptions sont un mécanisme qui traite les conditions considérées comme sortant du cadre de l'exécution normale d'un service : pannes, valeurs de paramètres hors limites, etc. Lorsqu'une telle condition est détectée, l'exécution du service est proprement terminée (par exemple les ressources sont libérées) et le contrôle est rendu à l'appelant, avec une information su la nature de l'exception. Une exception peut ainsi être considérée comme un «rappel à sens unique». Le demandeur du service doit fournir un traitant pour chaque exception possible.

La notion de rappel peut encore être étendue. Le service fourni par B à A peut être demandé depuis une source extérieure, A fournissant toujours à B une ou plusieurs interfaces de rappel. Ce patron d'interaction (Figure 1.7b) est appelé *inversion du contrôle*, parce que le flot de contrôle va de B (le fournisseur) vers A (le demandeur). Ce cas se produit notamment lorsque B « contrôle » A, c'est-à--dire lui fournit des services d'administration tels que la surveillance ou la sauvegarde persistante; dans cette situation, la demande de service a une origine externe (elle est par exemple déclenchée par un événement extérieur tel qu'un signal d'horloge).

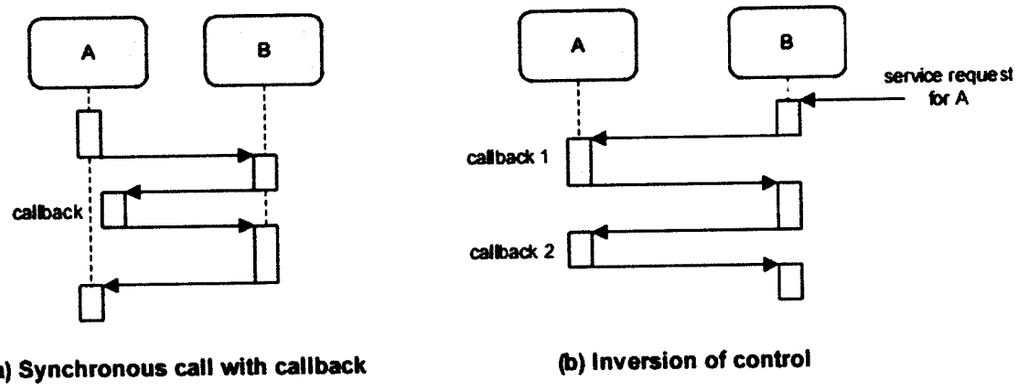


Figure 1.7 – Inversion du contrôle [4]

1.2.2.2 Interfaces

Un service élémentaire fourni par un composant logiciel est défini par une *interface*, qui est une description concrète de l'interaction entre le demandeur et le fournisseur du service. Un service complexe peut être défini par plusieurs interfaces, dont chacune représente un aspect particulier du service. Il y a en fait deux vues complémentaires d'une interface.

- la vue d'usage : une interface définit les opérations et structures de données utilisées pour la fourniture d'un service;
- la vue contractuelle : une interface définit un contrat entre le demandeur et le fournisseur d'un service.

La définition effective d'une interface requiert donc une représentation concrète des deux vues, par exemple un langage de programmation pour la vue d'usage et un langage de spécification pour la vue contractuelle.

Rappelons qu'aussi bien la vue d'usage que la vue contractuelle comportent deux partenaires. En conséquence, la fourniture d'un service implique en réalité deux interfaces : l'interface présentée par le composant qui fournit un service, et l'interface attendue par le client du service. L'interface fournie (ou serveur) doit être « conforme » à l'interface requise (ou client), c'est-à-dire compatible avec elle; nous revenons plus loin sur la définition de la conformité.

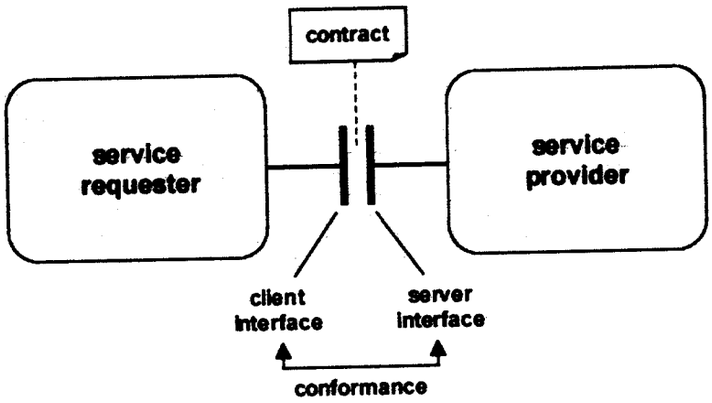


Figure 1.8 – Interfaces [4]

La représentation concrète d'une interface, fournie ou requise, consiste en un ensemble d'opérations, qui peut prendre des formes diverses, correspondant aux patrons d'interaction décrits en 1.2.2.1 :

- procédure synchrone ou appel de méthode, avec paramètres et valeur de retour; accès à un attribut, c'est-à-dire à une structure de données (cette forme peut être convertie dans la précédente au moyen de fonctions d'accès (en lecture, en anglais *getter* ou en écriture, en anglais *setter*) sur les éléments de cette structure de données);
- appel de procédure asynchrone;
- source ou puits d'événements;
- flot de données fournisseur (*output channel*) ou récepteur (*input channel*);

Le contrat associé à l'interface peut par exemple spécifier des contraintes sur l'ordre d'exécution des opérations de l'interface (par exemple ouvrir un fichier avant de le lire). Diverses notations, appelées Langages de Description d'Interface(IDL), ont été conçues pour décrire formellement des interfaces. Il n'y a pas actuellement de modèle unique commun pour un IDL, mais la syntaxe de la plupart des IDLs existants est inspirée par celle d'un langage de programmation procédural. Certains langages (par exemple Java, C#) comportent une notion d'interface et définissent donc leur propre IDL. Une définition d'interface typique spécifie la signature de chaque opération, c'est-à-dire son nom, son type et le mode de transmission de ses paramètres et valeurs de retour, ainsi que les exceptions qu'elle peut provoquer à l'exécution (le demandeur doit fournir des traitants pour ces exceptions).

La représentation d'une interface, avec le contrat associé, définit complètement l'interaction entre le demandeur et le fournisseur du service représenté par l'interface. En conséquence, ni le demandeur ni le fournisseur ne doit faire d'autre hypothèse sur son partenaire que celles explicitement spécifiées dans l'interface. En d'autres termes, tout ce qui est au-delà de l'interface est vu par chaque partenaire comme une «boite noire». C'est le *principe d'encapsulation*, qui est un cas particulier de la séparation des préoccupations (voir section 1.3). Le principe d'encapsulation assure l'indépendance entre interface et réalisation, et

permet de modifier un système selon le principe «je branche et ça marche» (*plug and play*) : un composant peut être remplacé par un autre à condition que les interfaces entre le composant remplacé et le reste du système restent compatibles.

1.2.3 Patrons architecturaux pour les systèmes intergiciels

1.2.3.1 Architectures multiniveaux

Architectures en couches

La décomposition d'un système complexe en niveaux d'abstraction est un ancien et puissant principe d'organisation. Il régit beaucoup de domaines de la conception des systèmes, via des notions largement utilisées telles que les machines virtuelles et les piles de protocoles.

L'abstraction est une démarche de conception visant à construire une vue simplifiée d'un système sous la forme d'un ensemble organisé d'interfaces, qui ne rendent visibles que les aspects jugés pertinents. La réalisation de ces interfaces en termes d'entités plus détaillées est laissée à une étape ultérieure de raffinement. Un système complexe peut ainsi être décrit à différents niveaux d'abstraction. L'organisation la plus simple (Figure 1.9a) est une hiérarchie de couches, dont chaque niveau *i* définit ses propres entités, qui fournissent une interface au niveau supérieur (*i*+1). Ces entités sont réalisées en utilisant l'interface fournie par le niveau inférieur (*i*-1), jusqu' à un niveau de base prédéfini (généralement réalisé par matériel).

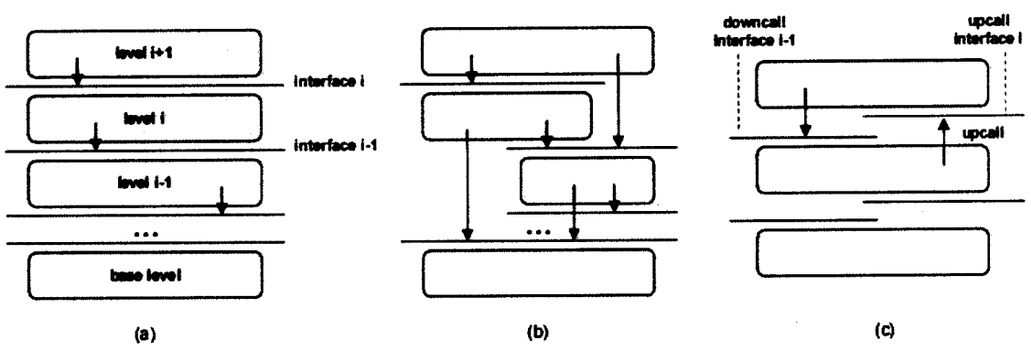


Figure 1.9 - Organisations de systèmes en couche [4]

L'interface fournie par chaque niveau peut être vue comme un ensemble de fonctions définissant une bibliothèque, auquel cas elle est souvent appelée API (*Application Programming Interface*). Une vue alternative est de considérer chaque niveau comme une machine virtuelle, dont le «langage» (le jeu d'instructions) est défini par son interface. En vertu du principe d'encapsulation, une machine virtuelle masque les détails de réalisation de tous les niveaux inférieurs. Les machines virtuelles ont été utilisées pour émuler un ordinateur ou un système d'exploitation au-dessus d'un autre, pour émuler un nombre quelconque de ressources identiques par multiplexage d'une ressource physique, ou pour réaliser

l'environnement d'exécution d'un langage de programmation (par exemple la *Java Virtual Machine* (JVM)).

Ce schéma de base peut être étendu de plusieurs manières. Dans la première extension (Figure 1.9b), une couche de niveau *i* peut utiliser tout ou partie des interfaces fournies par les machines de niveau inférieur. Dans la seconde extension, une couche de niveau *i* peut rappeler la couche de niveau *i*+1, en utilisant une interface de rappel (*callback*) fournie par cette couche. Dans ce contexte, le rappel est appelé « appel ascendant » (*upcall*) (par référence à la hiérarchie « verticale » des couches).

Bien que les appels ascendants puissent être synchrones, leur utilisation la plus fréquente est la propagation d'événements asynchrones vers le haut de la hiérarchie des couches. Considérons la structure d'un noyau de système d'exploitation. La couche supérieure (application) active le noyau par appels descendants synchrones, en utilisant l'API des appels système. Le noyau active aussi les fonctions réalisées par le matériel (par exemple mettre à jour la MMU, envoyer une commande à un disque) par l'équivalent d'appels synchrones. En sens inverse, le matériel active typiquement le noyau via des interruptions asynchrones (appels ascendants), qui déclenchent l'exécution de traitants. Cette structure d'appel est souvent répétée aux niveaux plus élevés : chaque couche reçoit des appels synchrones de la couche supérieure et des appels asynchrones de la couche inférieure.

Architectures multiétages

Le développement des systèmes répartis a promu une forme différente d'architecture multiniveaux. Considérons l'évolution historique d'une forme usuelle d'applications client-serveur, dans laquelle les demandes d'un client sont traitées en utilisant l'information stockée dans une base de données.

Dans les années 1970 (Figure 1.10a), les fonctions de gestion de données et l'application elle-même sont exécutées sur un serveur central (mainframe). Le poste du client est un simple terminal, qui réalise une forme primitive d'interface utilisateur.

Dans les années 1980 (Figure 1.10b), les stations de travail apparaissent comme machines clientes, et permettent de réaliser des interfaces graphiques élaborées pour l'utilisateur. Les capacités de traitement de la station cliente lui permettent en outre de participer au traitement de l'application, réduisant ainsi la charge du serveur et améliorant la capacité de croissance (car l'addition d'une nouvelle station cliente ajoute de la puissance de traitement pour les applications).

L'inconvénient de cette architecture est que l'application est maintenant à cheval sur les machines client et serveur; l'interface de communication est à présent interne à l'application. Une modification de cette dernière peut maintenant impliquer des changements à la fois sur les machines client et serveur, et éventuellement une modification de l'interface de communication.

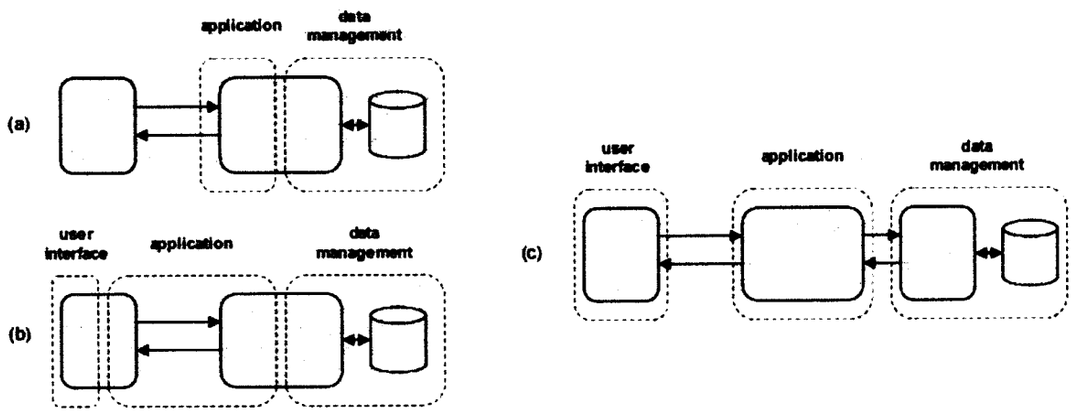


Figure 1.10 -Architectures multiétages [4]

Ces défauts sont corrigés par l'architecture décrite sur la Figure 1.10c, introduite à la fin des années 1990. Les fonctions de l'application sont partagées entre trois machines : la station client ne réalise que l'interface graphique, l'application proprement dite réside sur un serveur dédié, et la gestion de la base de données est dévolue à une autre machine.

Chacune de ces divisions « horizontales » est appelée un *étage* (en anglais *tier*). Une spécialisation plus fine des fonctions donne lieu à d'autres architectures multiétages. Noter que chaque étage peut lui-même faire l'objet d'une décomposition «verticale » en niveaux d'abstraction.

L'architecture multiétages conserve l'avantage du passage à grande échelle, à condition que les serveurs puissent être renforcés de manière incrémentale (par exemple en ajoutant des machines à une grappe). En outre les interfaces entre étages peuvent être conçues pour favoriser la séparation de préoccupations, puisque les interfaces logiques coïncident maintenant avec les interfaces de communication. Par exemple, l'interface entre l'étage d'application et l'étage de gestion de données peut être rendue générique, pour accepter facilement un nouveau type de base de données, ou pour intégrer une application patrimoniale, en utilisant un adaptateur (section 2.3.4) pour la conversion d'interface.

1.2.3.2 Objets répartis

Programmation par objets

Les objets ont été introduits dans les années 1960 comme un moyen de structuration des systèmes logiciels. Il existe de nombreuses définitions des objets, mais les propriétés suivantes en capturent les concepts les plus courants, particulièrement dans le contexte de la programmation répartie.

Un *objet*, dans un modèle de programmation, est une représentation logicielle d'une entité du monde réel (telle qu'une personne, un compte bancaire, un document, une voiture, etc.). Un

objet est l'association d'un état et d'un ensemble de procédures (ou méthodes) qui opèrent sur cet état. Le modèle d'objets que nous considérons a les propriétés suivantes :

- *Encapsulation.* Un objet a une interface, qui comprend un ensemble de méthodes (procédures) et d'attributs (valeurs qui peuvent être lues et modifiées). La seule manière d'accéder à un objet (pour consulter ou modifier son état) est d'utiliser son interface. Les seules parties de l'état visibles depuis l'extérieur de l'objet sont celles explicitement présentes dans l'interface; l'utilisation d'un objet ne doit reposer sur aucune hypothèse sur sa réalisation. Le type d'un objet est défini par son interface. Comme indiqué en 1.2.2.2, l'encapsulation assure l'indépendance entre interface et réalisation. L'interface joue le rôle d'un contrat entre l'utilisateur et le réalisateur d'un objet. Un changement dans la réalisation d'un objet est fonctionnellement invisible à ses utilisateurs, tant que l'interface est préservée.
- *Classes* et *instances*. Une *classe* est une description générique commune à un ensemble d'objets (les instances de la classe). Les instances d'une classe ont la même interface (donc le même type), et leur état a la même structure; mais chaque instance a son propre exemplaire de l'état, et elle est identifiée comme une entité distincte. Les instances d'une classe sont créées dynamiquement, par une opération appelée *instanciation*; elles peuvent aussi être dynamiquement détruites, soit explicitement soit automatiquement (par un ramasse-miettes) selon la réalisation spécifique du modèle d'objet.
- *Héritage*. Une classe peut dériver d'une autre classe par spécialisation, autrement dit par définition de méthodes et/ou d'attributs supplémentaires, ou par redéfinition (surcharge) de méthodes existantes. On dit que la classe dérivée *étend* la classe initiale (ou classe de base) ou qu'elle *hérite* de cette classe. Certains modèles permettent à une classe d'hériter de plus d'une classe (héritage multiple).
- *Polymorphisme*. Le polymorphisme est la capacité, pour une méthode, d'accepter des paramètres de différents types et d'avoir un comportement différent pour chacun de ces types. Ainsi un objet peut être remplacé, comme paramètre d'une méthode, par un objet « compatible ». La notion de compatibilité, ou conformité est exprimée par une relation entre types, qui dépend du modèle spécifique de programmation ou du langage utilisé.

Rappelons que ces définitions ne sont pas universelles, et ne sont pas applicables à tous les modèles d'objets (par exemple il y a d'autres mécanismes que les classes pour créer des instances, les objets peuvent être actifs, etc.), mais elles sont représentatives d'un vaste ensemble de modèles utilisés dans la pratique, et sont mises en œuvre dans des langages tels que Smalltalk, C++, Eiffel, Java, ou C#.

Objets distants

Les propriétés ci-dessus font que les objets sont un bon mécanisme de structuration pour les systèmes répartis.

- L'hétérogénéité est un trait dominant de ces systèmes. L'encapsulation est un outil puissant dans un environnement hétérogène : l'utilisateur d'un objet doit seulement connaître une interface pour cet objet, qui peut avoir des réalisations différentes sur différents sites.
- La création dynamique d'instances d'objets permet de construire un ensemble d'objets ayant la même interface, éventuellement sur des sites distants différents; dans ce cas l'intergiciel doit fournir un mécanisme pour la création d'objets distants, sous la forme de fabriques.
- L'héritage est un mécanisme de réutilisation, car il permet de définir une nouvelle interface à partir d'une interface existante. Il est donc utile pour les développeurs d'applications réparties, qui travaillent dans un environnement changeant et doivent définir de nouvelles classes pour traiter des nouvelles situations. Pour utiliser l'héritage, on conçoit d'abord une classe (de base) générique pour capturer un ensemble de traits communs à une large gamme de situations attendues. Des classes spécifiques, plus spécialisées, sont alors définies par extension de la classe de base. Par exemple, une interface pour un flot vidéo en couleur peut être définie comme une extension de celle d'un flot vidéo générique. Une application qui utilise des flots d'objets vidéo accepte aussi des flots d'objets en couleur, puisque ces objets réalisent l'interface des flots vidéo (c'est un exemple de polymorphisme).

La manière la plus simple et la plus courante pour répartir des objets est de permettre aux objets qui constituent une application d'être situés sur un ensemble de sites répartis (autrement dit, l'objet est l'unité de répartition; d'autres méthodes permettent de partitionner la représentation d'un objet entre plusieurs sites). Une application cliente peut utiliser un objet situé sur un site distant en appelant une méthode de l'interface de l'objet, comme si l'objet était local. Des objets utilisés de cette manière sont appelés objets distants, et leur mode d'interaction est l'appel d'objet distant (*Remote Method Invocation*); c'est la transposition du RPC au monde des objets.

Les objets distants sont un exemple d'une organisation client-serveur. Comme un client peut utiliser plusieurs objets différents situés sur un même site distant, des termes distincts sont utilisés pour désigner le site distant (le site *serveur*) et un objet individuel qui fournit un service spécifique (un objet *servant*). Pour que le système fonctionne, un intergiciel approprié doit localiser une réalisation de l'objet servant sur un site éventuellement distant, envoyer les paramètres sur l'emplacement de l'objet, réaliser l'appel effectif, et renvoyer les résultats à l'appelant. Un intergiciel qui réalise ces fonctions est un courtier d'objets répartis (en anglais *Object Request Broker*, ou ORB).

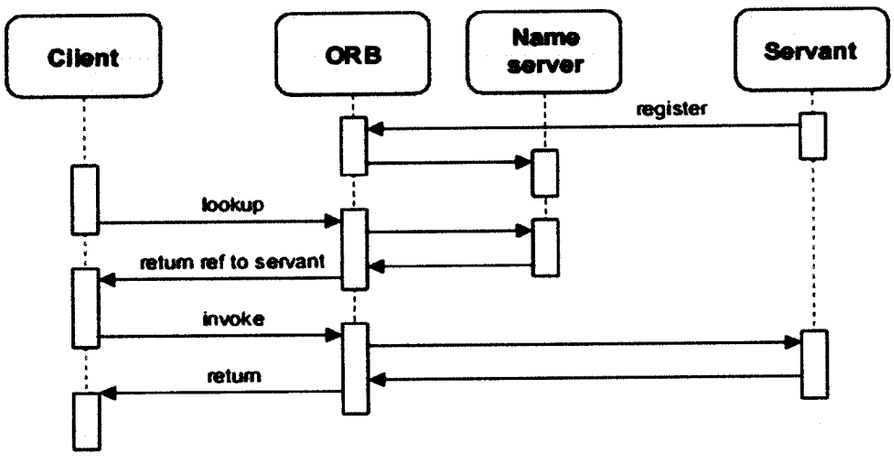


Figure 1.11 – Appel de méthode à distance [4]

La structure d'ensemble d'un appel à un objet distant (Figure 1.11) est semblable à celle d'un RPC : l'objet distant doit d'abord être localisé, ce qui est généralement fait au moyen d'un serveur de noms ou d'un service vendeur (*trader*); l'appel proprement dit est ensuite réalisé. L'ORB sert de médiateur aussi bien pour la recherche que pour l'appel.

1.3 Quelques principes d'architecture

1.3.1 Séparation des préoccupations

En génie logiciel, la séparation des préoccupations (*separation of concerns*) est une démarche de conception qui consiste à isoler, dans un système, des aspects indépendants ou faiblement couplés, et à traiter séparément chacun de ces aspects.

Les avantages attendus sont de permettre au concepteur et au développeur de se concentrer sur un problème à la fois, d'éliminer les interactions artificielles entre des aspects orthogonaux, et de permettre l'évolution indépendante des besoins et des contraintes associés à chaque aspect. La séparation des préoccupations a une incidence profonde aussi bien sur l'architecture de l'intergiciel que sur la définition des rôles pour la répartition des tâches de conception et de développement.

La séparation des préoccupations peut être vue comme un «méta-principe », qui peut prendre diverses formes spécifiques dont nous donnons quatre exemples ci-après.

- le principe d'encapsulation dissocie les préoccupations de l'utilisateur d'un composant logiciel de celles de son réalisateur, en les plaçant de part et d'autre d'une définition commune d'interface.

- Le principe d'abstraction permet de décomposer un système complexe en niveaux (voir section 1.2.3.1), dont chacun fournit une vue qui cache les détails non pertinents, qui sont pris en compte aux niveaux inférieurs.
- La séparation entre politiques et mécanismes est un principe largement applicable, notamment dans le domaine de la gestion et de la protection de ressources. Cette séparation donne de la souplesse au concepteur de politiques, tout en évitant de « sur-spécifier » des mécanismes. Il doit par exemple être possible de modifier une politique sans avoir à réimplémenter les mécanismes qui la mettent en œuvre.
- Le principe de la persistance orthogonale sépare la définition de la durée de vie des données d'autres aspects tels que le type des données ou les propriétés de leurs fonctions d'accès.

Dans un sens plus restreint, la séparation des préoccupations vise à traiter des aspects dont la mise en œuvre (au moins dans l'état de l'art courant) est dispersée entre différentes parties d'un système logiciel, et étroitement imbriquée avec celle d'autres aspects. L'objectif est de permettre une expression séparée des aspects que l'on considère comme indépendants et, en dernier ressort, d'automatiser la tâche de production du code qui traite chacun des aspects. Des exemples d'aspects ainsi traités sont ceux liés aux propriétés « extra-fonctionnelles » telles que la disponibilité, la sécurité, la persistance, ainsi que ceux liés aux fonctions courantes que sont la journalisation, la mise au point, l'observation, la gestion de transactions. Tous ces aspects sont typiquement réalisés par des morceaux de code dispersés dans différentes parties d'une application.

La séparation des préoccupations facilite également l'identification des rôles spécialisés au sein des équipes de conception et de développement, tant pour l'intergiciel proprement dit que pour les applications. En se concentrant sur un aspect particulier, la personne ou le groupe qui remplit un rôle peut mieux appliquer ses compétences et travailler plus efficacement.

1.3.2 Evolution et adaptation

Les systèmes logiciels doivent s'accommoder au changement. En effet, les spécifications évoluent à mesure de la prise en compte des nouveaux besoins des utilisateurs, et la diversité des systèmes et organes de communication entraîne des conditions variables d'exécution. S'y ajoutent des événements imprévisibles comme les variations importantes de la charge et les divers types de défaillances. La conception des applications comme celle de l'intergiciel doit tenir compte de ces conditions changeantes : l'évolution des programmes répond à celle des besoins, leur adaptation dynamique répond aux variations des conditions d'exécution.

Pour faciliter l'évolution d'un système, sa structure interne doit être rendue accessible. Il y a une contradiction apparente entre cette exigence et le principe d'encapsulation, qui vise à cacher les détails de la réalisation.

Ce problème peut être abordé par plusieurs voies. Des techniques pragmatiques, reposant souvent sur l'interception, sont largement utilisées dans les intergiciels commerciaux. Une approche plus systématique utilise la réflexivité. Smith, B. C. in [4] définit un système

réflexif comme un système qui fournit une représentation de lui-même permettant de l'observer et de l'adapter, c'est-à-dire de modifier son comportement. Pour être cohérente, cette représentation doit être *causalement connectée* au système : toute modification apportée au système doit se traduire par une modification homologue de sa représentation, et vice versa. Les *méta-objets* fournissent une telle représentation explicite des mécanismes de base d'un système, ainsi que des protocoles pour examiner et modifier cette représentation. La programmation par aspects, une technique destinée à assurer la séparation des préoccupations, est également utile pour réaliser l'évolution dynamique d'un système.

1.4 Défis de l'intergiciel

Les concepteurs des futurs systèmes intergiciels sont confrontés à plusieurs défis.

- Performances. Les systèmes intergiciels reposent sur des mécanismes d'interception et d'indirection, qui induisent des pertes de performances. L'adaptabilité introduit des indirections supplémentaires, qui aggravent encore la situation. Ce défaut peut être compensé par diverses méthodes d'optimisation, qui visent à éliminer les coûts supplémentaires inutiles en utilisant l'injection[6] directe du code de l'intergiciel dans celui des applications. La souplesse d'utilisation doit être préservée, en permettant d'annuler, en cas de besoin, l'effet de ces optimisations.
- Passage à grande échelle. A mesure que les applications deviennent de plus en plus étroitement interconnectées et interdépendantes, le nombre d'objets, d'usagers et d'appareils divers composant ces applications tend à augmenter. Cela pose le problème de la capacité de croissance (*scalability*) pour la communication et pour les algorithmes de gestion d'objets, et accroît la complexité de l'administration (ainsi, on peut s'interroger sur la possibilité d'observer l'état d'un très grand système réparti, et sur le sens même que peut avoir une telle notion). Le passage à grande échelle rend également plus complexe la préservation des différentes formes de qualité de service.
- Ubiquité. L'informatique ubiquitaire (ou omniprésente) est une vision du futur proche, dans laquelle un nombre croissant d'appareils (capteurs, processeurs, actionneurs) inclus dans divers objets physiques participent à un réseau d'information global. La mobilité et la reconfiguration dynamique seront des traits dominants de ces systèmes, imposant une adaptation permanente des applications. Les principes d'architecture applicables aux systèmes d'informatique ubiquitaire restent encore largement à élaborer.
- Administration. L'administration de grandes applications hétérogènes, largement réparties et en évolution permanente, soulève de nombreuses questions telles que l'observation cohérente, la sécurité, l'équilibre entre autonomie et interdépendance pour les différents sous-systèmes, la définition et la réalisation des politiques d'allocation de ressources, etc.

1.5 Conclusion

Les architectures des systèmes intergiciels ont connu différentes évolutions (voir section 1.2.3). Une de ces évolutions ayant le plus attitré l'attention fût l'intergiciel à objets répartis.

En effet, les technologies intergicielles ont pour but le support d'un modèle de développement de haut niveau focalisé sur des questions métiers dans le cadre d'une exécution des applications en environnement réparti, et le modèle de développement de la conception et de la programmation orientée objet occupe une place importante dans le domaine du génie logiciel.

Les objets cependant proposés comme première approche du concept de réutilisation introduit par McIlroy [5] pour la mise en place d'une industrie du logiciel où les applications seraient conçues par assemblages de composants ont montré leurs limites en termes d'unité de composition tant sur le plan métier que sur le plan de la répartition. Un grand nombre de patrons de conception furent mis au point pour répondre à ces limites, mais de par leur complexité et leur grand nombre, leur utilisation n'était possible que par les experts. De plus parmi les patrons proposés, certains étaient complémentaires et d'autres en « compétition », une situation rendant difficile le tri et même parfois le choix entre plusieurs implémentations du même algorithme.

L'émergence du concept de composant semble ainsi répondre à un besoin d'uniformisation et de systématisation face à cette profusion de recettes d'experts face aux problèmes posés. A ceci s'ajoute également une volonté évidente d'extraire certains concepts comme réellement fondamentaux, aussi fondamentaux que la notion d'objets elle-même. A l'instar des objets répartis nés du portage des concepts de l'objet en environnement réparti, les composants aussi donnèrent naissance aux composants répartis pour les environnements répartis. Le chapitre 2 consacré aux composants étudie les concepts de base de ce paradigme ainsi que leur représentation et cycle de vie.

Chapitre 2
Composants logiciels

2.1 Introduction

Les besoins croissants des utilisateurs de systèmes logiciels, l'évolution rapide du matériel, l'explosion des réseaux informatiques ont motivé l'émergence de nouvelles approches de développement d'applications à grande échelle comme celles basées sur le paradigme de composants.

Il semble naturel de rapprocher le paradigme de composant du paradigme objet, puisque tous deux ont en commun de privilégier le développement d'un système, et ce, depuis sa phase de spécification jusqu'à son implémentation. Bien que chacun d'eux porte l'accent sur des aspects différents du développement logiciel en termes de niveaux d'abstraction, de granularité et de mécaniques opératoires, il n'ya pas de frontières très nettes entre les deux. En ce sens, il n'est pas étonnant de retrouver des points communs tant au niveau des motivations, des techniques que des méthodes. D'ailleurs, Garlan, D. in [6] nous fait remarquer que les développeurs et concepteurs de systèmes à base de composants sont souvent tentés d'utiliser les modèles à objets comme support de spécification, de conception et d'implémentation. Chaque composant peut être représenté, par exemple par une classe, les interfaces des composants peuvent être représentées par des interfaces de classes et les interactions entre composants peuvent être définies en termes d'association et d'appels de méthodes.

La justification de ce choix selon Booch, G. in [6] tient au fait que les modèles à objets :

- sont familiers à une large communauté d'ingénieurs et de développeurs du logiciel ;
- fournissent souvent une correspondance directe de la spécification à l'implémentation ;
- sont supportés par des outils commerciaux ;
- reposent sur des méthodologies bien définies pour développer des systèmes à partir d'un ensemble de besoins.

Cependant, force est de constater que l'approche objet souffre d'un certain nombre de lacunes au regard de systèmes à base de composants. Nous en donnons ci-après les plus significatives :

- l'approche objet a montré des limites importantes en termes de granularité et dans le passage à l'échelle. Le faible niveau de réutilisation des objets est du en partie au fort couplage des objets. En effet, ces derniers peuvent communiquer sans passer par leur interface. Les composants avec un niveau d'abstraction plus élevé que les objets sont mieux adaptés à une industrie de réutilisation ;
- la structure des applications objets est peu lisible (un ensemble de fichiers) ;
- la plupart des mécanismes objets sont gérés manuellement (création des instances ; gestion des dépendances entre classes, appels explicites de méthodes, etc.);
- il y a peu ou pas d'outils pour déployer (installer) les exécutables sur les différents sites ;
- des architectures objets :
  - spécifient seulement les services fournis par les composants d'implémentation mais ne définissent en aucun cas les besoins requis par ces composants,
  - ne fournissent pas (ou peu) de support direct pour caractériser et analyser les propriétés non fonctionnelles,
  - fournissent un nombre limité de formes d'interconnexion de primitives (invocation de méthodes), rendant difficile la prise en compte d'interactions complexes,
  - proposent peu de solutions pour faciliter l'adaptation et l'assemblage d'objets,
  - prennent en compte difficilement les évolutions d'objets (ajout, suppression, modification, changement de mode de communication, etc.),
- en général, les modèles objets :
  - ne sont pas adaptés à la description de schémas de coordination et de communication complexes. En effet, ils ne se basent pas sur des techniques de construction d'applications qui intègrent d'une manière homogène des entités logicielles hétérogènes provenant de diverses sources,
  - disposent de faibles supports pour les descriptions hiérarchiques, rendant difficiles la description de systèmes à différents niveaux d'abstraction,
  - permettent difficilement la définition de l'architecture globale d'un système avant la construction complète (implémentation) de ses composants, en effet, les modèles objets exigent l'implémentation de leurs composants avant que l'architecture ne soit complètement définie,
- enfin, la recherche des relations d'interactions dans une architecture objet n'est pas une tâche facile (surtout en présence de pointeurs et autres) et requiert forcément l'inspection de tout le système.

Selon le Petit Robert, un composant est un « *élément qui entre dans la composition de quelque chose et qui remplit une fonction particulière* ».

Le terme composant existe dans le vocabulaire informatique depuis la naissance du génie logiciel. Il a cependant d'abord désigné des fragments de code alors qu'aujourd'hui il englobe toute unité de réutilisation [5].

A l'origine, l'approche par composants est fortement inspirée des composants de circuits électroniques. L'expérience gagnée dans cette discipline a largement contribué aux idées de composants et de réutilisation. Concernant les composants électroniques, il suffit de connaître leur principe de fonctionnement et la façon dont ils communiquent avec leur environnement (tension d'entrée, de sortie,...) pour construire un système complet sans pour autant dévoiler les détails de leur implémentation. Comme le souligne Cox, B.J. in [6], l'idée de circuits logiciels intégrés conduit à la notion d'éléments logiciels qui peuvent être connectés ou déconnectés d'un circuit plus complexe, remplacés et/ou configurés. Les constructeurs d'applications adoptent de plus en plus cette démarche. Il s'agit alors d'agencer des composants logiciels de natures très diverses pour construire une application, on ne parlera alors plus de programmation d'applications mais plutôt de composition d'applications.

Aujourd'hui, le développement d'applications à base de composants constitue une voie prometteuse. Pour réduire la complexité de ce développement, le coût de maintenance et accroître le niveau de réutilisabilité, Booch, G. ; Rumbaugh, J. & Jacobson, I. [7] proposent l'adoption de deux principes fondamentaux : *acheter plutôt que construire et réutiliser plutôt qu'acheter*.

Le concept de réutilisation de composant logiciel a été introduit par McIlroy [6] en 1968 dans une conférence de l'OTAN consacrée à la crise du logiciel suite à échec patent de la part de développeurs pour livrer des logiciels de qualité à temps et à prix compétitif. L'idée est de mettre en place une industrie du, logiciel pour accroître la productivité des développeurs et réduire le temps de mise sur le marché (*time-to-market*). Pour ce faire, les développeurs construiraient des applications à partir de composants logiciels sur étagère (COTS, *Commercial Off The Shelf*) plutôt que de les créer à partir de rien (*from scratch*). Ainsi, la construction d'applications pourrait se faire d'une manière plus rapide en assemblant des composants préfabriqués.

La popularité croissante des technologies logicielles à base de composants, à l'instar des technologies à objets, est entrain de s'installer et de se propager auprès de millions de programmeurs. Pour construire rapidement des applications, les nouvelles technologies logicielles adoptent de plus en plus le concept de « composant » comme clé de voûte de la réutilisation. Parmi ces technologies, nous pouvons citer : Activex, OLE, DCOM, .NET de Microsoft, EJB de SUN, CORBA de l'OMG. Elles permettent d'utiliser des composants et des objets à forte granularité, distribués et complexes.

L'intérêt accru des dix dernières années pour le développement de logiciels à base de composants est motivé principalement par la réduction des coûts et des délais de développement des applications. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant qu'à le concevoir, le coder, le tester, le déboguer et le documenter.

Cependant, plusieurs questions se posent dès qu'on aborde la problématique des composants. De fait, la situation des composants rappelle celle des objets au début des années 70 : une syntaxe disparate qui renvoie à une sémantique ambiguë.

En effet, l'utilisation de l'approche « composants » dans le développement d'applications « grandeur réelle » révèle rapidement des interrogations. Quelle est la définition d'un composant ? Quelle est sa granularité, sa portée ? Comment distinguer et rechercher les composants de manière rigoureuse, les manipuler, les assembler, les réutiliser, les installer dans des contextes matériels et logiciels variant dans le temps. Comment les administrer, les faire évoluer ? Etc.

Le but de ce chapitre est d'apporter les notions de base nécessaires à la compréhension du paradigme de composants, de caractériser leurs propriétés et enfin d'exposer leur cycle de vies.

2.2 Présentation des composants

Cette partie est organisée de la façon suivante. La section 2.2.1 fait un tour d'horizon sur les différentes acceptions des composants, depuis leur définition jusqu'à leur assemblage et leur réutilisation. La section 2.2.2 aborde les concepts de base permettant la représentation de composants. La section 2.2.3 relate le cycle de vie des composants. Enfin, avant de clore cette partie, nous résumons les principaux apports de l'approche composants ainsi que les problèmes engendrés par ces derniers.

2.2.1 Concepts de composant

Cette section donne certaines définitions du concept de composant, largement reconnues par la communauté scientifique et décrit successivement les dimensions d'un composant, les composants composites et l'idée de réutilisation d'un composant.

2.2.1.1 Notions et définitions

Il existe différentes définitions du concept de composant. Néanmoins, celle de Szyperski [8] semble être la plus consensuelle et la plus répandue dans la littérature. Ainsi un composant est défini comme suit :

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Cette définition est valable à différents niveaux de compréhension (spécification, conception, implémentation). Elle décrit les caractéristiques d'un composant logiciel ainsi que ses dépendances avec l'environnement extérieur sans toutefois porter des contraintes sur le type d'interaction avec d'autres composants ou le monde extérieur.

Une autre définition d'un composant était donnée par Meyer [9] :

A software component is a program element with the following two properties :

1. *It may be used by other program elements, or clients.*
2. *The clients and their authors do not need to be known to the component's authors.*

Enfin, Harris in [6] a donné également une définition bien acceptée dans la communauté « architecture logicielle » :

Un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standard pour pouvoir interopérer.

L'analyse de l'ensemble de ces définitions permet de déduire les propriétés suivantes, à savoir qu'un composant est :

- comme une unité de composition, il est *autodescriptif* : capable de disposer d'un mécanisme d'introspection permettant de connaître et de modifier dynamiquement ses caractéristiques ;
- *composable* : considéré comme une unité de composition, il est censé être connectable avec d'autre composants ;
- *configurable* : il doit être paramétrable via ses propriétés configurables selon un contexte d'exécution particulier ;
- *réutilisable* : il doit être une unité de réutilisation. Une étape d'adaptation sera sans doute nécessaire selon le contexte d'exécution ;
- *autonome* : il peut être déployé (c'est-à-dire diffusé et installé) et exécuté indépendamment des autres composants.

Aujourd'hui, bien qu'il existe une diversité considérable dans les propositions de modèles de composants, tous partagent une base conceptuelle similaire ou ontologie, considérée comme un fondement commun de concepts et d'intérêts pour la description de systèmes à base de composants. Selon divers auteurs cités par [6], les éléments principaux de cette ontologie sont les suivants:

- *Les composants* sont définis comme des unités de composition qui décrivent et/ou assurent des fonctions spécifiques, possèdent des interfaces de besoins et des interfaces de services et un contexte particulier d'exécution. Ils peuvent être déployés indépendamment et composés avec d'autres composants. Ils sont préfabriqués, pré-testés, s'auto-contiennent, et disposent de documentations appropriées et d'un statut de réutilisation bien défini. Les clients, serveurs, *blackboards*, bases de données sont des exemples types de composants. Dans la plupart des systèmes à base de composants, les composants peuvent avoir plusieurs interfaces, chaque interface définissant un point d'interaction entre un composant et son environnement.
- *Les interactions ou connecteurs* représentent les interactions entre composants. Les formes simples d'interaction, comme les pipes, les appels de procédures, les événements en sont des exemples. Cependant, Les connecteurs peuvent aussi représenter des interactions plus complexes, telles qu'un protocole client-serveur ou un lien SQL entre une base de données et une application. Les connecteurs peuvent

aussi disposer d'interfaces qui définissent les rôles joués par les divers participants à l'interaction.

- *Les propriétés* représentent les informations sémantiques des composants et de leurs interactions.
- *Les contraintes/contrats* représentent les moyens permettant à un modèle d'architecture de rester valide durant sa durée de vie et de prendre en compte l'évolution et le remplacement des composants logiciels dans cette architecture. Ces contraintes peuvent inclurent des restrictions sur les valeurs permises de propriété, sur l'utilisation d'un service offert par un composant et garantir par exemple la validité des résultats retournés par ce service.
- *Une architecture* est une spécification des composants d'un système et de leurs interactions. La motivation pour définir une architecture est de fournir un plan précis (ou métamodèle) approprié pour prédire le comportement d'un système avant de le construire et pour guider son développement.
- *La composition/assemblage* permet de construire des applications complexes à partir de composants simples. La composition ou assemblage permet de lier un composant demandant des services à d'autres composants offrant ces dits services.

2.2.1.2 Les trois dimensions d'un composant

Un composant peut être de deux natures :

- *produit* : il s'agit d'une entité « *building block* » autonome passive (entité logicielle ou entité conceptuelle) qu'il est possible d'adapter. Les composants logiciels et les bibliothèques de fonctions mathématiques en font partie ;
- *processus* : un composant processus correspond à une suite d'actions qu'il faut réutiliser pour obtenir un produit final. Ces actions sont souvent encapsulées dans un processeur(ou unité de traitement).Un composant processus correspond en général à des fragments de démarche.

Au regard des travaux existants, un composant produit ou processus doit refléter trois dimensions comme le montre la figure 2.1 :

- son niveau d'abstraction ;
- son mode d'expression ;
- son domaine.

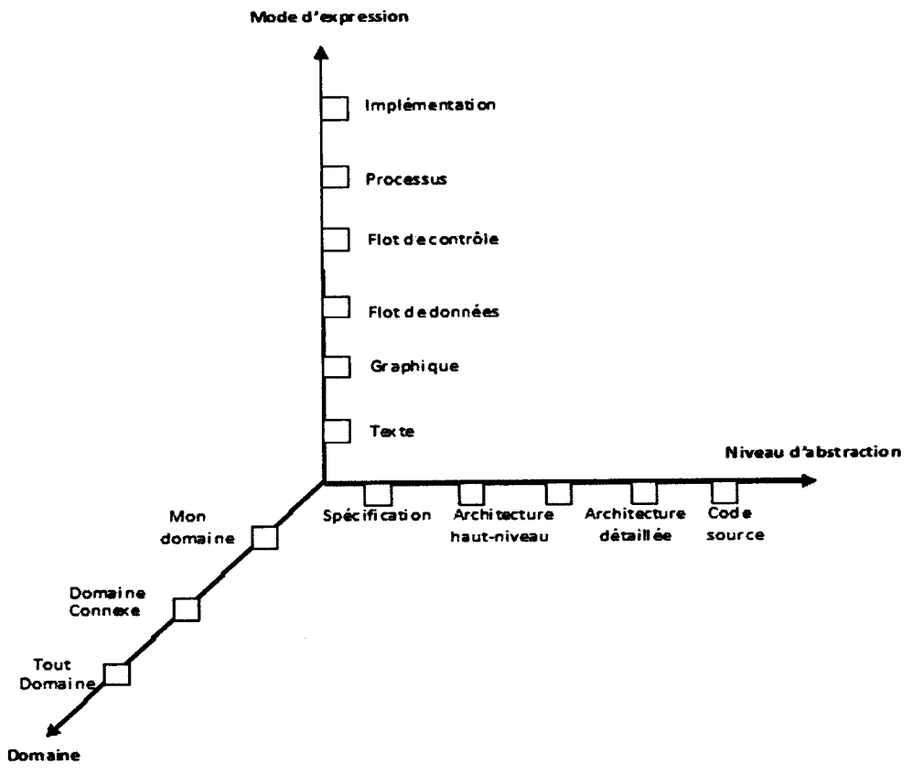


Figure 2.1-Les trois dimensions d'un composant [6]

La première dimension, ou niveau d'abstraction, permet d'exprimer les degrés de raffinement d'un composant, et ce, depuis sa spécification considérée comme étant le plus haut niveau d'abstraction, à son code source représentant le plus bas niveau. Chaque niveau d'abstraction peut lui être associé à *un niveau de transparence* qui définit le niveau de visibilité des détails internes d'un composant lors de (sa) son (ré) utilisation. Le niveau de transparence d'un composant peut être de type :

- *boîte noire* : l'interface du composant fait abstraction de son implémentation et l'interaction avec l'extérieur se fait uniquement à travers l'interface. Dans ce cas, aucun détail autre que l'interface, n'est fourni à l'utilisateur. Mais on donne la possibilité de remplacer un composant qui réalise la même interface ;
- *boîte blanche* : le composant rend transparent tous les détails de son implémentation. L'interaction avec l'extérieur peut se réaliser non seulement à travers l'interface du composant mais aussi à travers son implémentation. Ce type de composant a l'avantage de fournir toute l'information relative à son implémentation ;
- *boîte grise* : il s'agit d'un niveau de transparence intermédiaire entre les composants de type boîte noire et ceux de type boîte blanche. Les détails d'implémentation peuvent être révélés pour comprendre la réalisation du composant, mais ne peuvent

être sujets à des modifications émanant de l'extérieur. L'interaction se fait uniquement à l'interface.

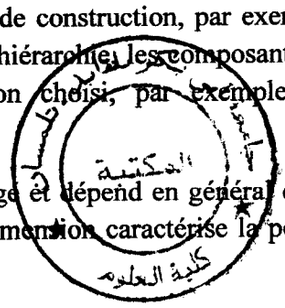
La deuxième dimension, ou mode d'expression, permet de décrire les différents modèles de représentation d'un composant (représentation textuelle, graphique, flot de données, implémentation). La description d'un composant doit être simple, compréhensible avec une sémantique pas nécessairement définie de manière formelle mais, au moins, claire. En effet, les participants à l'élaboration de composants (concepteurs, développeurs, utilisateurs, managers, etc.) peuvent avoir recours à différents modes d'expression. Les utilisateurs peuvent se contenter d'une description graphique de haut niveau (type boîtes et flèches), les développeurs voudront détailler par exemple les modèles de connecteurs et de composants, alors que les managers peuvent requérir une vue du processus de développement.

Le mode d'expression d'un composant dépend également de sa granularité. La notion de granularité de composant recouvre selon les langages de représentation aussi bien des unités atomiques telles que les structures de données, les fonctions mathématiques mais également de véritables structures complexes composées d'éléments et de liens entre ces éléments.

Du point de vue de la granularité, nous pouvons citer :

- *les composants dit simples ou de fine granularité* : directement liés à une construction syntaxique du langage de développement (exprimé souvent à l'aide d'un langage d'implémentation) comme les classes (en programmation objet), les packages (programmation ADA) ou les fonctions (programmation usuelle) ;
- *les composants à granularité moyenne* : par exemple, les composants logiciels utilisés dans la production du logiciel ou les patrons de conception (souvent exprimés de façon textuelle ou graphique) ;
- *les composants dits complexes ou à forte granularité* : le composant s'apparente à une véritable structure complexe comme les infrastructures logicielles ou « *frameworks* », les systèmes d'exploitation, les serveurs, les serveurs, les *blackboards,* etc.

La troisième dimension, ou domaine, reflète les différents domaines structurés en couche et sur lesquels repose un système donné. Cette structuration permet à un composant appartenant à une couche domaine particulière de ne pouvoir interagir qu'avec seulement les composants de la même couche ou de la couche adjacente. Les composants du « Tout domaine » constituent la couche de plus bas niveau et sont indépendants du domaine d'application, c'est le cas par exemple de systèmes d'exploitation, de compilateurs, de systèmes de base de données, etc. Au niveau plus haut, « Domaine connexe », on trouve des composants appartenant à des domaines connexes à l'application en cours de construction, par exemple des interfaces utilisateurs, des simulateurs. Enfin en haut de la hiérarchie, les composants de « mon domaine » sont spécifiques au domaine d'application choisi, par exemple les composants de circuits intégrés.



Il est clair que le nombre de couches de domaines n'est pas figé et dépend en général de la maturité des domaines d'application. In fine, cette troisième dimension caractérise la portée

des composants, c'est-à-dire leur degré de réutilisabilité. Nous pouvons recenser dans la littérature :

- *les composants génériques ou indépendants d'un domaine* : ce sont des composants généraux qui ne dépendent pas d'un domaine particulier. Les objets graphiques, les patrons de conception de Gamma et al. [10] et les types de données abstraits sont des exemples de ce type de composants ;
- *les composants métiers ou dépendants du domaine* : le concept de composant métier résulte de celui d'objet métier. Les composants métiers sont des composants réutilisables à travers les applications d'un même domaine ;
- *les composants orientés applications* : ce sont des composants spécifiques à une application donnée. Ils sont peu réutilisables. Ce type de composant est généralement engendré par des réutilisations ad-hoc non planifiées.

2.2.1.3 Composants composites

Un composant peut être composé d'autres composants, on l'appelle alors « composant composite ». Dans cette section, nous appellerons « composant interne » les composants inclus dans un composant composite.

Les composants internes peuvent être assemblés afin de remplir les services du composant composite. Le composant composite dispose de ses propres interfaces (voir section 2.2.2), et donc de ses propres points de connexion (voir section 2.2.2) et on utilise les correspondances pour faire le lien entre les points de connexion des interfaces des composants internes et ceux du composant composite. Ainsi, pour les services du composant composite rendus par les composants internes, on utilise les correspondances pour indiquer quels points de connexion de quel composant interne rendent les services du composant composite. Cependant, il est toujours possible de rendre les services par des implémentations propres au composant composite.

La mise en œuvre de la composition peut se faire au moment de la compilation, mais aussi au moment de l'exécution. Les compositions les plus utilisées sont :

- *composition structurelle :* ce type de composition est guidé par les interfaces fournies et les interfaces requises des composants. La connexion est assurée à l'aide de protocoles de communication élémentaires (messages, appels de procédures, événements). Cependant, les protocoles complexes (incompatibilité de spécification des interfaces entre composants) nécessitent souvent l'utilisation d'adaptateurs entre les composants pour régler les incompatibilités ;
- *composition comportementale :* l'objectif de ce type de composition est d'avoir un comportement global du composite à partir d'un ensemble de comportements individuels de ses composites ;
- *composition fonctionnelle :* l'objectif de ce type de composition est d'avoir une vue fonctionnelle globale d'une application en assemblant des vues fonctionnelle

existantes. La programmation par aspect est une technique représentative de ce type de composition. Chaque vue fonctionnelle est appelée aspect. L'application est vue comme une composition d'aspects ou de services. Les aspects ne sont pas forcément encapsulés dans les composants structuraux, mais en général ils leur sont transversaux.

2.2.1.4 La réutilisation d'un composant : une idée simple

Contrairement aux approches usuelles dont le développement consiste à partir de rien et à tout réinventer, la réutilisation est une approche qui permet de créer une nouvelle application à partir d'éléments existants.

Le concept de base de la réutilisation d'un composant est expliqué comme suit par McIlroy in [6] :

Développer un composant d'une taille raisonnable et le réutiliser. Ensuite, étendre l'idée du « composant code » à celui des composants besoins, analyse, conception et test. Toutes les étapes du processus de développement sont sujettes à la réutilisation. De cette façon, tous les intervenants dans ce processus minimisent le travail redondant. Ils augmentent la fiabilité du travail, vu que chaque composant réutilisé a déjà été contrôlé et inspecté au cours de son développement originel. Ainsi, le temps de production d'un composant se trouve réduit de façon drastique.

Comme le souligne un auteur rapporté par [6], des expériences menées dans certains projets de firmes internationales montrent un réel gain de temps et de coût dans la réutilisation systématique de composants : AT&T, (niveau de réutilisation entre 40 et 92%), Motorola (jusqu'à 85%), Ericsson (jusqu'à 90%) et Hewlett-Packard (entre 25 et 50%).

Cependant, dans la pratique il est souvent difficile d'expliciter des critères clairs de réutilisation, c'est-à-dire la possibilité d'identifier si un composant a la capacité d'être réutilisé ou d'être candidat. Et même si ces critères sont exhibés, on est confronté à un autre problème lié au manque de flexibilité dans les composants potentiellement réutilisables. En effet, la capacité à adapter un composant pour répondre à un nouveau besoin ou à une nouvelle architecture est souvent limitée. Il n'est donc pas suffisant de disposer d'un modèle de composant riche et expressif, encore faut-il lui associer les bons mécanismes et les bonnes règles pour pouvoir le réutiliser.

Afin de favoriser la réutilisation de composants, le groupe de travail « Architectures de logiciels et réutilisation de composants » de l'Observatoire français des techniques avancées recommande pour l'élaboration d'architectures logicielles à base de composants réutilisables de respecter les principes suivants :

- rester indépendant des infrastructures : il est nécessaire de s'abstraire le plus possible des architectures dépendantes d'une couche intergicielle sujette à des modifications ;
- concevoir les architectures avec les composants : l'idée de réutiliser les composants est liée à celle de les organiser en architecture ;

- mettre en place un processus de développement basé sur la réutilisation de composants ;
- exploiter les nouvelles formes de collaboration : Internet et les logiciels libres démontrent bien que la collaboration client-fournisseur n'est pas l'unique façon de collaborer. Il faut bien entendu en tirer partie.
- rendre les systèmes réutilisables et flexibles par la métamodélisation : le fait de monter en abstraction favorise pleinement la réutilisation.

2.2.2 Représentation d'un composant

Après avoir donné quelques éléments de définition relatifs au paradigme de composant, cette section envisage la représentation d'un composant. Un composant est généralement représenté par les éléments suivants (voir Figure 2.2) :

- *son type :* la définition abstraite du composant ;
- *son implémentation :* la mise en œuvre des aspects fonctionnels et non-fonctionnels de son type ;
- *son instance :* une instance de composant est, au même titre qu'une instance d'objet, une entité exécutable dans un système. Elle est définie par une référence unique, à savoir un type de composant et une implémentation particulière de ce type.

La spécification d'un composant repose d'abord et essentiellement sur son type. Un type de composant est caractérisé par deux éléments : ses interfaces avec leurs modes de connexion et ses propriétés configurables ou non.

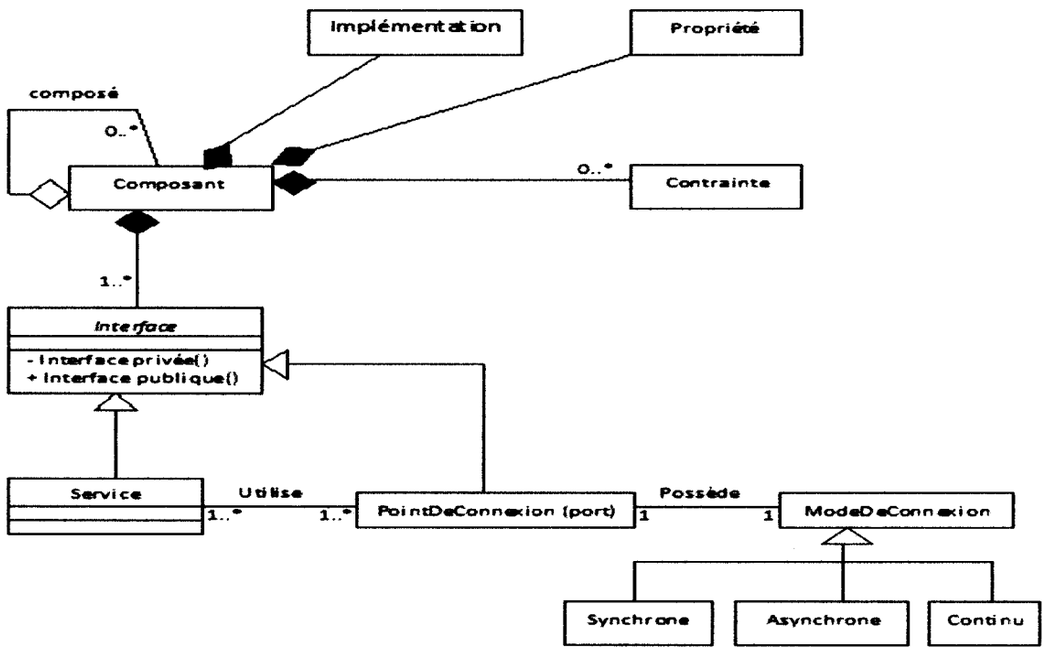


Figure 2.2-Métamodèle des principaux éléments d'un composant [6]

Interfaces

Une interface spécifie les services fournis et requis par un type de composant. Ces derniers lui permettent d'interagir avec son environnement, y compris avec d'autres types de composants.

L'interface sert à déclarer les services fournis et requis et permet de faire le lien avec les autres composants. C'est la seule partie visible d'un composant. L'interface décrit entièrement ce que fait le composant et ce dont il a besoin pour fonctionner. En général, l'intérieur du composant est une boîte noire que les utilisateurs n'ont pas besoin de connaître.

Une interface est composée de deux éléments :

- *les points* de connexion sont les points d'interaction (ils peuvent être des besoins ou des services) entre le composant et son environnement ;
- *les services* décrivent d'une part, le comportement fonctionnel du composant en expliquant ce qu'il fait, on parlera alors de services fournis, et d'autre part, les fonctionnalités dont il a besoin pour fonctionner, il s'agit alors de services requis.

Points de connexion

Le point de connexion est le point d'interaction entre le composant et l'extérieur du composant. Ce point, appelé également port, peut être soit un port fourni (souvent appelé port service) soit un port requis (souvent appelé port besoin).

- *le port requis* est le point de connexion qui exprime le fait que le composant nécessite un service pour fonctionner ;
- *le port fourni* est le point de connexion qui permet de fournir un service aux autres composants.

Ce sont les seules parties véritablement manipulables de l'extérieur et on peut seulement se connecter à ces ports pour recevoir le service voulus ou fournir le besoin demandé. Un port possède aussi un mode de connexion synchrone, asynchrone ou continu et seuls deux ports avec le même mode de connexion peuvent être reliés.

Il existe trois modes de connexion possibles pour les points de connexion : les modes synchrone, asynchrone et de diffusion en continu [6]:

- *Mode de connexion synchrone :* il est généralement utilisé pour les cas de producteurs/consommateur. Ce mode est utilisé pour les services qui ne peuvent être rendus que sous certaines conditions (temporelles ou applicatives) et qui nécessitent une synchronisation entre le composant offrant le service et celui le recevant. Ainsi, le composant client (demandeur de services) appelle une opération de l'interface fournie par le composant serveur et reste bloqué jusqu'au retour de l'opération ;
- *Mode de connexion asynchrone :* il est utilisé pour les services sans contraintes de synchronisation entre les deux composants. Le service peut donc toujours être rendu quelque soit le moment où on le demande ;
- *Mode diffusion en continu :* il est défini par un protocole qui crée un pont de communication continue entre le composant client et le composant serveur.

Services

Les services permettent d'exprimer la sémantique des fonctionnalités fournies et requises par les composants. On peur regrouper les services par catégorie, cela permet de faciliter la recherche de composants. Un service peut utiliser plusieurs points de connexion pour exécuter sa tâche.

Typologie d'interfaces

Dans la littérature nous pouvons recenser différents types d'interfaces qui permettent plus ou moins de compléter la caractérisation d'un composant. Parmi celles-ci, nous citons :

- *interfaces de services fournis :* elles permettent à l'environnement du composant (utilisateur ou autre composant) d'accéder à une fonctionnalité donnée offerte par le composant. Un composant réagit aux changements de son environnement via ses interfaces fournies ;
- *interfaces de services requis :* elles permettent au composant d'accéder à une fonctionnalité donnée de son environnement. Un composant peut faire réagir son environnement via ses interfaces requis.

Les interfaces de services fournis et requis peuvent être soit publiques ou privées :

- *Interfaces publiques :* elles sont utilisées par un composant pour communiquer avec son environnement. Ces interfaces sont visibles et accessibles de l'environnement du composant ;
- *Interfaces privées :* elles sont utilisées par un composant composite pour communiquer avec ses sous-composants ou composants internes. Elles ne sont pas visibles de l'extérieur. Elles sont accessibles uniquement de l'intérieur d'un composant composite, par le composant lui-même ou ses sous-composants.

A l'aide des interfaces privées, un composant composite peut structurer et repartir la définition de ses fonctionnalités dans ses sous-composants.

Une interface de services requis avec accès privé permet au composant composite d'utiliser une fonctionnalité prédéfinie dans un de ses sous-composants. Le composite peut ainsi faire réagir ses sous-composants.

Une interface de services fournis avec accès privé permet aux sous-composants d'accéder aux fonctionnalités de leur composant père et aussi notifier celui-ci en fonction de l'évolution des sous-composants.

Propriétés

Les propriétés servent en général à documenter les détails d'une architecture, d'un composant, d'un connecteur, etc. relevant de leur conception et de leur analyse. Elles deviennent utiles

lorsqu'elles sont utilisées par des outils à des fins de manipulations, d'affichage, d'analyse, etc.

Pour un composant, les propriétés peuvent concerner aussi bien sa structure, son comportement ou ses fonctionnalités. Elles peuvent être paramétrées et configurées selon un contexte d'exécution particulier. Par ailleurs, il existe d'autres propriétés dites non fonctionnelles qui représentent les services utilisées par le composant et qui ne font pas partie des services applicatifs. Ce sont des services de base fournis par l'architecture à travers le serveur et les conteneurs (sécurité réseaux, traçage, etc.).

Les contraintes peuvent également être considérées comme un type spécial de propriétés, mais comme en général elles jouent un rôle clé dans la conception d'architectures à base de composants, les auteurs et développeurs de logiciels fournissent souvent une syntaxe spéciale pour les décrire.

Exemple 2.1

La figure 2.3 montre un exemple d'architecture d'un système client serveur. Cet exemple, emprunté à Gamma et *.al* in [6], définit comment le système client serveur peut être représenté en utilisant les concepts de base d'un modèle de composant (Figure 2.4). Dans ce système, le composant *client* est défini comme un demandeur de services et le composant *serveur* comme un fournisseur de services. Le client a une interface publique (`send-request`) pour communiquer avec le serveur. De son côté, le serveur dispose d'une interface publique (`send-response`) pour communiquer avec le client. Par ailleurs, il est composé de trois composants (Connection-Manager, Security-Manager et Database) et possède trois interfaces privées (`connection-port`, `security-port` et `database-port`) pour communiquer entre eux.

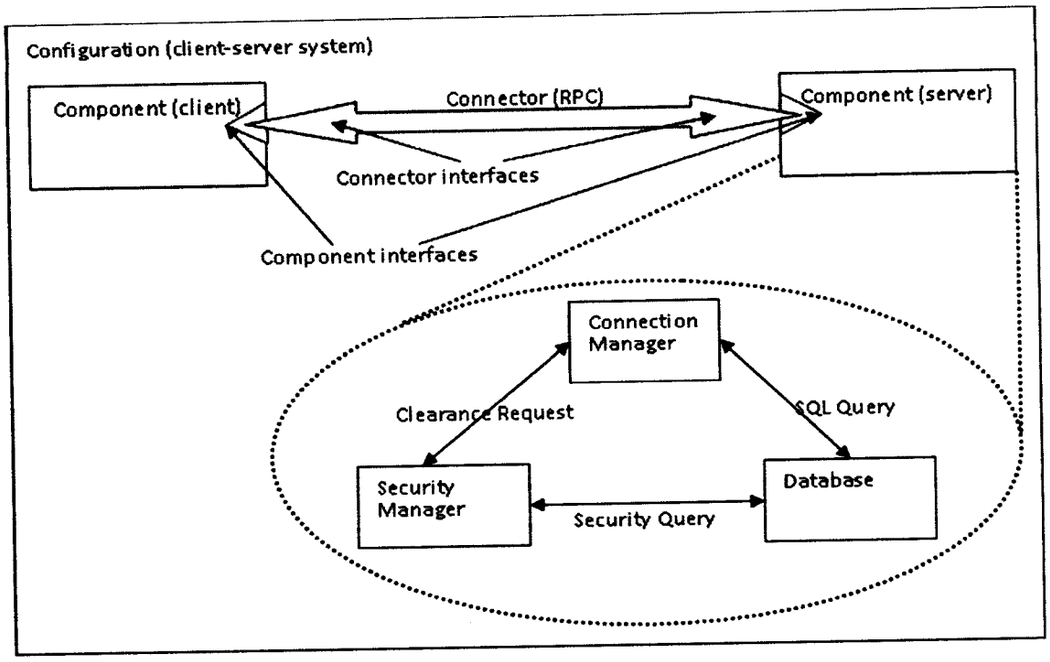


Figure 2.3-Exemple d'architecture du système client-serveur [6]

La communication entre le client et le serveur est réalisée à l'aide d'un connecteur (RPC) qui possède deux interfaces publiques (`caller` et `callee`), l'une est reliée au client et l'autre au serveur. Le serveur définit par ailleurs une propriété indiquant que le nombre maximum de clients qui peuvent solliciter ses services est fixé à un. Le mode de connexion entre le client et le serveur est synchrone.

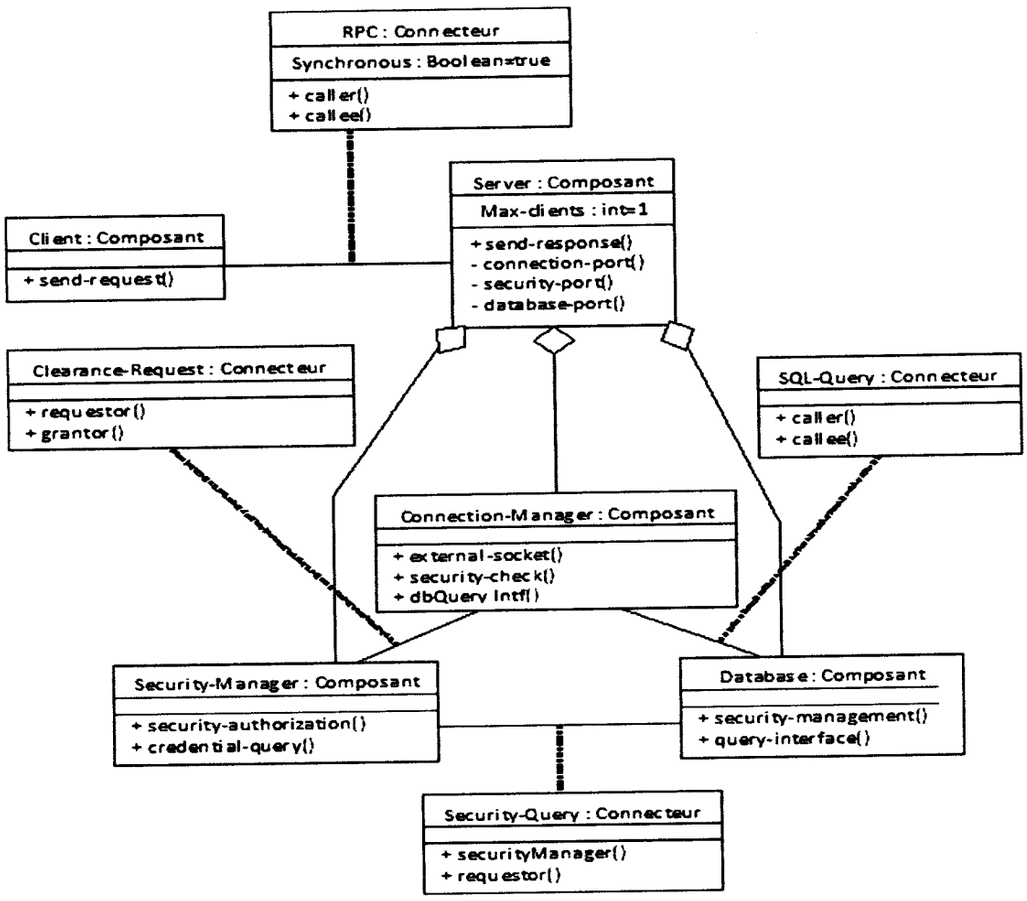


Figure 2.4-Représentation de l'exemple du client

2.2.3 Cycle de vie d'un composant

Le cycle de vie d'un composant suit les étapes de définition et d'analyse des besoins, de conception, d'implantation, de paquetage et diffusion des implantations, d'assemblage/composition, de déploiement et d'exécution.

Définition et analyse des besoins en terme de composant

Cette étape est essentielle dans le processus de construction de logiciel puisqu'elle a comme objectif de produire les composants réutilisables. Des techniques basées sur l'analyse de domaine, la recherche de similarité, d'analogie et de généricité peuvent être utilisées pour identifier systématiquement des composants candidats à la réutilisation. Le résultat de cette étape est en général un ensemble de modèles d'analyse, de diagrammes, par exemple en UML (*Unified Modeling Language* [7]), décrivant les besoins en termes de fonctionnalités d'un type de composant. Cette phase doit être menée sans tenir en compte les contraintes techniques du futur modèle de composants. Quant à la qualité des composants obtenus du point de vue de leur réutilisabilité, elle reste encore très subjective et, à notre connaissance, peu de travaux s'y sont intéressés.

Conception des types de composants

La conception consiste à spécifier les types des composants à partir de la définition et de l'analyse des besoins. Le type de composant est la définition abstraite d'un composant. Cette étape est concernée par la représentation, l'organisation, l'accès, l'adaptation et l'intégration de composants. L'utilisation d'un langage est importante lors de la phase de conception. Ce langage servira notamment à spécifier les trois aspects du composant : ses interfaces, ses propriétés configurables et locales et ses modes d'interaction avec les autres types de composants. Lors de cette étape, il est indispensable de prévoir les mécanismes qui permettent d'adapter, d'intégrer et de rechercher et sélectionner des composants [6]:

- *l'adaptation :* les composants sélectionnés ont souvent besoin d'être soumis à des modifications afin qu'ils soient adéquats pour une utilisation particulière. Les techniques d'adaptation sont nombreuses, depuis la modification du composant jusqu'au mécanisme d'instanciation en passant par la paramétrisation et la spécialisation ;
- *l'intégration :* la réutilisation d'un composant devient effective si celui-ci est intégré à l'application en cours de développement. Le problème d'intégration est simple si les langages de spécification, conception et implémentation sont les mêmes ;
- *la recherche et la sélection :* cette activité consiste à rechercher dans la bibliothèque un composant qui réponde à un problème particulier de développement. Cependant, il arrive souvent que plusieurs composants soient candidats aux besoins. Dans ce cas, il est nécessaire de mettre en œuvre un ou plusieurs mécanismes de sélection qui choisissent le composant le plus proche des besoins exprimés.

Implantation des types de composants

L'implantation d'un composant correspond à la mise en œuvre de son type. Elle doit se réaliser en respectant les spécifications du type de composant visé. Les langages de programmation utilisés pour l'implantation des composants peuvent varier d'un modèle de composant à un autre (EJB utilise en général Java, etc.). En général l'implantation d'un type de composant se résume à l'implantation des parties fonctionnelles. Cependant, en pratique, il s'avère nécessaire de pouvoir exprimer les contraintes techniques de l'implantation (système d'exploitation, *multithreading,* etc.).

Paquetage et diffusion des implantations de composants

La diffusion d'une implantation de composant se fait grâce au paquetage de cette implantation. Une fois les composants implantés, ils sont d'abord archivés dans des paquetages. Un paquetage permet de regrouper dans une même unité le type de composant, son implantation, un ensemble de fichiers descripteurs qui fournissent des informations sur le composant et aident à son déploiement. La diffusion d'un composant nécessite de normaliser le format et le contenu des paquetages (dans EJB et CCM, les paquetages sont respectivement de format JAR et ZIP). Cette normalisation permet en effet de faciliter les échanges de

composants entre développeurs et de favoriser leur réutilisation via des outils multifournisseurs.

Assemblage/composition

L'assemblage consiste à composer tous les composants d'une application. Un assemblage(ou composition) reflète l'architecture d'une application et devrait être lui-même un composant pour être assemblé à son tour. Un assemblage implique non seulement la configuration de chaque composant, mais aussi celle de leurs interconnexions. Pour qu'un assemblage soit diffusé à son tour, et utilisables par des tiers, il devrait être empaqueté. Dans ce cas, le paquetage contient en plus une description de l'assemblage et de la configuration de chaque composant.

Déploiement

Le déploiement d'un composant consiste à l'extraire de son paquetage et à l'installer dans son environnement. Pour mener à bien cette opération (automatique ou semi-automatique), il faut disposer, ce qui n'est pas souvent le cas, de structures d'accueil permettant d'une part d'instancier des composants et de les réceptionner aux endroits désirés et d'autre part de les exécuter. Dans le cas d'EJB, le déploiement des composants est automatique et consiste à mettre en place leur conteneur et à les y instancier et configurer. Le processus de déploiement se base en général sur les descripteurs du composant contenus dans le paquetage pour instancier le composant, vérifier sa compatibilité avec la structure d'accueil et éventuellement le configurer.

Exécution et utilisation

L'étape d'exécution correspond à l'exploitation effective du composant. Les acteurs potentiels de cette phase sont les administrateurs d'applications distribuées qui vont installer et maintenir les applications et les utilisateurs finaux qui vont exploiter ces services.

2.3 Modèles de composants logiciels

Les plateformes intergicielles basées sur la notion de composant proposent des solutions aux limites des approches objets de la génération précédente. Au nombre de ces limites, la séparation des préoccupations entre code fonctionnel, ou logique métier, et code non fonctionnel, ou logique système, se trouve réellement au cœur des développements centrés sur la notion de composant logiciel. Dans ce but, les modèles de composants sont généralement architecturés selon le principe de *l'inversion du contrôle,* également appelé « principe d'Hollywood ». L'idée est de placer le contrôle applicatif, et notamment tout ce qui concerne les interactions entre couches logicielles métier et systèmes, sous l'autorité d'entités extérieures, gérées au sein de la plateforme intergicielle (figure 2.5).

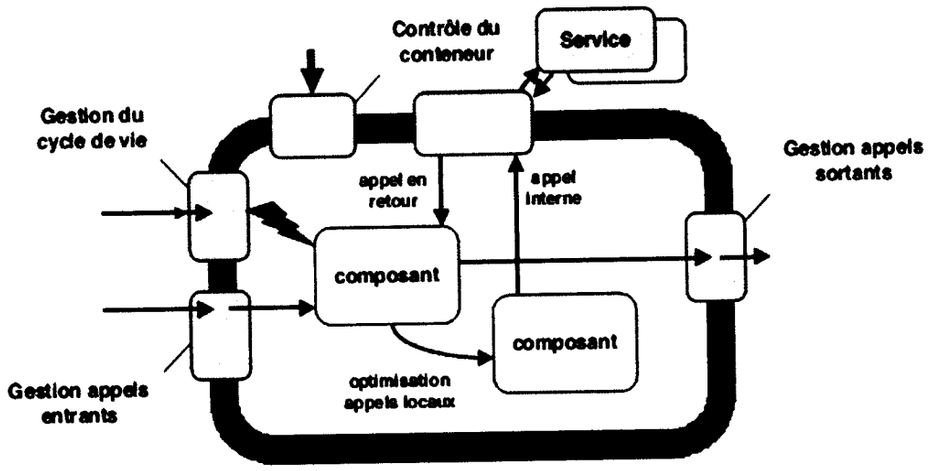


Figure 2.5-Schéma générique de la réalisation de l'inversion du contrôle [11]

Dans le cadre des composants répartis, plusieurs approches ont été proposées pour réaliser cette inversion du contrôle : les approches à conteneurs, les approches hiérarchiques ainsi que des approches plus expérimentales comme les modèles réflexifs et/ou génératifs, notamment certains travaux issus de la programmation par aspects. Il est intéressant de noter que ces différentes approches ne sont pas incompatibles entre elles ; elles correspondent plutôt à des points de vue différents, et parfois conciliables, sur la réalisation du principe d'inversion du contrôle.

2.3.1 Modèles à conteneurs

Dans les plateformes industrielles EJB et CCM, qui ciblent essentiellement les systèmes d'informations d'entreprises et les architectures trois tiers, les composants logiciels sont avant tout considérés comme des entités métier. La notion complémentaire de composants systèmes commence également à émerger. Mais dans tous les cas de figure, les composants sont introduits comme entités spécialisées, qui doivent interagir avec d'autres concepts indépendants. En particulier, l'articulation entre code métier (implémenté par les composants) et code système (implémentés par les services du tiers système) est placé sous l'autorité d'entités indépendantes que l'on nomme *conteneurs.*

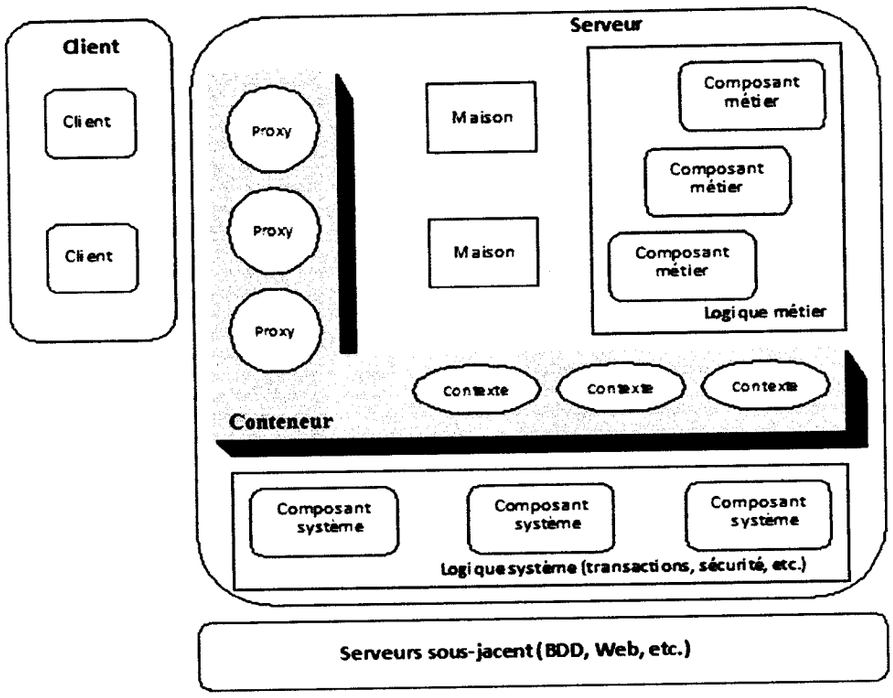


Figure 2.6- Architectures à conteneurs [1]

Comme le montre la figure 2.6, les conteneurs se situent dans les architectures trois tiers à la croisée des trois logiques client, métier et système. Leur rôle primordial est de contrôler les composants métiers, en fonction à la fois des requêtes client et des ressources système sous-jacentes. Pour cela, la mise en œuvre des conteneurs repose sur deux principes fondamentaux :

- mécanismes d'interception ;
- mécanismes de gestion contextuelle.

Les interactions entre les clients et le serveur reposant sur des modèles de communication variés sont conceptualisées par la notion de *proxy*. En plus de gérer la communication, les *proxys* sont également employés pour distinguer les requêtes de nature métier (par exemple, achat d'un article en ligne) des requêtes système (ouverture d'une transaction), on parle dans ce cas de rôle d'*interception.* Une partie de la gestion de l'identité et du cycle de vie des composants métiers est parfois déléguée à des entités spécifiques nommées *maisons (home).* Le conteneur est également responsable des interactions entre les composants métiers et la logique système sous-jacent. Pour les EJB, un unique type de conteneur, dit conteneur EJB, englobe l'ensemble des fonctionnalités système supportées par la plateforme. Dans les approches les plus récentes, et notamment CCM, des composants de service sont de plus en plus introduits en tant qu'intermédiaire supplémentaires dans la couche système. Les *contextes,* pour leur part, permettent de conserver au sein du conteneur des informations reflétant, en termes de contrôle, l'état des composants métier. Une partie de cet état contextuel

est visible par le code métier, le reste étant encapsulé de façon opaque dans la logique système.

2.3.2 Modèles hiérarchiques

Dans la métaphore componentielle, la notion de *composition hiérarchique* éveille de façon grandissante l'intérêt des architectes logiciels. La possibilité de créer des composants de haut niveau dits composites par composition de composants de niveau d'abstraction d'inférieure représente un principe de construction logiciel à la fois naturel et expressif. Si cette notion apparaît lentement mais sûrement dans les approches industrielles, les modèles hiérarchiques issus du monde de la recherche arrivent à maturité. Parmi les nombreuses approches proposées, nous pouvons distinguer les spécifications Fractal qui ont l'avantage d'être à la fois portables et extensibles (voir chapitre 3). Ces spécifications sont développées au sein du consortium ObjectWeb. Dans ce modèle sont distingués trois catégories complémentaires de composants :

- *composants basiques :* ce sont des composants minimaux avec lesquels ont communique uniquement par appels de méthodes sur leur interface. Cette couche est avant tout spécifiée pour permettre l'interopérabilité avec les environnements objets traditionnels ;
- *composants primaire :* ils ajoutent aux composants basiques un *contrôleur* chargé d'un certain nombre de fonctionnalités liées au contrôle du composant ;
- *composants composites :* ces composants ajoutent aux primaires un *contenu*, sous la forme d'une interconnexion de sous-composants, eux même basiques, primaires ou composites.

La grande nouveauté du modèle concerne donc la troisième catégorie qui donne corps à la notion de composition hiérarchique. Un composant composite Fractal possède la structure représentée sur la figure 2.7.

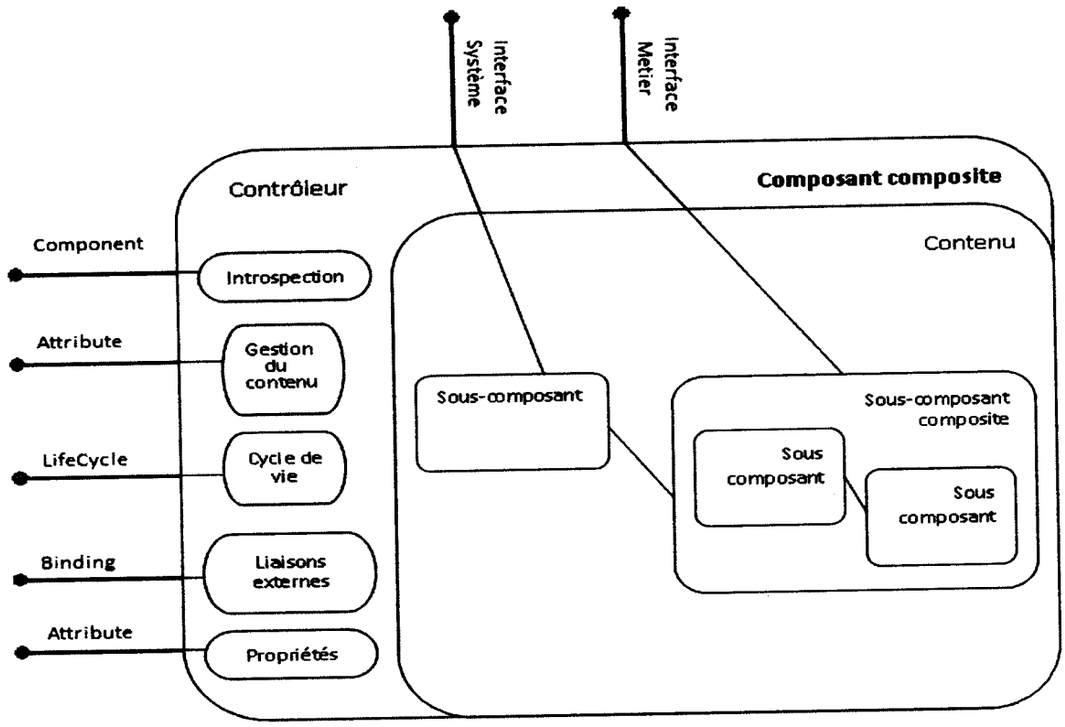


Figure 2.7-Modèle de composant hiérarchique Fractal [1]

Un composant composite est subdivisé en deux parties distinctes : la partie *contrôleur* et la partie *contenu*. Un contrôleur Fractal ressemble à une généralisation du principe de conteneur des architectures EJB et CCM, réalisant tout comme ce dernier une inversion du contrôle, propice à la séparation des préoccupations. Les interfaces standard pour les contrôleurs sont cependant de plus bas niveau dans le cadre de Fractal

2.3.3 Modèles réflexifs et génératifs

L'inversion du contrôle, c'est-à-dire l'externalisation du contrôle sur les composants applicatifs, est répétons-le encore, au cœur des différentes plateformes intergicielles à base de composants répartis. Dans les plateformes « traditionnelles », les interfaces de contrôle sont fournies par des entités (conteneurs ou autres contrôleurs) dont l'implémentation est généralement assez opaque ; elle n'est pas (ou peu) accessible au développeur applicatif. Dans les modèles réflexifs te génératifs, en revanche, les implémentations des principes d'inversion du contrôle sont ouvertes. Les modèles réflexifs sont basés sur une séparation entre un niveau base, fonctionnel, et un niveau méta, pour la gestion de propriétés non fonctionnels. Les liens entre ces deux couches reposent sur des principes d'introspection qui permettent au niveau méta de « découvrir » les fonctionnalités disponibles au niveau base. En complément, on introduit le principe d'*intercession* qui permet la prise de contrôle par le niveau méta à certains moments clé de l'exécution des applications réflexives. Dans le cadre des plateformes componentielles, l'idée est généralement de fournir les principes de composition- éventuellement hiérarchique- à la fois au niveau base et au niveau méta (figure 2.8). La

nature des composants méta est variable d'une implémentation à l'autre. Par exemple dans la plateforme OpenORB développée à l'université de Lancaster en Grande-Bretagne, on trouve des méta-composants essentiellement dédiés aux applications multimédia (streaming, qualité de service, etc.) [Gord01].

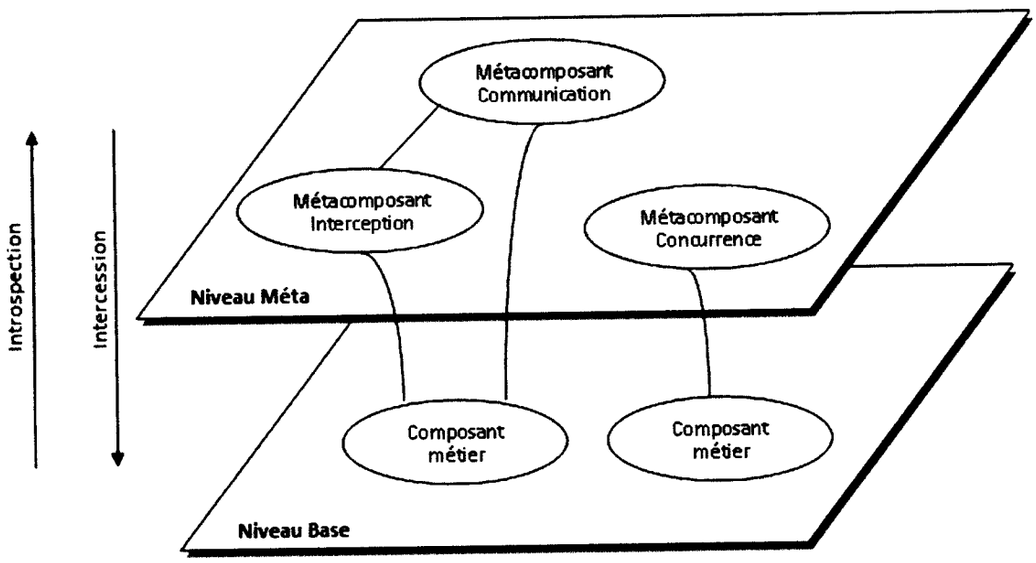


Figure 2.8-Exemple de modèle réflexif [1]

Les applications génératives proposent, tout come les approches réflexives de réifier certains aspects de l'implémentation de fonctionnalités de contrôle pour les rendre accessibles au niveau applicatif. En revanche, les technologies sous-jacentes reposent essentiellement sur des principes de génération de code et/ou de manipulation au vol de code-octet (bytecode). Le domaine très en vogue de la *programmation par aspect* repose principalement sur de telles techniques génératives. Parmi les aspects de contrôle les plus fréquemment réifiés, citions entre autre l'interception des invocations de méthodes permettant d'encadrer ces invocations par des prés et/ou post-traitement. Un intérêt supplémentaire génératif est de permettre l'introduction des principes d'intercession en tant que primitives de langages de programmations (points de coupure, conseil, etc.). Récemment, certains principes de la programmation par aspect ont été introduits pour « ouvrir » l'implémentation de conteneurs EJB industriels, notamment dans le conteneur JBoss [JBOS] et Spring [Sprin].

2.4 Langages de description d'architecture

Les développements récents au sein des standards et technologies des systèmes à bases de composants ont favorisé des approches de développement de systèmes logiciels à base de composants (component-based systems). Ces approches permettent de mettre en place une

véritable industrie de composants logiciels réutilisables afin de métriser le coût du logiciel, de réduire son temps de développement et de faciliter la réutilisation des composants produits. En outre, ces approches se basent toutes sur une définition précise de la notion d'architecture logicielle qui décrit un système en termes de composants et de relations entre ces composants. L'architecture d'un logiciel constitue donc son épine dorsale. De ce fait, la maîtrise du logiciel est subordonnée à celle de son architecture. Aussi, lorsqu'on s'intéresse à l'architecture d'un logiciel, on peut raisonner en termes de composants. Dans ce cadre, résoudre un problème en le découpant en composants puis, en combinant ces composants, permet de concevoir plus facilement les applications et aussi d'améliorer leur niveau de réutilisabilité. Une approche à base de composants semble donc indispensable lors des phases du cycle de vie du logiciel. Ainsi, le monde académique s'est intéressé à la formalisation de la notion d'architecture logicielle à base de composants en proposant des langages de description d'architecture tels que Acme, SOFA ou Fractal qui favorisent la conceptualisation et la formalisation des principales caractéristiques des composants académiques.

Par ailleurs, le modèle architectural d'un système fournit un modèle du dit système à un haut niveau d'abstraction en termes de composants qui assurent les fonctions de calcul, et de connecteurs qui relient les composants et coordonnent leurs interactions pour satisfaire des contraintes globales d'intégrité (invariants structurel, coordination, etc.) et des contraintes de qualité (fiabilité, sécurité, évolutivité, etc.).

Un modèle architectural représente donc un niveau d'abstraction élevé pour les différents utilisateurs d'un système et leur permet de raisonner sur ses propriétés (fonctionnelles ou non fonctionnelles) et d'établir des extensions et des décisions d'évolution. La communauté d'architecture logicielle a accompli depuis une dizaine d'année des progrès considérables portant sur la spécification, la conception et le développement de langages de description d'architecture.

L'architecture logicielle d'un système définit donc sa structure à un haut niveau d'abstraction, exposant son organisation brute comme une collection de composants communicant les uns avec les autres. Une architecture bien définie permet à un concepteur de raisonner sur les propriétés de son système à un niveau d'abstraction élevé . Parmi ces propriétés, nous pouvons citer la compatibilité entre les composants [AlGa97], la conformité aux standards , la performance et la fiabilité [12].

Cette partie est organisée comme suit : dans la section 2.2.4.1, on essaie de comprendre la notion d'architecture logicielle en se basant sur la norme IEEE Std 1471-2000. La section 2.2.4.2 décrit les langages de description d' architecture en expliquant leur définition, leurs concepts de base, leurs mécanismes opérationnels et leur cycle de vie.

2.4.1 Notion d'architecture logicielle (IEEE STD 1471-2000)

Même s'il est souvent employé, le terme architecture logicielle (en anglais, *software architecture*) n'a pas de définition bien établie. Néanmoins dans le domaine du génie logiciel, on ne trouve pas de définition contradictoire. On a choisi dans cette partie de présenter un

ensemble de définitions, visant les descriptions architecturales de systèmes logiciels, conforme à l'IEEE Std 1471-2000 : il s'agit de recommandations pratiques de l'IEEE pour les descriptions architecturales de systèmes logiciels. Elles cherchent à instaurer un référentiel commun au sien duquel seront codifiés les éléments commun à différentes initiatives de description architecturale, et à influencer les infrastructures de descriptions architecturales, admises dans les pratiques de description architecturales de systèmes logiciels et fournissent un cadre référentiel dans lequel les futurs développements en technologie d'architectures logiciels peuvent être déployés. La figure 2.9 montre le modèle conceptuel de description architecturale tel qu'il est défini dans l'IEEE Std 1471-2000 in [12].

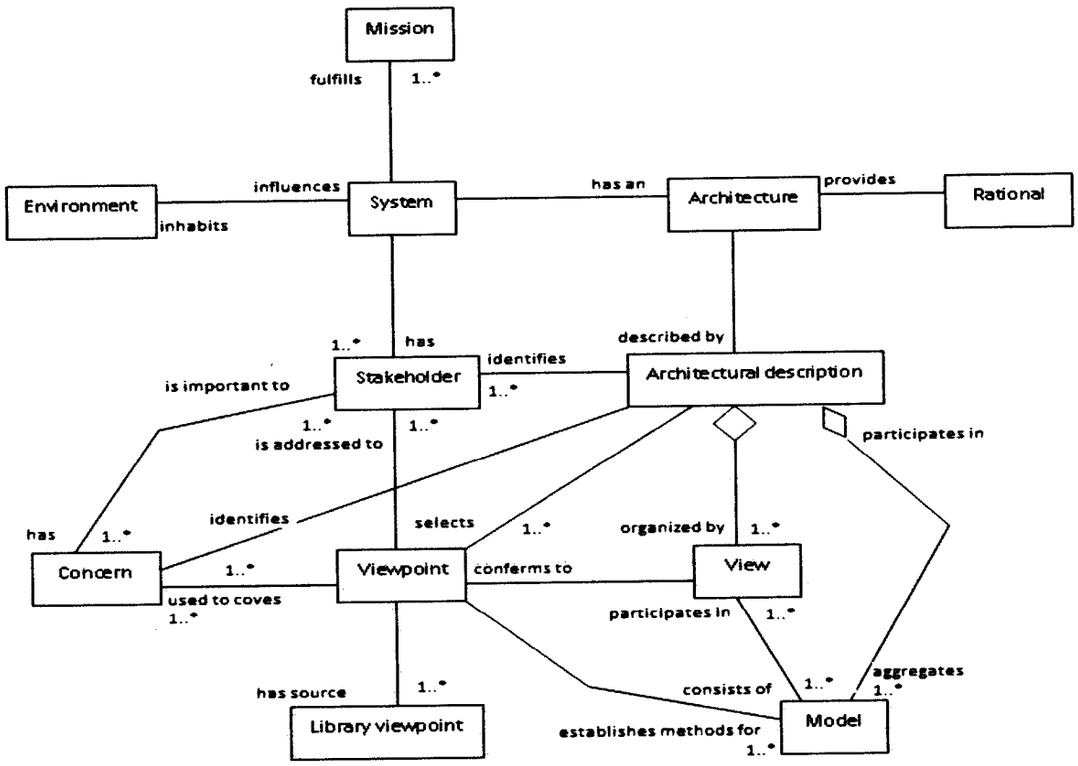


Figure 2.9-Modèle de description architecturale de l'IEEE Std 1471-2000 [12]

Dans ces recommandations un système est défini comme une collection de composants organisés afin d'accomplir une fonction ou un ensemble spécifique de fonctions. Le terme système englobe des applications individuelles, des systèmes dans le sens traditionnel, des sous-systèmes, des systèmes de systèmes, des lignes de produits, etc. A partir de cette définition, il s'ensuit que n'importe quoi peut être un système pourvu qu'il satisfasse certains buts (à savoir accomplir une ou plusieurs fonctions) et on choisit de le voir comme tel.

Un système est hébergé dans un environnement, tandis que l'environnement d'un système peut l'influencer. L'environnement, parfois appelé contexte, détermine les conditions et les circonstances de développement opérationnel, politique et autres influences sur le dit système.

L'environnement peut inclure d'autres systèmes qui interagissent avec le système cible, soit directement par l'intermédiaire d'interfaces soit indirectement via d'autres voies.

Un système peut intéresser une ou plusieurs personnes que nous appelons intervenants (utilisateur final, développeur, concepteur, etc.). Un intervenant (en anglais, *stakeholder*) peut trouver un ou plusieurs centre d'intérêt pour le système. Ces centres d'intérêt concernent aussi bien le développement du système, sa conception, son utilisation ou tout autre aspect important comme les fonctionnalités, la performance, la sécurité, la fiabilité, la sureté, etc.

Un système existe pour accomplir une ou plusieurs « missions » dans son environnement. Il doit répondre à un ou plusieurs objectifs d'un ou plusieurs intervenants dans le développement de ce système.

Un système a une architecture qui peut être représentée par une « description architecturale ». A noter la distinction entre l'architecture d'un système, qui est de niveau conceptuel de la description de cette architecture, qui est de niveau concret.

La description architecturale est définie comme une collection de produits pour documenter une architecture. Elle peut être décomposée en une ou plusieurs « vues ». Chaque vue couvre un ou plusieurs centres d'intérêt des intervenants. Une vue est définie comme une représentation d'un système entier dans la perspective de satisfaire un ensemble relatif d'intérêts des intervenants. Elle est crée selon des règles et des conventions définies dans un point de vue. Un point de vue est défini comme une spécification des conventions pour construire et utiliser une vue.

En plus des informations décrites dans les vues, une description architecturale peut contenir d'autres informations telles que la vue d'ensemble d'un système et/ou son support de raisonnement. Cette information n'est par forcément décrite selon une définition de point de vue, mais peut être trouvée dans des recommandations de documentation organisationnelle.

Une description architecturale sélectionne un ou plusieurs points de vue à être utilisés. Ce choix dépend bien entendu des centres d'intérêt des intervenants. L'ISO *Reference Model of Open Distributed Processing* (RM-ODP) a choisi cinq points de vue, mais l'IEEE Std 1471-2000 ne prescrit aucun point de vue particulier. Un point de vue peut être défini avec la description architecturale, mais il peut aussi être défini ailleurs et être seulement utilisé dans la description architecturale. De tels points de vue définis à l'extérieur sont appelés des points de vue de bibliothèque.

Enfin, une vue peut se composer d'un ou plusieurs modèles et un modèle peut participer à une ou plusieurs vues. Chaque modèle est défini selon les méthodes établies dans la définition correspondante du point de vue. La description architecturale agrège les modèles qui sont organisés en vues.

2.4.2 Langages de description d'architectures

Les langages de description d'architectures sont des langages formels qui peuvent être utilisés pour représenter l'architecture d'un système logiciel. Comme l'architecture devient un thème important dans les systèmes de développement et d'acquisition, des méthodes pour spécifier de façon non ambiguë une architecture deviennent indispensables. Par architecture, nous entendons les composants que contient un système, les spécifications comportementales de ces composants, les modèles et les mécanismes de leurs interactions (connecteurs), et enfin un modèle définissant la topologie d'un système (configuration). Notons qu'un système simple se compose habituellement de plus d'un type de composants : modules, taches, fonctions, etc. Une architecture peut choisir le type du composant le plus approprié, ou peut inclure différentes vues du même système.

Les architectures ont été souvent représentés par des schémas informels de types « boîtes-et-lignes » dans lesquels la nature des composants, leurs propriétés, la sémantique de connexion, et le comportement du système dans l'ensemble sont mal définis (si toutefois ils le sont). Même si ces notations donnent souvent une image intuitive de la construction d'un système, elles échouent généralement quand il s'agit de répondre aux questions suivantes :

- que sont les composants ? sont-ils des modules qui existent seulement à l'étape de conception, mais sont compilés ensemble avant l'exécution ? Sont-ils des taches ou des processus de différents modules assemblés à la compilation, et formant des unités d'exécution ?
- que font les composants ? Comment se comportent-ils ? Sur quels autres composants comptent-ils ?
- que signifient les connexions ?, Signifient-elles : envois des données à, envois de commandes à, des appels, ou une certaine combinaison de tout cela ? Quels sont les mécanismes utilisés pour réaliser ces connexions ?

C'est pourquoi il est important de trouver des notations appropriées pour décrire ces systèmes. En effet, de bonnes notations permettent de les documenter clairement, de raisonner sur leurs propriétés et d'automatiser leur analyse.

Une approche pour décrire les systèmes à base de composants est d'utiliser les notations de l'approche objet comme le fait **Szyperski** in [12]. Chaque composant peut être représenté par une classe, les interfaces de composants peuvent être représentées par des interfaces de classes, et les interactions entre composants peuvent être définies en termes d'associations. La modélisation par objets de systèmes à base de composants présente plusieurs avantages. En effet, les notations objets sont familières à un grand nombre de développeurs et d'utilisateurs de logiciels. Elles fournissent un lien direct entre la conception et l'implémentation de systèmes. Elles sont supportées par les outils commerciaux. Elles disposent de méthodes bien définies pour développer des systèmes à partir des besoins. Toutefois, ces notations ont montré des limites importantes en termes de granularité et dans le passage à l'échelle (section2.2.1).

Une autre alternative, qui répond à ces problèmes, est d'utiliser les langages de description d'architectures (ADL).

2.4.2.1 Les concepts de base des ADL

D'après Smeda et *.al* [12], les ADL trouvent leurs racines dans les langages d'interconnexion de modules (*Module Interconnection languages*) des années 1970. Ils sont aujourd'hui dans une phase de maturité. Parmi les représentants actuels, nous citons : Darwin, C2, Unicon, Wright, Acme, Fractal, SOFA, OpenCom et ArchJava.

Nous pouvons définir un ADL comme suit : un ADL est un langage qui fournit des caractéristiques pour modéliser une architecture conceptuelle d'un système logiciel. Il fournit aussi bien une syntaxe concrète qu'un cadre conceptuel pour caractériser des architectures. Le cadre conceptuel reflète les caractéristiques du domaine pour lequel l'ADL est prévu. Il englobe aussi leur théorie sémantique (par exemple les réseaux de Pétri , les machines à états).

Les concepts de base d'une description architecturale sur lesquels s'accordent l'ensemble des travaux portant sur les ADLS consultés par [12] reposent sur les *composants*, les *connecteurs* et les *configurations architecturales*. Un ADL doit fournir les notations et les sémantiques nécessaires pour leur spécification explicite. Ceci nous permet de déterminer si une notation particulière est un ADL ou pas. Par ailleurs, afin d'inférer la moindre information sur une architecture, il faut au minimum disposer des interfaces de composants. Sans cette information, une description architecturale n'est qu'une collection d'identificateurs interconnectés.

Plusieurs aspects du paradigme composant sont utiles, mais non nécessaires. Leurs avantages sont reconnus et admis par certains concepteurs d'ADL, mais leurs absence ne nuit pas au fait qu'un langage donné ait un statut d'ADL. Parmi ces aspects, nous citons les contraintes, les propriétés non fonctionnelles et les styles architecturaux. D'autres aspects pas toujours présents dans la plupart des ADL comme l'héritage, la généricité, le raffinement, la traçabilité peuvent également enrichir le pouvoir expressif et opératoire des ADL. La figure 2.10 présente un méta-modèle montrant les concepts de base des langages de base des langages de description d'architecture ainsi que les relations qu'ils entretiennent ensemble.

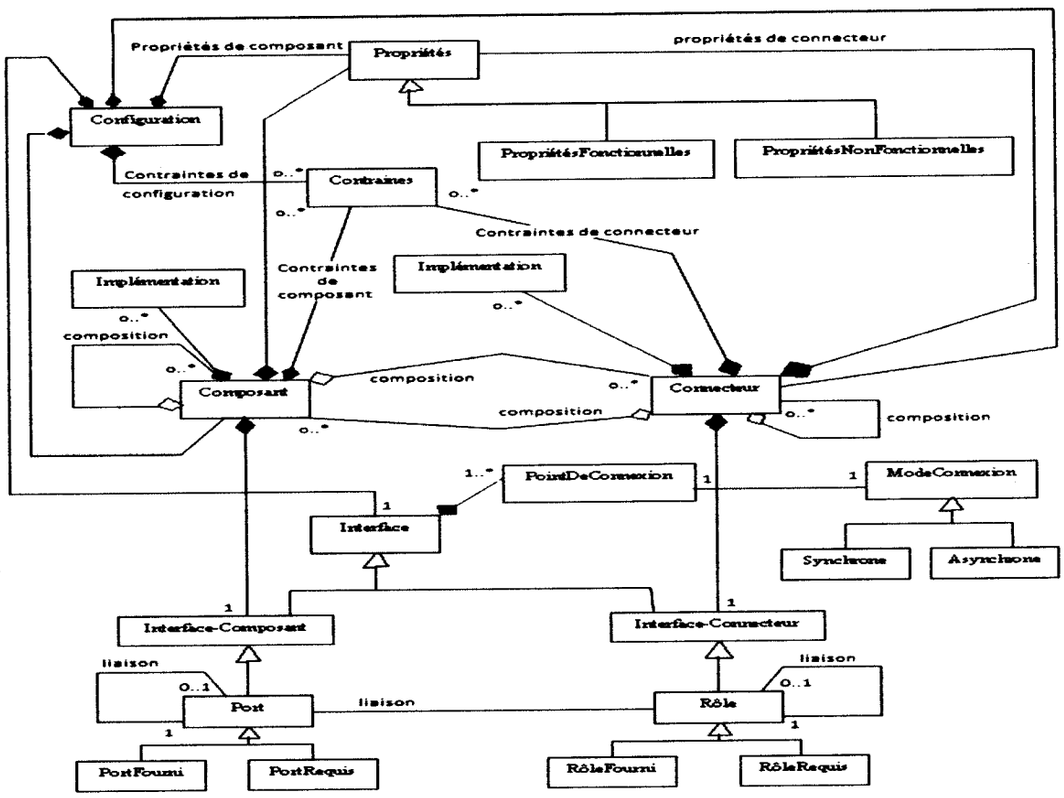


Figure 2.10-Les concepts de base des langages de description d'architecture [12]

Les composants

Un composant est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Un composant dans une architecture peut être aussi petit qu'une procédure simple ou aussi grand qu'une application entière. Il peut exiger ses propres données et/ou espace d'exécution comme il peut les partager avec d'autres composants. Tous les ADL existants modélisent les composants sous une forme ou une autre et sous divers noms. Par exemple Fractal, Wright, Acme, ArchJava les désignent tout simplement comme *components* alors SOFA les qualifie de *Frames*. Dans cette section, nous présentons les aspects des composants que nous estimons essentiels et devant être définis dans tout ADL : les interfaces, les types, les contraintes et les propriétés non fonctionnelles.

- Les interfaces : L'interface d'un composant est un ensemble de points d'interaction entre le composant et le monde extérieur. Comme dans les classes de l'approche objet ou les spécifications de paquetage d'Ada, une interface d'un composant dans un ADL indique les services (messages, opérations et variables) que le composant fournit. Les ADL doivent également fournir des moyens pour exprimer les besoins d'un composant, c'est-à-dire les services requis à partir d'autres composants dans l'architecture. Une interface définit ainsi les engagements qu'in composant peut tenir et les contraintes liées à son utilisation. Elles permettent également un certain degré de

raisonnement, certes limité, portant sur la sémantique d'un composant. Dans la plupart des ADL, les interfaces d'un composant portent le nom de ports. Deux types de ports peuvent être distingués : les ports services (fournis), qui exportent les services des composants, et les ports besoins (requis), qui importent les services vers les composants.

- Les types : Les types de composants sont des abstractions qui encapsulent la fonctionnalité dans des blocs réutilisables. Un type de composant peut-être instancié plusieurs fois dans une même architecture ou peut être réutilisé dans d'autres architectures. Les types de composants sont des abstractions qui encapsulent la fonctionnalité dans des blocs réutilisables. La modélisation explicite des types facilite également la compréhension et l'analyse d'une architecture, étant donné que les propriétés d'un type sont partagées pat toutes ses instances.
- Les types : Les sont considérées comme un type spécial de propriété. Elles doivent être spécifiées afin de respecter les utilisations prévues d'un composant et d'établir les dépendances parmi ses éléments internes. Les contraintes peuvent être définies soit dans un langage de contrainte séparé, soit peuvent être spécifiées directement en utilisant les notations de l'ADL hôte.
- Les propriétés non fonctionnelles : Les spécifications des propriétés non fonctionnelles des composants sont nécessaires pour permettre : la simulation de leur comportement, leur analyse, leur traçabilité depuis leur conception jusqu'à leur implémentation, l'aide dans la gestion de projet (par exemple, en définissant des conditions de performance rigoureuses, le développement d'un composant peut devoir être assigné au meilleur ingénieur). Peu d''ADL offrent une structure d'accueil des propriétés non fonctionnelles malgré tous leurs avantages potentiels.

Les connecteurs

Les connecteurs sont des blocs de construction architecturaux utilisés pour modéliser les interactions entre les composants et les règles qui régissent ces interactions. Ils correspondent aux lignes dans les descriptions de type « boîtes-et-lignes ». Ils sont des entités architecturales qui lient des composants ensemble agissent en tant que médiateur entre eux. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure, et l'émission d'événements. Les connecteurs peuvent également représenter des interactions complexes, comme un protocole de client-serveur ou lien de SQL entre une base de données et une application [12]. Contrairement aux composants, les connecteurs peuvent ne pas correspondre à des unités de compilation [11]. Cependant, les spécifications de connecteurs dans un ADL peuvent également contenir des règles pour implémenter un type spécifique d'un conteneur.

De manière générale, les connecteurs dans les langages de description d'architectures peuvent être classés en trois groupes : les connecteurs implicites, comme ceux par exemple de Darwin [12] ; les ensembles énumérés de connecteurs prédéfinis, comme ceux par exemple d'Unicon

et de SOFA [12]; les connecteurs dont les sémantiques sont définis par les utilisateurs, comme ceux par exemple de Wright [12].

Premier groupe : dans les ADL comme Darwin ou Fractal, les connexions entre composants sont spécifiées en termes de liaison directe (*binding*) entre les ports « requis » et les ports « fournis ». La sémantique des ces connecteurs est définie dans l'environnement sous-jacent et la communication entre les composants prend en compte cet environnement. D'autres langages décrivent les connecteurs implicitement comme Rapide [12] et MetaH [12]. Les *bindings* de Darwin, Rapide et les connexions de MetaH sont modélisées en ligne et ne peuvent en aucun cas être renommés ou réutilisés.

Deuxième groupe : Un connecteur dans Unicon est spécifié par un protocole. Un protocole est défini par un type de connecteurs (Unicon possède sept types de connecteurs prédéfinis : FileIO, Pipe, Procedure Call, RPC, Scheduler, RT, Data Acces et PL Bundler, un ensemble de propriétés, et des rôles typés qui servent comme points d'interaction avec les composants. Les connecteurs d'Unicon ne peuvent pas être instanciés ni évoluer et ne peuvent être composés que de connecteurs uniquement.

Troisième groupe : dans le langage Wright, les connecteurs sont définis comme un ensemble de « rôles » et une « glu ». Chaque rôle définit le comportement d'un participant dans l'interaction. La glu définit comment les rôles interagissent entre eux. Les connecteurs dans Wright sont définis par les utilisateurs et peuvent évoluer via des instanciations de leurs différents paramètres. Ils peuvent être composés d'autres connecteurs.

Les concepts caractérisant les connecteurs sont les *interfaces*, les *types*, les *contraintes* et les *propriétés non fonctionnelles*.

- Les interfaces : Afin de permettre une connexion correcte entre composants et d'assurer leur communication dans une architecture, un connecteur doit exporter les services qu'il prévoit. Par conséquent, l'interface d'un connecteur est un ensemble de points d'interaction entre lui-même et les composants qui lui sont attachés. Les interfaces d'un connecteur portent le nom de rôles. Deux types de rôles existent : le rôle fourni et le rôle requis.
- Les types : La communication au niveau architecture peut faire appel à des protocoles complexes. Pour abstraire ces protocoles et les rendre réutilisables, les ADL doivent modéliser les connecteurs comme des types. Ceci peut être fait de deux manières : soit en tant que types extensibles définis en termes de protocoles de communication et qui sont indépendants de l'implémentation, soit en tant que types intégrés et énumérés basés sur leurs mécanismes d'implémentation. Seuls les ADL qui modélisent les connecteurs comme des entités de première classe distinguent les types de connecteurs de leurs instances.
- Les contraintes : Afin d'assurer les protocoles d'interaction prévus, d'établir les dépendances intraconnecteur, et de fixer les conditions d'utilisation des connecteurs,

des connecteurs doivent être spécifiées. Un exemple d'une contrainte simple est la restriction du nombre de composants qui interagissent à travers un connecteur donné.

- Les propriétés non fonctionnelles : Les propriétés non fonctionnelles d'un connecteur ne sont pas forcément obtenues des spécifications de sa sémantique. Elles représentent en général les informations additionnelles nécessaires pour l'implémentation correcte d'un connecteur. Modéliser les propriétés non fonctionnelles d'un connecteur permet : la simulation de leur comportement, de leur analyse, et la sélection de connecteurs appropriés et leurs correspondances [12].

Les configurations

Une configuration représente un graphe de composants et de connecteurs et définit la façon dont ils sont reliés entre eux. Cette notion est importante pour déterminer si les composants sont bien reliés, leurs interfaces s'accordent, les connecteurs correspondants permettent une communication correcte, et la combinaison de leurs sémantiques aboutit au comportement désiré. En appui des modèles de composants et de connecteurs, les descriptions des configurations permettent l'évaluation des aspects distribués et concurrents d'une architecture, par exemple, la possibilité de déterminer des verrous, de connaître les potentiels de performance, de fiabilité, de sécurité, et ainsi de suite.

Les descriptions des configurations permettent également d'analyser les architectures. Par exemple déterminer si une architecture est « trop profonde », ce qui peut affecter la performance due au trafic de messages à travers plusieurs niveaux, ou « trop large », ce qui peut conduire à trop de dépendances entre composants.

Le rôle clé des configurations est de faciliter la communication entre les différents intervenants dans le développement d'un système. Leur but est d'abstraire les détails des différents composants et connecteurs. Ainsi, elles décrivent le système à un haut niveau d'abstraction qui peut être potentiellement compris par des personnes avec différents niveaux d'expertise te de connaissance techniques. Nous expliquons ci-dessous les concepts qui caractérisent les configurations d'architecture.

- Les interfaces : pour permettre des interactions entre les configurations et permettre à des configurations de communiquer avec leurs composants, les interfaces de configurations doivent être fournies. Celles-ci sont principalement utilisées dans les couplages de sous-systèmes.
- Les types : Pour permettre la construction de différentes architectures d'un même système, les configurations doivent aussi être définies comme des classes instanciables. Ainsi on peut déployer une architecture donnée de plusieurs manières, sans récrire le programme de configuration/déploiement.
- Les contraintes : Les contraintes qui spécifient les dépendances entre les composants et les connecteurs dans une configuration sont aussi importantes que celles spécifiées au niveau des composants et des connecteurs. Cependant, la plus part des ADL se sont

focalisés plus sur des contraintes locales que celles définies au niveau des configurations. Par exemple, des contraintes de validité d'une configuration peuvent être définies comme un ensemble de contraintes entre les composants et les connecteurs. A leur tour, ces derniers sont exprimés par leurs interfaces et leurs protocoles. La performance décrit par une configuration dépendra de l'exécution de chaque élément architectural individuel, et la sécurité d'une architecture est toujours fonction de la sécurité de ses constituants.

- Les propriétés non fonctionnelles : Certaines propriétés non fonctionnelles ne sont reliées ni aux composants ni aux connecteurs et doivent être spécifiées au niveau des configurations. Les propriétés non fonctionnelles exprimées au niveau des configurations sont nécessaires pour choisir les composants et les connecteurs appropriés, pour réaliser des analyses, pour appliquer et faire respecter des contraintes de topologie, pour lier les composants architecturaux aux processeurs, et pour aider dans la gestion de projet.

Les styles architecturaux

Un autre aspect intéressant des ADL, même s'il n'est présente que dans certains d'entre eux, concerne les styles architecturaux. Selon Garlan, D. in [12], un choix d'un style architectural permet de :

- déterminer l'ensemble du vocabulaire désignant le type des entités de base (composants et connecteurs) telles que pipe, filtre, client, serveur, événement, processus, etc. ;
- spécifier l'ensemble des règles de configuration qui déterminent les compositions et les assemblages permis entre ces éléments. Par exemple pipes-filtres, client/serveur suivant la relation d'association n-1 ;
- donner la sémantique des configurations. Par exemple, un système multi-agents communiquant via le tableau noir signifie que c'est un partage de connaissances entre ces agents ;
- déterminer les analyses qui peuvent être réalisées sur un système construit selon un tel style. Par exemple, la vérification de la causalité dans un système distribué basé sur la communication par événement.

2.4.2.2 Les mécanismes opérationnels des ADL

Les architectures sont prévues pour décrire des systèmes logiciels à grande échelle qui peuvent évoluer dans le temps. Les modifications dans une architecture peuvent être planifiées ou non planifiées. Elles peuvent également se produire avant ou pendant la phase d'exécution. Les ADL doivent supporter de tels changements grâce à des mécanismes opérationnels. En plus, les architectures sont prévues pour fournir aux développeurs des abstractions dont ils ont besoin pour faire face à la complexité et à la taille des logiciels. C'est pourquoi les ADL doivent fournir des outils de spécification et de développement pour prendre en compte des systèmes à grande échelle susceptibles d'évoluer. Aussi, pour

améliorer l'évolution et le passage à l'échelle et pour augmenter la réutilisabilité et la compréhensibilité des architectures, des mécanismes spéciaux doivent être pris en compte par les ADL. Dans les paragraphes suivants, nous présentons ces différents mécanismes.

L'instanciation

Avec la définition susmentionnée, les composants et les connecteurs peuvent être instanciés plusieurs fois dans une architecture et chaque instance peut correspondre à une implémentation différente. L'instance d'un composant/connecteur/configuration est un objet particulier, qui est créé avec le respect du plan donné par son type de composant/connecteur/configuration. Toutes les instances d'un type de composant/connecteur/configuration doivent inclure la structure définie par ce type et se comportement de la même manière que ce dernier. Par exemple, si un ensemble de propriétés est défini pour un modèle particulier, chaque instance de ce modèle doit avoir les mêmes propriétés.

L'héritage et le sous-typage

L'héritage et le sous-typage sont deux manières différentes de réutiliser des modèles (de composant, de connecteurs ou de configurations). Alors que l'héritage permet la réutilisation du modèle lui-même ; le sous-typage permet la réutilisation des constituants d'un modèle.

- L'héritage : L'héritage est une représentation de la hiérarchie des abstractions, dans laquelle un sous-modèle hérite de un ou plusieurs modèles. Typiquement, le sous-modèle augmente ou redéfinit la structure et le comportement existants de son modèle. C'est un outil puissant pour l'évolution en permettant à un modèle hérité d'être modifié en ajoutant, en supprimant, ou en changeant sa structure. Une relation d'héritage voisine de celle disponible dans les langages à objets serait très appréciée. Pour permettre la prise en compte de systèmes sophistiqués et complexes, l'héritage dans les systèmes à base de composants est sélectif et dynamique. L'aspect sélectif permet de choisir les éléments à hériter et de les combiner (par exemple hériter l'interface d'un composant mais pas son implémentation, fusionner deux interfaces, etc.). L'aspect dynamique permet à des sous-modèles de modifier ce qu'ils ont hérité. L'héritage dans les ADL peut également être multiple, où un sous-modèle hérite de plusieurs modèles.
- Le sous-typage : Le sous-typage peut être défini par la règle suivante : un type x est le sous-type d'un type y si les valeurs du type x peuvent être utilisées dans n'importe quel contexte où le type y est prévu sans présenter d'erreurs.
  Selon Medvidovic *et al.*in [12], on distingue trois relations de sous-typage dans les architectures logicielles : le sous-typage d'interface, le sous-typage de comportement et le sous-typage d'implémentation. Le sous-typage d'interface exige que, pour qu'un composant C1 soit un sous-type d'interface d'un autre composant C2, il faut que C2 spécifie au moins les interfaces fournies et au plus les interfaces besoins de C1. Le sous-typage de comportement exige que chaque opération fournie du super-type ait

une opération fournie correspondante dans le sous-type. Enfin, le sous-typage d'implémentation peut être établi avec un contrôle syntaxique si les opérations du sous-type ont des implémentations identiques que les opérations correspondantes du super-type.

La composition

Les architectures sont nécessaires pour décrire des systèmes logiciels à différents niveaux de détails, où les comportements complexes sont soit explicitement représentés soit encapsulés dans des composants et des connecteurs. Un ADL doit pouvoir prendre en compte le fait qu'une architecture entière devienne un composant simple dans une plus grande architecture. Par conséquent, la prise en compte de la composition ou de la composition hiérarchique, est cruciale. Cependant, ce mécanisme exige que les composants aient des interfaces bien définies puisque leurs implémentations sont cachées. Comme nous le verrons dans section 4, plusieurs ADL utilisent ce mécanisme pour définir des configurations, où les systèmes sont définis comme des composants composites qui sont composés de composants et de connecteurs.

La généricité

La généricité se rapporte à la capacité de paramétrer des types. L'instanciation ne fournit pas de service pour paramétrer des types. Souvent, des structures communes dans les descriptions de systèmes complexes sont amenées à être spécifiées à plusieurs reprises. Bien que l'héritage et la composition permettent de réutiliser le code, cela ne permet pas de résoudre tous les besoins en réutilisation. Les type génériques permettent de réutiliser le code source en fournissant au compilateur la manière de substituer le nom des types dans le corps d'une classe. Ceci aide à la conception et à l'utilisation de bibliothèques de composant et de connecteurs. Ces dernières constituent des outils importants pour le développement rapide et efficace du logiciel à base de composants.

Le raffinement et la traçabilité

L'argument le plus souvent évoqué pour créer et utiliser des ADL est qu'ils sont nécessaires pour établir le pont entre les diagrammes informels de haut niveau de types « boîtes-et-lignes » et les langages de programmation qui sont considérés comme des langages de bas niveau. Comme les modèles architecturaux peuvent être définis à différents niveaux d'abstraction ou de raffinement, les ADL fournissent aux développeurs et aux concepteurs des outils expressifs et sémantiquement riches pour les spécifier. Les ADL doivent également permettre une traçabilité des changements à travers ces niveaux de raffinements. Notons que le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les ADL actuels.

2.4.2.3 Le cycle de vie des ADL

La construction d'une application à base de modèle de composants académique nécessite l'utilisation d'un modèle abstrait pour l'architecture logiciel. Un tel modèle est illustré par les langages de description d'architecture tels que Fractal, Acme, Fractal. En raison de la particularité des composants académiques, leur cycle de vie est différent de celui des composants industriels. Dans les composants industriels, on se concentre plus sur les étapes d'implémentation et de déploiement des composants, alors que dans les composants académiques, on se focalise sur l'analyse et la spécification des composants et leur relation.

De manière générale, le cycle de vie d'un modèle de composant académique commence par la phase d'analyse pour définir les composants et les connecteurs. Les composants représentent des unités de calcul ou des unités de stockage et les connecteurs définissent les interactions entre les composants. Les interfaces des composants et des connecteurs sont également définies dans cette phase. Pour chaque composant et connecteur, des propriétés non fonctionnelles et des contraintes peuvent être indiquées en cas de nécessité.

Dans la phase d'assemblage, les composants sont reliés à l'aide des connecteurs. Dans la phase de déploiement, les composants et les connecteurs sont instanciés et reliés ensemble pour former l'architecture logicielle. Enfin, la phase d'exécution n'est pas un objectif fixé par les modèles de composants académiques, et d'ailleurs n'existe pas pour la plupart d'entre eux. Par exemple, Acme reste un langage de modélisation et d'analyse et ne fournit pas un lien entre le modèle et son implémentation.

2.5 Conclusion

L'essence même des composants est de construire des systèmes bien structurés, qui soient compréhensibles de manière indépendante et qui constituent des blocs de construction réutilisables.

Comme le souligne fort justement Jacobson, I. in [6], la conception et le développement d'un modèle de composants doivent répondre aux critères suivants :

- *l'adéquation :* d'une part, le modèle de composants doit être capable de représenter les concepts de base existant dans la plupart des outils de développement de composants logiciels tels que ActiveX, JavaBeans, etc. Il doit fournir des abstractions communes pour ces concepts qui sont en général facilement reconnaissables par les utilisateurs de composants. D'autre part, on devrait pouvoir lier un tel modèle à des théories formelles existantes sur les composants logiciels, profitant ainsi de leurs preuves en les appliquant et en les réutilisant ;
- *l'expressivité :* comme d'ailleurs tous les systèmes, les systèmes à base de composants peuvent être caractérisés par leur structure et leur comportement, c'est-à-dire le type et l'organisation de leurs composants et des échanges d'informations mais aussi l'évolution de leur état et de leur structure. Les modèles de composants devraient être

capables d'exprimer adéquatement l'ensemble de ces aspects, sans pour autant imposer des contraintes drastiques et non réalistes sur les composants à représenter ;

- *la clarté :* suite aux recommandations précédentes, les modèles de composants doivent reposer sur un nombre minimal de concepts de base en définissant clairement les relations existantes entre eux, ce qui devrait faciliter la compréhension, la communication et l'application de ces modèles ;
- *l'ubiquité :* les modèles de composants doivent fournir des mécanismes pour prendre en compte le développement d'applications distribuées.

Plusieurs pistes de recherche concernant les composants méritent d'être explorés. La piste qui nous semble la plus prometteuse est l'interopérabilité de composants. Elle représente la clé de voûte de la réussite de nouveaux modèles de composants évolutifs et a comme faculté de permettre les interactions entre composants hétérogènes souvent distribués. Deux types d'interopérabilité sont souvent mentionnés dans la littérature. La première concerne l'interopérabilité sémantique qui permet la coopération de composants conçus selon des modèles conceptuels hétérogènes. La seconde a trait à l'interopérabilité technique qui permet à un composant à coopérer techniquement avec un autre composant en dépit de l'environnement différent (par exemple CORBA, EJB, etc.) qui constituent le système.

Le problème aujourd'hui est non seulement de faire communiquer entre eux des systèmes et composants hétérogènes, mais aussi de s'assurer que cette communication reste toujours valide et correcte.

Chapitre 3

Le modèle de composants Fractal

3.1 Introduction

En faisant respecter une stricte séparation entre *interface* et *implémentation* et en définissant explicitement l'architecture logicielle, la programmation orientée composant permet de faciliter la réalisation et la maintenance de systèmes logiciels complexes. En effet, ces deux principes constituent la base de deux propriétés essentielles : l'*adaptabilité* et l'*administrabilité* (c'est-à-dire l'observation cohérente, la sécurité, l'équilibre entre autonomie et interdépendance pour les différents sous-systèmes, etc.). Leurs apports dans la constitution d'unités de déploiement logiciel et dans la configuration en particulier, sont bien clairs : ils rendent possible la configuration statique dans le but de permettre le portage dans des environnements de déploiement arbitraires (construction de canevas destinés à des environnements contraints), l'évolution des besoins métier et technique (maintenance) et l'évolution organisationnelle (intégration, interopérabilité). Considérés comme entités d'exécution, les composants peuvent servir de base pour la reconfiguration. En permettant de spécifier clairement les sous-systèmes, les composants offrent un support naturel pour les actions de reconfiguration et constituent un concept naturel pour aborder l'étude, la réalisation ou l'administration d'un système. Associée aux techniques de la programmation à méta-objets, la programmation orientée composants peut rendre transparentes aux programmeurs nombre de complexités inhérentes au traitement des aspects non fonctionnels dans les systèmes logiciels, tels la répartition, la tolérance aux pannes, comme l'illustre le concept de conteneurs dans des modèles de composants comme Enterprise Java Bean (EJB), CORBA Component Model (CCM) ou Microsoft .NET.

Nombre de canevas pour la programmation orientée composants et de langages de description d'architecture ont été mis au point, néanmoins ces canevas et ces ADL fournissent un support limité pour l'extensibilité et l'adaptabilité.

Cette limitation implique de graves conséquences : elle empêche la possibilité de doter facilement et éventuellement dynamiquement les composants de nouveaux types de contrôles, prenant en charge par exemple les aspects non fonctionnels ; elle empêche les concepteurs et programmeurs d'applications d'effectuer des compromis importants tel que degré de

configurabilité contre performance et consommation de ressources ; elle peut également compliquer le portage du canevas dans différents environnements, par exemple les systèmes embarqués. Même avec des modèles de composants réflexifs comme OpenCOM , c'est-à-dire des modèles dotés explicitement d'un protocole à méta-objets permettant le contrôle de l'exécution des composants et introduisant un support pour la prise en charge des aspects non fonctionnels, nous ressentons la nécessité supplémentaire de pouvoir personnaliser les capacités réflexives des composants afin d'autoriser certains compromis qui sont d'une importance cruciale dans la construction des couches logicielles de bas niveaux comme les systèmes d'exploitation et les intergicielles.

Nous présentons dans ce chapitre un modèle de composants, appelé Fractal, qui s'affranchit des limites évoquées ci-dessus par la définition de composants dotés de capacités de contrôle personnalisables. En d'autres termes, les composants dans Fractal sont réflexifs, dans le sens que leur exécution et leur structure peuvent être réifiées et contrôlées à travers des interfaces bien spécifiées. Ces capacités réflexives ne sont toutefois pas fixées dans le modèle et peuvent être étendues et adaptées pour répondre aux exigences et contraintes du programmeur.

La section 3.2 présente les principes de base de la spécification Fractal. La section 3.3 présente le langage Fractal ADL qui permet de construire des assemblages de composants Fractal. Les plateformes mettant en œuvre la spécification Fractal sont présentées dans la section 3.4. L'accent est mis sur deux d'entre elles, Julia (section 3.4.1) et AOKell (section 3.4.2). Les autres plates-formes existantes sont présentées brièvement en section 3.4.3. La section 3.5 présente quelques bibliothèques de composants disponibles pour le développement d'applications Fractal. La section 3.6 compare Fractal à des modèles de composants existants. Finalement, la section 3.7 conclut ce chapitre.

3.2 Le modèle Fractal

Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (c'est-à-dire observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation.

Le modèle de composants Fractal a été défini par France Telecom R&D et l'INRIA. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET. Fractal est organisé comme un projet du consortium ObjectWeb pour le *middleware open source.* Les premières discussions autour du modèle Fractal, initiées dès 2000 à France Telecom R&D, ont abouti en juin 2002 avec la première version officielle de la spécification et la première version de Julia, qui est l'implémentation de référence de cette spécification. La spécification a évolué pour aboutir en septembre 2003 à une deuxième version comportant un certain nombre de changements au niveau de l'API. Dès le départ, un langage de description d'architecture, Fractal ADL, a été associé au modèle de composants. Basé sur une syntaxe *ad*

hoc au départ, il a évolué et sa version 2, définie en janvier 2004 et implémentée en mars 2004, est basée sur une syntaxe extensible.

La spécification Fractal [13] est disponible sur le site du consortium ObjectWeb.

3.2.1 Aperçu

Les caractéristiques principales du modèle Fractal sont motivées par l'objectif de pouvoir construire, déployer et administrer des systèmes complexes tels que des intergiciels ou des systèmes d'exploitation. Le modèle est ainsi basé sur les principes suivants :

- **composants composites** (c.-à-d. composants qui contiennent des sous-composants) pour permettre d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** (c.-à-d. sous-composants de plusieurs composites englobants) pour permettre de modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- **capacités d'introspection** pour permettre d'observer l'exécution d'un système.
- **capacités de (re)configuration** pour permettre de déployer et de configurer dynamiquement un système.

Par ailleurs, un autre objectif de Fractal est d'être utilisable pour une large variété d'applications, des systèmes embarqués aux systèmes d'informations en passant par les serveurs d'applications. Malheureusement, la prise en charge des fonctionnalités avancées de Fractal a un coût qui n'est pas toujours compatible avec les ressources limitées des environnements contraints.

Afin d'atteindre ces objectifs contradictoires, le modèle de composant Fractal n'est pas défini comme une spécification vaste et fixe que tous les composants doivent respecter, mais plutôt comme un système extensible de relations entre des concepts bien définis et les API correspondantes que les composants Fractal peuvent selon leur choix implémenter, suivant les fonctionnalités qu'ils peuvent ou veulent offrir aux autres composants. Cet ensemble de spécifications peut être organisé comme des « *niveaux de contrôle croissants* », c.-à-d. par ordre croisant de capacités réflexives (*introspection* et *intercession*), et nous le présentons dans les sections suivantes de cette façon.

Au plus bas niveau de contrôle, un composant est une entité d'exécution qui ne fournit aucune possibilité de contrôle aux autres composants. Les composants ainsi construits sont comparables aux objets instanciés dans les langages à objets comme Java. L'intérêt de ces composants réside dans le fait qu'ils permettent d'intégrer facilement des logiciels patrimoniaux.

Au niveau de contrôle suivant, qui peut être appelé niveau d' « introspection », un composant Fractal peut fournir une interface standard, similaire à l'interface `IUnknown` du modèle COM, qui permet de découvrir ses interfaces externes, ou en d'autres mots ses frontières.

Au niveau de contrôle supérieur, qui peut être appelé niveau de « configuration », un composant Fractal peut se doter d'interfaces pour l'introspection et la modification de son contenu. Dans le modèle Fractal, ce contenu est un ensemble d'autres composants, appelés *sous-composants* du composant englobant appelé alors *composant composite*, liés entre eux à travers des *liaisons*. Un composant peut en conséquence choisir de fournir des interfaces pour contrôler l'ensemble de ses sous-composants, l'ensemble des liaisons entre ces sous-composants, et ainsi de suite.

En plus de ces niveaux de contrôle, le modèle Fractal spécifie aussi un canevas pour l'instanciation des composants, ainsi qu'un simple système de types pour les composants Fractal. Mais comme les capacités de contrôle décrits ci-dessus, le canevas d'instanciation et le système de types sont optionnels. En fait, dans le modèle Fractal, *tout* est optionnel. Cela a des avantages et des inconvénients, que nous discuterons plus tard.

Le résultat de cette organisation modulaire et extensible (n'importe qui est libre de définir ses propres interfaces de contrôle, afin de fournir des capacités d'introspection et d'intercession), vu que le modèle Fractal n'est restreint à aucun langage de programmation, est que les composants peuvent être utilisés dans une large variété de situations, des systèmes d'exploitation aux plateformes intergicielles, des interfaces graphiques aux systèmes d'informations, et des configurations très efficaces mais non reconfigurables à des systèmes ou applications entièrement configurables et moins efficaces.

3.2.2 Niveaux de contrôle

3.2.2.1 Absence totale de contrôle

Au plus bas niveau de capacité de contrôle, un composant Fractal ne fourni aucune fonction d'introspection ou d'intercession aux autres composants. Un tel composant, appelé *composant de base*, peut être utilisé d'une seule manière, à savoir en invoquant des opérations sur ses interfaces de composant. Une *interface de composant* est un point d'accès à un composant qui implémente une *interface de langage*. *Interfaces de composant* et *interfaces de langage* ne doivent pas être confondus : une interface de composant est un point d'accès au composant qui *implémente* une interface de langage, une interface de langage est un *type*, dans le chapitre 2 lors de l'étude conceptuelle des composants, nous parlions plutôt de *port* (pour désigner une interface de composant) et d'*interface* (pour désigner une interface de langage). Mais dans la suite du chapitre nous préférerons à l'expression « un composant a une interface de composant qui implémente une interface de langage X » la suivante : « un composant a une interface X », dans le but d'améliorer la lisibilité.

3.2.2.2 Introspection

Au prochain niveau de capacité de contrôle, au-delà du niveau de base où les composants ne fournissent aucune fonction de contrôle, un composant Fractal peut fournir des fonctions d'introspection pour *introspecter* ses caractéristiques externes, c.-à-d. ses frontières. Cette section définit plus précisément les caractéristiques externes de composants Fractal, et spécifie les interfaces relatives à l'introspection de ces caractéristiques. Les interfaces

relatives à l'introspection (et la configuration) des caractéristiques internes des composants Fractal sont spécifiées dans la section suivante.

3.2.2.2.1 Caractéristiques externes d'un composant

Suivant son niveau d'observation, ou *échelle*, un composant Fractal peut être vu comme une boîte noire ou une boîte blanche. Quand il est vu comme une boîte noire, c.-à-d. quand son organisation interne n'est pas visible, les seuls détails visibles d'un composant Fractal sont les *points d'accès* à cette boîte noire, appelés *interfaces externes* (voir Figure 3.1). Chaque interface a un nom, qui permet de la distinguer des autres interfaces du composant (un composant peut avoir plusieurs interfaces implémentant la même interface de langage). Toutes les interfaces externes d'un composant doivent avoir des noms différents, mais deux interfaces dans deux composants différents peuvent avoir le même nom. Nous pouvons distinguer deux types d'interfaces : une interface *client* (ou *requise*) émet des invocations d'opérations, alors qu'une interface *serveur* (ou *fournie*) les reçoit.

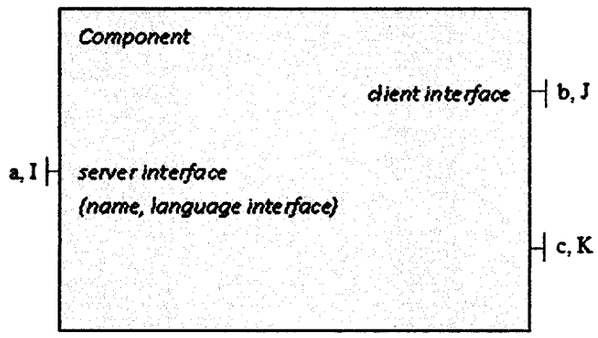


Figure 3.1-Vue externe d'un composant Fractal [13]

Les interfaces d'un composant peuvent être introspectées avec deux interfaces de langage, spécifiées dans les prochaines sections: l'une permet d'obtenir la liste des interfaces d'un composant, et l'autre permet d'introspecter les interfaces elles-mêmes. Ces deux interfaces sont bien sûr optionnelles, comme tout le reste dans le modèle Fractal : un composant peut choisir de fournir soit les deux interfaces, soit l'une d'entre elles ou aucune d'elles.

3.2.2.2.2 Introspection de composant

Afin de permettre de découvrir ses interfaces externes, un composant peut fournir une interface qui implémente l'interface `Component` (voir Figure 3.2). Cette interface de langage fournit deux opérations nommées `getFcInterfaces` et `getFcInterface`, qui peuvent être utilisées pour retrouver les interfaces du composant. La première opération n'a pas d'arguments, et retourne un tableau contenant toutes les interfaces externes, qu'elles soient client ou serveur, et incluant l'interface `Component`. La deuxième opération prend comme paramètre le nom d'une interface et retourne cette interface, si elle existe.

```
package org.objectweb.fractal.api;
interface Component {
   any[] getFcInterfaces ();
   any getFcInterface (string itfName) throws
NoSuchInterfaceException;
   Type getFcType ();
}
interface Type {
   boolean isFcSubTypeOf (Type t);
}
```

Figure 3.2-API d'introspection de composant

L'interface `Component` fournit aussi une opération `getFcType`, qui retourne le type du composant comme une référence de type `Type`. Cette interface définit une notion de type minimale, qui présentement définit une seule opération nommée `isFcSubTypeOf`, dont le role est de tester si un type donné est sous-type ou pas d'un autre type. Cette interface peut être étendue pour définir des systèmes de types plus utiles pour les composants et les interfaces, comme nous allons le voir plus bas.

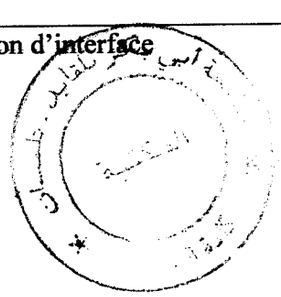
L'exception `org.objectweb.fractal.api.NoSuchInterfaceException` est déclenchée par l'opération `getFcInterface` lorsque l'interface requise est introuvable.

3.2.2.2.3 Introspection d'interface

Par défaut, les interfaces retournées par les opérations `getFcInterface` et `getFcInterfaces` fournissent l'accès aux interfaces requises, et rien de plus. En particulier, il est impossible d'obtenir les noms de ces interfaces. Afin de fournir de telles fonctions d'introspection d'interface, un composant peut assurer que les références retournées par les opérations précitées peuvent être transtypées en références de type `Interface` (voir Figure 3.3). Cette interface spécifie quatre opérations pour obtenir le nom d'une interface de composant, pour obtenir son type (comme une référence de type `Type`), pour obtenir l'interface `Component` du composant auquel il appartient, et pour tester si cette interface est interne ou pas.

```
package org.objectweb.fractal.api;
interface Interface {
   string getFcItfName ();
   Type getFcItfType ();
   Component getFcItfOwner ();
   boolean isFcInternalItf ();
}
```

Figure 3.3-API d'introspection d'interface



3.2.2.3 Introspection (introspection et intercession)

Au niveau de capacité de contrôle supérieur, au-delà du niveau « d'introspection » où les composants fournissent des interfaces permettant d'introspecter leurs caractéristiques externes, un composant Fractal peut fournir des interfaces de contrôle pour introspecter et reconfigurer ses caractéristiques *internes*. Cette section définit ces caractéristiques internes, et spécifie quelques interfaces utilisables pour les introspecter et les reconfigurer.

3.2.2.3.1 Structure interne d'un composant

En interne, un composant est formé de deux parties : un contrôleur (aussi appelé *membrane*), et un contenu (voir Figure 3.4). Le contenu d'un composant est composé d'un nombre fini d'autres composants, appelés sous-composants, qui sont sous son contrôle. Le modèle Fractal est ainsi récursif et permet aux composants d'être imbriqués dans d'autres composants à un niveau arbitraire. Un composant qui expose son contenu est appelé composant *composite*. Un composant qui n'expose pas son contenu, mais qui est doté au moins d'une interface de contrôle (voir plus haut) est appelé composant *primitif*. Un composant sans aucune interface de contrôle est appelé composant de base.

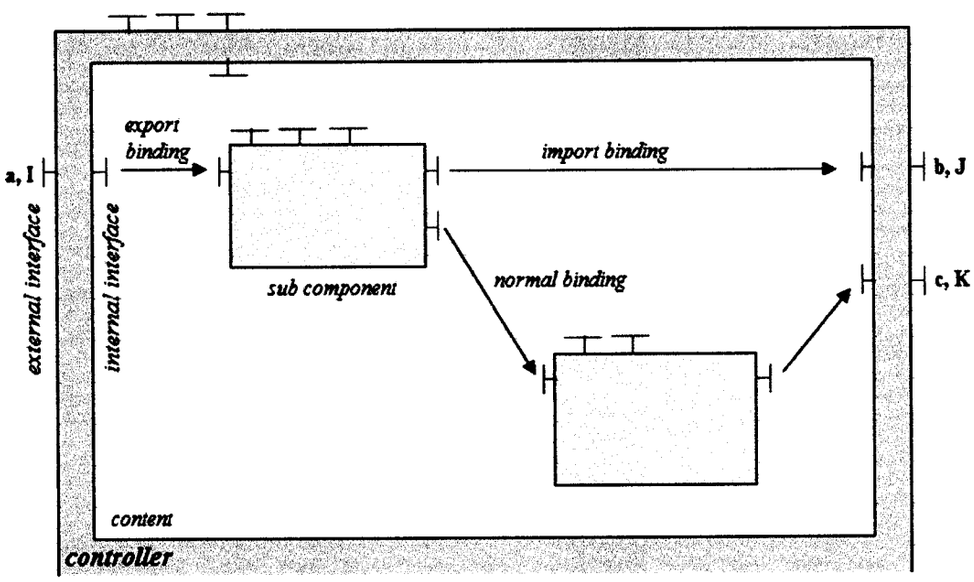


Figure 3.4-Vue interne d'un composant Fractal [13]

Le contrôleur d'un composant peut avoir des interfaces internes et externes. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes ne sont accessibles que de ses sous-composants. Toutes les interfaces externes d'un composant doivent avoir des noms différents, toutes les interfaces internes doivent avoir des noms différents, mais un composant peut avoir une interface externe et interne de même nom. Une interface *fonctionnelle* est une interface qui correspond à une fonctionnalité fournie ou requise par un composant, alors qu'une interface de *contrôle* est une interface *serveur* qui correspond

à un « aspect non fonctionnel », comme l'introspection, la configuration ou la reconfiguration, etc. Par convention, une interface est considérée comme interface de contrôle lorsque son nom est `component`, ou se termine par `-controller`. Toutes les autres interfaces sont des interfaces fonctionnelles.

Le contrôleur d'un composant incarne le contrôle de comportement associé à un composant particulier. En particulier, un contrôleur peut :

- fournir une *représentation causalement connectée* des sous-composants du composant ;
- intercepter les invocations d'opérations entrantes ou sortantes ciblant ou en provenance des sous-composants du composant ;
- superposer un contrôle de comportements aux comportements des sous-composants du composant, incluant la suspension, la vérification des liaisons et la reprise des activités des sous-composants.

Chaque contrôleur de composant peut être ainsi vu comme implémentant une sémantique de composition particulière des sous-composants du composant. Les capacités de contrôle d'un contrôleur ne sont pas fixées par le modèle. Par exemple, elles peuvent être basées principalement sur l'interception comme dans les canevas industriels de conteneurs par exemple, ou elles peuvent être inexistantes (c.-à-d. qu'aucun contrôle n'est exercé, dans ce cas, le contrôleur peut être toujours utile pour fournir une représentation du contenu et des relations d'imbrication).

Un composant peut faire parti du contenu de plusieurs composants composites. Un composant qui est partagé par deux composants ou plus est soumis au contrôle de leurs contrôleurs respectifs. La sémantique exacte de la configuration résultante est en général déterminée par un composant englobant tous les autres composants concernés par la configuration.

Une *liaison* est un chemin de communication entre des interfaces de composant. Le modèle Fractal fait une distinction entre les liaisons *primitives* et les liaisons *composites*. Une liaison primitive est une liaison entre une interface client et une interface serveur résidant toutes dans le même espace d'adressage. Une liaison composite est un chemin de communication entre un nombre arbitraire d'interfaces de composant, de types arbitraires. Ces liaisons sont représentées comme un ensemble de liaisons primitives et de *composants de liaison* (stubs, squelettes, etc.). Un composant de liaison est un composant Fractal normal fournissant une fonction pour la communication. Dans le paradigme composant, les composants de liaison sont généralement désignés par le terme *connecteurs* : donc il existe des connecteurs dans le modèle Fractal, quoique ce concept ne fasse pas partie des concepts de base comme les concepts de composants ou interfaces, raison pour laquelle aucune API ne lui est consacrée.

3.2.2.3.2 Contrôle d'attribut

Un *attribut* est une propriété configurable d'un composant, comme la couleur ou le texte d'un bouton, ou la taille maximale d'un pool. Les attributs sont généralement de types primitifs, et sont utilisés pour configurer l'état des composants en se passant de l'option des liaisons (il

est possible de configurer le texte d'un bouton, par exemple, en le liant à un composant texte ; mais le besoin requis ne justifie pas la complexité de cette solution). Un composant peut fournir une interface `AttributeController` pour permettre de lire et modifier ses attributs (voir Figure 3.5).

```
package org.objectweb.fractal.api.control;
interface AttributeController { }
```

Figure 3.5-API de contrôle des attributs

Le composant doit alors deriver une interface de cette interface, puisque l'interface `AttributeController` est en fait vide. L'interface dérivée doit contenir un getter et/ou un setter par attribut configurable. Fractal requiert que la signature des setters et getters suive les mêmes règles que celles du modèle Java Beans. Une interface de composant implémentant l'interface `AttributeController` doit être nommée `attribute-controller`.

3.2.2.3.3 Contrôle de liaison

Un composant peut fournir l'interface `BindingController` pour connecter et déconnecter ses interfaces client à d'autres composants à travers liaisons primitives (voir Figure 3.6).

```
package org.objectweb.fractal.api.control;
interface BindingController {
  string[] listFc ();
  any lookupFc (string clientItfName)
    throws NoSuchInterfaceException;
  void bindFc (string clientItfName, any serverItf)
    throws NoSuchInterfaceException, IllegalBindingException,
IllegalLifeCycleException;
  void unbindFc (string clientItfName)
    throws NoSuchInterfaceException, IllegalBindingException,
IllegalLifeCycleException;
}
```

Figure 3.6-API de contrôle des liaisons

Cette interface définit les operations suivantes:

- L'opération `listFC` retourne les noms des interfaces client du composant. Ce sont ces noms qui peuvent être passés en paramètre de l'opération `lookupFc`.
- L'opération `lookupFc` prend comme paramètre le nom d'une interface client du composant, qu'elle soit externe ou interne, et retourne l'interface serveur à laquelle cette interface client est liée, ou `null` si l'interface n'est pas liée.
- L'opération `bindFc` prend comme paramètres le nom d'une interface client du composant, qu'elle soit interne ou externe, et l'interface serveur d'un autre composant, et lie ces deux interfaces.

- L'opération unbindFc prend comme paramètres le nom d'une interface client du composant, qu'elle soit interne ou externe, et déconnecte cette interface.

Ces opérations peuvent lancer une exception NoSuchInterfaceException si l'interface client spécifiée n'existe pas, une exception IllegalLifeCycleException lorsque le composant n'est pas dans un état approprié pour le traitement des opérations et une exception org.objectweb.fractal.api.control.IllegalBindingException pour des erreurs relatives aux liaisons.

Une interface de composant implémentant l'interface BindingController doit être nommée binding-controller.

Les liaisons composites sont créées en utilisant une API spéciale spécifiée par le modèle pour l'accès aux interfaces de composants, qu'elles soient locales ou distantes (voir la spécification Fractal). Pour lier une interface client à une interface serveur distante, des liaisons primitives seront créées entre l'interface client et un composant de liaison spécifié par l'API précitée, et ensuite entre le composant de liaison et l'interface serveur désirée.

3.2.2.3.4 Contrôle de contenu

Un composant peut fournir l'interface ContentController pour permettre d'ajouter et supprimer des sous-composants à (de) son contenu (voir Figure 3.7).

```
package org.objectweb.fractal.api.control;
interface ContentController {
   any[] getFcInternalInterfaces ();
   any getFcInternalInterface (string itfName) throws
NoSuchInterfaceException;
   Component[] getFcSubComponents ();
   void addFcSubComponent (Component c)
      throws IllegalContentException,
IllegalLifeCycleException;
   void removeFcSubComponent (Component c)
      throws IllegalContentException,
IllegalLifeCycleException;
}
interface SuperController {
   Component[] getFcSuperComponents ();
}
```

Figure 3.7-API de contrôle de contenu

Cette interface définit trois opérations pour obtenir la liste des sous-composants d'un composant, et pour ajouter et supprimer des sous-composant au (du) composant :

- L'opération `getFcSubComponents` retourne la liste des sous-composants du composant dans un tableau de références `Component`. L'opération `getFcSuperComponents`, dans l'interface `SuperController` (voir Figure 3.7) fournit la fonction opposée : elle retourne les composites renfermant le composant, appelés alors ses *super* composants.
- L'opération `addFcSubComponent` prend comme paramètre une référence c de type `Component`, et ajoute le composant correspondant au contenu du composant.
- L'opération `removeFcSubComponent` prend comme paramètre une référence c de type `Component`, et supprime le composant correspondant du contenu du composant.

Un composant donné peut être ajouté au contenu de plusieurs composants. Un tel composant est dit *partagé* en ces composites. Les composants partagés sont paradoxalement utiles pour la préservation de l'encapsulation. Considérons par exemple un composant menu, un composant barre d'outils (voir Figure 3.8) et un bouton "undo" de barre d'outils correspondant aussi à un item de menu "undo". Il est naturel de représenter les items du menu et les boutons de la barre d'outils comme des sous-composants, encapsulés dans les composants menu et barre d'outils, respectivement. Mais sans le partage, cette solution n'arrange le bouton "undo" de la barre d'outils et l'item de menu, qui doivent avoir le même état (activé ou désactivé) : ces composants, ou un composant d'état associé, doivent être mis à l'extérieur des composants menu et barres d'outils. Avec les composants partagés, le composant d'état peut être partagé entre les composants menu et barre d'outils, afin de préserver l'encapsulation.

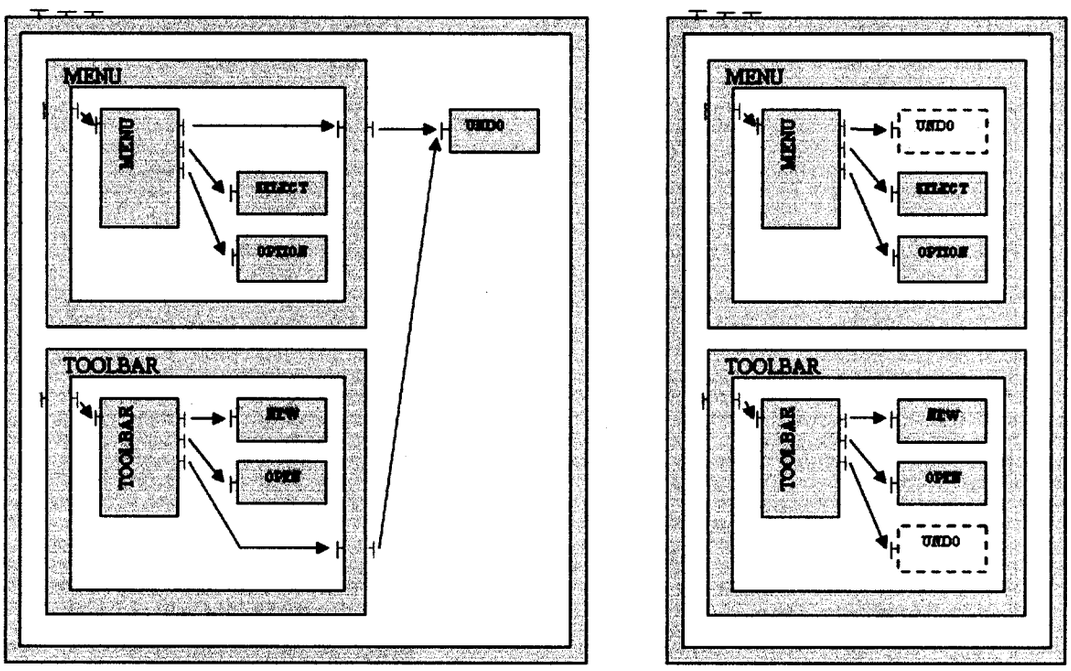


Figure 3.8-Avantage des composants partagés [13]

L'interface `ContentController` spécifie aussi deux operations pour obtenir les interfaces internes du composant, qui sont similaires aux opérations `getFcInterface` et `getFcInterfaces` de l'interface `Component`. Ces interfaces sont utiles pour lier les interfaces internes aux sous-composants.

Les opérations de l'interface `ContentController` peuvent lancer une `NoSuchInterfaceException` lorsqu'un composant n'est pas dans un état approprié pour traiter une opération et une exception `org.objectweb.fractal.api.control.IllegalContentException` pour les erreurs relatives au contrôle du contenu.

Une interface de composant implémentant l'interface `ContentController` (resp. `SuperController`) doit être nommée `content-controller` (resp. `super-controller`).

3.2.2.3.5 Contrôle du cycle de vie

Modifier un attribut ou une liaison, ou supprimer un sous-composant, avec les interfaces de contrôles décrites ci-dessus, et durant l'exécution des composants, peut être dangereux : des messages peuvent être perdus, l'état de l'application peut devenir incohérent, ou l'application peut tout simplement se planter. Afin d'offrir un support minimal pour aider dans la réalisation de telles reconfigurations dynamiques, un composant peut fournir l'interface `LifeCycleController` (voir Figure 3.9).

```
package org.objectweb.fractal.api.control;
interface LifeCycleController {
    string getFcState ();
    void startFc () throws IllegalLifeCycleException;
    void stopFc () throws IllegalLifeCycleException;
}
```

Figure 3.9-API de contrôle de cycle de vie

Cette interface spécifie deux opérations nommées `startFc` et `stopFc`, pour démarrer et arrêter un composant correctement. Elle spécifie également l'opération `getFcState` qui retourne l'état courant du composant.

3.2.3 Instanciation

Les API présentées dans les sections précédentes nous permettent d'utiliser, introspecter et reconfigurer des composants *existants*. Afin d'être utiles, ces API doivent être complétées par une API qui permet de créer des composants. Cette section décrit une telle API, basée sur les fabriques.

3.2.3.1 Fabriques

Dans cette API d'instanciation, les composants sont créés à partir d'autres composants appelés *fabriques* de composants. Nous distinguons dans cette API les fabriques de composants génériques, qui peuvent créer plusieurs types de composants, et les fabriques de composants standard qui créent un seul type de composants. Fabriques génériques et standard fournissent les interfaces `GenericFactory` et `Factory`, respectivement (voir Figure 3.10).

```
package org.objectweb.fractal.api.factory;
interface GenericFactory {
   Component newFcInstance (Type t, any controllerDesc,
any contentDesc)
      throws InstantiationException;
}
interface Factory {
   Type getFcInstanceType ();
   any getFcControllerDesc ();
   any getFcContentDesc ();
   Component newFcInstance () throws
InstantiationException;
}
```

Figure 3.10-API d'instanciation

L'interface `GenericFactory` spécifie une seule opération nommée `newFcInstance`. Cette operation prend comme paramètre le type du composant à créer, un descripteur de sa partie contrôleur et un descripteur de sa partie contenu. L'opération crée un composant correspondant à la description donnée et retourne son interface `Component`.

L'interface `Factory` fournit aussi une opération `newFcInstance`, mais cette dernière ne prend aucun paramètre, ceci reflète le fait que tous les composants créés par l'opération sont du même type, ont les mêmes descripteurs de contrôleur et de contenu. Ces informations sont retrouvées à partir des autres méthodes de l'interface.

Notons que la sémantique exacte des descripteurs de contrôleur et de contenu n'est pas spécifiée par la version courante de la spécification Fractal. Notons également que la spécification n'impose pas que les opérations `newFcInstance` retournent une nouvelle instance à chaque invocation, elles peuvent par exemple toujours retourner la même instance (patron singleton). La spécification impose par contre que les composants créés par une fabrique doivent être créés dans le même espace d'adressage que la fabrique.

L'exception
`org.objectweb.fractal.api.factory.InstantiationException` doit être lancée en cas d'échec de création de composant dans les opérations `newFcInstance` des deux interfaces. Une interface de composant implémentant `GenericFactory` (resp. `Factory`) doit être nommée `generic-factory` (resp. `factory`).

3.2.3.2 Patrons (templates)

Un *patron* (ou *moule*) est un type particulier de fabrique standard qui crée des composants qui lui sont quasiment « isomorphes ». Plus exactement, les composants créés par un patron *doivent* avoir les mêmes interfaces client et serveur que le composant patron (excepté pour l'interface `Factory`, qui est fournie par le patron, mais pas forcément par ces instances), mais peuvent avoir des interfaces de contrôle arbitraires. Les composants créés par un patron ont aussi les mêmes attributs que ce dernier. Un composant patron peut contenir plusieurs sous composants patron, liés entre eux à travers des liaisons. Les composants créés par le patron sont alors des composants qui contiennent autant de sous-composants que de sous patrons dans le patron et qui sont liés entre eux suivant les liaisons entre les sous patrons dans le patron (voir Figure 3.11). Si des sous patrons sont partagés, alors les sous-composants correspondants dans les composants créés par le patron le seront aussi.

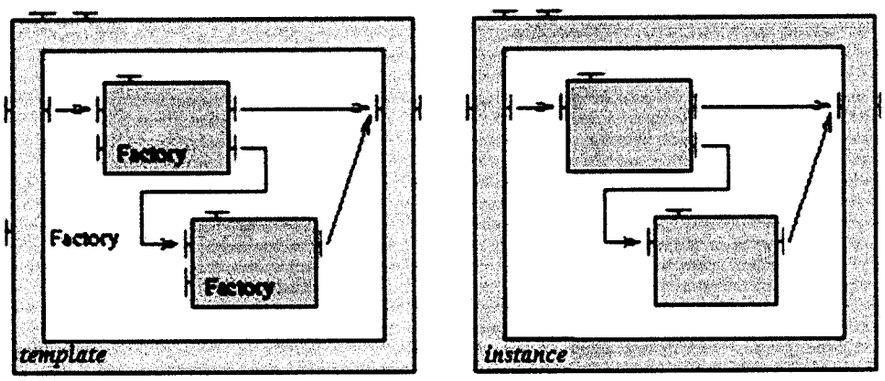


Figure 3.11-Un simple composant patron et un composant instancié à partir de lui [13]

Les patrons de composant sont créés à partir de fabriques de composant génériques, en faisant une invocation de la forme : `newFcInstance (type, templateControllerDesc, {controllerDesc, contentDesc})`, où `type` décrit les interfaces fonctionnelles client et serveur des composants que le patron devra créer, `templateControllerDesc` est le descripteur de la partie contrôleur du patron à créer, et `controllerDesc` et `contentDesc` sont les descripteurs des parties contrôleur et contenu que le patron devra créer (les accolades dénotent un tableau). La sémantique des deux deniers paramètres n'est pas spécifiée par la actuelle de la spécification Fractal.

Les patrons n'ont qu'une seule vraie utilité, c'est lorsque plusieurs composants doivent être créés à partir d'une représentation textuelle, par exemple un langage de description d'architecture (nous étudions dans une prochaine section Fractal ADL, le langage de description d'architecture associé à Fractal). Dans ce cas, au lieu d'analyser la représentation textuelle pour chaque instance à créer, il peut être plus efficace d'analyser le texte une seule

fois et de créer le patron de composant correspondant qui sera alors utilisé pour les futures instanciations.

3.2.3.3 Bootstrap

D'accord avec les API que nous venons de décrire, les composants sont créés à partir de fabriques de composants. Mais comment ces fabriques sont-elles créées ? Elles peuvent être créées à partir d'autres fabriques de composants, nous risquons là d'aboutir à une récursivité sans fin. Le problème est résolu par l'introduction d'une fabrique de composant d'amorçage (composant *bootstrap*), qui n'a pas besoin d'être créée explicitement et qui est accessible à partir d'un nom bien connu. Ce composant d'amorçage doit être capable de créer plusieurs types de composants, incluant les fabriques de composants. En d'autres mots, il doit fournir l'interface `GenericFactory`.

3.2.4 Système de types

Cette section définit un simple système de type pour les composants et les interfaces de composant. Ce système de type reflète les principales caractéristiques des interfaces de composant introduites dans la section 3.2.2.2, c.-à-d. leur nom, leur interface de langage et leur *rôle* (client ou serveur). Nous introduisons aussi deux nouvelles caractéristiques qui sont la *contingence* et la *cardinalité* d'une interface. La contingence d'une interface indique s'il est garanti que les opérations fournies ou requises d'une interface seront présentes ou non à l'exécution :

- les opérations d'une interface obligatoire sont toujours présentes. Pour une interface client, cela signifie que l'interface doit être liée pour que le composant s'exécute.
- les opérations d'une interface optionnelle ne sont pas nécessairement disponibles. Pour un composant serveur, cela peut signifier que l'interface interne complémentaire n'est pas liée à un sous-composant implantant l'interface. Pour un composant client, cela signifie que le composant peut s'exécuter sans que son interface soit liée.

La cardinalité d'une interface de type T spécifie le nombre d'interfaces de type T que le composant peut avoir. Une cardinalité *singleton* signifie que le composant doit avoir une, et seulement une, interface de type T. Une cardinalité *collection* signifie que le composant peut avoir un nombre arbitraire d'interfaces du type T. Ces interfaces sont généralement créées de façon paresseuse à l'exécution. Fractal n'impose aucune contrainte sur la sémantique opérationnelle des interfaces de cardinalité collection.

Le système de types permet également de décrire le type d'un composant comme l'ensemble des types de ses interfaces. Le type d'un composant est représenté par l'interface `ComponentType` (voir Figure 3.12). Cette interface fournit deux opérations `getFcInterfaceTypes` et `getFcInterfaceType`, retournant respectivement les types de toutes les interfaces du composant et le type d'une interface donnée, cette dernière opération déclenchant une exception `NoSuchInterfaceException` si l'interface requise n'existe pas.

Le type d'une interface de composant est représenté par l'interface `InterfaceType` voir (Figure 3.12). Cette interface spécifie des opérations permettant d'obtenir le nom, la signature, le type client ou serveur, la contingence et la cardinalité d'une interface de composant.

Les types de composant et d'interface peuvent être créés en utilisant une fabrique de type, représentée par l'interface `TypeFactory` (voir Figure 3.12). Cette interface fournit deux méthodes pour la création de types d'interface de composant et de types de composant. Une interface de composant implémentant l'interface `TypeFactory` doit être nommée `type-factory`.

```
package org.objectweb.fractal.api.type;
interface ComponentType extends Type {
   InterfaceType[] getFcInterfaceTypes ();
   InterfaceType getFcInterfaceType (string itfName)
throws NoSuchInterfaceException;
}
interface InterfaceType extends Type {
   string getFcItfName ();
   string getFcItfSignature ();
   boolean isFcClientItf ();
   boolean isFcOptionalItf ();
   boolean isFcCollectionItf ();
}
interface TypeFactory {
   InterfaceType createFcItfType (string name, string
signature, boolean isClient,
      boolean isOptional, boolean isCollection) throws
InstantiationException;
   ComponentType createFcType (InterfaceType[] itfTypes)
throws InstantiationException;
}
```

Figure 3.12-API du système de type

3.2.5 Niveaux de conformance

Nous l'avons souligné plus d'une fois, tout dans le modèle Fractal est optionnel. Un composant peut selon son choix fournir l'interface `Component`, l'interface `Interface`, des interfaces de contrôle de sa structure interne, il peut selon son choix utiliser le système de type. En plus de pouvoir faire ces choix, un composant peut simplement modifier la

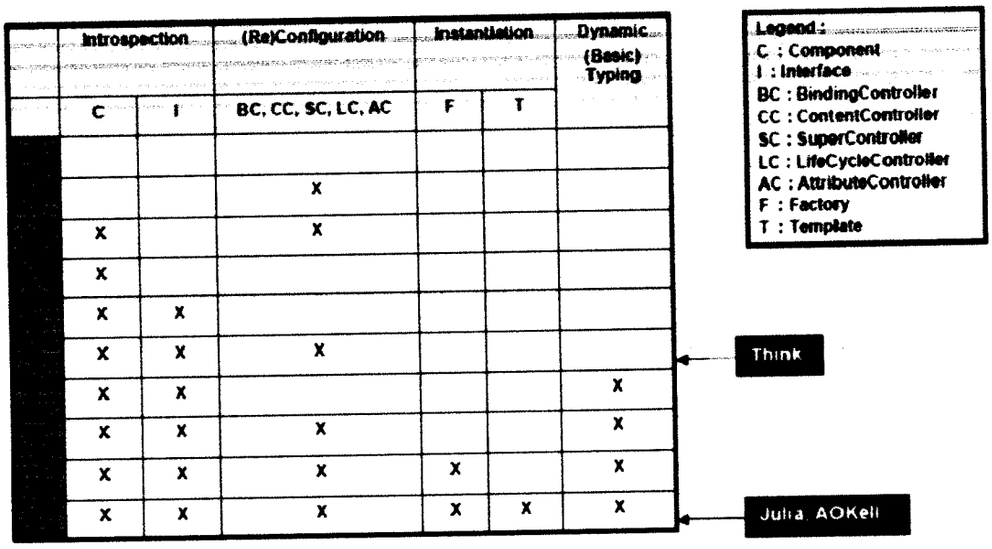


Figure 3.13-Niveaux de conformance du modèle Fractal [14]

3.3 Fractal ADL

Fractal fournit un langage de description d'architecture (ADL) dont la caractéristique principale est d'être extensible. La motivation pour une telle extensibilité est double. D'une part le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (`LoggerController`) soit ajouté à un composant. Il est nécessaire que l'ADL puisse être étendu facilement pour prendre en compte ce nouveau contrôleur, c'est-à-dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (par exemple *debug*, *warning*, *error*). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL : déploiement, vérification, analyse, etc.

Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. Nous présentons ces deux éléments dans les 3.3.1 et 3.3.2 sections suivantes. Le chapitre 4 dédié à la construction d'une application Fractal illustre des utilisations de Fractal ADL.

3.3.1 Le langage extensible

Le langage ADL de Fractal est basé sur le langage XML. Contrairement aux autres ADL qui fixent l'ensemble des propriétés (implantation, liaisons, attributs, localisation, etc.) qui doivent être décrites pour chaque composant, l'ADL Fractal n'impose rien. Il est constitué d'un ensemble (extensible) de modules permettant la description de divers aspects de l'application. Chaque module — à l'exception du module de base — s'applique à un ou

plusieurs autres modules, c'est-à-dire rajoute un ensemble d'éléments et d'attributs XML à ces modules. Le module de base définit l'élément XML qui doit être utilisé pour démarrer la description de tout composant. Cet élément, appelé `definition`, a un attribut obligatoire, appelé `name`, qui spécifie le nom du composant décrit.

Différents types de modules peuvent être définis. Un exemple typique de module est le module *containment* qui s'applique au module de base en permettant d'exprimer des relations de contenance entre composants. Ce module définit un élément XML component qui peut être ajouté en sous-élément d'un élément `definition` ou de lui-même pour spécifier les sous-composants d'un composant. Notons que l'élément `component` a un attribut obligatoire `name` qui permet de spécifier le nom du sous-composant. Fractal ADL définit actuellement trois autres modules qui s'appliquent soit au module de base, soit au module `containment` pour spécifier l'architecture de l'application : le module `interface` permet de décrire les interfaces d'un composant; le module implémentation permet de décrire l'implantation des composant primitifs; le module `controller` permet la description de la partie contrôle des composants.

Les modules ne servent pas uniquement à décrire les aspects architecturaux de l'application. Par exemple, Fractal ADL fournit des modules permettant d'exprimer des relations de référencement et d'héritage entre descriptions ADL. Le rôle principal de ces modules est de faciliter l''ecriture de définition ADL. Le module de référencement s'applique au module *`containment`* en ajoutant un attribut `definition` à l'élément component pour référencer une définition de composant. Le module d'héritage s'applique au module de base. Il permet à une définition d'en étendre une autre (via un attribut `extends`). Notons que l'héritage proposé par Fractal ADL est multiple.

3.3.2 L'usine extensible

L'usine permet de traiter les définitions écrites à l'aide du langage extensible Fractal ADL. Elle est constituée d'un ensemble de composants Fractal qui peuvent être assemblés pour traiter les différents modules de Fractal ADL décrits précédemment. Ces composants sont représentés sur la figure 3.14.

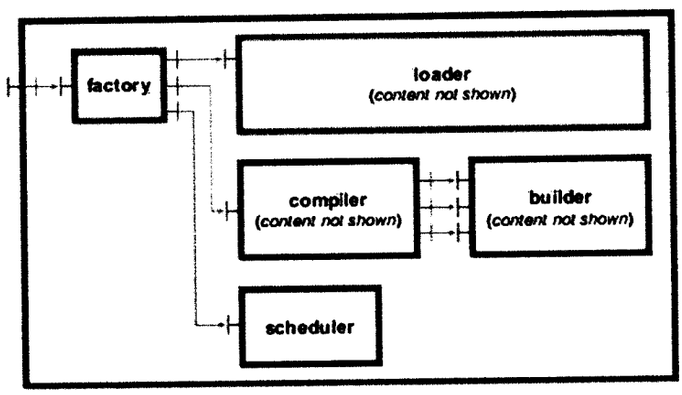


Figure 3.14-Architecture de l'usine Fractal ADL [15]

Le composant loader analyse les définitions ADL et construit un arbre abstrait correspondant (AST pour *Abstract Syntax Tree*). L'AST implante deux API distinctes : une API `générique` similaire à celle de DOM [W3C-DOM 2005] qui permet de naviguer dans l'arbre; une API typée qui varie suivant les modules qui sont utilisés dans Fractal ADL. Par exemple, si le module `interface` est utilisé, l'API typée contient des méthodes permettant de récupérer les informations sur les interfaces de composants (nom, signature, rôle, etc.). Le composant `loader` est un composite encapsulant une chaîne de composants primitifs (voir Figure 3.15). Le composant le plus à droite dans la chaîne (`basic loader`) est responsable de la création des AST à partir de définitions ADL. Les autres composants effectuent des vérifications et des transformations sur les AST. Chaque composant correspond à un module de l'ADL. Par exemple, le composant `binding loader` vérifie les liaisons déclarées dans l'AST, etc.

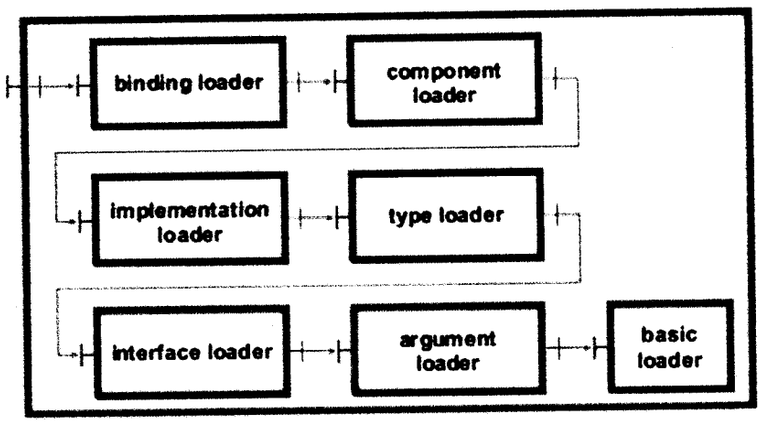


Figure 3.15-Architecture du composant loader [15]

Le composant compiler utilise l'AST pour définir un ensemble de tâches à exécuter (par exemple création de composants, établissement de liaisons). Le composant compiler est un composite encapsulant un ensemble de composants *compiler* primitifs (voir Figure 3.16). Chaque *compiler* primitif produit des tâches correspondant à un ou plusieurs modules ADL. Par exemple, le composant `binding compiler` produit des tâches de création de liaisons. Les tâches produites par un *compiler* primitif peuvent dépendre des tâches produites par un autre *compiler* primitif ce qui impose un ordre dans l'exécution des *compiler*. Par exemple, le *compiler* primitif qui produit les tâches de création des composants doit être exécuté avant les *compilers* en charge des liaisons et des attributs.

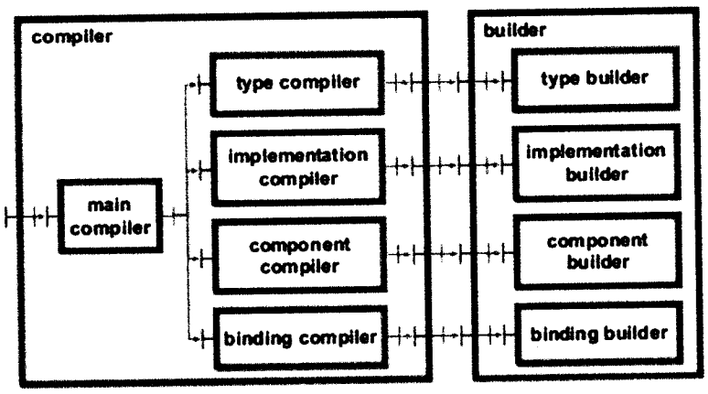


Figure 3.16-Architecture des composants `compiler` et `builder` [15]

Le composant builder définit un comportement concret pour les tâches créées par le `compiler`. Par exemple, un comportement concret d'une tâche de création de composant peut être d'instancier le composant à partir d'une classe Java. Le composant `builder` est un composite qui encapsule plusieurs *builders* primitifs (figure 3.16). Chaque canevas logiciel peut définir ses composants `builder`. Par exemple, Julia fournit actuellement quatre composants `builder` qui permettent respectivement de créer des composants avec l'API Java, l'API Fractal ou de produire du code source permettant d'instancier les composants à partir de ces deux API.

3.4 Plates-formes

Fractal est un modèle de composants indépendant des langages de programmation. Plusieurs plates-formes sont ainsi disponibles dans différents langages de programmation. Julia (section 3.4.1) l'implémentation de référence de Fractal, a été développée pour le langage Java. Une deuxième plate-forme Java, AOKell, développée plus récemment est présentée en section 3.4.2. Par rapport à Julia, AOKell apporte une mise sous forme de composants des membranes. La section 3.2.3 présente un aperçu des autres plates-formes existantes.

3.4.1 Julia

Julia est l'implémentation de référence du modèle de composant Fractal. Sa première version remonte à juin 2002. Julia est disponible sous licence open source LGPL sur le site du projet Fractal[2]. Les choix de conception faits pour l'implémentation de Julia sont expliqués dans [16] et [17].

Julia est un canevas logiciel écrit en Java qui permet de programmer les membranes des composants. Il fournit un ensemble de contrôleurs que l'utilisateur peut assembler. Par ailleurs, Julia fournit des mécanismes d'optimisation qui permettent d'obtenir un continuum allant de configurations entièrement statiques et très efficaces à des configurations dynamiquement reconfigurables et moins performantes. Le développeur d'application peut

[2] http://fractal.objectweb.org/

ainsi choisir l'équilibre performance/dynamicité dont il a besoin. Enfin, notons que Julia s'exécute sur toute JVM, y compris celles qui ne fournissent ni chargeur de classe, ni API de réflexivité.

Pour simplifier l'implémentation, les concepteurs ont fait les hypothèses suivantes (ces hypothèses seront peut-être levées dans les prochaines versions de Julia):

- concurrence: il est supposé qu'il y a au plus un thread de reconfigurations à la fois;
- protection: les composants sont supposés non malveillants envers la plate-forme et les autres composants. L'accès aux structures internes de Julia n'est donc pas protégé.

3.4.1.1 Principales structures de données

Un composant Fractal est formé de plusieurs objets Java que l'on peut séparer en trois groupes (figure 3.17) :

- Les objets qui implémentent le contenu du composant. Ces objets n'ont pas été représentés sur la figure. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
- Les objets qui implémentent la partie de contrôle(ou membrane) du composant (représentés en gris). Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle et des intercepteurs optionnels qui interceptent les appels de méthodes entrants et sortants. Les fonctions de contrôle n'étant généralement pas indépendantes, les contrôleurs et les intercepteurs possèdent généralement des références les uns vers les autres.
- Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.

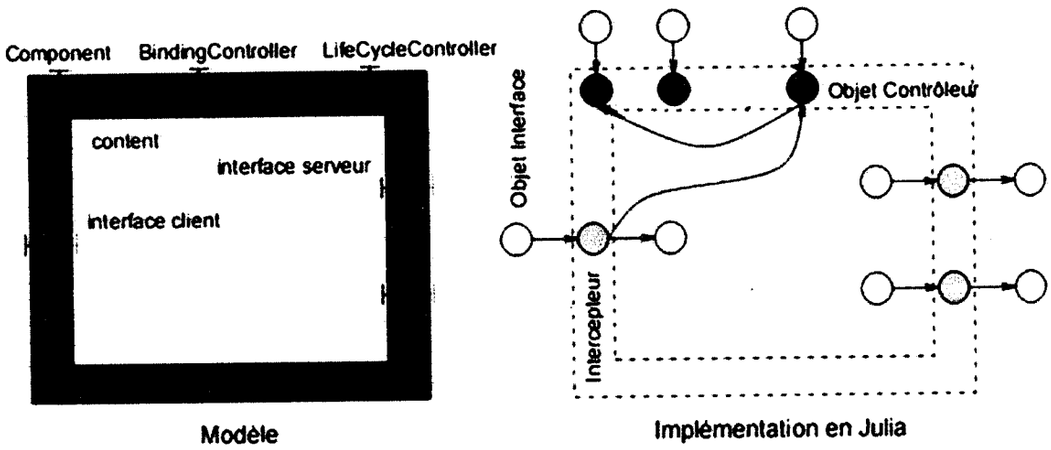


Figure 3.17-Un composant Fractal et son implantation Julia [15]

Le fait que chaque interface Fractal soit représentée par un objet Java séparé vient du fait que les références d'interfaces sont typées (c'est-à-dire qu'une une référence à une interface Fractal dont le type est `I` doit implémenter l'interface `Interface` et I).

Les objets qui composent la membrane d'un composant peuvent être séparés en deux groupes : les objets qui implémentent les interfaces de contrôle (en gris sur la figure 3.20), et les objets d'interception (en gris clair), optionnels qui peuvent intercepter des appels entrants ou sortants sur les interfaces fonctionnelles du composant. Ces objets implémentent respectivement les interfaces `Controller` et `Interceptor`. Ces objets peuvent avoir des références les uns vers les autres (puisque les aspects de contrôle ne sont généralement orthogonaux, ils doivent généralement communiquer entre eux).

Les objets qui implémentent le contenu peuvent être des sous-composants (pour les composants composites) ou des objets utilisateur (pour les composants primitifs).

Les composants de Julia peuvent être créés manuellement ou automatiquement. La première méthode peut être utilisée pour créer n'importe quel type de composant, alors que la méthode automatique est réservée aux composants respectant le système de base défini par la spécification Fractal qui fournit l'interface `Component` et l'interface des fonctions d'introspection. Dans chacune de ces méthodes, le composant doit être créé comme suit :

- création des objets représentant les interfaces du composant, de ses objets de contrôle, de ses objets d'interception et de son contenu ;
- initialisation des références des objets représentant les interfaces du composant vers le contenu, les objets d'interception et les objets de contrôle ;
- création d'un objet `InitializationContext`, et initialisation de cet objet avec les références aux objets créés précédemment ;
- initialisation des objets de contrôle et d'interception par appel de leur méthode `initFcController` avec l'objet `InitializationContext` précédent en paramètre, cette étape permet aux objets de contrôle et d'interception d'initialiser leurs références les uns vers les autres.

3.4.1.2 Composition des aspects de contrôle

Motivation

La spécification Fractal est supposée extensible, c'est-à-dire qu'on doit pouvoir lui ajouter de nouvelles interfaces de contrôle. Pour atteindre ce but, chaque interface de contrôle est implantée dans une classe séparée dans Julia, et le contrôleur, d'un objet est construit en reliant les objets de contrôle instanciés à partir de ces classes.

Mais cette séparation n'est pas suffisante car les « aspects » de contrôle peuvent se mélanger. C'est le cas par exemple avec les aspects liaisons et cycle de vie : la spécification dit qu'une liaison ne doit pas pouvoir être modifiée si un contrôleur de cycle de vie est présent et si le composant n'est pas stoppé. Il est donc nécessaire d'introduire dans la méthode `bindFc` (qui

ne devrait contenir idéalement que du code lié à la gestion des liaisons) du code pour tester la présence d'un contrôleur de cycle de vie et pour tester si le composant est stoppé.

Une première solution à ce problème serait d'utiliser de classes. L'aspect liaisons serait alors implémenté dans sa propre classe `B`, sans aucun code lié au cycle de vie, et l'aspect cycle de vie serait implémenté dans sa propre classe `L`, et dans une sous classe `LB` de `B`, qui surchargerait la méthode `bindFc` pour lui ajouter les vérifications précédentes. Mais cette solution ne marche plus avec plusieurs aspects. Par exemple, l'aspect « contenu » aurait besoin de sa propre classe `C` et d'une sous classe `CB` de `B`, qui surchargerait la méthode `bindFc` pour vérifier les contraintes liées aux liaisons autorisées. Mais pour utiliser ces trois aspects à la fois, il faudrait une sous classe de `LB` et `CB`, ce qui est impossible sans héritage multiple. Et même avec cette fonctionnalité, il resterait un problème d'explosion combinatoire entrainant de nombreuses redondances de code si on introduisait encore plus d'aspects, du fait que chaque aspect peut être implémenté de plusieurs façons différentes (par exemple Julia fournit quatre implémentations de l'interface `BindingController` : il faudrait donc fournir quatre sous classes, pratiquement identiques, similaires à `CB`, quatre sous classes similaires à `LB`, et ainsi de suite).

Implémentation

La solution adoptée dans Julia est l'utilisation de classes *mixin* [Bracha and Cook 1990] : une classe mixin est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. La classe mixin peut s'appliquer (c'est-à-dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe. Ces classes mixin sont appliquées au chargement à l'aide de l'outil ASM. Dans Julia, les classes mixin sont des classes abstraites développées avec certaines conventions. En l'occurrence, elles ne nécessitent pas l'utilisation d'un compilateur Java modifié ou d'un pré-processeur comme c'est le cas des classes mixins développées à l'aide d'extensions du langage Java. Par exemple la classe mixin JAM illustrée sur la partie gauche de la figure 3.18 s'écrit en Julia en pur Java (partie droite). Le mot clé `inherited` en JAM est équivalent au préfixe `super` utilisé dans Julia. Il permet de spécifier les membres qui doivent être présents dans la classe de base pour que le mixin lui soit appliqué. De façon plus précise, le préfixe `super` spécifie les méthodes qui sont surchargées par le mixin. Les méthodes qui sont requises mais pas surchargées sont spécifiées à l'aide du préfixe `this`.

L'application de la classe mixin `A` à la classe `Base` décrite sur la partie gauche de la figure 3.19 donne la classe `C55d992cb_0` représentée sur la partie droite de la figure 3.20.

Les classes mixin peuvent être *mixées*. Plus exactement, le résultat du mixage des classes mixin `M1`,..., `Mn`, *dans cet ordre*, serait une classe équivalente à la classe `Mn` étendant la classe `Mn-1`, elle-même étendant la classe `Mn-2`,..., elle-même étendant la classe `M1`. La condition à respecter pour mixer plusieurs classes mixin est que tous les attributs et méthodes requis par

une classe mixin `Mi` soient fournies par une classe mixin `Mj`, avec `j<i` (les attributs ou méthodes requis peuvent être fournis par des classes séparées).

```
mixin Compteur {
   inherited public void m ();
   public int count;
   public void m () {
      ++count;
      super.m();
   }
}
```

```
abstract class Compteur {
   abstract void _super_m ();
   public int count;
   public void m () {
      ++count;
      _super_m();
   }
}
```

Figure 3.18-Ecriture d'une classe mixin en JAM et en Julia [15]

```
abstract class Base {
   public void m () {
      System.out.println("m");
   }
}
```

```
public class C55d992cb_0 implements
                                    Generated {
   // from Base
   private void m$0 () {
      System.out.println("m");
   }
   // from A
   public int count;
   public void m () {
      ++count;
      m$0();
   }
}
```

Figure 3.19-Application d'une classe mixin [15]

Exemples

Julia utilise plusieurs classes mixin, pour séparer les codes liés aux divers «aspects » définis dans la spécification Fractal (cycle de vie, contenu, attributs,...). Par exemple :

- la classe `AttributeTemplateMixin` est ajoutée à chaque classe de contrôleur qui implémente l'interface `JuliaTemplate`, elle gère les attributs du template (par convention une classe mixin de la forme PréfixePostfixeMixin ne peut être appliquée que sur les classes qui implémentent l'aspect Postfix, et modifie ces classes pour gérer l'aspect Préfixe) ;
- les classes `BindingMixin`, `LifeCycleBindingMixin` et `ContentBindingMixin` sont appliquées aux classes de contrôle des liaisons et surchargent leurs méthodes afin de vérifier, avant toute modification de liaison, les contraintes générales, liées au cycle de vie et liées au contenu (respectivement).

3.4.1.3 Intercepteurs

Certains aspects de contrôle, comme le contrôle des liaisons, peuvent être implémentés définitivement de façon générique dans un seul objet. D'autres aspects de contrôle par contre doivent être implémentés en deux parties : une partie générique, et une partie non générique qui doit être liée au code de l'utilisateur. Dans Julia, cette partie non générique est implémentée par les objets d'interception.

Les classes d'interception, puisqu'elles ne sont pas génériques (elles implémentent une ou plusieurs interfaces fonctionnelles), ne peuvent pas être écrites à la main, contrairement aux classes de contrôle, et doivent par conséquent être générées automatiquement par un générateur de classes. Ce générateur prend en paramètre le nom d'un super classe, les noms d'une ou plusieurs interfaces applicatives, et les noms d'un ou plusieurs générateurs de code « d'aspect » d'interception. Il génère une sous classe de la super classe qui implémente toutes les interfaces spécifiées et qui pour chaque méthode applicative, implémente tous les « aspects » correspondant aux générateurs de code « d'aspect ».

Chaque générateur de code « d'aspect » peut modifier le code de chaque méthode applicative de façon arbitraire. Par exemple, un générateur de code A peut modifier une méthode

```
void m () { impl.m() }
```

en:

```
void m () {
  // pre code A
  try {
    impl.m();
  } finally {
    // post code A
  }
}
```

Alors qu'un autre générateur de code B modifierait la méthode en :

```
void m () {
  // pre code B
  impl.m();
  // post code B
}
```

Lorsqu'une classe d'interception est générée en utilisant plusieurs générateurs de code, les transformations effectuées par ces générateurs sont composées automatiquement. Par

exemple, si A et B sont utilisés pour générer une classe d'intercepteur, dans ce ordre, le résultat pour la méthode m précédente est le suivant :

```
void m () {
  // pre code A
  try {
    // pre code B
    impl.m();
    // post code B
  } finally {
    // post code A
  }
}
```

Notez que grâce à ce mélange de code (élémentaire) automatique, plusieurs aspects peuvent être gérés par un seul et même objet d'interception : il n'y a pas besoin d'utiliser des chaînes d'objets d'interposition, chaque objet correspondant à un aspect.

3.4.1.4 Optimisations

Julia offre deux mécanismes d'optimisation, intra et inter composants. Le premier mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant une partie de ses objets de contrôle. Pour ce faire, Julia fournit un outil utilisant ASM et imposant certaines contraintes sur les objets de contrôle fusionnés : par exemple, deux objets fusionnés ne peuvent pas implémenter la même interface. Nous distinguons quatre degrés dans l'optimisation intra-composant : l'optimisation obtenue grâce à la fusion des objets de contrôle (Figure 3.20 a), l'optimisation obtenue grâce à la fusion des objets de contrôle et des intercepteurs (Figure 3.20 b), l'optimisation obtenu grâce à la fusion des objets de contrôle, des intercepteurs et des objets utilisateur (Figure 3.20 c) et une dernière optimisation non illustrée ici, il s'agit de celle obtenue par la fusion des objets de contrôle et des objets utilisateur.

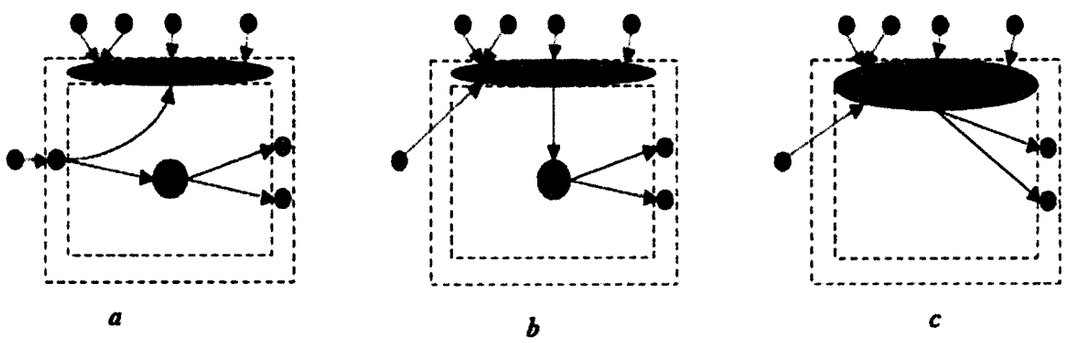


Figure 3.20-Optimisations intra composant [14]

Le second mécanisme d'optimisation a pour fonction d'optimiser les chaînes de liaison entre composants : il permet de court-circuiter les parties contrôle des composites qui n'ont pas

d'intercepteurs. Comme nous l'avons déjà expliqué, chaque interface de composant est représentée par un objet qui contient une référence vers un objet implantant réellement l'interface. Le principe du mécanisme de court-circuitage est représenté sur la figure 3.21 : un composant primitif est relié à un composite exportant l'interface d'un composant primitif qu'il encapsule. En conséquence, il existe deux objets de référencement d'interface (*r1* et *r2*). Seuls les appels du primitif sont interceptés (objet *i2*). En conséquence, Julia court-circuite l'objet *r2* et *r1* référence directement *i2*.

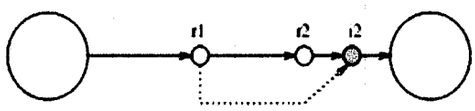


Figure 3.21-Optimisation des chaînes de liaison [15]

3.4.1.5 Fichier de configuration

Nous avons vu à la section 3.2.3.1 que le descripteur du contrôleur d'un composant faisait partie des paramètres requis par les fabriques de composants pour procéder à leur instanciation, nous avons même souligné que la sémantique de ce descripteur n'est pas spécifié par Fractal. Le choix d'une sémantique revient donc aux différentes implémentations de la spécification Fractal. Nous avons déjà eu l'occasion de rencontrer le descripteur « *primitive* » dans la section 3.1.1, où nous avons précisé la nature du mot. Julia a donc choisi d'associer des chaînes de caractères à chaque type de descripteur de composant, la définition de ces chaînes se trouvant dans un fichier configuration dont le chemin est indiqué à Julia pendant son lancement. Julia définit par défaut dans ce fichier de configuration treize descripteurs de composant : `primitive`, `parametricPrimitive`, `composite`, `parametricComposite`, etc. Puisque Fractal est extensible, l'utilisateur peut modifier ces descripteurs ou en définir pour ses propres contrôleurs. Les descripteurs `primitive` et `composite` définissent des composants correspondants aux types correspondants dans la spécification. De tels descripteurs fournissent toutes les informations nécessaires à la mise en place de la membrane du composant : les interfaces de contrôle, les objets implémentant ces interfaces, les générateurs de classes d'interception, les options d'optimisations, etc.

La syntaxe utilisée dans le fichier de configuration est la suivante :

```
(x (1 2 3) )        # x est égal à(1 2 3)

(y (a b c 'x) )  # y est égal à(a b c (1 2 3))
```

La figure 3.25 illustre par exemple la définition correspondante à la description des composants primitifs. Par exemple pour la description `'component-itf`(voir Figure 3.22), nous trouverons dans le fichier la définition suivante :

```
(component-itf

  (component org.objectweb.fractal.api.Component)

)
```

Il en est de même pour les autres descriptions similaires.

```
(primitive
  ('interface-class-generator
    ( 'component-itf
      'binding-controller-itf
      'super-controller-itf
      'lifecycle-controller-itf
      'name-controller-itf )
    ( 'component-impl
      'container-binding-controller-impl
      'super-controller-impl
      'lifecycle-controller-impl
      'name-controller-impl )
    ( (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
       org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator ) )
    org.objectweb.fractal.julia.asm.MergeClassGenerator
    'optimizationLevel
```

Interfaces de contrôle

Implémentations

Intercepteurs

Option d'optimisation

Figure 3.22-Définition du descripteur *primitive* dans le fichier de configuration de Julia

3.4.2 AOKell

Comme Julia, le canevas logiciel AOKell est une implémentation complète des spécifications Fractal. Le respect de l'API Fractal permet ainsi d'exécuter telle quelle, avec AOKell, des applications conçues pour Julia ou vice-versa. AOKell est disponible sous licence open source LGPL sur le site du projet Fractal. Le développement de AOKell a débuté en décembre 2004.

Le canevas logiciel AOKell diffère de Julia sur deux points : l'intégration des fonctions de contrôle dans les composants est réalisée à l'aide d'aspects et les contrôleurs sont implémentés eux-mêmes sous forme de composants. Par rapport à Julia qui utilise un mécanisme de mixin et de la génération de bytecode à la volée avec ASM, l'objectif d'AOKell est de simplifier et de réduire le temps de développement de nouveaux contrôleurs et de nouvelles membranes.

La suite de cette section présente le principe de mise sous forme de composants des membranes et la façon dont AOKell utilise les aspects.

Membranes componentisées

La membrane d'un composant Fractal est composée d'un ensemble de contrôleurs. Chaque contrôleur est dédié à une tâche précise : gestion des liaisons, du cycle de vie, etc. Loin d'être complètement autonomes, ces contrôleurs collaborent entre eux afin de remplir la fonction qui leur est assignée. Par exemple, lors du démarrage d'un composite, son contenu doit être visité afin de démarrer récursivement tous les sous-composants. De ce fait, le contrôleur de cycle de vie dépend du contrôleur de contenu. Plusieurs autres dépendances de ce type peuvent être exhibées entre contrôleurs.

Jusqu' à présent ces dépendances étaient implémentées sous la forme de références stockées dans le code des contrôleurs. L'idée d'AOKell est d'appliquer à la conception de la membrane de contrôle le principe qui a été appliqué aux applications : extraire du code les schémas de dépendances et programmer celui-ci sous forme de composants. En « spécifiant contractuellement les interfaces » [8] de ces composants de contrôle, AOKell espère favoriser leur réutilisation, clarifier l'architecture de la membrane et faciliter le développement de nouveaux composants de contrôle et de nouvelles membranes. Cette approche devrait également permettre d'apporter plus de flexibilité aux applications Fractal en permettant d'adapter plus facilement leur contrôle à des contextes d'exécutions variés ayant des caractéristiques différentes en terme de gestion des ressources (mémoire, activité, etc.).

Ainsi, AOKell est un canevas logiciel dans lequel les concepts de composant, d'interface et de liaison sont utilisées pour concevoir le niveau applicatif et le niveau de contrôle. Une membrane AOKell est un assemblage exportant des interfaces de contrôle et contenant un nombre quelconque de sous-composants. Chacun d'eux implémente une fonctionnalité de contrôle particulière. Comme expliqué précédemment, chaque composant de contrôle est aussi associé à un aspect qui intègre cette logique de contrôle dans les composants de niveau applicatif. La figure 3.23 présente le schéma de principe de cette solution. Par soucis de clarté, la membrane de contrôle du troisième composant applicatif a été omise.

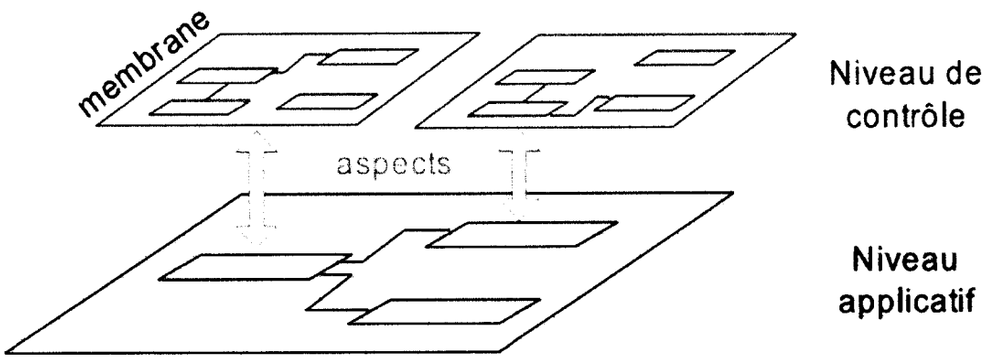


Figure 3.23-Les niveaux de composant du canevas logiciel AOKell [15]

La membrane la plus courante dans les applications Fractal est celle associée aux composants primitifs. L'architecture de cette membrane est illustrée figure 3.24. Cette membrane fournit cinq contrôleurs pour gérer le cycle de vie (LC), les liaisons (BC), le nommage (NC), les références vers les composants parents (SC) et les caractéristiques communes à tout composant Fractal (Comp).

L'architecture présentée figure 3.24 illustre le fait que la fonction de contrôle des composants primitifs n'est pas réalisée simplement par cinq contrôleurs isolés, mais est le résultat de leur coopération. Comparée à une approche purement objet, l'implémentation des membranes sous la forme d'un assemblage permet de d'écrire explicitement les dépendances entre contrôleurs. Elle permet également d'aboutir à des architectures logicielles de contrôle plus explicites, plus évolutives et plus maintenables. De nouvelles membranes peuvent être développées en étendant les existantes, ou en en développement des nouvelles.

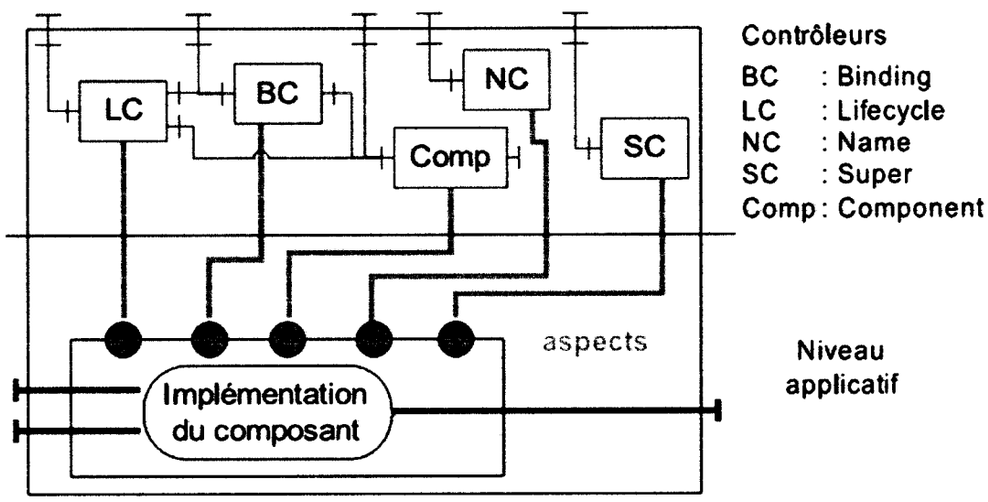


Figure 3.24-Membrane de contrôle pour les composants primitifs [15]

Le code Fractal ADL suivant fournit la description de la membrane de la figure 3.25. La DTD permettant d'écrire cet assemblage et l'API Java permettant de le manipuler sont exactement les mêmes que ceux utilisés pour les composants Fractal applicatifs.

```
<definition
name="org.objectweb.fractal.aokell.lib.membrane.primitive.Primi
tive">

  <!-- Composants de contrôle inclus dans une membrane
  primitive -->
  <component name="Comp" definition="ComponentController"/>
  <component name="NC" definition="NameController"/>
  <component name="LC"
  definition="NonCompositeLifeCycleController"/>
  <component name="BC"
  definition="PrimitiveBindingController"/>
  <component name="SC" definition="SuperController"/>

  <!-- Export des interfaces de contrôle -->
  <binding client="this.//component"
  server="Comp.//component"/>
  <binding client="this.//name-controller" server="NC.//name-
  controller"/>
  <binding client="this.//lifecycle-controller"
  server="LC.//lifecycle-controller"/>
  <binding client="this.//binding-controller"
  server="BC.//binding-controller"/>
  <binding client="this.//super-controller" server="SC.//super-
  controller"/>

  <!-- Liaisons entre composants de contrôle -->
  <binding client="BC.//component" server="Comp.//component"/>
  <binding client="LC.//binding-controller"
  server="//BC.binding-controller"/>
  <binding client="LC.//component" server="Comp.//component"/>

  <controller desc="mComposite"/>
</definition>
```

Intégration des contrôleurs à l'aide d'aspects

Les modèles de composants comme EJB ou CCM fournissent des environnements dans lesquels les composants sont hébergés par des conteneurs fournissant des services techniques. Par exemple, les spécifications EJB définissent des services de sécurité, persistance, transaction et

de gestion de cycle de vie. La plupart du temps, cet ensemble de services est fermé et codé en dur dans les conteneurs. Une exception notable est le serveur J2EE JBoss dans lequel les services peuvent être accédés via des aspects définis à l'aide du canevas logiciel JBoss AOP. De nouveaux services peuvent être définis qui seront alors accédés à l'aide de leurs aspects associés.

L'idée générale illustrée par le serveur JBoss est que les aspects, tout en fournissant un mécanisme pour *modulariser* les fonctionnalités transverses, permettent également d'intégrer de façon harmonieuse de nouveaux services dans les applications. Cela illustre également une des bonnes pratiques de la programmation orientée aspect : il est conseillé de ne pas implémenter directement les fonctionnalités transverses dans les aspects, mais d'y implémenter simplement la logique d'intégration (c.-à-d. comment la fonctionnalité interagit avec le reste de l'application) et de déléguer la réalisation concrète de la fonctionnalité à un ou plusieurs objets externes. On obtient ainsi une séparation des préoccupations quasi optimale entre la logique d'intégration et celle du service à intégrer.

Cette pratique est mise en œuvre dans AOKell : chaque contrôleur est associé à un aspect chargé de l'intégration de la logique du contrôleur dans le composant. La logique d'intégration repose sur deux mécanismes : l'injection de code et la modification de comportement. Le premier mécanisme est connu dans AspectJ, un langage pour la programmation par aspects assez connu, sous la dénomination de déclaration inter-type (en anglais ITD pour I*nter-Type Declaration*). Avec ce mécanisme, les aspects peuvent déclarer des éléments de code (méthodes ou attributs) qui étendent la définition de classes existantes (d'où le terme ITD car les aspects sont des types qui déclarent des éléments pour le compte d'autres types, c.-à-d. des classes). Dans le cas d'AOKell, les méthodes des interfaces de contrôle sont injectées dans les classes implémentant les composants. Le code injecté est constitué d'une souche qui délègue le traitement à l'objet implémentant le contrôleur.

Le second mécanisme, la modification de comportement, correspond en AspectJ à la notion de code dit *advice*. Ainsi, la définition d'un aspect est constituée de *coupes* et de code *advice*. Les coupes sélectionnent un ensemble de points de jonction qui sont des points dans le flot d'exécution du programme au tour desquels l'aspect doit être appliqué. Les blocs de code *advice* sont alors exécutés autour de ces points. Le code advice est utilisé dans AOKell pour intercepter les appels et les exécutions des opérations des composants. Par exemple, le contrôleur de cycle de vie peut rejeter des invocations tant qu'un composant n'a pas été démarré.

Avec les mécanismes d'injection de code et de modification de comportement, les aspects intègrent de nouvelles fonctionnalités dans les composants et contrôlent leur exécution. La réalisation concrète de la logique de contrôle est déléguée par l'aspect au contrôleur implémenté sous la forme d'un objet.

3.4.3 Autres plates-formes

Au-delà de Julia et de AOKell, plusieurs autres plates-formes de développement Fractal existent. On peut citer en particulier, THINK pour le langage C, ProActive pour le langage Java, FracTalk pour SmallTalk, Plasma pour C++ et FractNet pour les langages de la plate-forme .NET.

THINK est une implémentation de Fractal pour le langage C. Elle vise plus particulièrement le développement de noyaux de systèmes qu'ils soient conventionnels ou embarqués. THINK est associé à KORTEX qui est une librairie de composants système pour la gestion de la mémoire, des activités (processus et processus légers) et de leur ordonnancement, des systèmes de fichiers ou des contrôleurs matériels (IDE, Ethernet, carte graphique, etc.). Les architectures matérielles les plus courantes comme celles à base de processeurs PowerPC, ARM ou x86 sont supportées. Le développement d'applications avec THINK passe par l'utilisation d'un langage pour la définition des interfaces (IDL) des composants et un langage pour la description des assemblages de composants (ADL). Dans le cadre des travaux en cours sur THINK v3, l'ADL est en cours de convergence vers Fractal ADL (voir section 3.3).

ProActive est une implémentation de Fractal pour le langage Java. Elle vise plus particulièrement le développement de composants pour les applications s'exécutant sur les grilles de calcul. ProActive est basée sur la notion d'objet/composant actif. Chaque composant est muni d'un ensemble de *threads* qui lui sont propres et qui gèrent l'activité de ce composant. Une deuxième caractéristique originale des composants Fractal/ProActive réside dans leur sémantique de communication : un mécanisme d'invocation de méthode asynchrone est utilisé. Cela permet d'obtenir de façon transparente des interactions non bloquantes dans lesquelles les composants client poursuivent leurs traitements pendant l'exécution des services invoqués.

Finalement, trois autres implémentations du modèle Fractal existent : FracTalk pour SmallTalk, Plasma pour C++ et FractNet pour les langages de la plate-forme .NET. Cette dernière est basée sur AOKell.

3.5 Bibliothèques

Cette section présente quelques bibliothèques majeures existantes à ce jour pour le développement d'applications à base de composants Fractal. Plusieurs bibliothèques de composants Fractal ont été développées. Il s'agit dans plupart des cas de fournir des briques de base pour la construction d'intergiciels.

Parmi les bibliothèques existantes nous pouvons citer :

- Dream, une bibliothèque de composants dédiée à la construction d'intergiciels orientés messages dynamiquement configurables plus ou moins complexes : de simples files de messages distribuées à des systèmes publication/abonnement complexes.
- Le projet Perseus définit une bibliothèque de composants Fractal pour la construction de services de persistance de données. Il fournit des composants de base pour la gestion de

cache, de réserves (pools), de politiques de concurrence (mutex, lecteurs/écrivain FIFO, lecteurs/écrivain avec priorité aux écrivains, optimiste), de journalisation et de gestion de dépendances.

- Le projet GoTM est une bibliothèque de composants Fractal pour la construction de moniteurs transactionnels. Il permet de construire des moniteurs transactionnels conformes à différents standards (par exemple JTS, OTS ou WSAT) et supportant différentes formes de protocoles de validation.

3.6 Comparaison avec les autres modèles

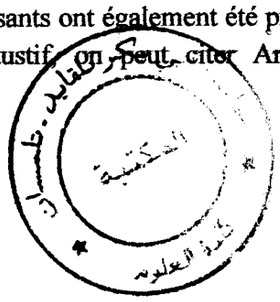
De nombreux modèles de composants ont été proposés ces dix dernières années. Dans le domaine des intergiciels, les propositions peuvent être classées selon qu'elles sont issues d'initiatives industrielles, de la communauté du logiciel libre ou d'équipes de recherche académiques.

Initiatives industrielles La première catégorie comprend les modèles issus d'initiatives industrielles comme EJB, COM+/.NET, CCM ou OSGi. Les caractéristiques de ces modèles varient, allant de composants avec des propriétés d'introspection (COM+), à des composants avec liaisons et cycle de vie (OSGi et CCM). Le modèle Fractal est, quant à lui, entièrement réflexif et introspectable, autorise des liaisons selon différentes sémantiques de communication et fournit un modèle hiérarchique autorisant le partage. Par ailleurs, EJB, CCM et COM+/.NET sont tous les trois accompagnés d'un modèle figé de services techniques offerts par des conteneurs aux composants. Par contraste, Fractal est basé sur un modèle ouvert, dans lequel les services techniques sont entièrement programmables via la notion de contrôleur.

Plus récemment, sous l'impulsion d'IBM, l'initiative SCA a défini un modèle de composants pour des architectures orientées services. Le projet Tuscany fournit une implémentation en Java et en C++ de ces spécifications. SCA propose la notion de liaison pour l'assemblage et de module pour la création de hiérarchies de composants. SCA n'impose pas une forme prédéterminée de liaison, mais autorise l'utilisation de différentes technologies, comme SOAP, JMS ou IIOP pour mettre en œuvre ces liaisons. De même, Fractal autorise différents types de liaisons et n'impose pas de technologie particulière.

Initiatives du monde du logiciel libre Dans la catégorie des modèles de composants issus d'initiatives de type logiciel libre, nous pouvons notamment citer Avalon qui est un modèle de composants général, Kilim, Pico et Hivemind qui ciblent la configuration de logiciel, Spring, Carbon et Plexus qui ciblent les conteneurs de composants de type EJB. De manière générale, ces modèles sont moins ouverts et extensibles que ne l'est Fractal.

Initiatives académiques Plusieurs modèles de composants ont également été proposés par des équipes de recherche académiques. Sans être exhaustif, on peut citer ArchJava, FuseJ, KComponent, OpenCOM v1 et v2.



3.7 Conclusion

Ce chapitre a présenté le modèle de composants Fractal, les principales plates-formes le mettant en œuvre, le langage de description d'architecture et les bibliothèques permettant de développer des systèmes à base de composants Fractal.

La section 3.2 est consacrée à la présentation du modèle de composant Fractal. C'est un modèle hiérarchique au sens où les composants peuvent être soit primitif, soit composite et contenir des sous-composants (primitifs ou composites). Deux parties sont mises en avant dans un composant Fractal : le contenu et la membrane. Cette dernière fournit un niveau méta de contrôle et de supervision du contenu. Elle est composée d'entités élémentaires, les contrôleurs, qui implémentent des interfaces dites de contrôle. Le modèle de composant Fractal est ouvert au sens où il ne présuppose pas un ensemble fini et figé de contrôleurs : de nouveaux contrôleurs peuvent être ajoutés par les développeurs en fonction des besoins.

L'API Fractal permet de construire et d'assembler des composants élémentaires afin de construire des applications complètes. Le langage de description d'architecture Fractal ADL (voir section 3.3) permet de décrire ces architectures de façon plus concise qu'avec une simple API. Fractal ADL est un langage ouvert basé sur XML. Contrairement à nombre d'ADL existants, la DTD de Fractal ADL n'est pas figée et peut être étendue avec de nouvelles balises permettant d'associer des caractéristiques supplémentaires aux composants.

Le modèle Fractal est indépendant des langages de programmation. La section 3.4 a présenté deux plates-formes, Julia et AOKell, mettant en œuvre ce modèle pour le langage Java. D'autres plates-formes existent pour les langages SmallTalk, C, C++ et les langages de la plate-forme .NET. Julia est la plate-forme de référence du modèle Fractal. AOKell a été développé par la suite et apporte une approche à base de composants de contrôle pour le développement des membranes. AOKell utilise en plus des techniques issues de la programmation orientée aspect pour l'intégration des niveaux de base et de contrôle.

Plusieurs bibliothèques de composants sont disponibles pour faciliter la tâche des développeurs Fractal. Nous en avons présentées quelques unes dans la section 3.5, dont Dream, qui permet de développer des intergiciels. Nous aurions pu également mentionner les outils qui existent autour de Fractal, comme Fractal GUI qui permet de concevoir graphiquement une architecture de composants et de générer des squelettes de code, Fractal Explorer qui est une console graphique d'administration d'applications Fractal, Fractal RMI qui permet de construire des assemblages de composants distribués communicants via un mécanisme d'invocation de méthodes distance.

Après une première version stable diffusée en juillet 2002, la version 2 des spécifications Fractal parue en septembre 2003 a permis d'en consolider l'assise. Fractal est maintenant une spécification stable et mature grâce à laquelle de nombreux systèmes et applications ont pu être développés. Par ailleurs, de nombreuses activités de recherches sont conduites autour de Fractal. Au-delà de leur intérêt propre, ces travaux devraient également servir de terreau pour compléter

Fractal sur plusieurs points qui sont actuellement peu pris en compte comme le paquetage (packaging), le déploiement, la sémantique des interfaces collection ou les modèles de composants pour les architectures orientées service.

Chapitre 4
Exemple d'application Fractal

4.1 Présentation de l'application

Ce chapitre montre un exemple d'usage d'une plate-forme de niveau 3.3 (voir section 3.25), afin d'illustrer comment les API que nous avons définies peuvent être définies pour créer, assembler et reconfigurer des configurations de composants. Nous avons choisi dans notre exemple la plateforme Julia. L'exemple en question est une simple application composée de deux composants primitifs imbriqués dans un composant composite (voir Figure 4.1). Le premier composant primitif, que nous désignerons par composant serveur, celui à droite, fournit une interface fonctionnelle serveur `s` de type `Service` et une interface de contrôle d'attributs `ServiceAttributes`. L'autre composant primitif, que nous désignerons par composant client, fournit une interface serveur `m` de type `Main` exportée par le composant composite et une autre interface client `s` de type `Service` liée à l'interface `s` du composant serveur.

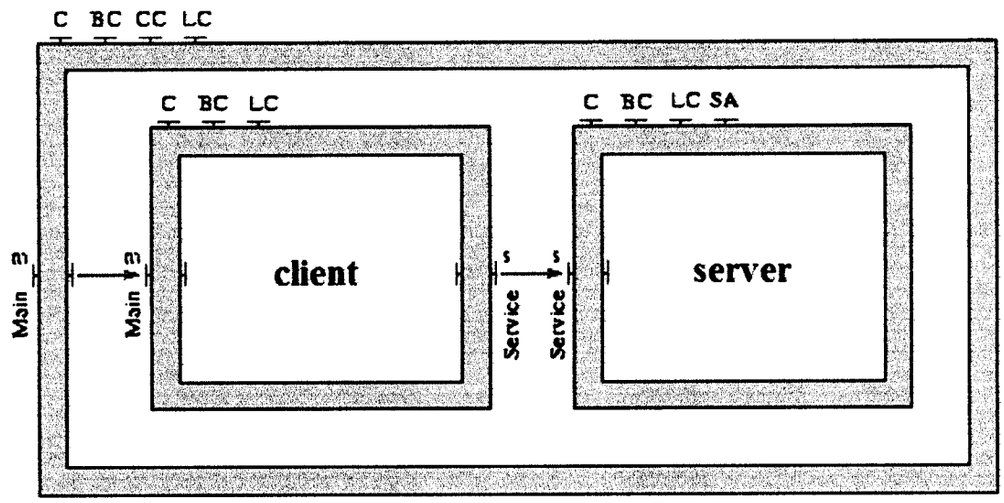


Figure 4.1-Architecture de l'application

Le fonctionnement de l'application est le suivant : à travers l'interface m du composant racine, un utilisateur peut demander un affichage de message, le composant composite délègue alors l'appel au composant client, le composant client traite l'appel en invoquant sur l'interface s du composant serveur l'opération qui finalement affichera le message.

L'exemple, aussi simple soit-il, illustre parfaitement la démarche à suivre pour la création de n'importe quelle application Fractal.

Nous expliquons dans la section 4.2 le choix de notre environnement de développement, la section 4.3 explique de façon détaillée les étapes de l'implantation qui se fait d'abord avec l'API Fractal dans la section 4.3.1 et ensuite avec Fractal ADL dans la section 4.3.2

4.2 Environnement de développement

Nous aurions bien pu éditer notre exemple avec un simple éditeur de texte, le compiler et l'exécuter directement en ligne de commandes avec JDK. Mais nous avons préféré utiliser un environnement de développement doté de fonctions d'assistance pour la construction d'applications, que ce soit pour l'édition du programme ou la configuration de l'application, et permettant ainsi d'augmenter de façon considérable la productivité du développeur. Notre choix s'est porté sur l'environnement Borland JBuilder 2006 à cause de la convivialité offerte par cet environnement (voir figure 4.2).

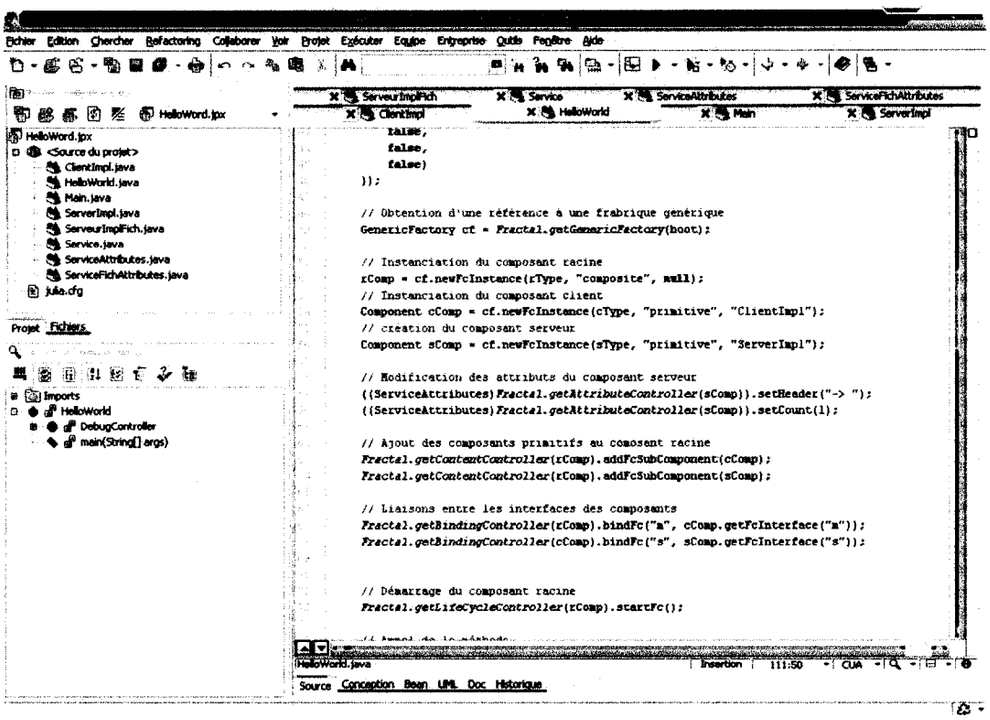


Figure 4.2-Interface de l'environnement de développement Borland JBuilder 2006

4.3 Implémentation de l'application

Nous allons utiliser deux méthodes pour la construction de l'application, d'abord la méthode faisant directement usage de l'API Fractal, dans la section 4.3.1 et ensuite celle basée sur l'utilisation de Fractal ADL dans la section 4.3.2.

4.3.1 Implémentation avec l'API Fractal

Dans cette section nous créons et assemblons nos composants en faisant directement usage de l'API Fractal. Dans la section 4.3.1.1 nous expliquons la démarche suivie pour la configuration et le lancement de notre application, la section 4.3.1.2 interprète le résultat de l'exécution suite à cette première configuration et dans la section 4.3.1.3 nous montrons comment nous avons procédé à la reconfiguration de l'application.

4.3.1.1 Configuration et lancement de l'application

Nous avons dit que c'est la plateforme Julia que nous allons utiliser pour notre application. Nous avons pu télécharger Julia à partir du site d'ObjectWeb[3]. Julia est définie comme une bibliothèque de classes. Comme nous avons réalisé l'application sous JBuilder, il nous a fallu ajouter la bibliothèque de Julia à notre projet pour pouvoir l'utiliser (voir Figure 4.3). Nous aurions pu à vrai dire retarder l'ajout de la bibliothèque de Julia jusqu'aux étapes de compilation et d'exécution, mais le fait de le faire maintenant nous permet de bénéficier de la fonctionnalité d'audit de code de JBuilder, cette fonctionnalité permet d'assister le développeur dans l'écriture du code, en lui signalant par exemple des incohérences ou en l'aidant dans la recherche de la classe ou de la méthode appropriée suivant le contexte courant. Nous pouvions passer en ce moment à l'implémentation proprement dite.

La création de toute application Fractal à base de composants de niveau 1 ou supérieur passe par les étapes suivantes : création des types de composant à partir des types d'interface, instanciation des composants à partir de types des composant, éventuelles configurations individuelles des composants, liaisons des composants, éventuelles imbrications de composants et enfin démarrage des composants. Nous allons suivre les mêmes étapes pour notre application.

Considérez que toutes les instructions que nous allons commentées dans cette section et dont la classe de provenance ne sera pas spécifiée feront partie de la méthode `main` (principale) de la classe `HelloWord`, la classe de lancement de l'application.

La première étape, c'est donc la création des types de composant. Pour cela, nous obtenons une référence au *bootstrap component*, et en suite une référence à son interface `TypeFactory`:

[3] fractal.objectweb.org

```
// Obtention d'une  référence au bootstrap component
Component boot = Fractal.getBootstrapComponent();
// Obtention d'une  référence à une fabrique de type
TypeFactory tf = Fractal.getTypeFactory(boot);
```

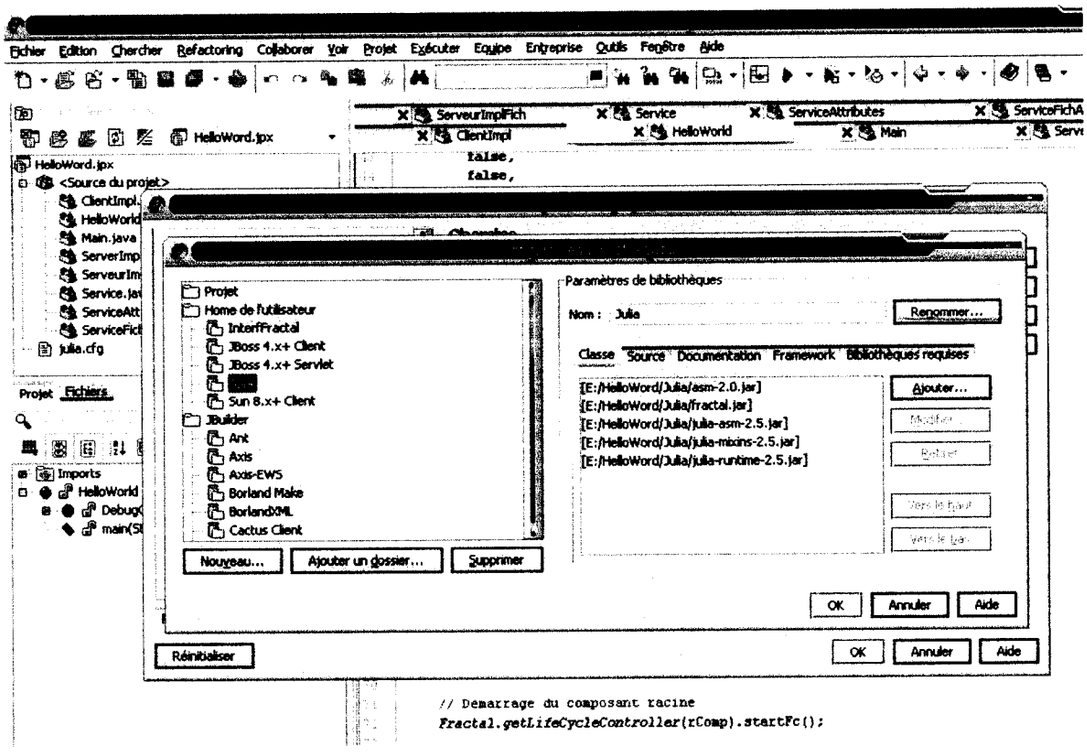


Figure 4.3-Configuration d'une nouvelle bibliothèque pour l'intégration de Julia au projet

Nous pouvons à present créer les types des composants racine, client et serveur comme suit:

```
29
30      // Type du composant racine
31      ComponentType rType = tf.createFcType(new InterfaceType[] {
32        tf.createFcItfType("m", "Main", false, false, false)
33      });
34      // Type du composant client
35      ComponentType cType = tf.createFcType(new InterfaceType[] {
36        tf.createFcItfType("m", "Main", false, false, false),
37        tf.createFcItfType("s", "Service", true, false, false)
38      });
39      // Type du composant serveur
40      ComponentType sType = tf.createFcType(new InterfaceType[] {
41        tf.createFcItfType("s", "Service", false, false, false),
42        tf.createFcItfType(
43          "attribute-controller",
44          "ServiceAttributes",
45          false,
46          false,
47          false)
48      });
```

Nous rappelons que l'opération de création de type d'interface prend comme paramètres, et dans cet ordre, le nom de l'interface, le type de l'interface, le type client ou serveur (la valeur faux pour serveur), la contingence (la valeur faux pour obligatoire) et la cardinalité (la valeur faux pour singleton). Le code ci-dessus permettra de doter le composant racine d'une interface fonctionnelle serveur `m` de type `Main`, le composant client d'une interface fonctionnelle serveur `m` de type `Main` et d'une interface fonctionnelle client `s` de type `Service`, et le composant serveur d'une interface fonctionnelle serveur `s` de type `Service` et d'une interface de contrôle d'attributs de type `ServiceAttributes`.

Nous pouvons à présent créer les composants, pour cela nous devons obtenir une référence à une fabrique générique:

```
49
50      // Obtention d'une référence à une frabrique générique
51      GenericFactory cf = Fractal.getGenericFactory(boot);
52
```

Nous procédons enfin à l'instanciation des composants :

```
53      // Instanciation du composant racine
54      Component rComp = cf.newFcInstance(rType, "composite", null);
55      // Instanciation du composant client
56      Component cComp = cf.newFcInstance(cType, "primitive", "ClientImpl");
57      // création du composant serveur
58      Component sComp = cf.newFcInstance(sType, "primitive", "ServerImpl");
59
```

Le premier paramètre de la méthode d'instanciation est le type du composant, le deuxième est le descripteur de la membrane du composant et le dernier est l'objet d'implémentation des interfaces fonctionnelles. Le composant composite se limitant à déléguer ses appels au composant client, nous pouvons remarquer qu'il n'a pas d'objet d'implémentation.

Les chaînes « composite » et « primitive » ont déjà été rencontrées dans le chapitre précédent, elles correspondent à des descripteurs utilisés par Julia pour configurer la membrane des composants. La première chaîne décrit la membrane d'un composant composite et la seconde celle d'un composant primitif. Ces descripteurs sont définis dans un ou plusieurs fichiers de configuration dont il faut indiquer le(s) chemin(s) à Julia pour qu'elle puisse appliquer les définitions correspondantes. Cela se fait en spécifiant la valeur d'une propriété système définie par Julia, soit au lancement de l'application comme paramètre de la machine virtuelle ou à l'exécution, mais dans ce dernier cas indiquer à tout prix la valeur de la propriété système avant toute utilisation de l'API Fractal au risque de ne pouvoir lancer Julia. Nous avons spécifié la propriété système comme suit :

```
17
18      System.setProperty("julia.config","conf/julia.cfg");
19
20
```

Nous interrompons l'explication des étapes de configuration pour nous attarder un instant sur les classes d'implémentation des composants client et serveur. Nous commençons par la classe d'implémentation du composant client, nommée `ClientImpl`.

La classe implémente pour le composant client son interface fonctionnelle serveur `Main` qui est définie comme suit :

```
1
2  public interface Main {
3      void main (String[] args);
4  }
```

Comme le composant doit être lié au composant serveur, la classe implémente en plus l'interface de contrôle `BindingController`, son implémentation est la suivante :

```
import org.objectweb.fractal.api.control.BindingController;

public class ClientImpl implements Main, BindingController {

  private Service service;

  public void main (final String[] args) {
    service.print("hello world");
  }

  public String[] listFc () {
    return new String[] { "s" };
  }

  public Object lookupFc (final String cItf) {
    if (cItf.equals("s")) {
      return service;
    }
    return null;
  }

  public void bindFc (final String cItf, final Object sItf) {
    if (cItf.equals("s")) {
      service = (Service)sItf;
    }
  }

  public void unbindFc (final String cItf) {
    if (cItf.equals("s")) {
      service = null;
    }
  }
}
```

La classe `ServerImpl` quant à elle implémente pour le composant serveur l'interface fonctionnelle `Service` et l'interface de contrôle `ServiceAttributes` définies comme suit :

```
public interface Service {
  void print (String msg);
}
```

```
import org.objectweb.fractal.api.control.AttributeController;

public interface ServiceAttributes extends AttributeController {
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
}
```

La classe est implémentée comme suit :

```
public class ServerImpl implements Service, ServiceAttributes {

    private String header = "";

    private int count = 0;

    public void print (final String msg) {
      new Exception() {
        public String toString () {
          return "Serveur: appel de la méthode d'écriture";
        }
      }.printStackTrace();
      System.err.println("Serveur: début de l'écriture...");
      for (int i = 0; i < count; ++i) {
        System.err.println(header + msg);
      }
      System.err.println("Serveur: écriture terminée ");
    }

    public String getHeader () {
      return header;
    }

    public void setHeader (final String header) {
      this.header = header;
    }

    public int getCount () {
      return count;
    }

    public void setCount (final int count) {
      this.count = count;
    }
  }
```

Nous fermons la parenthèse des classes d'implémentation et continuons avec la configuration de notre application. La prochaine action consiste à configurer les attributs du composant serveur :

```
    // Modification des attributs du composant serveur
    ((ServiceAttributes)Fractal.getAttributeController(sComp)).setHeader("-> ");
    ((ServiceAttributes)Fractal.getAttributeController(sComp)).setCount(1);
```

La première instruction ci-dessus fixe la valeur de l'attribut qui détermine l'en-tête des messages et la seconde fixe la valeur de l'attribut qui détermine le nombre d'affichage d'un message.

Nous allons maintenant ajouter les composants primitifs au composant composite :

```
    // Ajout des composants primitifs au comosant racine
    Fractal.getContentController(rComp).addFcSubComponent(cComp);
    Fractal.getContentController(rComp).addFcSubComponent(sComp);
```

Nous lions d'abord l'interface m du composite à celle du composant client et ensuite l'interface s du client à celui du composant serveur :

```
71      // Liaisons entre les interfaces des composants
72      Fractal.getBindingController(rComp).bindFc("m", cComp.getFcInterface("m"));
73      Fractal.getBindingController(cComp).bindFc("s", sComp.getFcInterface("s"));
74
```

Nous démarrons l'ensemble des composants en démarrant le composant composite :

```
76      // Démarrage du composant racine
77      Fractal.getLifeCycleController(rComp).startFc();
78
```

Nous somme maintenant en mesure d'effectuer des appels sur les interfaces fonctionnelles de notre application :

```
79      // Appel de la méthode
80      ((Main)rComp.getFcInterface("m")).main(null);
81
```

En réalité, nous ne pourrons pas exécuter notre application si nous n'effectuons pas encore deux actions : l'indication de la plateforme d'implémentation, et comme dans notre cas c'est Julia, il faut en plus indiquer à cette dernière la valeur de la propriété spécifiant la classe de chargement. Vous vous trompez si vous pensez que les API utilisées provenaient de Julia, toutes les API utilisées dans cet exemple sont les API provenant des packages de la spécification Fractal, distincts des packages de Julia. Le lien entre cette API et celle des plateformes se fait suivant le design pattern *brigde* : les méthodes implémentées dans les packages Fractal ne se réfèrent qu'à des interfaces devant être fournies par les plateformes, si bien que s'il est possible de compiler l'application en ne s'approvisionnant que des packages de la spécification Fractal, on se verra incapable de l'exécuter sans les implémentations réelles qui ne sont fournies que par les plateformes.

Voici donc l'instruction indiquant à l'API Fractal que nous choisissons comme plateforme Julia:

```
14
15      System.setProperty("fractal.provider","org.objectweb.fractal.julia.Julia");
16
17
```

Voici maintenant celle par laquelle nous indiquons à Julia le type de chargeur de classes :

```
17
18      System.setProperty("julia.loader","org.objectweb.fractal.julia.loader.DynamicLoader");
19
```

En définissant la valeur du chargeur de classes de cette façon, nous indiquons à Julia de générer dynamiquement et ensuite de charger les classe requises pour l'instanciation des

composants (nous avons vu que Julia générait beaucoup de classes pour l'instanciation d'un composant). Ce processus dynamique repose sur l'API de réflexion de Java qui n'est pas disponible sur toutes les JVM. Dans ce cas la solution consiste à générer les classes requises avant le lancement de l'application, et ensuite à indiquer au lancement de l'application le *classpath* des classes générées qui pourront alors être chargées par le chargeur de classes satique, à la place de l'instruction précédente on indiquerait alors plutôt :

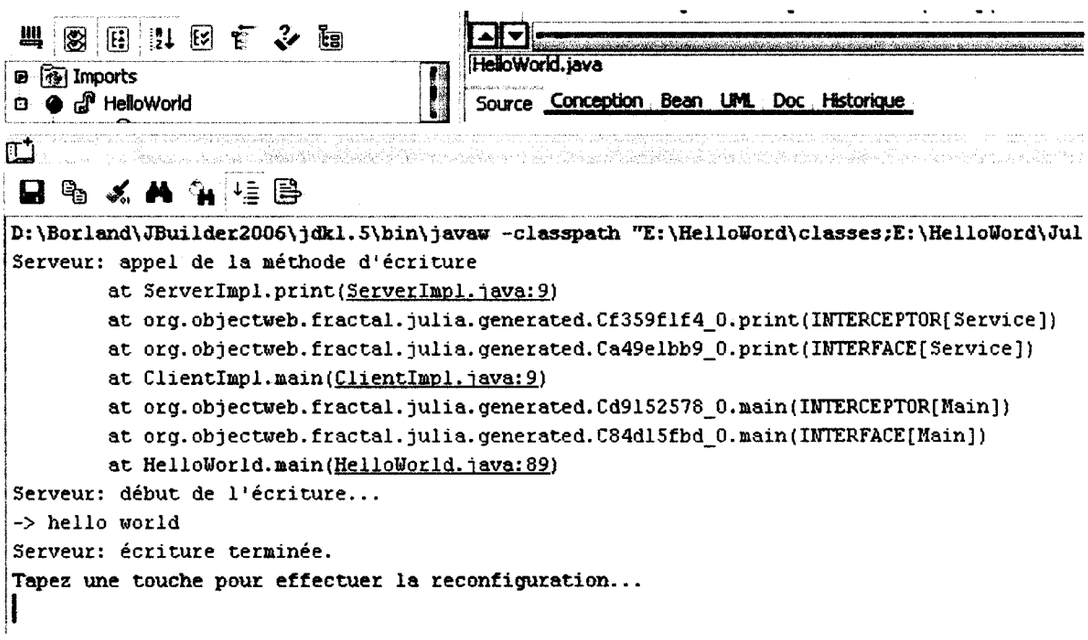
```
System.setProperty("julia.loader","org.objectweb.fractal.julia.loader.BasicLoader");
```

Mais les classes générées devraient auparavant avoir été générées par un chargeur dynamique, auquel on aurait indiqué le dossier de destination des classes générées en spécifiant la valeur d'une propriété système comme suit :

```
System.setProperty("julia.loader.gen.dir","class");
```

Notez que la valeur que nous avons affectée à la propriété n'est qu'un simple choix.

Après avoir ajouté ces dernières instructions, nous pouvons enfin exécuter l'exemple, le résultat est le suivant :



Le résultat est interprété dans la section suivante.

4.3.1.2 Interprétation de l'exécution

Revoyons encore l'implémentation de la méthode `print` de l'interface `Service` dans la classe `ServerImpl` :

```
8   public void print (final String msg) {
9     new Exception() {
10      public String toString () {
11        return "Serveur: appel de la méthode d'écriture";
12      }
13    }.printStackTrace();
14    System.err.println("Serveur: début de l'écriture...");
15    for (int i = 0; i < count; ++i) {
16      System.err.println(header + msg);
17    }
18    System.err.println("Serveur: écriture terminée.");
19  }
20
```

Au début de la méthode, nous lançons une exception et affichons la chaîne d'appels ayant conduit à l'exception, nous pourrons ainsi mettre en évidence les objets générés par Fractal et la façon dont ils ont collaboré pour fournir le résultat.

En nous référant au résultat de la section précédente, nous voyons que l'exception commence à se propager à partir de la méthode `print` de l'objet `ServerImpl`, cela se comprend tout à fait puisque c'est de là que nous avons lancé l'exception. Ensuite l'exception se propage à un intercepteur implémentant l'interface Service, il s'agit de l'intercepteur associé au composant serveur, cet intercepteur est requis pour la réalisation du service de gestion du cycle de vie. Le résultat nous indique que l'exception se propage ensuite à un objet d'interface implémentant l'interface `Service`, il s'agit de l'objet d'interface associé à l'interface fonctionnelle `s` du composant serveur. L'exception continue sa propagation et atteint cette fois l'objet `ClientImpl` dans sa méthode `main`, cela aussi se comprend si nous savons que c'est à partir de cette méthode que nous avons invoqué le service du composant serveur. L'exception continue toujours sa propagation et atteint cette fois l'intercepteur associé au composant client, cet intercepteur est requis pour la même raison que celle de la présence de l'intercepteur associé au composant serveur. L'exception se propage après à l'objet d'interface associé à l'interface fonctionnelle m du composant composite, en court-circuitant (voir section 3.4.1.4) l'objet d'interface associé à l'interface fonctionnelle du composant client. L'exception s'arrête enfin au niveau de la méthode `main` de la classe `HelloWord`, la classe où l'invocation initiale a été lancée.

Toujours en nous référant au résultat de l'exécution de la section précédente, nous constatons qu'après le lancement de l'execption, la méthode `print` procède à l'affichage du message et se termine.

Les indirections précédentes sont illustrées par la figure 4.4.

Vous constatez en observant le résultat de l'exécution à la section précédente qu'à la fin de l'exécution nous invitons l'utilisateur à lancer la phase de reconfiguration en tapant une touche. Nous expliquons dans la section suivante les étapes de cette phase de reconfiguration.

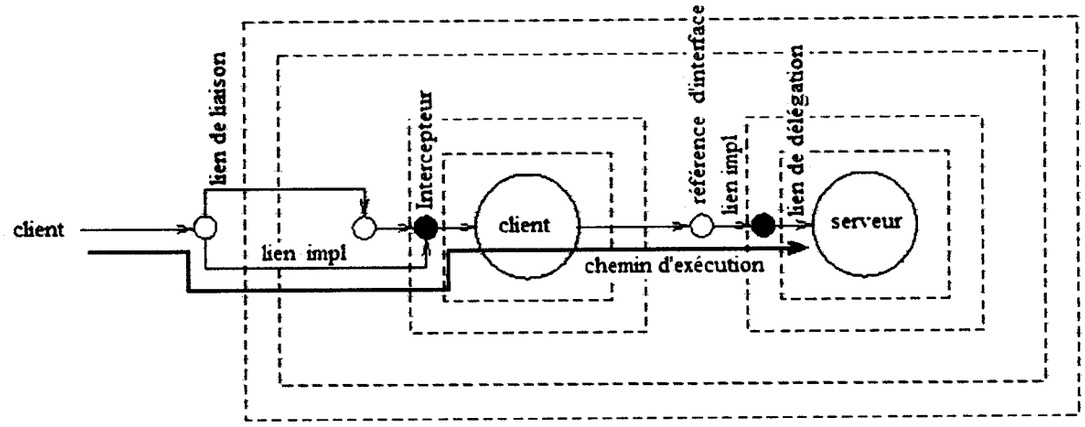


Figure 4.4-Chemin d'exécution de l'appel

4.3.1.2 Reconfiguration

Nous souhaitons maintenant procéder à la reconfiguration de notre configuration de composants afin de remplacer le composant serveur, en effet nous voulons disposer cette fois d'un composant serveur qui affiche le message non pas sur la sortie d'erreur mais dans un fichier dont on lui passera le chemin à travers un de ses attributs. Le nouveau composant sera donc doté d'une nouvelle interface de gestion d'attributs différente de la première, définie comme suit :

```
import org.objectweb.fractal.api.control.AttributeController;

public interface ServiceFichAttributes extends AttributeController {
    void setFilepath(String path);
    String getFilepath();
    void setHeader(String header);
    String getHeader();
}
```

La classe d'implémentation du nouveau composant est nommée `ServeurImplFich`, elle implémente l'opération `print` de l'interface `Service` comme suit :

```
public class ServeurImplFich implements Service, ServiceFichAttributes
{
    public void print(String msg) {
        try
        {
            o=new PrintWriter(new FileOutputStream(this.filePath));
            o.println(this.header+":"+msg);
            o.close();
            System.out.println("Le message a été déposé avec succès dans le fichier : "+
                               this.filePath);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
```

Pour procéder au remplacement du composant composite, nous devons déconnecter le composant client et le composant serveur, supprimer le composant serveur du contenu du composite, créer le nouveau composant serveur, l'ajouter au composant composite et lier le composant client au nouveau composant serveur. Mais ces opérations ne peuvent être effectuées qu'une fois les composants stoppés, nous commençons donc par stopper les composants en stoppant le composant composite :

```
101
102     // Arrêt des composants
103     Fractal.getLifeCycleController(rComp).stopFc();
104
```

Nous déconnectons le composant client du composant serveur :

```
105     // Déconnection du composant client et du composant serveur
106     Fractal.getBindingController(cComp).unbindFc("s");
```

Nous supprimons le composant serveur du composite :

```
108     // Suppréssion du composant serveur
109     Fractal.getContentController(rComp).removeFcSubComponent(sComp);
110
```

Nous créons le type du nouveau composant serveur, il ne diffère de l'ancien type que par l'interface de gestion des attributs :

```
110
111     // Création d'un nouveau composant serveur
112     sType = tf.createFcType(new InterfaceType[] {
113          tf.createFcItfType("s", "Service", false, false, false),
114          tf.createFcItfType(
115            "attribute-controller",
116            "ServiceFichAttributes",
117            false,
118            false,
119            false)
120     });
```

Nous instancions le nouveau composant serveur :

```
122
123     // Instanciation du nouveau composant serveur
124     sComp=cf.newFcInstance(sType,"primitive","ServeurImplFich");
```

Nous établissons le chemin du fichier de destination du message :

```
125
126     // Configuration des attributs du nouveau composant serveur
127     ((ServiceFichAttributes)Fractal.getAttributeController(sComp)).
128             setFilepath("Message File.txt");
```

Nous ajoutons le nouveau composant au composite :

```
129
130     // Ajout du nouveau composant serveur au composant racine
131     Fractal.getContentController(rComp).addFcSubComponent(sComp);
```

Nous lions le composant client au nouveau composant serveur :

```
132
133     // Liaison du composant client au  nouveau composant serveur
134     Fractal.getBindingController(cComp).bindFc("s",sComp.getFcInterface("s"));
135
```

Nous reprenons alors l'exécution de l'application :

```
76      // Démarrage du composant racine
77      Fractal.getLifeCycleController(rComp).startFc();
78
```

Nous pouvons à nouveau invoquer les opérations des interfaces fonctionnelles de nos composants :

```
140
141     // Appel de la méthode m de l'interface Main
142     ((Main)rComp.getFcInterface("m")).main(null);
143
```

Ces sont donc ces instructions qui sont exécutées lorsque nous somme invités à tapez une touche afin de procéder à la reconfiguration. La nouvelle configuration permet de déposer le message dans un fichier. Voici le message affiché sur la console après l'invocation de la méthode `main` de l'interface `m` du composant composite après la reconfiguration:

```
D:\Borland\JBuilder2006\jdk1.5\bin\javaw -classpath "E:\HelloWord\classes;E:\Donnees\Cours
Serveur: appel de la méthode d'écriture
        at ServerImpl.print(ServerImpl.java:9)
        at org.objectweb.fractal.julia.generated.Cf359f1f4_0.print(INTERCEPTOR[Service])
        at org.objectweb.fractal.julia.generated.Ca49e1bb9_0.print(INTERFACE[Service])
        at ClientImpl.main(ClientImpl.java:9)
        at org.objectweb.fractal.julia.generated.Cd9152578_0.main(INTERCEPTOR[Main])
        at org.objectweb.fractal.julia.generated.C84d15fbd_0.main(INTERFACE[Main])
        at HelloWorld.main(HelloWorld.java:89)
Serveur: début de l'écriture...
-> hello world
Serveur: écriture terminée.
Tapez une touche pour effectuer la reconfiguration...

Le message a été déposé avec succès dans le fichier : Message File.txt
```

4.3.2 Implémentation avec Fractal ADL

Dans cette section, nous construisons l'architecture de notre application avec Fractal ADL. Si nous avons déjà fait ce travail dans les sections précédentes en utilisant directement l'API Fractal, nous allons voir que Fractal ADL est l'outil le mieux approprié pour la construction de l'architecture d'une application Fractal.

Nous allons aussi profiter pour introduire l'usage d'un outil important dans l'administration d'applications Fractal, il s'agit de Fractal Explorer.

La section 4.3.2.1 présente la construction de l'architecture de l'application avec Fractal ADL et le lancement de l'application à partir de cette définition d'architecture. Nous pourrons ainsi constater que nous obtenons les mêmes résultats que dans la section précédente mais avec moins d'effort. La section 4.3.2.2 explique comment exploiter cette définition d'architecture en utilisant Fractal Explorer.

4.3.2.1 Définition des architectures avec Fractal ADL

Nous avons choisi de définir les composants dans des fichiers séparés, le composant composite est donc défini en utilisant des références aux définitions des composants primitifs. Si ce choix rend l'architecture du composite moins lisible (pour comprendre l'architecture du composant, il faut consulter les définitions des composants référencés), il a l'avantage de procurer une plus grande souplesse dans la réutilisation des définitions de composants.

La définition du composant client est la suivante :

HelloWorldExplorer | ServeurImpl.fractal | ServeurImplFich | S
ClientImpl.fractal | HelloWorld | HelloWorldExplc

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="ClientImpl">
  <interface name="m" role="server" signature="Main"/>
  <interface name="s" role="client" signature="Service"/>
  <content class="ClientImpl"/>
</definition>
```

La définition est assez intuitive, nous définissons un composant nommé `ClientImpl`, doté d'une interface fonctionnelle serveur `m` de type `Main` et d'une interface fonctionnelle client `s` de type `Service` et ayant pour classe d'implémentation la classe `ClientImpl`. Nous remarquons que la cardinalité et la contingence des interfaces ne sont pas définies, par défaut elles sont à *singleton* et *mandatory* respectivement. Le composant ne comportant pas de composant imbriqué, il est automatiquement de type primitif, n'empêche que nous aurions pu le préciser quand même, comme dans la définition suivante.

La définition suivante est celle du composant serveur :

HelloWorldExplorer | ServeurImpl.fractal | ServeurImplFich | S

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="ServeurImpl">
  <interface name="s" role="server" signature="Service"/>
  <content class="ServerImpl"/>
  <attributes signature="ServiceAttributes">
    <attribute name="header" value="-> "/>
    <attribute name="count" value="1"/>
  </attributes>
  <controller desc="primitive"/>
</definition>

```

Nous définissons avec l'élément `attributes` l'interface de contrôle des attributs du composant, nous affectons aussi des valeurs à ces attributs toujours dans l'élément `attributes`.

Nous pouvons maintenant écrire la définition du composant composite :

HelloWord.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="HelloWorld">
  <interface name="m" role="server" signature="Main"/>
  <component name="client" definition="ClientImpl"/>
  <component name="serveur" definition="ServeurImpl"/>
  <binding client="this.m" server="client.m"/>
  <binding client="client.s" server="serveur.s"/>
</definition>
```

La définition dote le composant composite, en plus d'une interface fonctionnelle serveur `m` de type `Main`, de deux sous-composants correspondants aux définitions précédentes, elle réalise la liaison entre l'interface `m` du composite et l'interface `m` du sous-composant client et entre l'interface `s` du sous-composant client et l'interface `s` du sous-composant serveur.

Avec ces différentes définitions, nous pouvons instancier un composant composite semblable en tous les points au composite de la section précédente. L'instanciation se fait en utilisant une fabrique fournie par la bibliothèque de Fractal ADL, en réalité cette fabrique délègue les opérations à des fabriques concrètes, comme par exemple celles de Julia, pour cela il faudrait d'abord que Julia soit spécifiée comme plateforme cible et qu'elle soit en conséquence configurée pour être opérationnelle. Ce qui est fait comme suit (à noter que toutes les instructions qui vont suivre sont dans la méthode `main` (de lancement) d'une nouvelle classe que nous avons nommée `HelloWorldExplorer`) :

```
    System.setProperty("fractal.provider","org.objectweb.fractal.julia.Julia");
    System.setProperty("julia.loader",
                       "org.objectweb.fractal.julia.loader.DynamicLoader");
    System.setProperty("julia.config","conf/julia.cfg");
```

Ce sont les mêmes opérations effectuées dans la section précédente.

Nous obtenons ensuite une référence à une fabrique de Fractal ADL qui reposera sur les fabriques de Fractal (argument `FactoryFactory.FRACTAL_BACKEND`) :

```
    Factory f=FactoryFactory.getFactory(FactoryFactory.FRACTAL_BACKEND);
```

Puisque la fabrique de Fractal ADL reposera sur les fabriques d'une plateforme de Fractal, il faudra lui donner une information qui lui permettra d'exploiter cette plateforme. Fractal ADL permet en plus de déclarer dans les définitions ADL des arguments dont les valeurs peuvent être définies avant l'instanciation. La solution pour passer toutes ces informations à la fabrique consiste à lui passer au moment de l'instanciation une table de mappage les

contenant. En effet la fabrique en parcourant les clés de la table est susceptible de retrouver toutes les informations qui lui sont destinées. Par exemple n'ayant pas défini d'arguments dans nos définitions ADL, nous avons juste besoin de lui donner l'information lui permettant d'accéder aux Fabriques de Julia :

```
30      Map m=new HashMap();
31      m.put("bootstrap",Fractal.getBootstrapComponent());
```

A noter que la clé `bootstrap` est définie par Fractal ADL comme celle correspondant au *bootstrap component* de la plateforme cible.

Nous pouvons à présent instancier notre définition ADL en passant le nom du fichier la contenant et la table de mappage définie précédemment :

```
33      Component rComp=(Component)f.newComponent("HelloWorld",m);
34
```

Nous venons de créer un composant similaire en tous les points au composant de la section précédente.

4.3.2.2 Administration de notre application avec Fractal Explorer

Nous montrons dans cette section comment utiliser l'outil Fractal Explorer pour administrer notre application.

Fractal Explorer est une console graphique d'administration d'applications Fractal. Fractal Explorer permet de découvrir les composants d'une application, d'introspecter leurs interfaces et leurs contenus et de les reconfigurer à travers leurs contrôleurs. Fractal Explorer met également à notre disposition les fabriques de la plateforme sous-jacente permettant de construire de nouveaux types d'interfaces et de composants et d'instancier de nouveaux composants qui pourraient être liés ou ajoutés aux composants de l'application. La figure 4.5 montre l'interface de Fractal Explorer administrant notre application.

Fractal Explorer permet d'afficher les applications chargées suivant plusieurs niveaux de détail. Ces niveaux de détail peuvent être choisis à partir du menu *Roles* (voir Figure 4.4). Les différentes options de ce menu sont :

- la vue *service* : cette vue affiche les interfaces serveurs de l'application;
- la vue *user* : cette vue représente la vue d'un utilisateur de l'application, elle n'affiche que les interfaces fonctionnelles de l'application ;
- la vue *advanced* : cette vue affiche le composant de haut niveau de l'application et ses sous-composants ainsi que leurs interfaces fonctionnelles ;
- la vue *administrator* : comme les contrôleurs sont disponibles dans cette vue, elle permet à travers ces derniers d'effectuer des actions de reconfiguration, comme par

exemple l'ajout d'un composant à un composite, la modification des liaisons entre les composants ou encore le changement du nom d'un composant ;

- la vue *architect* : avec cette vue, il devient possible d'instancier une application, de créer des types de composant et de les instancier.

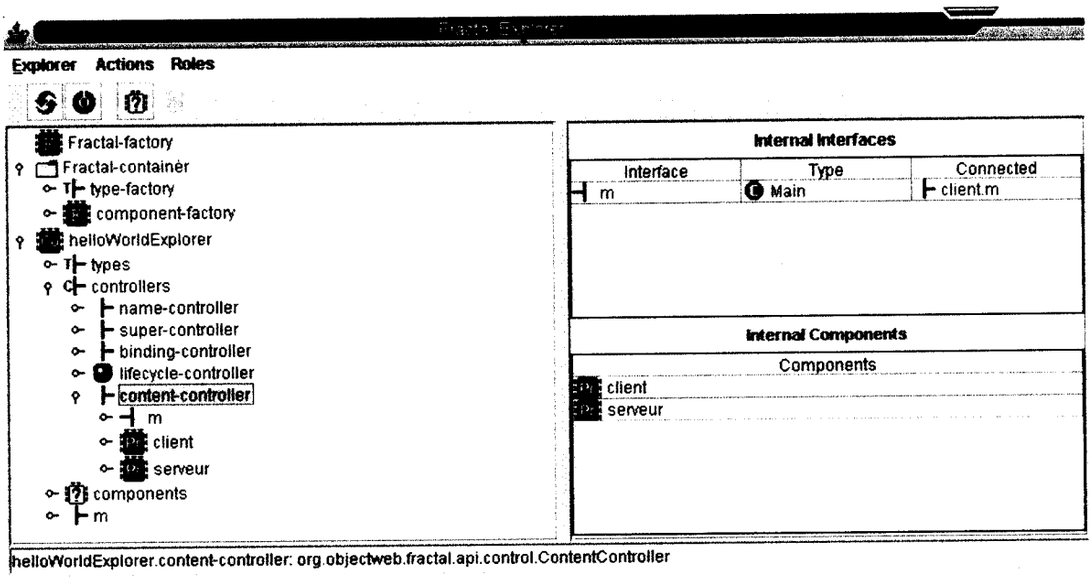


Figure 4.5-Fractal Explorer en train d'administrer l'application HelloWorld

Fractal Explorer est elle-même un composant Fractal ayant la structure suivante (voir Figure 4.6) :

- une interface client (appelée `fcAplI`) pour lier l'interface `Component` de l'application Fractal à Fractal Explorer ;
- un attribut (appelé `configFiles`) dont on peut modifier la valeur et qui contient la liste des chemins des fichiers de configuration destinés à Fractal Explorer elle-même, ces fichiers peuvent contenir par exemple des informations permettant de personnaliser l'affichage de Fractal Explorer. Mais il faut noter que Fractal Explorer peut être utilisé sans contrôleur d'attributs, c'est d'ailleurs ce que nous allons faire.

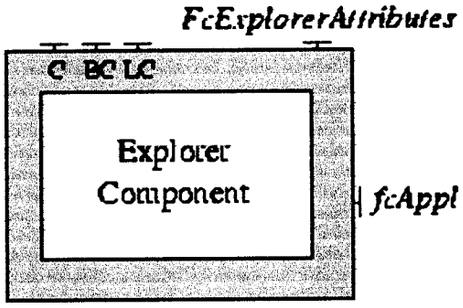


Figure 4.6-Structure de Fractal Explorer

Pou utiliser Fractal Explorer, il suffit d'écrire une définition ADL contenant un sous-composant référençant la définition d'un composant composite (`org.objectweb.fractal.explorer.BasicFractalExplorer`) auquel on passera en arguments le nom à afficher pour notre application une fois Fractal Explorer lancée et le chemin de la définition ADL de notre application. Le composant composite encapsulant déjà le composant Fractal Explorer, il va se servir des informations que l'on vient de lui fournir pour créer un second sous-composant correspondant à notre application et connectera finallement l'interface `fcApll` de Fractal Explorer à l'interface `Component` de notre application, et Fractal Explorer est lancé. Mais pour que tout cela fonctionne il faudrait évidemment d'abord instancier la première définition dont nous avons parlé. Pour notre application voilà comment elle est définie :

HelloWorldExplorer.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="HelloWorldExplorer">

  <component name="explorer"
    definition="org.objectweb.fractal.explorer.BasicFractalExplorer(helloWorldExplorer,HelloWorld)" />

</definition>
```

Nous n'avons qu'à instancier cette définition ADL comme nous l'avons fait dans la section précédente et Fractal Explorer sera lancée avec notre application comme sur la figure 4.5 :

```
        Component rComp=(Component)f.newComponent("HelloWorldExplorer",m);
```

Chapitre 5
Conclusion générale et perspectives

Ce mémoire a porté sur l'étude du modèle de composants Fractal, un intergiciel à base de composants. Nous avons d'abord du faire un tour d'horizon des systèmes répartis afin de saisir les spécificités propres à ces systèmes qui entraînèrent la naissance de la notion d'intergiciel. Nous avons ensuite interprété l'intergiciel comme le logiciel intermédiaire masquant la complexité de l'infrastructure sous-jacente à une application distribuée et lui fournissant un ensemble de services communs afin de rendre possible un développement de haut niveau focalisé sur les questions métier. Les systèmes intergiciels à objets repartis qui furent perçus avec espoir à un certain moment montrèrent leurs limites à cause du faible usage de la séparation des préoccupations dans le paradigme objet. En réponse, nous avons assisté à la naissance du concept de composant qui s'affranchissait des limites du modèle objet.

Orientés dans leurs débuts avec des modèles de composants comme EJB et CCM vers la construction des applications, les composants virent à partir de 2000 leur champ d'application s'étendre aux couches inférieures, c.-à-d. les intergiciels et les systèmes d'exploitation. Il s'agit toujours d'avoir des entités logicielles aux interfaces spécifiées contractuellement, mais il s'agit également d'avoir des plates-formes suffisamment légères pour ne pas perturber les performances du système.

Le modèle de composants Fractal répond à ces conditions. Il s'agit d'un modèle général dédié à la construction, au déploiement et à l'administration de systèmes logiciels complexes. Ces objectifs motivent les caractéristiques du modèle de composants : composants composites, composants partagés, capacités d'introspection et de reconfiguration. Nous avons vu que la spécification Fractal, contrairement à d'autres modèles de composants était extensible, permettant ainsi aux développeurs de configurer leurs composants suivant les ressources des environnements de déploiement.

Parmi les nombreuses implémentations du modèle, L'accent a été mis sur Julia et AOKell, deux implémentations de la spécification en Java. Pour modulariser les aspects de contrôle, Julia s'appuie sur un mécanisme de classes mixin. La classe résultant du mixage de plusieurs classes mixin est générée par un générateur de classe appelé générateur de classe mixin. AOKell par rapport à Julia apporte une componentisation de la membrane des composants et assure le lien entre les composants et les contrôleurs à l'aide des aspects. AOKell est donc un canevas qui fait usage de la notion de composant aussi bien pour la construction du niveau

métier que de la membrane. Les concepteurs d'AOKell espèrent en définissant explicitement les relations entre les contrôleurs favoriser la clarification de l'architecture de la membrane des composants et la réutilisation des contrôleurs. L'utilisation d'aspects pour lier composants et contrôleurs permet au niveau fonctionnel d'accéder directement à la membrane et permet d'aboutir à une séparation des préoccupations optimale entre logique d'intégration et contrôle à intégrer.

Fractal est accompagné d'un langage de description d'architecture, Fractal ADL. Bien qu'il soit possible de construire et lier les composants d'une application, il est clair qu'un langage de description d'architecture décrit de façon plus concise l'architecture de l'application. Puisqu'il est associé à une spécification extensible, Fractal ADL aussi est spécifié de manière à être extensible, pour pouvoir prendre en charge l'ajout de nouveaux contrôleurs.

De nombreuses bibliothèques de composants Fractal à l'heure actuelle sont disponibles, nous pouvons citer Dream pour la construction d'intergiciels orientés messages (les MOM, Middleware Oriented Message), la bibliothèque de composants définie par le projet Perseus pour la construction de services de persistance ou encore la bibliothèque de composants définie par le projet GoTM pour la construction de moniteurs transactionnels.

Fractal tient une place de choix parmi les autres modèles de composants à cause de son caractère réflexif, sa prise en charge des composants hiérarchiques et des composants partagés et surtout par-dessus tout à cause de son extensibilité.

Dans notre travail qui a porté essentiellement sur l'étude de la spécification Fractal, l'aspect application a été relégué au second plan. Nous nous sommes contentés de présenter un exemple d'application illustrant l'utilisation de l'API Fractal. Nous espérons donc que les promotions des années à venir compléteront notre travail avec la réalisation d'applications Fractal d'envergure exploitant les possibilités de Fractal que nous avons évoquées mais que nous n'avons pas mises en pratique. Et nous espérons que ces réalisations seront celles auxquelles Fractal est principalement destiné, c'est-à-dire celles des applications de bas niveaux : les intergiciels et les systèmes d'exploitation.

Liste des acronymes

ADL : Architecture Definition language

API : Application Programming Interface

COM : Component Object Model

CORBA : Common Object Request Broker Architecture

DCOM : Distributed Component Object Model

EJB : Enterprise JavaBean

HTML : Hyper Text Markup Language

IEEE : Institute of Electrical and Electronic Engineer

IIOP : Internet Inter-Orb Protocol

JAM : Java with Mixin

JTS : Java Transaction Service

OLE : Object Linking and Embedding

OMG : Object Management Group

OTAN : Organisation du Traité de l'Atlantique Nord

OTS : Object Transaction service

RPC : Remote Procedure Call

SCA : Service Component Architecture

SOAP : Simple Object Acces Protocol

SQL : Structured Query Language

XML : eXtended Markup Language

Liste des figures

Références bibliographiques

[1] Peschanski, F. & Briot, J.-P. – Architecture de composants répartis. In : Oussalah, M. ; 2005 –Ingénierie des composants. Ed. Vuibert Informatique. Paris. pp.247-279.

[1] Krakowiak, S. – Introduction à l'intergiciel. In : Krakowiak, S. ; Coupaye, T ; Quéma V. ; Seinturier, L. ; Stefani, J.-B ; Dumas, M. ; Fauvet, M.-C. ; Déchamboux, P. ; Riveill, M. ; Beugnard, A. ; Emsellem, D. & Donsez, D. ; 2007 – Intergiciel et Construction d'Applications Réparties **[en ligne]**. Disponible sur http://sardes.inrialpes.fr/ecole/livre/pub/Chapters/Fractal/fractal.html (consulté le 13/12/2007).

[3] Bieber, G & Carpenter, J. ; 2002 – Introduction to Service-Oriented Programming **[en ligne]**. Disponible sur http://www.openwings.org (consulté le 16/02/2008).

[4] Krakowiak, S. – Patrons et canevas pour l'intergiciel. In : Krakowiak, S. ; Coupaye, T ; Quéma V. ; Seinturier, L. ; Stefani, J.-B ; Dumas, M. ; Fauvet, M.-C. ; Déchamboux, P. ; Riveill, M. ; Beugnard, A. ; Emsellem, D. & Donsez, D. ; 2007 – Intergiciel et Construction d'Applications Réparties **[en ligne]**. Disponible sur http://sardes.inrialpes.fr/ecole/livre/pub/Chapters/Patterns/patterns.html (consulté le 13/12/2007).

[5] McIlroy, D. – Mass-produced software components. In : Proceeding of the 1st International Conference on Software Engineering, Garmisch Pattenkirschen, Germany, October 1968.

[6] Oussalah, M. ; Khammaci, T. & Smeda, A. – Les composants : concepts et définitions de base. In : Oussalah, M. ; 2005 –Ingénierie des composants. Ed. Vuibert Informatique. Paris. pp.1-19.

[7] Booch, G. ; Rumbaugh, J. & Jacobson, I. ; 1998 –The Unified Modeling language User Guide. Ed. Addison-Wesley. Massachusetts.

[8] Szyperski, C. ; 2002 – Beyond Object-Oriented Programming. 2ème édit. Ed. Addison-Wesley.

[9] Meyer, B. ; 1999 – On to components, Technical Report, IEEE Computer.

[10] Gamma, E., Helm, R., Johnson, R. & Vlissides, J.; 1995 – Design Patterns, Elements of Reusable Object-Oriented Software. Ed. Addison-Wesley.

[11] Krakowiak, S. ; 2006 – Patrons et canevas pour l'intergiciel. In École d'été sur les Intergiciels et sur la Construction d'Applications Réparties **[en ligne].** Disponible sur http://sardes.inrialpes.fr/ecole/2006/ (consulté le 17/03/2008).

[12] Smeda, A. ; Khammaci, T. & Oussalah, M. – Les modèles de composants académiques. In : Oussalah, M. ; 2005 –Ingénierie des composants. Ed. Vuibert Informatique. Paris. pp.45-86.

[13] Bruneton, E ; Coupaye, T & Stefani, J.B. ; 2004 – The Fractal Component Model **[en ligne]**. Disponible sur http://fractal.objectweb.org/specification/index.html (consulté le 13/12/2007).

[14] Seinturier, L. ; 2006 – Le système de composants Fractal. In: École d'été sur les Intergiciels et sur la Construction d'Applications Réparties **[en ligne].** Disponible sur http://sardes.inrialpes.fr/ecole/2006/ (consulté le 17/03/2008).

[15] Coupaye, T. ; Quéma, V. ; Seinturier, L. ; Stefani, J.-B. – Le système de composants Fractal. In : Krakowiak, S. ; Coupaye, T ; Quéma V. ; Seinturier, L. ; Stefani, J.-B ; Dumas, M. ; Fauvet, M.-C. ; Déchamboux, P. ; Riveill, M. ; Beugnard, A. ; Emsellem, D. & Donsez, D. ; 2007 – Intergiciel et Construction d'Applications Réparties **[en ligne]**. Disponible sur http://sardes.inrialpes.fr/ecole/livre/pub/ChaptersFractal/fractal.html (consulté le 13/12/2007).

[16] ObjectWeb – Document de mise en œuvre de Julia **[en ligne]**. Disponible sur http://fractal.objectweb.org/overview-summary0.html (consulté le 13/12/2007).

[17] Bruneton, E. ; Coupaye, T. ; Leclercq, M. ; Quéma, V. & Stefani, J.B. ; 2003 – The Fractal Component Model and Its Support in Java. In: SOFTWARE-PRACTICE AND EXPERIENCE **[en ligne]**. Disponible sur http://fractal.objectweb.org/model.html (consulté le 17/03/2008).