Stage de DEA RSD, mars à juin 2004

Florence Guitton

le 23 juin 2004

Table des matières

Introduction

Ce rapport présente le travail que j'ai réalisé au cours de mon stage de DEA Réseaux et Systèmes distribués au sein de l'équipe SmartTools de l'INRIA Sophia Antipolis.
Le sujet de ce stage est l'étude des technologies à base de composants du monde industriel et académique. Le point de départ de cette étude a été le modèle de composant de SmartTools. L'approche qui à été adoptée pour concevoir ce modèle de composant consiste dans un premier temps à définir un modèle abstrait puis dans un deuxième temps à l'implanter à l'aide d'une certaine technologie. L'objectif du stage a été, par une comparaison plus fine de ce modèle de composant avec les multiples modèles de composant, de proposer des pistes de recherche. Cette étude a permis de mieux mettre en valeur les qualités d'extension et de dynamisme du modèle de composant SmartTools.
La démarche a donc consisté à faire un état de l'art sur les modèles de composants en étudiant en particulier le modèle de SmartTools et la bibliographie sur le sujet.

Pour synthétiser cette étude j'ai défini un modèle abstrait (indépendant de toute technologie) qui m'a permis de clarifier le vocabulaire employé. J'ai obtenu un modèle minimal, unifiant les concepts communs aux différents modèles étudiés. J'ai alors cherché à prouver, à l'aide de transformations abstraites, que le modèle proposé contenait bien tous les concepts de base. Ainsi j'ai défini un modèle de transformations, modèle que j'ai appliqué pour obtenir des transformations informelles de notre modèle vers SmartTools, le CCM et les Web Services. L'approche que je viens de décrire est promue par l'approche MDA (concevoir un modèle indépendant et le projeter ensuite vers une plate-forme). L'étude a alors mis en évidence des propriétés intéressantes de SmartTools, masquées par la spécification des besoins ayant abouti à ce modèle. J'ai alors cherché à les rendre plus explicite et applicable à tous les modèles de composants. Mon travail s'est porté plus particulièrement sur l'assemblage dynamique des composants et l'extension des composants.

La première partie de ce document propose l'analyse des différents modèles de composant étudiés selon la taxonomie définie ci-dessous. Dans la deuxième partie je propose le modèle abstrait élaboré à partir de cette étude selon une progression chronologique. Tout d'abord le noyau du modèle et sa validation

par sa transformation dans des technologies existantes. Puis l'extension de ce modèle afin de prendre en compte les propriétés intéressantes de SmartTools.

Il n'existe pas une définition unique du composant logiciel, celle retenue par les participants au premier Workshop on Component Oriented Programing est la suivante: "A Software component is a unit of composition with contactualy specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to compostion by third parties." Dans un composant, le point important pour nous est la séparation entre les propriétés fonctionnelles et non fonctionnelles. Les propriétés fonctionnelles sont les services fournis par le composant, les propriétés non fonctionnelles désignant les propriétés liées à la façon dont sont implantées les propriétés fonctionnelles (performance, fiabilité, qualité de service...). Le composant encapsule donc la partie fonctionnelle (ou métier) tandis que son conteneur fournit la partie non fonctionnelle, gérant parfois le cycle de vie du composant. Bien souvent celà se traduit par une notion de "boîte noire", c'est à dire que ce qui se trouve dans le composant proprement dit est caché, seules ses interfaces sont visibles de l'extérieur.
Selon [31] un type de composant est la définition abstraite d'une entité logicielle. Il est caractérisé par trois éléments; ses interfaces (l'ensemble des méthodes du composant accessibles de l'extérieur), les modes de coopération avec les autres types de composants et ses propriétés configurables. Le type de composant sera aussi appelé modèle abstrait de composant. Pour pouvoir comparer plus aisément les modèles existant les auteurs proposent une taxonomie basée sur sept points:

1. Analyse des besoins en terme d'applicatif et de composant: exprimer les besoins de l'application en termes fonctionnels et spécifier les fonctionnalités au niveau du composant. C'est une phase importante puisqu'elle permet de décrire ce que l'on veut faire.
2. Conception des types de composants: Spécifier les types de composants dans le modèle à partir de l'expression des besoins. L'écriture des spécifications se fait à partir e ce qui a été identifié au cours de la phase d'analyse.
3. Implantation des types de composants: Elle doit se faire conformément au type de composant visé. Le développeur doit pouvoir se focaliser sur la valeur ajoutée du composant, la "logique métier". L'ensemble des implantations non-fonctionnelles est idéalement prise en charge et automatiquement générée à partir des différentes descriptions.
4. Diffusion des implantations de composants: Elle se fait à travers la mise à disposition du paquetage de l'implantation du composant. En général le paquetage contient l'implantation des composants et un descriptif de la manière de les assembler pour obtenir l'application souhaitée.
5. Assemblage des types de composants: La construction d'une application correspond à la réalisation d'un assemblage de composants, il reflète l'architecture de l'application. Il est souhaitable qu'un assemblage de composants soit un composant, et ainsi utilisable comme tel pour être assemblé

à d'autres composants. C'est la notion de composant composite.

6. Déploiement des implantations de composants et des applicatifs: Idéalement il est automatisé. Il nécessite la mise à disposition de structures d'accueil pour réceptionner les instances de composants.
7. Utilisation des instances de composants et des applicatifs: L'utilisation des instances de composants peut se faire de manière statique à travers des programmes clients prévus à cet effet, ou dynamiquement en découvrant et en exploitant à l'exécution les interfaces des instances de composants.

Nous nous servirons de cette taxonomie dans la suite de ce rapport afin de déterminer la place du modèle de composant de SmartTools par rapport aux autres technologies composant. Mais avant, voici un bref apperçu de quelques modèles de composants dont je ne développerais pas l'étude.
Pour le **modèle Durra** développé dans les années 90 les composants (ou tâches) sont les éléments de base d'une application. Chacun offre une interface constituée de ports. Ce sont des points d'entrée (API du composant) et de sortie (ses dépendances fonctionnelles). Les propriétés et les dépendances non fonctionnelles du composant sont décrites par les attributs. Les interactions entre tâches se font à travers des composants de communication, différents protocoles pouvant être développés. L'architecture de l'application est l'ensemble des composants et de la description des différentes configurations utilisées lors de l'exécution. Les composants sont instanciés à l'initialisation de l'application.
Dans **le modèle Darwin** le composant est une unité de traitement encapsulant du code applicatif; il est décrit par une interface contenant les services fournis et requis. Les unités de configuration sont des composants constitués d'autres composants. Ils contiennent la description des composants de l'application ainsi que leurs interactions. Ils coopèrent selon un modèle de communication connu par la signature des composants. Il existe une notion de pré-déclaration permettant de n'instancier les composants que lors de leur premier appel et non à l'initialisation de l'application.
Pour **Unicon** un composant est une entité d'encapsulation des fonctions d'un module logiciel ou des données. L'interface définie les fonctions et données accessibles et requises, mais aussi le mode d'accès et l'utilisation de ces fonctions. Une application est formée par un ensemble de composants.
Olan propose un modèle pour la construction d'applications réparties. Le composant encapsule le logiciel, le composant composite étant une unité de structuration. Les connecteurs spécifient le protocole de communication permettant la connexion de composants hétérogènes.
COM est la proposition de Microsoft en terme de composants logiciels. Le modèle est constitué du composant: une entité binaire (l'implantation du comportement des interfaces), des interfaces et d'un mode d'action. Les composants se connectent et se déconnectent de manière dynamique. DCOM offre la possibilité de créer des applications distribuées.
Le modèle .NET est proposé par Microsoft et permet au développeur d'écrire des applications logiques dans différents langages et de générer des composants. Ce sont des boîtes noires. Son objectif est de faciliter la conception d'applica-

tions et de services Web.
Le projet ASPECT s'intéresse à la partie IHM du composant. Les composants sont des composants d'un des modèles décrits précédemment. Ils sont définis par une interface métier (contenant l'ensemble des méthodes et des attributs fournis aux autres composants) et implémentés par une classe. Tout composant client d'un autre composant utilise nécessairement son interface. Il s'agit maintenant de lui ajouter une IHM, cette dernière étant considérée elle aussi comme un composant. Elles sont décrites dans un langage basé sur XML (SUNML) qui définit un ensemble d'éléments de base de l'IHM (bouton ...). L'IHM est construite en assemblant ces éléments de bases. Les instances des différents éléments sont assemblées sous la racine de l'IHM (l'OIA dans sa forme abstraite, l'OIC une fois projetée sur une plateforme). Plusieurs IHM sont possibles pour un composant, l'IHM et le composant métier peuvent être distribués sur des sites différents. Les relations entre le composant métier et le composant IHM peuvent être exprimées de manière statique ou dynamique.
Lorsque plusieurs composants métiers sont assemblés il est nécessaire de réaliser une référence dynamique entre leurs IHM. La fusion des éléments redondant des IHM est actuellement à l'étude.
[17] décrit **Apache Avalon**, un modèle de composant orienté serveur. Dans Avalon on a la séparation entre les interfaces (les rôles) et l'implantation (c'est une boîte noire implantant les services offerts par les interfaces), un système de "service registry" permettant la découverte des services et la génération des composants de l'extérieur (notion d'Inversion of Control). C'est le conteneur qui instancie le composant et l'enregistre auprès du service manager, ce dernier gérant son cycle de vie (les méthodes gérant le composant, l'ordre d'appel, les conditions d'appel). Un composant peut ne servir qu'un client à la fois ou plusieurs... Avalon est découpé en différents sous-projets, Framework qui définit le modèle de composant, Excalibur qui l'implémente fournissant notamment différents types de conteneurs, Phoenix permettant de gérer l'environnement des serveurs d'applications.

Dans le prochain chapitre nous allons appliquer la taxonomie définie précédemment à différents modèles de composants; SmartTools, le Corba Component Model, les EJB, les Web services et Fractal.

Chapitre 1

Etude des modèles de composant

1.1 Analyse des besoins

définir complètement un nouveau modèle permet d'être en adéquation parfaite avec les besoins particuliers du projet et de ne pas devoir entrer dans un moule de pensée (implicite avec tout modèle de composant). Mais, le gros handicap c'est qu'il est à la charge du projet de fournir tous les outils nécessaires à la production de composants ainsi que leur support d'exécution. SmartTools est un outil conçu pour répondre spécifiquement aux besoins métiers. Les EJB s'intéressent aux applications réparties orientées transactions; c'est un modèle de composant cotés serveur pour plateforme Java. Fractal s'attache à la reconfiguration dynamique des composants. Les services (Web Services) webs sont dédiés à l'internet. Les applications visées par les composants CORBA sont des applications réalisées à partir de composants hétérogènes distribués. Le modèle DCOM est particulièrement adapté aux plateformes Microsoft.

1.2 Les modèles de composants

Dans ce chapitre nous allons nous intéresser au deuxième point de la taxonomie citée dans l'état de l'art, la spécification du type de composant. Un composant est une entité logicielle fournissant des services, **la partie métier**. Le composant encapsule son état et ses données et n'est accessible de l'extérieur qu'à travers ses **interfaces** par l'intermédiaire de **ports**. Il est placé dans une structure, structure chargée de la partie non fonctionnelle de l'application: **le conteneur**. Cette séparation de la partie fonctionnelle et non fonctionnelle d'un composant est une notion primordiale. Dans ce chapitre nous allons étudier ces différents points dans l'approche SmartTools et dans les autres modèles de composants.

1.2.1 SmartTools

- La partie métier est une spécificité de SmartTools. Le coeur du composant contient le modèle (on parle de formalisme). Sur ce modèle sont appliqués des visiteurs qui effectuent les traitements demandés. Par contre le modèle est une notion optionnelle, il existe des composants dans SmartTools qui ne décrivent pas ce formalisme. Comme un composant "classique" leur partie métier est uniquement constituée de code.
- le conteneur gère les communications du composant avec l'extérieur, c'est à dire la découverte des services et la connexion des composants lorsqu'ils sont compatibles entre eux. Il gère les messages, effectue l'appel des services correspondant à chaque message. Il peut avoir trois états différents: ON, OFF ou SLEEP pendant lesquels on peut l'utiliser ou non. Sa première tache est de connecter le composant au générateur de composants. Un conteneur a une façade, un nom et un fichier de ressources (appelè cdml). Le CDML permet de décrire le composant. La particularité de SmartTools est que les composants décident eux même de se connecter et que c'est une connexion per à per.
- Lorsque le formalisme est présent dans le document des traducteures (parsers) lui sont associés.
- La façade: se situe à la frontière entre le composant et le conteneur. Elle connaît le contenu du composant et sait où diriger les requêtes des clients transmises par le conteneur.
- les ports: in permet d'appeler les méthodes du composant, out de recevoir des appels de l'extérieur, inout permet de préciser qu'il faut attendre le résultat.
- behavior: contient la description des actions sur les représentations des composants (la représentation du formalisme) et comment les utiliser. Ce fichier permet une extension des services offerts par le composant de type "vue" en fonction des besoins du composant logique qui lui est associé.
- Les langages utilisés pour décrire les composants: Lml est le langage permettant de décrire les vues attachées au composant. C'est à dire les composants capables de visualiser le composant métier. absynt est un langage de description du langage métier sous forme de syntaxe abstraite. Cdml, langage de description de déploiement. Il décrit le type de composant, les services qu'il offre, ses ports, la configuration de ses attributs. Cosynt est un langage de syntaxe concrète permettant de spécifier les afficheurs et de générer les analyseurs syntaxiques associés.

1.2.2 Le Corba Component Model

D'après les spécifications du *Corba Component Model* [34] un type de composant est un ensemble de caractéristiques pouvant être décrits par un langage de description, il définit les services fournis et les besoins du composant (quels sont les composants avec lesquels il doit se connecter). Je me suis aussi appuyée

sur [30] pour écrire cette partie de mon rapport.

- La partie métier est encapsulée dans le composant et de fait elle est non visible par le client. Cette partie est laissée à la charge du développeur. Il existe quatre types différents de composants (*service, cession, process et entity*).
- les conteneurs, serveurs de conteneurs et maisons de composants: Un conteneur est un support d'exécution générique qui peut accueillir différents types de composants, mais ayant toutefois des caractéristiques systèmes communes (en terme de persistance par exemple). Les conteneurs reposent sur des serveurs de conteneurs. Ces serveurs sont des processus, au sens Unix du terme, qui permettent d'activer des conteneurs à la demande et selon les besoins. Il existe encore les "maisons de composants" (*home*). Il s'agit d'un gestionnaire d'instances de composants, s'occupant principalement du cycle de vie. Chaque maison gère un type de composant.
- Les interfaces se situent à la frontière entre le composant et le conteneur. Elles connaissent le contenu du composant et savent où diriger les requêtes des clients transmises par le conteneur. Dans le Corba Component Model elles portent aussi le nom de port. Quatre types de ports sont définis dans ce modèle. **Une facette** est une interface fournie par un composant et qui est utilisée par des clients en mode synchrone. Elle représente un point de vue sur le composant (les méthodes du composant y sont regroupées de manière logique) et permettent la découverte des services offerts par le composant (propriété d'introspection). Plusieurs références sur une même facette peuvent exister ce qui permet à plusieurs clients de se connecter en même temps. Un client peut aussi naviguer entre les interfaces selon ses besoins. **Un réceptacle** est une interface utilisée par un composant en mode synchrone. Il permet d'assembler les composants entre eux (la connexion se fait entre un réceptacle et une facette). Plusieurs connexions sur un même réceptacle sont possibles. **Un puit d'événements** est une interface fournie par un composant et utilisée par ses clients en mode asynchrone. Ici le composant fournit des événements aux consommateurs. **Une source d'événements** est une interface utilisée par un composant en mode asynchrone. Elle permet au composant de recevoir des événements d'un type donné. Le composant est consommateur. Un composant peut supporter plusieurs interfaces.
- Une instance de composant est identifiée par sa référence et les références à ses facettes si elles existent.
- Les attributs sont les propriétés configurables du composant.

1.2.3 Les EJB

Les informations qui suivent sont issues de [26] et de [28]

- Il s'agit de composants , utilisant des standards ouverts pour la description, la découverte et l'intégration (WSDL: *web services description langage*, SOAP: *simple object access protocol*). Le modèle s'attache à décrire

la manière dont des composants peuvent coopérer entre eux. Il existe trois types de composants: *session, entity et message driven.* Dans le premier type le client commence une session avec le composant qui se comporte comme une application, exécutant un traitement pour le client qui peut inclure des transactions avec des bases de données. Dans le second type le client accède à un composant qui est une entité d'une base de données. Le dernier, enfin, répond à un unique message de client de manière asynchrone.

- les conteneurs et serveurs d'EJB: Les composants (ou Beans) sont isolés grâce aux conteneurs. Les conteneurs sont regroupés dans des serveurs et tous deux implantent les propriétés non fonctionnelles. Le client ne communique pas directement avec le composant mais passe par le conteneur. Les EJB server gèrent les ressources nécessaires au composant EJB tandis que le conteneur gère son cycle de vie. C'est le conteneur qui instancie le composant.
- Les EJB définissent des contrats entre le conteneur et le composant. Le contrat fourni au client (par l'intermédiaire du conteneur) une vue du composant indépendante de la plateforme. Il permet la portabilité du composant sur différents serveurs. Il fournit aussi un format de fichier standard pour faire des paquetages de composants.
- Les interfaces sont au nombre de deux et sont spécifiques au type de composant. L'*Home Interface* permet la gestion du composant tandis que la *Remote Interface* permet l'accès aux méthodes du composant. La *Remote Interface* publie les méthodes accessibles par les clients, intercepte les invocations des clients et appelle la méthode appropriée (c'est la *Component Interface* de l'instance). Tandis que la *Home interface* permet au client de créer, supprimer ou trouver les EJB. Les clients peuvent découvrir les EJB à travers l'API standard JNDI (Java Naming and Directory Interface).

1.2.4 Les services web

D'après [37] un service web est un système logiciel permettant des interactions à travers Internet. Il possède une interface décrivant comment les autres systèmes peuvent agir avec lui en utilisant le protocole SOAP. Ils sont décrits en WSDL. Un service web se trouve au sein d'un web server et est en relation avec des "objets métier" se trouvant dans des serveurs d'applications le tout formant un web services provider. Le service web fournit une interface qui sera interrogée par le client grâce à SOAP. Un web service est composé:

- de messages décrivant les noms et types d'un ensemble de champs à transmettre (paramètres d'une invocation, valeur du retour, ...);
- d'un type de point d'entrée (*porttype*) décrivant un ensemble d'opérations. Chaque opération a zéro ou un message en entrée, zéro ou plusieurs messages de sortie ou d'erreurs. Les opérations sont décrites par un nom et un ou plusieurs attributs suivant: *One-way* reçoit un message (input), *Request-response* reçoit un message (input) et retourne un message corrélé

(output) ou un ou plusieurs messages d'erreurs (fault), *Solicit-response* envoie un message (output) et reçoit un message corrélé (input) ou un ou plusieurs messages d'erreurs (fault), *Binding HTTP* : 2 requêtes HTTP par exemple, Notification envoie un message de notification (output).

- *des binding* spécifiant une liaison d'un porttype à un protocole concret (SOAP1.1, HTTP1.1, MIME, ...). Un *porttype* peut avoir plusieurs liaisons;
- des ports spécifiant un point d'entrée (*endpoint*) comme la combinaison d'un binding et d'une adresse réseau;
- service, une collection de points d'entrée (*endpoint*) relatifs.

1.2.5 Le modèle JavaPod

Il s'agit ici de définir une plate-forme à composants qui permette d'associer aux composants des propriétés non fonctionnelles. Elle est décrite dans [13].

- Le composant est un objet pouvant avoir plusieurs interfaces d'accès. Cet objet est un ensemble d'objets au sens des langages de programmation à objets.
- Le conteneur est la partie système correspondant au composant. Il l'encapsule, interceptant toutes les communications du composant avec l'extérieur et gérant ses propriétés non fonctionnelles. Il offre des talons et squelettes pour permettre les communications avec l'extérieur.
- Le connecteur est un objet distribué permettant de lier différents composants. Pour celà il est constitués de plusieurs talons et squelettes qui appartiennent aux conteneurs des composants. Plus précisément ce sont des interfaces entre les connecteurs et les composants, appartenant aux deux à la fois. Un talon permet au composant d'envoyer des messages vers l'extérieur, tandis qu'il les reçoit à travers le squelette. Chacun possède un type, c'est à dire l'ensemble des signatures des méthodes qu'un composant peut appeler sur un talon ou qu'un squelette peut appeler sur un composant.

1.2.6 Fractal

D'après [12] un composant Fractal est une membrane ayant un contenu et offrant de interfaces, les points d'accès au composant. L'intérieur est une boîte noire constituée de code ou de composants implantant les services fournis par les interfaces.
Le rôle du contrôleur est de fournir une représentation de son contenu vers l'extérieur, d'intercepter les messages entrant et sortant, d'ajouter certains comportements à ceux définis dans le composant.

En conclusion, le type de composant comporte la partie métier fournissant des services fonctionnels à travers des interfaces. La partie métier se trouvant dans une structure d'accueil gérant les services non fonctionnels et les appels entre le composant et le monde extérieur.

1.3 Implantation

Implantation de la partie fonctionnelle par le développeur et génération de la partie non fonctionnelle.

1.3.1 SmartTools

Dans SmartTools le développeur doit porter la plus grande attention au formalisme et écrire la partie métier. A lui ensuite de spécifier l'affichage de son composant dans l'interface graphique.

1.3.2 Le Corba Component Model

Un Component Implementation Framework est défini pour automatiser l'implantation des composants, pour celà il utilise le Component Implementation Definition Langage: le concepteur de composant décrit le composant à l'aide du langage IDL et la structure de l'implantation ainsi que les propriétés non fonctionnelles à partir du langage CIDL. Un compilateur génère alors la partie non fonctionnelle, le squelette ainsi que le fichier de description. Le développeur lui implante la partie métier.

1.3.3 Les EJB

La classe du composant est générée par le conteneur. Il fournit aussi l'implantation des propriétés non fonctionnelles.

1.3.4 Fractal

Le modèle dispose de *factories*. Ces composants permettent de créer d'autres composants, soit des *factories* génériques pouvant produire n'importe quel type de composant soit des *factories* standards ne pouvant produire qu'un type de composant.

1.4 Diffusion des implantations, déploiement

1.4.1 SmartTools

Les composants SmartTools sont distribués dans des fichiers j

1.4.2 Corba Component Model

Un composant est "packagé" dans une archive au format zip contenant l'implantation du composant, son descripteur et une configuration par défaut permettant le déploiement. Ce paquetage de composant est utilisé tel quel, ou par un architecte pour être inclus dans un assemblage et produire des applications.

Pour être diffusable, un assemblage de composants est lui aussi "packagé", regroupant dans une archive les implantations des composants, le descripteur de l'assemblage (permettant de spécifier les services utilisés par le composant) et la configuration des composants pour ce contexte précis. Les paquetages d'assemblage sont ensuite diffusables ou réutilisables pour participer à une nouvelle composition. Enfin, les paquetages de composants et d'assemblages sont déployés pour instancier les applications sur leurs sites d'exécutions. il est possible de choisir des sites d'exécution à partir d'un poste d'administration. Pour celà, en plus de fournir un environnement d'exécution (un serveur de conteneurs), un site doit offrir un service de chargement de code, il est ainsi possible de charger une implantation de composant et sa maison associée sur un site distant. Une fois les conteneurs instanciés, les implantations de composants et de maisons chargées, les composants et les maisons instanciés, il ne reste plus qu'à inter-connecter les instances de composants entre elles pour former l'application. La connexion est statique au démarrage.

1.4.3 EJB

Les Beans sont "packagés" dans un fichier EJB.jar. C'est un fichier jar classique contenant les classes des composants et un descripteur XML. Le Bean est installé dans l'EJB server à l'aide d'un outil de déploiement fourni par le serveur. Ensuite, une intervention est nécessaire pour paramétrer les attributs du Bean (mode de transaction, niveau de sécurité...).

1.5 Utilisation des instances de composants

1.5.1 SmartTools

Les composants SmartTools communiquent directement entre eux. Lorsqu'un composant cherche à en contacter un autre il demande au ComponentManager d'établir la connexion. Si le composant demandé n'est pas instancié le ComponentManager en crée une instance.

1.5.2 Le Corba Component Model

Un client interagit avec un composant CORBA au travers des deux formes d'interfaces externes : une interface de maison et une ou plusieurs interfaces d'application. Les interfaces de maison permettent au client d'avoir accès à des références d'interface d'application implantées par le composant.

1.5.3 Les services web

Le client interroge l'annuaire UDDI pour trouver le serveur hébergeant le service web qui l'intŕesse. Il interroge alors le serveur qui lui fournit le format d'appel de ses services (son contrat). Le client peut alors envoyer sa requête dans

le bon format en utilisant le protocole SOAP, le serveur lui retournera ensuite le résultat.

Chapitre 2

Une proposition de modèle de composants

Dans ce chapitre nous nous proposons, à partir de l'étude précédente, de préciser le vocabulaire utilisé dans la description des modèles. Ces choix sont par définition arbitraires et donc discutables mais ils serviront à préciser les différentes notions développées dans la suite de ce rapport. Le modèle abstrait de composant obtenu comporte quatre parties:

1. La partie encapsulée;
2. La partie "encapsulante";
3. les services non fonctionnels;
4. la communication.

2.1 Des définitions

Dans les chapitres suivants nous allons nous employer a expliciter ces notions une à une. Mais tout d'abord tentons de définir le composant.
Composant: qui sert à former, qui entre dans la composition de .
Pour le modèle Darwin [31] c'est une unité de traitement encapsulant le code applicatif. Pour Unicon c'est une entité qui encapsule des fonctions d'un module logiciel ou de données. Olan indique simplement que le composant encapsule le logiciel. Pour JavaPod [13] c'est une entité encapsulant un état interne accessible uniquement par des méthodes. COM le défini comme une entité binaire, ou plus exactement l'implantation du comportement des interfaces. Pour ODP il s'agit d'un objet de traitement encapsulant son état interne et ses traitements. Le Corba Component Model, [33] indique qu'il fournit un ensemble de comportements bien définis. Fractal [19] parle d'entités accessibles par des appels de méthodes à travers des interfaces. Elle fournit les services des interfaces. Pour les Web Services il s'agit d'une unité logique applicative.
Les termes communs à toutes ces définitions sont unité/entité et encapsulation.

Finalement **un composant est une unité logicielle encapsulant son état interne et qui est accessible par des appels de méthodes à travers des interfaces.**

2.1.1 La partie encapsulée

Métier, du latin ministerium: profession considérée relativement au genre de travail qu'elle exige. Occupation manuelle ou mécanique qui permet de gagner sa vie.
Partie métier: partie réalisant le travail que l'utilisateur attend de l'application. C'est elle qui est encapsulée dans le composant.
Façade:*Cotés où est située l'entrée principale d'un bâtiment. Apparence masquant une piètre réalité.* C'est la liste des méthodes disponibles dans le composant. **La façade offre les connaissances sur la partie métier** et se trouve à la frontière entre "l'encapsulé" et "l'encapsulant".
Méthodes: *ensemble de procédés, de moyens pour arriver à un résultat.* Les actions du composant.

2.1.2 La partie "encapsulante"

Interface: mot anglais d'origine latine, limite commune à deux systèmes.
Pour le modèle Darwin l'interface décrit le composant. L'interface d'UNICON définie les fonctions et données accessibles et requises. Dans les EJB les interfaces fournissent des opérations métier. Quand à ODP, ce modèle regroupe les opérations du composant dans une interface. Dans le Corba Component Model les composants disposent d'interfaces indiquant les services requis et fournis. La notion d'interface fournie permet d'indiquer les fonctionnalités accessibles à partir de cette interface. La notion d'interface requise marque la nécessité de disposer des fonctionnalités indiquées dans l'interface et appartenant à d'autres composants, c'est une liaison entre deux composants. Les interfaces permettent l'accès aux composants et contiennent la définition du mode d'appel vers le composant (synchrone, asynchrone...). Dans le modèle Fractal les composants sont liés par leurs interfaces: elles décrivent les fonctionnalités offertes par le composant (interface fournie) mais indiquent aussi les fonctionnalités nécessaires au composant (interface requise) . Dans SmartTools la façade connaît le contenu du composant et ce sont les ports qui permettent d'y accéder. Les EJB sont accessibles par leurs interfaces: l'interface publie les méthodes accessibles, intercepte les invocations et appelle la méthode appropriée du composant. Les interfaces des Web services décrivent comment les systèmes peuvent agir avec eux.
En conclusion une interface est à la frontière entre le composant et le reste du monde. On peut identifier deux taches principales remplies par l'interface; permettre au monde extérieur de connaître le composant et interagir avec lui. Dans notre modèle nous avons choisi de différencier ces deux rôles. La façade permet la connaissance du composant et l'interface est à la frontière entre le composant et le monde extérieur. **L'interface définie les messages envoyés**

et reçus par le composant. Dans notre modèle on dispose aussi de la notion de multi-interfaces: pour un même composant différentes interfaces peuvent être définies c'est à l'instanciation que le choix de l'une d'entre elle sera fait.

Conteneur: *Récipient métallique servant à contenir des marchandises ou des substances afin d'assurer leur transport, leur manutention, leur conservation et de se protéger de leur nocivité.*
Dans JavaPod le conteneur est la partie système correspondant au composant. Il l'encapsule, interceptant les communications entre l'extérieur et le composant et gère les propriétés non fonctionnelles. Les EJB sont isolés grâce à leur conteneur, les communications avec l'extérieur passant obligatoirement par ce dernier qui fournit les interfaces. Une autre fonctionnalité du conteneur d'EJB est la gestion du cycle de vie du composant. Dans le Corba Component Model un conteneur est un support d'exécution générique accueillant les composants et prenant en charge certains aspects non fonctionnels. Dans SmartTools le conteneur gère les communications du composant avec l'extérieur. Il intercepte les messages et effectue les appels au composant. Pour [32] le conteneur réalise la séparation du fonctionnel et du non fonctionnel en prenant en charge ce dernier.

Le conteneur réalise l'encapsulation du composant, interceptant et gérant les messages avec le monde extérieur. Certains modèles lui associe les propriétés non fonctionnelles.

message:*Ce que l'on transmet (objet, informations...)*. Dans les web services les points d'entrées sont associés à des messages. Les messages décrivent les informations transmises d'un composant à l'autre (appel de méthodes, paramètres, valeurs de retour...).

2.1.3 la communication

Dans SmartTools les ports définissent le mode de communication entre composants. Pour le modèle Durra les ports constituent les points d'entrée (d'accès) et de sortie du composant. Les web services ont des ports spécifiant les points d'entrée dans le composant. Pour Accord un port est un point d'accès aux opérations du composant. Nous retiendrons pour notre modèle que le **port définit le mode de communication du composant avec l'extérieur.** La première chose à définir est le flux: le message est-il "in" (c'est un appel à une méthode du composant) ou "out" (c'est un appel de ce composant à un service fonctionnel rendu par un autre composant). Vient ensuite la contrainte : le service appelé en out est-il obligatoire pour faire démarrer l'instance du composant ou est-il simplement optionnel. On peut avoir un port out-obligatoire ou out-optionnel alors qu'une contrainte sur un port in n'aura pas d'influence sur le composant et son environnement. Il faut aussi définir un mode: synchrone, asynchrone... Enfin il est nécessaire de déterminer la cardinalité du port: y a-t-il un échange 1-1, 1-n?

Connecteur: Le projet Accord [1] explicite les interactions entre composants

à l'aide des connecteurs. Le connecteur assure la mise en oeuvre du protocole associé à l'interaction. Pour le modèle OLAN les connecteurs spécifient le protocole de communication permettant la connexion de composants hétérogènes. Dans JavaPod les composants sont reliés par des connecteurs, ces derniers représentant n'importe quel type de liaison (client/serveur, flot de données...). Pour [14] le connecteur représente la communication entre les composants. **les connecteurs masquent les mécanismes de communication entre composants. Ainsi, si le port définit le mode de communication le connecteur le met en oeuvre.**

2.1.4 Les services non fonctionnels

Une notion fondamentale des composants, non encore évoquée dans ce chapitre, est la séparation des parties fonctionnelles et non fonctionnelles. La première est prise en charge par le composant proprement dit; la seconde va être définie ici. Ce qui est fonctionnel touche au métier de composant, à ce que l'on attend de lui. Ce qui est non fonctionnel concerne tous ce qui n'est pas spécifique à l'application mais qui lui est néanmoins indispensable (la sécurité, la distribution, la persistance...). Pour [11] le fonctionnel est "le quoi" tandis que le non-fonctionnel est "le comment". Ce sont les mécanismes régissant l'exécution de l'application.
Dans les composants ces aspects sont souvent pris en charge au niveau du conteneur (JavaPod, EJB, CCM) . Dans Fractal c'est les contrôleurs qui sont en charge de la partie non fonctionnelle, ils interceptent aussi les messages de et à destination de l'extérieur et permettent l'introspection sur le composant.

2.1.5 Conclusion

De cette étude se dégage les éléments suivants: un composant est une entité logicielle encapsulant son état interne dans la partie métier, la façade décrivant les méthodes accessibles. L'encapsulation est réalisée grâce au conteneur qui intercepte les messages entre le composant et l'extérieur (par l'intermédiaire de ses interfaces) et gère les propriétés non fonctionnelles nécessaires au composant. Pour établir la communication entre composants il faut disposer de points d'entrée (les ports) définissant le mode de communication et le réaliser à l'aide des connecteurs.

2.2 Validation du modèle

Le modèle proposé dans ce chapitre s'appuie sur les notions définies ci-dessus. Il servira de base de départ à nos propositions pour décrire un modèle générique. Pour celà je me suis inspirée plus particulièrement du modèle de SmartTools.

La première étape a consisté à décrire notre modèle à l'aide d'une syntaxe abstraite. Nous avons choisi de nous inspirer d'Absynt, le langage déclaratif de SmartTools permettant de définir des syntaxes abstraites. Notre objectif étant de présenter le modèle selon un semi-formalisme.
Un composant a des attributs (propriétés configurables du composant) et est constitué d'une partie métier contenant une façade; cette dernière ayant un nom (dans ce cas on utilise un mécanisme d'introspection pour déterminer les méthodes de la partie métier) ou exposant un ensemble de méthodes de la partie métier. D'un conteneur qui expose une interface, cette dernière acceptant des messages. De ports définissant la communication du composant et de services non fonctionnels. Remarquons que dans la définition du composant j'ai indiqué qu'il pouvait avoir un ensemble de conteneurs. En effet, un conteneur est lié à l'interface que le composant expose vers l'extérieur, or dans notre modèle nous posons comme hypothèse qu'un composant peut disposer de plusieurs interfaces. C'est à l'instanciation du composant que le choix est fait entre les différentes interfaces proposées ce qui conduit à la génération d'un des conteneurs.
Chacune de ces notions a des attributs dont je vais décrire la sémantique. L'attribut "name" est le nom du composant, de l'interface, des services non fonctionnels, des ports, de la façade, des méthodes, du conteneur et des messages. L'interface et le conteneur ont aussi un "type", il permet de les classer. L'attribut *extends* indique l'existence de la notion d'héritage au sens *include*. Le composant hérite de l'interface et il faut ensuite implanter les comportement correspondant à ses messages. Le message a un *port*, ce qui permet de déterminer son mode de communication. Ainsi un port a un *mode* (synchrone, asynchrone...), une *cardinalité* (relation 1-1 entre les composants, 1-n ...), un *flux* (en entrée ou en sortie), des *constraints* (obligatoire ou optionnel) et un *protocole de communication*. C'est la composition de ces cinq attributs qui permet de déterminer le mode de communication associé au port.
Le fait de définir une façade et une interface permet de prévoir une clause d'adaptation entre les messages et les méthodes afin de pouvoir définir des messages ne contenant pas tous les arguments des méthodes.

```
 Formalism of model is

Root is Top;

Operator and type definitions {
  Top = component(Attribute[] attribute, Metier? metier, Container [] container, Services [] services,
                  Ports [] ports);

  Attribute = attribute();

  Metier = metier( Facade facade);
    Facade = facade(), methods (Method [] method);
      Method = method (Arg [] arg);
        Arg = argument();

  Container = container (Interface [] interface);
    Interface = interface (Message [] message);
      Message = message(Parameter [] parameter);
```

```
        Parameter = parameter();

  Services = services(Service [] service);
    Service = service () ;


  Ports = ports( Port [] port);
    TypePort = typeport () ;

}

Attribute definitions {
REQUIRED name as java.lang.String in component, metier, interface, service, typeport, facade, method,
                                     container, message, parameter, attribute;
REQUIRED type as java.lang.String in component, interface, parameter, attribute;
REQUIRED extends as java.lang.String in component;
REQUIRED port as java.lang.String in message;
REQUIRED methode as java.lang.String in message;
REQUIRED mode as java.lang.String in typeport;
REQUIRED cardinality as java.lang.String in typeport;
REQUIRED flux as java.lang.String in typeport;
REQUIRED constraint as java.lang.String in typeport;
REQUIRED protocol as java.lang.String in typeport;
}
```

2.2.1 Une instance de composant

On considère un composant simple nommé *ComposantSimple*. Il a une partie métier que j'ai appelée "fonctionnel" et sa façade utilise le mécanisme d'introspection pour identifier les méthodes disponibles au sein du composant. Pour simplifier les choses il n'a qu'un conteneur. Le premier message permet de connaître la liste des services fonctionnels offerts par ce composant et le second lui indique qu'il doit se connecter avec un composant extérieur.

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="ComposantSimple" type="composant" extends="">
    <attribute name="attribut1" type="String"/>
    <metier name="fonctionnel">
        <facade name="facade"/>
    </metier>
    <container name="conteneur">
        <interface name="interface1" type="typeInterface">
            <message name="ListeService"  port="port1" methode="getService">
              <parameter name="parametre1" type="type1"/>
            </message>
            <message name="Connected" port="port2"methode="connectTo">
              <parameter name="parametre2" type="type2"/>
            </message>
        </interface>
    </container>
    <services>
        <service name="securite"/>
    </services>
    <ports>
        <typeport name="port1" mode="asynchrone" cardinality="1-n" flux="entree"
                                      constraint="optionnel" protocol="protocol1"/>
        <typeport name="port2" mode="asynchrone" cardinality="1-1" flux="sortie"
                                    constraint="obligatoire" protocol="protocol2"/>
    </ports>
</component>
```

2.3 Le modèle abstrait de composant et les autres modèles

Je vais maintenant chercher à donner des règles permettant de projeter notre modèle vers différents modèles étudiés dans la bibliographie.

Nous travaillons dans ce chapitre à la transformation des modèles. Pour celà nous proposons un langage simple permettant d'exprimer le passage d'un modèle à l'autre de manière réversible.

2.3.1 SmartTools

D'après l'étude précédente un composant SmartTool a:

- un nom et un type (avec une notion d'héritage, le composant hérite du type),
- un conteneur,
- une façade,
- des attributs de configuration,

- des dépendances vis à vis d'autres modules de code,
- une notion de service (port) qui recouvre: le fait que le service soit en entrée ou en sortie
  le mode de communication synchrone ou asynchrone
  le nom logique du service et la méthode à appeler dans la façade
  les paramètres de la méthode.

Les quatre premières notions se retrouvent dans notre modèle. Les autres notions sont présentes dans les deux modèles mais sous des termes différents. Les dépendances vis à vis des autres modules de code se retrouvent au niveau de l'attribut *constraint* du port. Le nom logique des services est dans l'interface de notre composant; les autres notions des services des composants SmartTools dans les ports de notre modèle.
Nous allons utiliser un langage inspiré du Cosynt (langage permettant de spécifier comment afficher la syntaxe concrète du modèle) de SmartTools afin d'exprimer les règles de transformation entre modèles. Il s'agit d'un processus réversible, c'est pourquoi nous utilisons le signe ":". Lorsque le signe "!" est employé, celà signifie qu'on a à faire à une notion qui n'est pas présente dans l'autre modèle.

```
component(Metier, Container, Services, Ports, Attribute) {@name
                                  @type @extends}:component(Documentation!, Container, Facade,
                                            Formalism!, Parser!, Lml!, Bahavior!, Dependance!,
                                              Attribute, Port)){@name, @type, @extends, @ns!};
                        attribute(){@type, @name}:attribute(){f(@javatype), @name};
                metier(Facade, Formalism) {@name}:(facadeclass(){@name, @userclassname!},
                                                                  formalism() {@name});
                            facade(Method) {@name}:facadeclass(){@name, @userclassname!};
                               formalism() {@name}:formalism() {@name};
                     container(Interface) {@name}:containerclass() {@name} ;
                       parameter() {@name, @type}:parameter(){@name, f(@javatype)};
                                Services(Service):
                                service() {@name}:
                                      Ports(Port):
                  interface(Message) {@name @type}:
                  message(Parameter) {@name @port}:(F(port), message(Parameter) {@name});
                        parameter() {@name, @type}:parameter(){@name, f(@javatype)};
port() {@name @mode @cardinality @flux @constraint @protocol}:
                                                  :parser(Extension){@type, @generator,
                                                                        @classname};
                                                  :extension(){@name};
                                                  :lml(){@file, @name};
                                                  :behavior() {@file};
                                                  :documentation();

      Avec F:
           asynchrone obligatoire entree : input;
           asynchrone obligatoire sortie : output;
           synchrone                     : inout;
      et f:
           type:javatype;
```

F est une fonction indiquant comment traduire les notions de input, output et inout de SmartTools dans notre modèle; tandis que f permet de transformer un type en javatype. On constate alors que, hormis le conteneur et la façade, les concepts permettant de regrouper des notions proches sont absents. Il s'agit de ce que nous avons nommé Métier, Interface, Services et Ports. Les méthodes,

même si elles existent comme pour nous, ne sont pas explicitées dans le modèle de SmartTools. Elles apparaissent comme arguments des ports. D'ailleurs, les ports de SmartTools recouvrent les notions de port et de message pour nous. SmartTools a enrichi ce modèle minimal pour répondre a ses besoins propres, la création de composants métiers à partir de la description d'un langage ou d'une structure de données. Le composant logique contient le formalisme du langage: il permet de construire les structures de données du document et de valider les données échangées entre les composants.
les protocoles de lecture d'un document (parser) en fonction des extensions définies par l'utilisateur.
les composants vues (composants de visualisation) qui sont associés par défaut au composant logique(lml): ces composants sont nécessaires à l'interface graphique de l'application.
les informations pour construire les menus spécifiques au langage dans l'interface utilisateur (behavior): il s'agit d'ajouter des services aux composants vues pour les adapter aux composants logiques.

2.3.2 Le CCM

Procédons de la même manière pour le Corba Component Model. Dans [23] le CCM est décrit comme suit. Un composant a un identifiant, une liste d'évènements et des interfaces. Les évènements sont de type *emit*, *consume* ou *publish*. Un composant a aussi des interfaces utilisées ou fournies. Pour compléter cette description notons que le composant est placé dans un conteneur prenant en charge les services non fonctionnels et que la partie métier est une boîte noire.

```
component(Metier, Container, Services, Ports, Attribute) {@name
                                  @type @extends}:component(Attribute, Events!, Interfaces);
                           attribute(){@type, @name}:atribute();
                  metier(Facade, Formalism) {@name}:
                              facade(Method) {@name}:
                                 method(Arg) {@name}:
                                               arg():
                                 formalism() {@name}:
                          container(Interface) {@name}:container()
                   interface(Message) {@name @type}:interfaces();
                   message(Parameter) {@name @port}:F(events!);
                         parameter() {@name, @type}:
                                   Service(Service):
                                  service() {@name}:
                                        Ports(Port):
port() {@name @mode @cardinality @flux @constraint @protocol}:

       Avec F:
            synchrone obligatoire entree : receptacle, interface fournie, methode;
            synchrone obligatoire sortie : facet;
            aynchrone, entree            : event sink;
            asynchrone, sortie           : event source;
```

2.3.3 Les Web Services

D'après [37] Un web service est composé:

- de types;
- de messages;
- d'opérations;
- d'un type de point d'entrée, le porttype;
- de binding;
- des ports;
- de service.

Remarquons que dans le cas des Web Services tous les échanges se font selon le mode requête/réponse.

```
component(Metier, Container, Services, Ports, Attribute) {@name}
                                  @type @extends}:component(Type,Operation, Message, PortType,
                                                                Binding, Service);
                          attribute(){@type, @name}:
                                                  @type:type();
                    metier(Facade, Formalism) {@name}:
                              facade(Method) {@name}:
                                 method(Arg) {@name}:operation(MessageEntree, MessageSortie) {@name};
                                                  arg():
                                 formalism() {@name}:
                         container(Interface) {@name}:
                     interface(Message) {@name @type}:porttype(Operartion, MessageEntree,
                                                              MessageSortie) {@ame @endpoint!};
                     message(Parameter) {@name @port}:binding(Operation, ProtocolEntree,
                                                             ProtocolSortie){@name @type};
                           parameter() {@name, @type}:message(Type) {@name};
                                    Service(Service):
                                   service() {@name}:
                                        Ports(Port):service(Port, Binding) {@name};
port() {@name @mode @cardinality @flux @constraint @protocol}:port() {@name};
                                           protocol:protocol();

avec MessageEntree, MessageSortie des Messages. ProtocolEntree, ProtocolSortie des Protocoles.
```

2.4 Connecter des composants

Nous disposons des éléments minimum pour fabriquer un composant. Nous allons maintenant faire des propositions pour permettre à ces composants de vivre et d'interagir ensemble. Pour celà il faut disposer d'un moyen de connaître les services. Nous allons voir que la solution retenue est celle d'un annuaire.
"Connecter deux composants c'est faire en sorte qu'ils travaillent ensemble. Que l'un puisse demander un service à l'autre (invocation de méthodes par l'intermédiaire des interfaces). Ce là nécessite de connaître la référence du composant, le protocole de communication."
Dans le Corba Component Model [30] la connexion se fait entre une facette (une interface exposant les méthodes de son composant) et un réceptacle (du second composant voulant utiliser les services du premier) et de manière statique. Les connexions sont indiquées dans le fichier de déploiement et établies

au démarrage par passage des références des facettes. Un client obtient une référence sur une facette en invoquant une opération sur le composant visé. Un réceptacle permet à un type de composant d'accepter une référence d'objet (c'est-à-dire une référence de facette, de composant ou d'objet au sens CORBA 2). Cette relation est appelée connexion. Le réceptacle est une abstraction qui se manifeste concrètement dans un composant par un jeu d'opérations pour établir et gérer les connexions. Un réceptacle permet donc d'assembler des instances de composants. Les réceptacles peuvent être de deux types : des réceptacles simples (une seule référence), ou des réceptacles multiples (plusieurs références d'objets peuvent être fournies). Remarquons que WSDL permet de décrire les services comme un ensemble d'opérations et de messages abstraits reliés à des protocoles et des serveurs réseaux. Dans le projet Fractal c'est le contrôleur qui vérifie que toutes les connexions obligatoires (une référence sur le nom de l'interface) sont établies avant d'autoriser le lancement l'application. JavaPod quand à lui dispose de connecteurs (composant reliant deux interfaces ou plus) constitués d'objets distribués entre les composants à lier. Les composants obtiennent une référence de connecteur identifiant un unique connecteur et décrivant les extensions à assembler pour construire la partie du connecteur appartenant au composant ayant demandé la connexion. Les EJB et les Web services utilisent des objets d'interpositions. Pour les EJB, le client obtient la référence voulue par l'intermédiaire de l'Home Object (qui est aussi chargé du cycle de vie des EJB) ou d'un annuaire (JNDI). Une fois la référence obtenue, le client invoque les méthodes de l'EJB à travers son interface en utilisant RMI. Dans le cas des cession beans une connexion est établie entre le client et l'EJB jusqu'à la fin des transactions. Un cession bean peut utiliser un entity bean (mais c'est le seul client de l'entity bean possible) pour l'accès aux bases de données. On est ici dans un mode synchrone pour répondre aux besoins des EJB qui sont une technologie orientée transactions. Les web services quand à eux répondent à une logique de requête/réponse sous forme XML. Le client recherche le Web Services dans un annuaire puis établit directement un lien avec le serveur trouvé. L'accès aux fonctions du Web Services se fait en suivant le protocole SOAP. Les liens sont établis de manière dynamique et sont transitoires. Pour SmartTools le componentManager joue le rôle d'annuaire. La méthode ConnecteTo effectue le mapping des services. Les in (services en entrée permettant d'accéder à une méthode du composant) du premier composant sont parcourus et pour chacun d'eux on regarde si il existe un out (service en sortie) correspondant sur le deuxième composant (à partir du nom du service). Si c'est le cas les ports sont connectés avec des listeners. L'opération est répétée du deuxième composant vers le premier. Une fois la connexion effectuée les composants dialoguent directement entre eux.

Il ressort de notre étude les éléments suivants:

On peut associer les interfaces des composants désirant communiquer directement ou via un connecteur)(élément permettant d'assembler des composants en utilisant leurs interfaces) qui est lui aussi un composant. Pour établir une connexion il faut disposer de la référence du composant avec lequel on veut se connecter. Un annuaire fournit ces références. Afin que l'annuaire les connaisse,

c'est aux composants de s'y inscrire et de se des-inscrire.
Il existe deux types de connexion, la connexion statique (voir plus loin) et la connexion dynamique (elle permet d'obtenir la référence d'un composant en cours d'exécution de l'application). Remarquons que la connexion statique est la plus simple à mettre en oeuvre. Les liaisons entre composants sont définies dans le fichier de déploiement (fichier explicitant les composants à instancier... Il peut porter différents noms; par exemple dans SmartTools il se nomme lml). Pour chaque composant instancié on regarde de quels services il a besoin, on cherche ces services dans les autres composants présents et si on les trouve on lie les composants au demandeur. C'est le mode de connexion choisi par le Corba Component Model. Il est possible aussi de choisir une connexion dynamique comme pour les Web Services. La connexion entre deux composants se fait en cours d'exécution toujours selon le même schéma: demande du service à un annuaire, l'annuaire envoie la référence sur le composant correspondant et les deux composants établissent la liaison. Mais qui décide de l'initialisation de la connexion? C'est ce que nous allons étudier dans le paragraphe suivant.
Que ce soit dans SmartTools, les Web Services ou dans les EJB se sont les clients qui sont à l'origine de la connexion. Au cours de leur vie ils ont besoin d'un service et établissent la connexion avec le composant pouvant le leur fournir. Finalement la connexion dynamique est le résultat, à un instant de l'exécution d'un composant, du besoin d'utiliser un service disponible sur un autre composant. Le composant demandeur contacte alors l'annuaire qui lui transmet la référence du composant fournisseur.
Revenons maintenant sur la notion d'annuaire. Dans les Web Services l'UDDI a vocation a être un annuaire mondial d'entreprises permettant d'automatiser les communications entre prestataires et clients. Pour celà il existe plusieurs entrées dans l'annuaire permettant de décrire l'entité offrant le service, la description du service et comment construire les messages à envoyer à ce service. Les Web Services doivent s'inscrire auprès de l'annuaire. Les EJB eux interrogent l'Home Object pour obtenir des références sur les EJB Object (l'interface de l'EJB). Home Object instancie et détruit les EJB et possède donc leur références. Nous avons à faire ici à deux logiques différentes. Bien que visant toutes deux à construire des applications réparties la première est destinée à des composants mis à disposition à travers l'internet et fonctionnant en mode client/serveur et bâtis sur des technologies différentes, la seconde construit des applications réparties classiques.
Voici maintenant deux scénari possible de fonctionnement, l'un avec un annuaire unique et l'autre avec plusieurs annuaires:
Premier scénario: Un client a besoin d'un service, il contacte l'annuaire. L'annuaire recherche le composant susceptible de fournir le service demander. Ce composant est instancié, il retourne la référence au composant demandeur. Ce composant n'est pas instancié, l'annuaire l'instancie et retourne la référence au composant demandeur.
Deuxième scénario: Il existe plusieurs annuaires. Le composant fait sa demande à l'annuaire dont il connaˆla référence. Soit cet annuaire trouve le service et on est ramené au premier scénario, soit il ne le trouve pas et il transmet la requête

et la référence du composant demandeur à l'annuaire dont il a la référence et ainsi de suite jusqu'à ce que le service soit trouvé. L'annuaire renvoie alors la réponse directement au composant demandeur. Il faut prévoir une condition d'arrêt, au cas ou le service ne soit répertorié dans aucun annuaire. On retiendra aussi comme solution de faire instancier les composants par les annuaires. Ainsi un composant sera obligatoirement référencé par l'un d'eux. Cette instanciation est initiée par un autre composant. De la même manière, à la demande du composant concerné ou d'un autre c'est l'annuaire qui a en charge la destruction des instances de composants. D'autres scénari sont bien sur envisageables.
En conclusion la solution retenue pour notre modèle est une solution totalement dynamique. En cours de vie un composant peut avoir besoin des services d'un autre composant et les lui demander. De plus nous choisissons de construire une solution distribuée et non centralisée par l'utilisation du deuxième scénario. Ce choix découle du fait que nous voulons construire un modèle le plus générale possible et capable de répondre à la plupart des besoins. Dans le paragraphe suivant nous allons formaliser le modèle de notre annuaire.

2.4.1 L'absynt de notre annuaire

Un annuaire est un composant qui contient la liste des composants du système. Un composant du système est, pour notre annuaire, une référence et une liste d'interfaces contenant des services fonctionnels.

```
 Formalism of directory is
Root is Top;
Operator and type definitions {
  Top = directory (ComponentType[] componentType);
  ComponentType = componentType(Interface [] interface);
}

Attribute definitions {
REQUIRED name as java.lang.String in ComponentType, directory, interface;
REQUIRED reference as java.lang.String in ComponentType;
}
```

Mais cet annuaire est aussi un composant. On pourrait donc le faire hériter du composant de base défini dans le chapitre décrivant le modèle de composant. Pourtant l'absynt que je viens de définir est spécifique au métier de ce composant. Ce phénomène se reproduira pour chaque métier: il va donc falloir étendre notre modèle de composant pour pouvoir prendre en compte le définition formelle des métiers:
Formalisme :*latin formalis, attachement excessif aux formes, aux formalités. Mettre sous forme de signes logiques ou mathématiques rigoureusement définis.*
Le formalisme est la description formelle de la partie métier du composant.

```
 Formalism of complexComponent is
```

```
Root is Top;
Operator and type definitions {
  Top = complexComponent () extend component;


  Metier = metier( Facade facade, Formalism? formalism);
    Facade = facade(), methods (Method [] method);
      Method = method (Arg [] arg);
        Arg = arg();
    Formalism = formalism();
}

Attribute definitions {
REQUIRED name as java.lang.String in metier, facade, methode, formalism;
}
```

Prenons comme exemple notre annuaire, nous obtenons alors un composant dont le formalisme n'est rien d'autre que l'absynt du directory. Lorsqu'un composant invoque l'annuaire il peut chercher un service, dans ce cas il attend une réponse (la référence du composant rendant ce service fonctionnel) et le port associé doit utiliser une communication synchrone. On peut aussi vouloir ajouter un service dans la liste de l'annuaire (en ajoutant un composant ou complétant une interface d'un composant existant) ou encore supprimer un service.

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="Annuaire" type="composant" extends="">
    <metier name="annuaire">
        <facade name="facade"/>
        <formalism name=directory.absynt/>
    </metier>
    <container name="conteneur">
        <interface name="interface" type="typeInterface">
            <message name="findService''  port=''port1'' method=''getService''>
               <parameter  name=''nameService '' type=''service''/>
            </message>
            <message name=''addService'' port=''port2'' method=''''addService/>
               <parameter name =''nameService'' type=''service''/>
               <parameter name =''nameComponent'' type=''Component''/>
               <parameter name =''nameInterface'' type=''Interface''/>
            </message>
            <message name=''deleteService'' port=''ina'' method=''deleteService''/>
               <parameter name =''nameService'' type=''service''/>
               <parameter name =''nameComponent'' type=''Component''/>
               <parameter name =''nameInterface'' type=''Interface''/>
            </message>
```

```
        </interface>
    </container>
    <services>
        <service name="securite"/>
    </services>
    <ports>
        <port name=''port1'' mode=''synchrone'' cardinality=''1-n'' flux=''entree''
                                    constraint=''optionnel'' protocol=''protocol1''/>
        <port name="port2'' mode=''asynchrone'' cardinality=''1-n'' flux=''entree''
                                   constraint=''optionnel'' protocol=''protocol2''/>
    </ports>
</component>
```

2.5 Assembler des composants

Les points précédents nous ont permis de décrire le noyau minimal de notre modèle. Nous allons maintenant l'étendre en essayant de mettre en avant sa puissance.

Nous avons vue ce qu'est un composant et comment il communique avec d'autres composants. Intéressons nous maintenant à la manière de les lier entre eux, c'est à dire de créer une application à partir de différents composants. Dans un premier temps nous allons réfléchir à la définition d'un langage permettant de spécifier les liens entre composants. Puis nous chercherons à exprimer le dynamisme de l'architecture dans notre langage.

2.5.1 Définition d'une ébauche d'ADL pour notre modèle

D'après [29] la définition admise est qu'un ADL spécifie les composants d'un système, leurs interfaces, les connecteurs (en tant que lieu d'interaction entre les composants) et la configuration architecturale. Les ADL permettent de décrire la structure de l'application, les communications entre composants logiciels ainsi que le schéma d'instanciation des composants au fur et mesure de l'exécution de l'application. Dans [2] les auteurs reprennent les définitions données par Medvidovic et Taylor. Un composant est défini par son interface (les messages échangés), son type (la partie implantation), sa sémantique et ses contraintes (les deux derniers points concernent le comportement du composant, l'aspect dynamique, les liens obligatoires avec d'autres composants...). Le connecteur définit les règles d'interactions entre les composants. Il possède une interface (notre port), un type (la définition du protocole utilisé), des contraintes (les limites d'utilisation du protocole). Vient enfin les configurations d'architectures d'applications qui définissent la structure et le comportement des applications formées de composants et de connecteurs. Les auteurs proposent ensuite une étude de différents langages. UniCon qui permet de définir des composants, des connecteurs et de les assembler mais qui dépend de l'implantation. Darwin qui

prend en compte l'aspect dynamique en décrivant les interactions entre composants. Rapid permet de simuler le comportement de l'application définie. C2 est, à l'origine, dédié aux interfaces graphiques. Il permet de définir le composant par son interface (décrivant les messages acceptés par les ports), ses méthodes et son comportement (explicitant ce que le composant doit faire à la réception d'un message et notamment les méthodes à appeler). L'architecture contient les composants impliqués, la définition des connecteurs (avec la politique de filtrage qu'ils doivent appliquer aux messages) et la topologie proprement dite qui indique quels ports sont reliés par quels connecteurs. C2 fournit aussi un moyen d'exprimer l'ajout et le retrait de composant dans l'architecture. C2, comme Olan ont pour particularité de permettre la réutilisation de logiciel existant en l'encapsulant pour en faire un composant. Olan contient aussi une notion de connecteur permettant d'exprimer avec qui et comment les composants communiquent. De plus cet ADL prend en compte la notion de dynamicité, prévoyant pour un composant client la possibilité d'instancier un autre composant à n'importe quel moment, de déclarer une instance d'un composant qui ne sera réellement instanciée qu'au premier appel.... ACME veut unifier les concepts utilisés dans les autres ADL. IL spécifie les architectures de manière syntaxique sans se focaliser sur la sémantique. D'après [25] il définit sept types d'entités: les composants, les connecteurs, le système (l'application, le résultat de la connexion des composants par les connecteurs), les ports (points d'interactions entre le composant et son environnement), les rôles (ils représentent les participants à une connexion), les représentations (différents niveaux de représentation d'une architecture), les repmaps (permettant de lier les ports des composants internes d'un composant composite avec les ports externes de ce dernier). ACME déclare que ces sept types sont suffisants pour décrire une architecture de composants. A celà doivent être ajoutées des informations auxiliaires permettant de répondre à des besoins spécifiques tels que la sémantique du système, lest types de données échangées, les protocoles de communication... Ce là se traduit dans ACME par des listes de propriétés attachées aux sept éléments de base.
La définition des ADL est une définition riche dont nous ne retiendront pour l'instant qu'une petite partie, celle permettant de décrire la topologie des applications. Parmi les modèles étudiés les EJB et le Corba Component Model disposent de fichier de déploiement leur permettant de préciser au démarrage les composants devant se connecter entre eux. Le plus souvent ces fichiers ont la forme de fichier jar. Pour les EJB [27] décrit le descripteur de déploiement, il contient entre autre le nom du composant, le nom du package contenant sa classe, le type de transactions supportées par le composant, la sémantique des messages échangés, la description des politiques de sécurité, des autorisations pour les méthodes... Le modèle fractal décrit un ADL [24] dont un exemple d'utilisation est donné ci-dessous:

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client">
```

```
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

Cet exemple définit un composant composite "HelloWord". Il a une interface, un rôle et une signature. Vient ensuite la description des composants qui le forme (ici les composants client et server); nom, interface et façade; et la définition des liens entre ces composants à partir du mot clé "binding" (chemin de communication entre les interfaces des composants).
SmartTools, quand à lui, définit un langage permettant d'exprimer les liens entre des composants particuliers, des vues, et des composants logiques (les vues permettent un affichage "lisible" pour l'utilisateur du composant logique). Il s'agit d'un langage XML permettant de décrire l'interface utilisateur associée à une application. l'interface graphique est constituée de composants de type vue (voir le paragraphe sur le modèle de composant de SmartTools). Il s'agit donc de lier un/des composants logiques avec des composants vues. La définition de LML est la suivante:

```
formalism of lml is

Root is Top;

Operator and type definitions {
  Top= Layout(Frame[] frame);

  Frame = frame(Set[] set);

  Set = split(),
        view();
}

Attribute definitions{
REQUIRED title as java.lang.String in frame, set, view;
REQUIRED StatusBar as java.lang.String in frame;
REQUIRED width as java.lang.String in frame;
REQUIRED height as java.lang.String in frame;
REQUIRED dynTabSwitch as java.lang.String in frame;
REQUIRED orientation as java.lang.String in split;
REQUIRED position as java.lang.String in split;
```

```
REQUIRED viewType as java.lang.String in view;
REQUIRED docRef as java.lang.String in view;
REQUIRED styleSheet as java.lang.String in view;
REQUIRED transform as java.lang.String in view;
REQUIRED behavior as java.lang.String in view;
}
```

Ainsi on a une ou plusieurs fenêtres, chacune disposant d'un titre, d'une barre de statuts en haut de la fenêtre, d'une hauteur et d'une largeur. Cette fenêtre contient un ensemble de Set, qui sont soit une vue soit deux vues côte à côte. Les vues ont un titre, un type (cet attribut permet d'instancier la vue), la référence sur le composant logique affiché dans la vue (une vue dans SmartTools est une représentation graphique d'un composant logique), le style d'affichage, la méthode utilisée pour transformer l'AST du composant logique et un fichier décrivant les actions à ajouter à la vue pour l'adapter parfaitement au composant logique qu'elle représente (behavior).
LML défini donc un binding (un connectTo pour SmartTools) entre les composants vues et les composants logiques de la manière suivante:

- instanciation des composants selon les attributs spécifiés;
- vérification que toutes les conditions nécessaires sont remplies pour démarrer les composants;
- extension dynamique des composants vues par les composants logiques à l'aide de "behavior"

Voici un exemple de fichier lml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="SmartTools: model demo" statusBar="on"
         width="1000"
         height="800"
         dynTabSwitch="off">
    <set title="model example for this GUI">
      <split orientation="1" position="68">
        <view title="Layout model.absynt"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="file:contrib/components/model/model.absynt"
              styleSheet="resources:css/absynt-styles.css"
              transform="resources:xsl/absynt-bml.xsl" />
       <view title="Layout example"
              behavior=""
      viewType="fr.smarttools.core.view.GdocView"
      docRef="file:demos/model/model.model"
              styleSheet="resources:css/xml-styles.css"
      transform="resources:xsl/genericXml.xsl" />
       </split>
```

```
    </set>
  </frame>
</layout>
```

Deux notions se retrouvent ici, une notion d'affichage graphique permettant d'ordonner les différentes vues (à travers les frame, set et split), et une notion implicite d'ADL puisque ce fichier permet de spécifier la connexion entre un composant logique et un composant vue (la première vue est connectée au document logique contenant l'absynt du composant model, la seconde à une instance de ce model).
En conclusion Fractal définit d'abord les composants puis leur liens. SmartTools quand à lui admet les connexions comme implicite et s'attache plus particulièrement à décrire l'organisation des composants dans l'interface graphique.
Nous allons maintenant proposer une première version de notre ADL, version n'exprimant que les connexions établies au démarrage de l'application:

```
<definition name="Modele">
  <binding>
     <Instance name="vue1" interface=''interface1'' attribut=''attribut1''/>
     <Instance name="model.absynt" interface=''interface1'' attribut=''attributAbsynt''/>
  <binding/>
  <binding>
     <Instance name="vue2" interface=''interface2'' attribut=''attribut2''/>
     <Instance name="model.model" interface=''interface2'' attribut=''attributModel''/>
  <binding/>
</definition>
```

Nous venons de définir l'application "Modele" en terme de composants interconnectés. Binding définit une connexion entre une liste de composants; les instances de composants sont identifiés par leur nom, leur interface et possèdent des attributs utiles à l'instanciation. Dans notre modèle nous privilégions une approche dans laquelle la description des composants et celle de l'application sont séparées. Cette approche nous parait plus propre car elle rend plus aisée la réutilisation des composants dans des applications différentes, et permet l'évolution d'une partie du modèle indépendamment des autres.

2.5.2 Une architecture dynamique

Nous cherchons maintenant à introduire la notion de dynamisme dans la composition, c'est à dire à prendre en compte les changements de l'application au cours de son exécution. [8] différencie trois catégories de dynamisme:

1. le dynamisme interactif dans lequel les connexions et les composants sont fixes mais les données varient;
2. le dynamisme structurel qui concerne l'ajout et le retrait de composants;
3. le dynamisme architectural permettant de changer la configuration des composants.

Le premier point ne nous intéressant pas, nous allons nous appliquer à obtenir le dynamisme pour les autres points.
Pour [10] un modèle de système de composants doit posséder les caractéristiques suivantes pour prendre en compte la composition dynamique:

1. Un composant ne doit agir avec son environnement que selon des interactions "bien définies" (ce qui est réalisé par l'encapsulation et les interfaces);
2. Un composant doit pouvoir être identifié sans ambigu¨é et de manière unique;
3. Le modèle de composant doit permettre la composition dynamique ou l'assemblage de composants pour former un nouveau composant de niveau supérieur. (La composition dynamique est rendu nécessaire par le besoin d'ajouter des services en cours d'exécution, de remplacer un composant défaillant...)
4. Le modèle doit permettre la gestion des ressources systèmes (gestion du multi-threading, multiplexage des connexions, partage de données répliquées sur plusieurs serveurs...);
5. Il doit fournir des contrôleurs, il s'agit de composants capables de gérer et de contrôler l'exécution d'un ensemble de composants. Pour celà il peut intercepter les messages afin d'y inclure des post et pré conditions modifiant le comportement du composant (ce qui rejoint le troisième point);
6. IL faut avoir la possibilité de modifier dynamiquement l'assemblage existant des composants (création de nouveaux composants simples, de nouvel assemblage de composants, migration d'un composant -simple ou composite- vers un autre assemblage de composants).

Notre modèle répond déjà aux deux premiers points: les interfaces permettent de définir les interactions avec l'environnement et le nom est un identifiant. La notion de contrôleur n'a pas été traitée spécifiquement dans notre modèle mais on voit qu'il est facile de l'intégrer au niveau du conteneur. Le point trois concerne la notion de composant composite qu'il nous faudra développer pour compléter le modèle proposé. La gestion des ressources système ne sera pas étudiée ici. Notons que le modèle que nous proposons n'est pour l'instant qu'à l'état débauche et que ce stage ne m'offre pas le temps nécessaire pour le compléter.
xADL2.0 [15] propose de distinguer le design-time model, un modèle statique de l'architecture; du run-tume model, un modèle dynamique permettant (pour l'instant) de définir des composants optionnels (à l'instanciation une condition de garde est évaluée pour savoir si l'élément optionnel doit être instancié ou non). Il prend aussi en compte la possibilité d'avoir différentes architectures (variants) et l'évolution des éléments de l'architecture (versions).

Reprenons ici l'étude des ADL prenant en compte la notion de dynamisme: Darwin, Rapide et Olan auxquels on peut ajouter l'ADL de Fractal qui tente de répondre au dynamisme architectural.

2.5.3 Le dynamisme dans l'instanciation des composants

Darwin peut spécifier deux types d'instanciation de composants. La première, l'instanciation dynamique, permet de spécifier la création d'une instance de composant en lui fournissant des paramètres d'initialisation. La seconde, l'instanciation paresseuse, est une pré-déclaration des instances effectivement créées, non pas lors de l'initialisation du composite, mais lors d'un premier appel vers l'instance. Néanmoins, cette forme d'instanciation n'est pas paramétrable comme l'instanciation dynamique.
Voici un exemple tiré de [4]. Il s'agit de construire un composant lazypipe qui est une composition d'un nombre indéterminé de filtres. Le composant filter est un composant primitif, il ne déclare que ses ports. Le composant lazypipe est un composant composite, qui déclare ses ports, les instances des composants qui le composent et le schéma d'interconnexion de ces instances.

```
component filter {
        provide left <port, int>;
        require right <port, int>,
                output <port, int>;
}

component lazypipe {
        provide input;
        require output;
        inst
              head : filter;
              tail : dyn lazypipe ; //recursivit, on instantie le composant lui-meme
        bind
              input -- head.left;
              head.right -- tail.input;// l'utilisation de cette communication declenche
la creation du composant tail, instances de lazypipe.
              tail.right -- output
              head.output -- output;
}
```

Dans cet exemple, l'instance queue est déclarée mais pas effective. La première fois qu'une communication partant de la tête vers la queue a lieu, le composant queue est créé puis le service input est appelé. Dans cet exemple il est impossible de fournir des paramètres d'initialisation aux composants crées. Pour celà un autre mécanisme est disponible dans Darwin, l'instanciation dynamique. Voici un autre exemple tiré de [4]:

```
Component client (int nbAnnuaire) {
         require lookupAnnuaire[nbAnnuaire] <port, string, string >;
}
Component Annuaire(int pays){
         provide lookup <port, string, string>;
```

```
}
component ClientFactory{
require createClient <component>;
}

Component Application {
       const int nbMaxAnnuaire = 2;
 // createur de clients
       inst clientFact : ClientFactory;
// ensemble d'annuaire
       array Annuaires[0..nbMaxAnnuaire-1] : Annuaire;
//creation dynamique de clients
       bind clientFact.createClient -- dyn Client(nbMaxAnnuaire);
       forall k:[0..nbMaxAnnuaire-1] // itrateur, cre les annuaires
                                     // et les interconnexions entre tout
                                    // client et les annuaires
              inst Annuaires[k] : Annuaire(k);
              bind Client.lookupAnnuaire[k] -- Annuaires[k].lookup;
}
```

Chaque client peut utiliser deux annuaires particuliers. Il cherche le nom à trouver dans l'un ou l'autre. Il existe aussi un composant de création de client (ClientFactory) qui permet de créer dynamiquement un nouveau client désirant utiliser un annuaire. L'application est formée d'un ensemble de clients dynamiquement créés. Il n'est pas possible de connaître le nombre de clients a priori puisque celui-ci est choisi par la mise en oeuvre du client. Néanmoins un nombre maximum d'annuaire est spécifié dans la configuration. A chaque fois que le service de création createClient est appelé, un client est créé.
Étudions maintenant OCL, un langage offrant lui aussi des possibilités pour gérer dynamiquement les composants d'une application. Olan propose un modèle d'assemblage logiciel et OCL est le langage de configuration dérivant de ce modèle. OCL a des notions de composants simples et composites. Dans OCL, deux concepts existent : l'instanciation paresseuse identique à celle présente dans Darwin et l'instanciation dynamique permettant de créer ou de détruire des instances de composants à la demande. Cette instanciation est rendue possible par le concept de "Collections". Ce sont des ensembles, bornés ou non, de composants ayant la même interface. Une collection permet d'ajouter ou de supprimer des composants en cours d'exécution. Il existe deux connecteurs spécifiques, qui lorsqu'ils sont utilisés pour la communication déclenchent la création ou la suppression d'une instance de composant dans la collection: CreateInCollection et DeleteInCollection.

```
interface ClientItf {
         provide Init();
         require Lookup_Annuaire( in string pays,in string cle, out string adresse);
         require Create_Annuaire(in string pays);
```

```
        }
...
implementation AppliImpl : AppliItf
          use AnnuaireItf, AdminItf, ClientItf
   {
                client = dyn instance ClientItf(); // instantiation paresseuse
                annuaires = collection [0..n] of AnnuaireItf;
                admin = instance AdminItf();
...
                client.Create_Annuaire(p) -> annuaires.Init() using createInCollection(p);
... }
```

Dans l'exemple précédent, l'application demande la création de l'objet client lors de sa première utilisation (l'instanciation paresseuse) et ajoute un annuaire dans la collection d'annuaires auxquels se connecte le client (utilisation du connecteur createInCollection).
Dans SmartTools il existe un mécanisme permettant de créer un composant à la demande d'un autre composant, le composant initiateur pouvant transmettre des paramètres pour l'instanciation.

2.5.4 Le dynamisme dans les connexions entre composants

Rapide propose un mécanisme permettant de connecter des composants à la demande de l'un d'entre eux. Prenons l'exemple de [4]. Un client veut se connecter à l'annuaire d'un pays particulier. Pour celà il envoie un message (une application est construite sous la forme de modules ou composants communiquant par échange de messages ou d'événements) indiquant de quel pays doit être l'annuaire et se connecte à l'annuaire qui renvoie la bonne réponse:

```
connect
      ?C.Lookup_Annuaire(?cle, ?pays) and ?A.Pays() == ?pays ||> ?A.Lookup(?cle);;
```

C est le client, Lookup-Annuaire une fonction du client permettant de rechercher un annuaire en fonction de sa clé et de son pays. A est un annuaire et Pays une fonction retournant le pays de l'annuaire. Si le pays demandé est le même que celui de l'annuaire alors la connection a lieu.

2.5.5 Notre proposition

Des modèles précédents nous retiendrons:

1. L'instanciation par nécessité, c'est à dire au premier appel vers le composant;
2. La possibilité pour un composant de demander l'instanciation d'un autre composant et d'agir sur sa configuration;
3. la possibilité de transcrire les modifications de l'application au cours de son exécution :création et suppression de composants, interconnexions se

modifiant dynamiquement en fonction des propriétés des composants existants dans l'application, etc;

4. la possibilité de définir des composants optionnels, notamment par l'utilisation de conditions de garde.

On trouve dans cette liste deux notions, une notion d'architecture (la description de l'assemblage des composants) et une notion temporelle (schéma d instanciation: comment les composants sont ils créés dans le temps).

```
<definition name="Application">
  <component>
     <Instance name="vue1" interface=''interface1'' attribut=''attribut1''/>
     <Instance name="modelAbsynt" interface=''interface1'' attribut=''attributAbsynt''
     <Instance name="modelModel" interface=''interface2'' attribut=''attributModel''/>
      <Load name=''vueL'' interface=''interface2'' attribut=''attributL'' >
  </component>
  <optional>
     <if =''modelAbsynt existe''/>
         <Instance name="vue1" interface=''interface1'' attribut=''attribut1''/>
  </optional>
  <binding modelAbsynt.interface1 -> vue1.interface1/>
  <binding modelModel.interface2 -> vueL.interfaceL/>
</definition>
```

Le modèle initial a évolué. Les instances de composants sont maintenant déclarées dans une partie nommée "component". Le mot clé "Instance" permet de déclarer les composants à instancier au lancement de l'application (les instances existent et sont identifiées par un nom). Le mot clé "Load" permet de déclarer des composants qui ne seront réellement instanciés que lors de leur premier appel (ces composants sont chargés mais pas instanciés). Une seconde partie, délimitée par le mot clé "optional" permet de déclarer des composants qui ne seront instanciés que si la condition de garde décrite dans le "if" est respectée. La partie suivante permet de déclarer les connexions entre composants, chaque connexion étant précédée par le mot clé binding, l'initiateur de la connexion étant l'instance de gauche et le destinataire l'instance écrite à droite du signe "-¿".
Nous avons ainsi défini un langage permettant de décrire la topologie de base de l'application. Les propriétés de dynamicité permettent d'exprimer son évolution en cours d'exécution. Ainsi un composant voulant se connecter à un autre non encore présent dans l'application peut le faire charger par la déclaration "load" et se connecter à lui en déclarant un "binding".

Pour réaliser des applications distribuées il faudra compléter le langage pour qu'il puisse nous permettre de décrire la localisation géographique des composants. Olan [4] permet de définir le site sur lequel s'exécute le composant. Le site est choisi à partir de critères détaillés tels que son nom, son adresse IP, le type de système d'exploitation, la charge CPU moyenne du site au moment du déploiement, la charge utilisateur moyenne du site au moment du déploiement.

Il est aussi possible de laisser le choix du site au support d'exécution. Dans un premier temps nous choisissons de laisser la possibilité d'indiquer le lieu (identifié par son nom et son adresse IP) d'insanciation du composant au moment de sa création, ou de laisser faire le système. La déclaration des instances de composants devient alors:

```
<component>
    <Instance name="vue1" interface=''interface1'' attribut=''attribut1''
                            nameNode=''noeud1'' IPAdress=''adressseIP1''/>
    <Instance name="modelAbsynt" interface=''interface1'' attribut=''attributAbsynt''/>
    <Instance name="modelModel" interface=''interface2'' attribut=''attributModel''/>
     <Load name=''vueL'' interface=''interface2'' attribut=''attributL''
                           nameNode=''noeud2'' IPAdress=''adressseIP2''/ >
 </component>
```

Il s'agit bien sur d'une ébauche de proposition. Comme toute ébauche elle est incomplète et comme toute proposition criticable. Elle nous a permis mal grès tout d'exprimer la topologie de l'application en fournissant les moyens d'y intégrer des notions de dynamisme permettant de décrire son évolution au cours du temps.

2.6 L'ajout de services non fonctionnels

Dans les chapitres précédents nous avons proposé un modèle de composant puis nous l'avons fait "vivre" en exprimant le processus de communication entre les composants, créant une structure évolutive. Mais un composant peut aussi avoir besoin d'évoluer. Construit pour des besoins particuliers la nécessité de le modifier peut apparaître au cours de sa vie. On peut alors choisir de stopper les services offerts par ce composant le temps d'effectuer les modifications, mais alors on ne répondrait pas aux besoins de toutes les applications qui ne peuvent tolérer aucune interruption. C'est pourquoi dans ce chapitre nous allons réfléchir à une proposition pour ajouter des services à des composants de manière dynamique. L'évolution est à l'initiative du développeur ou du système lui-même, capable de prendre en compte les modifications de son environnement.
Citons d'abord le projet Fractal qui définit des politiques d'adaptation à un contexte particulier. La programmation par aspect [11], quand à elle, consiste à définir dans le code de l'application des points de jonctions (en début de méthode, de boucle...) sur lesquels on pourra faire exécuter le code des aspects que l'on veut ajouter, Jac utilise la programmation par aspect pour développer des applications réparties en Java. Dans Javapod les aspects non fonctionnels sont désignés comme des extensions. Ces extensions sont ensuite composées pour produire le conteneur (il est chargé d'assurer les aspects non fonctionnels nécessaires au composant). Un modèle de composition a été défini pour répondre à ce but. Ils peuvent être ajouter ou retirer de manière dynamique. SmartTools quand à lui, utilise la notion de "behavior". Il s'agit d'un fichier permettant de décrire des actions et les messages associés, ces derniers pouvant avoir des

attributs. Des menus sont ensuite déclarés dont les boutons correspondent aux actions définies plus haut. C'est un mécanisme permettant d'étendre des composants de type "vue" en ajoutant des menus permettant de réaliser des actions qui ne font pas partie du comportement par défaut du composant. On obtient un ajout de service sur un message. Voici un exemple partiel du behavior de Component Manager:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<behaviors>

  <actions>
    <action name="Quit">
      <msg name="exit"/>
    </action>
    <action name="Connect To CM">
      <msg name="connectTo" chooser="cm.ConnectToDialogBox">
        <msgattr name="id_src"
                 value="ComponentsManager"/>
        <msgattr name="type_dest"
                 value=""/>
      </msg>
    </action>
    <action name="Debug">
      <msg name="connectTo">
        <msgattr name="id_src"
                 value="ComponentsManager"/>
        <msgattr name="type_dest"
                 value="worldStateGraph"/>
      </msg>
    </action>
...
  </actions>

  <menus>
    <menu name="Component">
      <item action="Add Component"/>
      <item action="Available Components"/>
      <item action="Connect To CM"/>
      <item action="Reload Component"/>
      <item action="separator"/>
      <item action="Debug"/>
      <item action="separator"/>5
      <item action="Serialize World"/>
    </menu>
...
```

```
  </menus>

  <toolbar>
  </toolbar>

  <keyb>
  </keyb>

  <mouse>
  </mouse>

</behaviors>
```

Nous retiendront ici qu'à un message donné correspond une action à réaliser (un appel de méthode):

```
    <action name="Connect To Component">
      <msg name="connect" >
        <msgattr name="comp1"/>
      </msg>
    </action>
```

Ce chapitre n'offre qu'une ébauche de l'extension de service et aurait mérité d'être approfondi. Le temps m'étant compté son étude approfondie est reportée à plus tard. On pourrait faire un parallèle avec le chapitre sur LML. Ce langage, sous l'apparence de définir une interface graphique cache un outil de description de l'architecture de l'application. Ici derrière la description des menus et autre boutons se cache un mécanisme d'extension de service que nous aimerions rendre explicite.

Conclusion

Au cours de ce stage d'initiation à la recherche j'ai cherché à proposer les concepts de base d'un modèle à composant. Je me suis appuyée pour celà sur les travaux existants et notamment sur le modèle de composant proposé par SmartTools; modèle qui s'est avéré être un modèle complet et puissant vis à vis des autres modèles de composants.
J'ai suivi l'approche MDA de création de modèle indépendant de la plateforme. J'ai défini un modèle de composant abstrait dont l'intérêt est de clarifier les concepts et le vocabulaire des modèles de composants. Ainsi j'ai reformulé de manière plus générique le modèle de SmartTools tout en prenant en compte les autres modèles de composant.
La suite logique de cette démarche est de montrer que ce modèle abstrait est projetable vers une technologie particulière. Pour celà j'ai proposé un langage de transformation d'un modèle à l'autre (encore simple et qui demande à être enrichi).
L'étude des spécificités du modèle de composant de SmartTools m'a amené à travailler sur l'architecture des application composants à travers les langages ADL (Architectural Description Langage) et le langage LML (LML est le langage permettant de décrire les vues attachées au composant. C'est à dire les composants capables de visualiser le composant métier) de SmartTools. Enfin j'ai initié un travail pour mieux comprendre les extensions de services mais il demande à être poursuivi et approfondi.

L'ébauche de transformation de modèle demanderai à être poursuivie. Dans SmartTools il existe déjà un outil de transformation appelé COSYNT (langage permettant de spécifier comment afficher la syntaxe concrète) du modèle qui pourrait être une base de départ à notre langage de transformation de modèle. En ce qui concerne les aspects innovant de SmartTools, et plus particulièrement l'extension de service, mon début de généralisation demanderai à être poursuivi à travers une implantation qui en montrerai tous l'intérêt.

Bibliographie

[1] Projet ACCORD. "Assemblage de composants par contrats. État de l'art de la standardisation"

[2] Projet ACCORD. "Etat de l'art sur les Langages de Description d'Architecture (ADLs)"

[3] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joël Fillon, Didier Parigot, Claude Pasquier, Claudio Sacerdoti Coen. "SmartTools: a Development Environment Generator based on XML Technologies"

[4] BELLISSARD LUC. "CONSTRUCTION ET CONFIGURATION D'APPLICATIONS REPARTIES", thèse.

[5] L. Berger. "Junction Point Aspect: A Solution to Simplify Implementation of Aspect Languages and Dynamic Management of Aspect Programs"

[6] Mireille Blay-Fornarino, Anne-Mary Dery, Olivier Nano, Michel Riveill. "AN abstract model for integrating and composing services in component platforms"

[7] Mireille Blay-Formarino, Anne-Marie Dery, Audry Occello, Michel Riveill. "Vers une adaptation dynamique cohérente des composants"

[8] Robert Pawel Bialek. "The Architecture of a Dynamically Updatable, Component-based System"

[9] E Brunetton. "Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties"

[10] E. Bruneton, T. Coupaye, J. B. Stefani. "Recursive and Dynamic Software Composition with Sharing"

[11] Noury M.N. Bouraqadi-Saâdani, Thomas Ledoux "Le point sur la programmation par aspets"

[12] E. Bruneton,T. Coupaye, JB Stefani http://fractal.objectweb.org/specification/index.html

[13] Eric Bruneton, Michel Riveill. "JavaPod: une plate-forme à composants adaptable et extensible"

[14] Thomas Bures. "Constraint-Based Generation of Connectors"

[15] Eric M. Dashofy, André van der Hoek, Richard N. Taylor. "A Highly-Extensible, XML-Based Archtecture Description Langage"

[16] Laurence Duchien, Gérard Florin, Renaud Pawlak, Lionel Seinturier. " JAC a flexible solution for aspect-oriented programming in java"

[17] Humberto Cerventes, Richard S Hall. “A conceptual comparaison of Avalon, Fractal and Gravity”

[18] Olivier Charra, Aline Senart. “Adaptable and extensible bindings in distributed environments”

[19] Philippe Collet, Roger Rousseau. “Contracting Hierarchical Components”

[20] Carine Courbis, Pascal Degenne, Alexandre Fau. “Un modèle abstrait de composants adaptables”

[21] Carine Courbis, Philippe Lahire, Didier Parigot. “Towards Domain-Driven Development: Approach and Implementation”

[22] Carine Courbis, Anthony Finkelstein. “Towards an Aspect Weaving BPEL Engine”

[23] May Dehayni, Louis Féraud. “Attribute grammars as tools for model-to-model transformations”

[24] http://fractal.objectweb.org/tutorials/adl/index.html

[25] David Garlan, Robert Monroe, David Wile. “ACME: An Architecture Description Interchange Language”

[26] http://java.sun.com/developer/technicalArticles/ebeans/IntroEJB/index.html

[27] http://java.sun.com/dtd/ejb-jar-2-0.dtd

[28] Entreprise JavaBeans Specification version 2.1

[29] Christophe Mareschal. “Adaptation d’un langage de descrition d’architecture à l’expression du comportement, applications.”

[30] Marvie R., Merle P. “Vers un modeèle de composants pour CESURE: le CORBA Component Model”

[31] Marvie R., Pellegtini M.-C.”Modèles de composants, un état de l’art”

[32] Philippe Merle, Mathieu Vadet. “ Les conteneurs ouverts dans les plates-formes à composants”

[33] OMG. “CORBA Component Model, v3.0”

[34] http://www.omg.org/docs/formal/02-06-69.pdf

[35] Projet accord, lot 1 livrable 4. ”Modele abstrait d’assemblage de composants par contrat”

[36] Projet Aspect. “Livrable 4, Assemblage dynamique de composants logiciels”

[37] http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/introduction