République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Informatique Académique

13/8 45

Thème :

Conception et Développement d'un Outil Graphique pour la Modélisation des Systèmes à base des Composants et des Aspects

Encadré Par :

Dr Hannousse A.H

Présenté par :

Aguibi Leyla

Nouiri Souhayla

Juin 2013

remerciment

C'est avec un grand plaisir que nous apportons ce modeste Travail à tous ceux qui nous ont gratifié de leur soutien et de leur Confiance.

***Louanges à Dieu,** qui nous a donné vie et santé pour le parachèvement de ce travail. Et nous voudrions exprimer toute notre gratitude à :*

***Monsieur Hanousse** .A pour la confiance qu'ils nous ont témoigné en acceptant de diriger ce travail et pour nous avoir accordé de son temps et avoir mis à notre disposition ses compétences et ses conseils pour une meilleure maîtrise du sujet.*

Tous les enseignants du département d'informatique,** qui ont assisté à nos débuts en informatique, pour leurs précieux conseils. Spésialement **Monsieur bourahouma.N

***Nos collèques de fin de cycle,** qui nous ont donné leurs encouragements toute la durée de réalisation de ce travail.*

***Nos familles,** qui durant nos études, nous ont toujours donné la possibilité de faire ce que nous voulions et qui ont toujours cru en nous.*

Toutes les personnes qui nous ont aidé et soutenu de prés ou de loin tout le long de ce travail.

Souhayla&leyla

Dédicace

Je remercie dieu de m'avoir donné le courage, la volonté, la force et la patience pour réaliser ce modeste travail, que je dédie :

A mon très cher père AbdAziz

Pour tout ce qu'il a fait pour moi il a toujours à mes cotes à

M'encourager et m'aide.

A mon très chère mère linda

Pour leur encouragement et leur sacrifice et attention durant la période de mes études, pour toute affectation q'elle m'a donnée.

« Que die vous garde dans le paradis, je vous aime mes parents »

A mon frère 'amine' et mes sœurs

A ma partenaire et mon binôme leyla qui était comme une soeur

A mes amis « tanissia, basouma,hanneneG,hanneneK,Nesrine, hiba, sara, meriem, aicha, soumia, nabila, Ahlem,Khaled,chamse,Abdraouf houssine,Aziz,Abdou ,Rahim Nadir »

A mes collègue Khetab Ahmed et Riad.

SouSou

Dédicace

Je remercie dieu de m'avoir donné le courage, la volonté, la force et la patience pour réaliser ce modeste travail, que je dédie :

A mon très cher père Saleh

Pour tout ce qu'il a fait pour moi il a toujours à mes cotes à

M'encourager et m'aide.

A mon très chère mère Fatma

Pour leur encouragement et leur sacrifice et attention durant la période de mes études, pour toute affectation q'elle m'a donnée.

« Que die vous garde dans le paradis, je vous aime mes parents »

A mes frères et ces femmes et ces enfants Arwa et Mouhamed raouf

A mes sœurs

A ma partenaire et mon binôme souhayla qui était comme une soeur

A mes amis Nesrine, hiba, Bessam, sara, meriem, aicha, soumia, nessma, hanane, nabila, Khaled.

A mes collègue Khetab Ahmed et Riad.

Leyla

Table des Matières

Table des Figures

Liste des abréviations

Abréviation	Désignation
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
MDA	Model Driven Architecture
OMG	Object Management Group
POA	Programmation Orienté Aspect
POC	Programmation Orienté Composer
CBSE	Component Based Software Engineering

Résumé

Les deux paradigmes de programmation (composants et aspects) montrent séparément leurs efficacités dans la séparation des préoccupations métiers (fonctionnement de base), des préoccupations techniques (performance, persistance, temps réel, etc.) des applications. Cette séparation améliore la réutilisabilité et facilite la maintenance des systèmes. Le mélange des deux paradigmes est une approche prometteuse assurant une meilleure réutilisabilité et maintenance des systèmes logiciels et matériels complexes. Différent modèles mélangeant les composants et les aspects ont été élaborés cette dernière décennie. Malheureusement, la plupart de ces systèmes resteront théoriques (manque des outils pratiques).

Dans le cadre de ce mémoire nous contribuons par le développement d'un outil graphique qui permet de modéliser les systèmes à base des composants et des aspectes et génère une description ADL qui peut être exécuté par une environnement d'exécution adéquat. À ce propos nous avons utilisées les outils MDA de l'environnement Eclipse pour : (1) définir un métamodèle générique, (2) décrit graphiquement des systèmes à composants et aspects sous forme des modèles équivalents au méta modèle, puis (3) générer la description ADL correspondante.

Introduction

De nos jours, nous vivons une évolution rapide dans le domaine de développement des logiciels. Chacun de sa manière, essaie d'améliorer la qualité de production durant le cycle de développement de logiciels en passant par toutes les étapes visant à atteindre les besoins et les objectifs des utilisateurs. Dans ce contexte, l'ingénierie des logiciels basée composants fait une séparation des préoccupations métiers du système en des entités clairement définies, appelées composants. Ces composants réutilisables sont définis et composés ensemble. Les modèles à composants utilisent un langage de description d'architecture (ADL : Architectural Description Language) pour décrire la topologie de l'application ; avec un langage ADL on décrit l'ensemble des éléments qui comporte le système et les différentes liaisons et connections entre eux. Un langage ADL permet ainsi à l'utilisateur de structurer les différents éléments qui comportent le système et ceci dans le but de simplifier la gestion de ce système et d'offrir une présentation plus claire et facile à interpréter.

D'autre part,la programmation par aspects est un paradigme de programmation qui concentre sur la modularité et réutilisabilité des préoccupations techniques (synchronisation, persistance, contraintes temps réels, etc.) qui peuvent apparaître dans différentes applications. Ces préoccupations particulières ne peuvent pas être encapsulées dans des entités uniques ordinaires (classe, méthode, objet ou même composant). Le paradigme aspect propose une entité nommée aspects pour l'encapsulation de ce type des préoccupations. Un aspect décrit le comportement de la préoccupation (advices) et les endroits où ce comportement doit être exécutée dans le système de base (points de jointures). L'aspect doit être tissé au système de base afin

d'obtenir le système final désiré, ce tissage peut être effectué d'une manière statique ou dynamique.

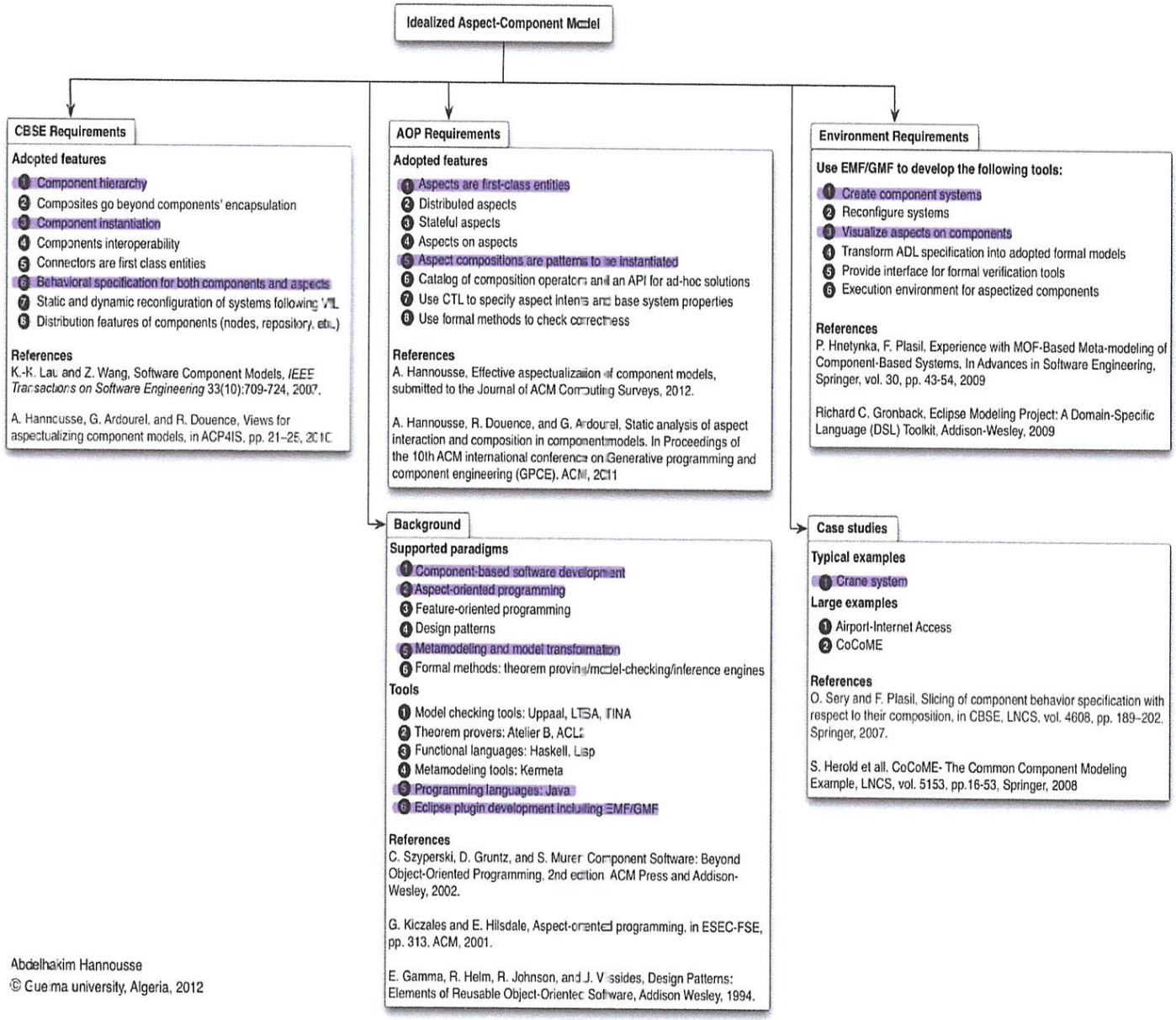
De ce fait, ces deux paradigmes semblent complémentaires puisque ensemble ils assurent une meilleur encapsulation et séparation des préoccupations métiers des préoccupations techniques des applications. Cette séparation améliore la réutilisabilité et facilite la maintenance des systèmes logiciels et matériels complexes. Différents

modèles mélangeant des composants et des aspects ont été élaborés. Malheureusement, ces différents modèles ont des limites : (1) manque de considération des comportements des éléments architecturaux (aspects et composants), une description structurelle est uniquement proposée ; (2) manque des outils de vérification en cas de présence de plusieurs préoccupations techniques en même temps ; (3) manque des outils graphiques permettant la visualisation de l'architecture globale des systèmes (aspects, composants, leurs comportements et interactions).

Notre contribution consiste à proposer un outil graphique qui permette d'une part à décrire des systèmes à composants et à aspects avec les interactions possibles entre les différents éléments architecturaux ; et d'autre part de visualiser et d'éliminer les conflits structurels qui peuvent apparaître dans le cas de tissage de plusieurs aspects à la fois. De façons générales, notre contributions fait partie d'un grand projet proposé par notre encadreur Dr. A.H. Hannousse dont le but est de proposer une meilleure composition des deux paradigmes aspects et composants afin d'assurer une meilleur réutilisabilité, modularité et maintenance des systèmes logiciels et matériels complexes. La figure ci-dessous montre les différentes étapes et objectifs de ce projet où les points indiquant notre contribution sont marqués en rouge. D'une manière plus précise, notre travail consiste à : (1) définir un méta-modèle pour la modélisation structurelle et comportementale des systèmes à base des composant et des aspects, et (2) générer une description ADL qui correspond aux systèmes compatibles avec le méta-modèle décrit dans la première étape. La description ADL généré peut être exécutée par des environnements d'exécutions adéquats comme celui de Fractal Julia [Bruneton, 2006].

Le présent mémoire est organisé en quatre chapitres. Les deux premiers chapitres introduits, respectivement, la programmation par aspects et la programmation par composants ; et le troisième décrit en détail notre contribution à la composition de ces

deux paradigmes en décrivant notre méta-modèle unifié des systèmes à composants et à aspects. Finalement, le chapitre quatre décrit l'implémentation de notre outil graphique.



Abdelhakim Hannousse

CHAPITRE I.

Programmation par Aspects

Les différents paradigmes de programmation existants aujourd'hui (fonctionnels, procédurales, et objets) visent a améliorer la modularité et la réutilisabilité des logiciels. Sachant que la modularité augmente la lisibilité et la compréhensibilité du code source des l'applications, ce qui facilite leurs maintenance ; la réutilisabilité réduit le temps nécessaire pour la disponibilité de l'application sur le marché (Time-to-Market). Par exemple, le paradigme objet propose d'une part l'encapsulation des préoccupations fonctionnelles des systèmes dans des entités distinctes appelées classes, cela rendre les systèmes plus modulaires ; et d'autre part, il permet la séparation de l'interface d'un

objet de son implémentation, ce qui augmente la réutilisabilité de ces objets.Cependant, certaines préoccupations non fonctionnelles telles que : la sécurité, la synchronisation, la persistance, etc. ne parviennent pas à être encapsulées dans des classes uniques et distinctes comme les autres ; elles se propagent sur plusieurs classes et dans différentes endroits dans la même classe. Ces préoccupations particulières sont appelées *des préoccupations transverses*. La programmation par aspects ne servira pas uniquement à nettoyer le code, mais aura un grand impact sur tout le processus de développement de logiciel. En fait, la programmation par aspects fournit une solution pour le problème d'encapsulation des préoccupations en général et les préoccupations transversesen particulier.

Dans ce chapitre, nous présentons le paradigme aspect en décrivons les problèmes rencontrées par les approches classiques de modélisation puis montrons en quoi la programmation par aspects propose une solution plus élégante.

1. Problèmes des approches classiques

Dans n'importe quel système logiciel en trouve deux types des préoccupations ou couches d'application, les préoccupations fonctionnelles, applicatives ou métiers et les préoccupations non fonctionnelles ou techniques. Le premier type des préoccupations décrit les fonctionnalités de base de système, alors que le dernier décrit l'ensemble des propriétés liées à l'exécution, la performance ou le contrôle de système.

Voici quelques exemples des deux types de préoccupations :

- *Préoccupations métiers* : fonction de calcul de la TVA dans une application comptable, fonction de calcul de charge salariale dans une application de gestion du personnel,
- *Préoccupations techniques* : gestion des données (persistance des données, contrôle d'accès aux données), débogage (génération de la trace), performance (contraintes temps réels), etc.

Dans les paradigmes de programmation classiques (procédurales ou objets), il est presque impossible d'éliminer le couplage qui existe entre ces deux types de

préoccupations dans une application donnée. Par exemple, le traçage d'exécution du code et l'authentification pour l'accès aux services d'un système peuvent apparaître à l'intérieur de ses différentes procédures ou méthodes. Cela fait soit par un simple appel à la méthode ou à la procédure implémentant la préoccupation technique ou carrément par l'ajout du bloc de code de la préoccupation ; ce couplage semble inévitable. De ce fait, à l'intérieur du code des préoccupations métiers, on trouve du code de gestion d'authentification et de traçage, etc. Les préoccupations métiers sont alors dépendantes des préoccupations non fonctionnelles ou techniques. La figure 1.1 illustre clairement ce problème de dépendance. Cette forte dépendance pose deux problèmes principaux : l'*éparpillement* et l'*enchevêtrement* de code.

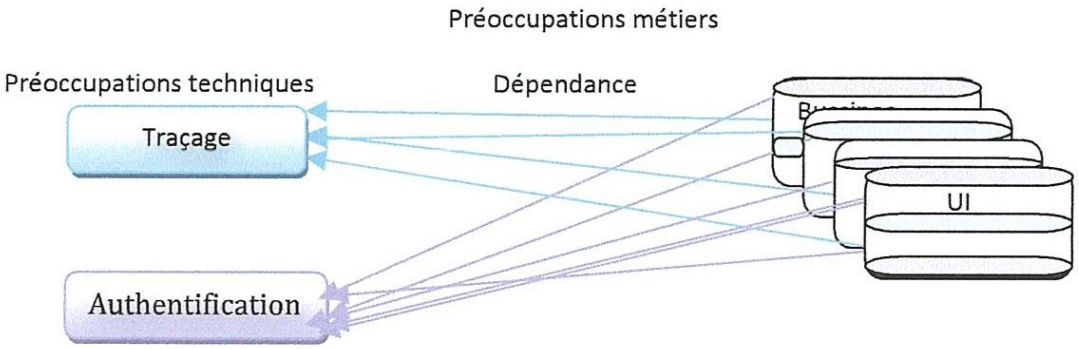


Figure 1.1. Problème du couplage des préoccupations métiers et techniques

1.1. L'éparpillement de code (*Code Scattering*) :

C'est le cas de la dispersion du code correspondant à une préoccupation technique dans la description des différents modules d'un système. Par exemple, dans un système utilisant une base de données, l'authentification est une préoccupation implémentée dans tous les modules qui accèdent à la base (Figure 1.2).

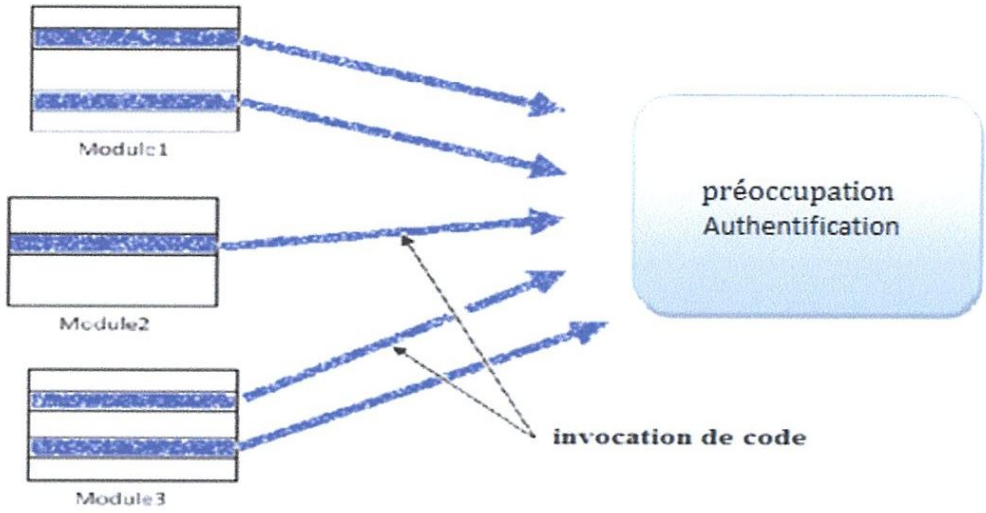


Figure 1.2. Éparpillement du code de la préoccupation Authentification

1.2. L'enchevêtrement du code (Code Tangling) :

C'est le cas de la présence simultanée des plusieurs morceaux de codes des différentes préoccupations techniques dans le même module, ce qui complique la compréhension du code de ce module et sa maintenance. Ce phénomène est appelé enchevêtrement du code (voir Figure 1.3).

Figure 1.3. Enchevêtrement du code

Combinés ensembles, l'éparpillement et l'enchevêtrement de code affectent la modélisation et le développement de l'application de nombreuses façons [Baltus2001] :

- *Traçage difficile* : Les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les différentes préoccupations et leurs implémentations,

- *Diminution de la productivité* : La prise en considération de plusieurs préoccupations au sein d'un même module empêche le programmeur de se focaliser uniquement sur son premier but, ce qui diminue sa productivité,

- *Diminution de la réutilisation du code* : Les modules implémentant à la fois des préoccupations métiers et autres préoccupations techniques ne peuvent pas être réutilisés tel quelles par d'autre systèmes nécessitant des préoccupations métiers similaires,

- *Maintenance et évolutivité du code difficile* : Lorsqu'on souhaite faire évoluer le système, on doit modifier de nombreux modules à la fois. Modifier chaque sous-système pour effectuer les modifications souhaitées peut conduire à des incohérences.

La meilleure solution à l'éparpillement et à l'enchevêtrement de code est de bien séparer les implémentations des préoccupations métiers des les implémentations des préoccupations techniques, c'est le but principal de la programmation par aspects.

2. La programmation par aspects

La programmation par aspects (AOP) [Kickzales 1997] est une nouvelle technique de programmation. Elle a été définie par Gregor Kickzales et son équipe (du laboratoire de recherche PARC de Xerox) en 1996, qui ont permis de simplifier l'écriture des programmes informatique en les rends plus modulaires et plus facile à évoluer. L'AOP peut être considérer comme étant une extension aux différents autres paradigmes.

La programmation par aspects permet d'implémenter les différentes préoccupations d'un système indépendamment les unes des autres, puis de les assembler selon des règles bien définies afin d'avoir le système final. En conséquence, l'approche AOP offre

une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements.

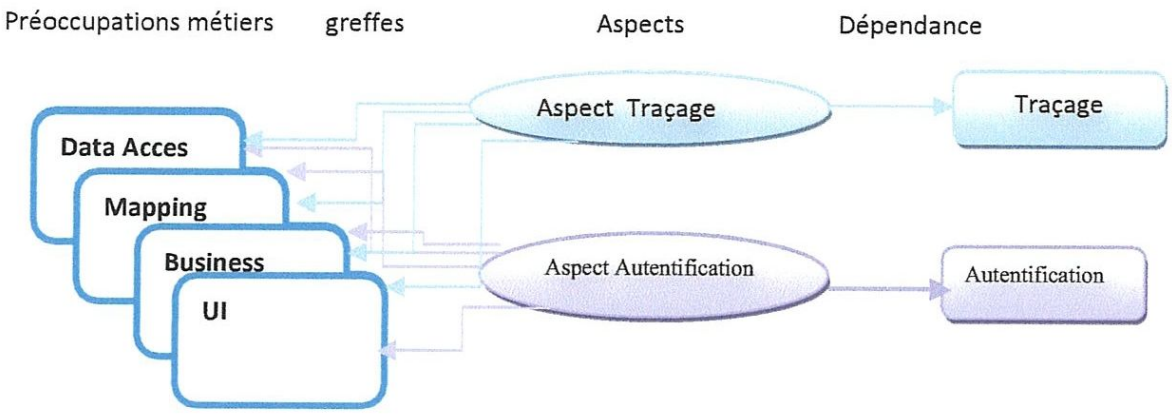


Figure 1.4. Séparation des préoccupations métiers des préoccupations techniques

Suivant l'approche AOP, le couplage entre les différentes préoccupations peut être réduit de façon très importante, ce qui présente de nombreux avantages [Gabsi2011]:

- *Maintenance aisée* : le code des préoccupations techniques peut être maintenus plus facilement du fait de son détachement de son utilisation,
- *Meilleure réutilisation* : toute préoccupation peut être réutilisée séparément dans d'autres systèmes nécessitants des fonctionnalités similaires,
- *Gain de productivité* : le programmeur ne se préoccupe que de la préoccupation du système qui le concerne, ce qui simplifie son travail.

2.1. Le modèle Général

Une application à base d'aspects est constituée d'un *programme de base* défini par un ensemble de modules implémentant les préoccupations métiers de l'application et d'autres modules indépendants nommés *aspects* chacun décrit une préoccupation technique. Le programme de base et les aspects sont développés séparément. Les aspects sont ensuite, tissés au programme de base à l'aide d'un outil spécifique nommé *tisseurd'aspects* (aspect weaver). Ce dernier produit un nouveau programme qui

contient les fonctionnalités de système de base et celles fournies par les aspects (Figure 1.5).

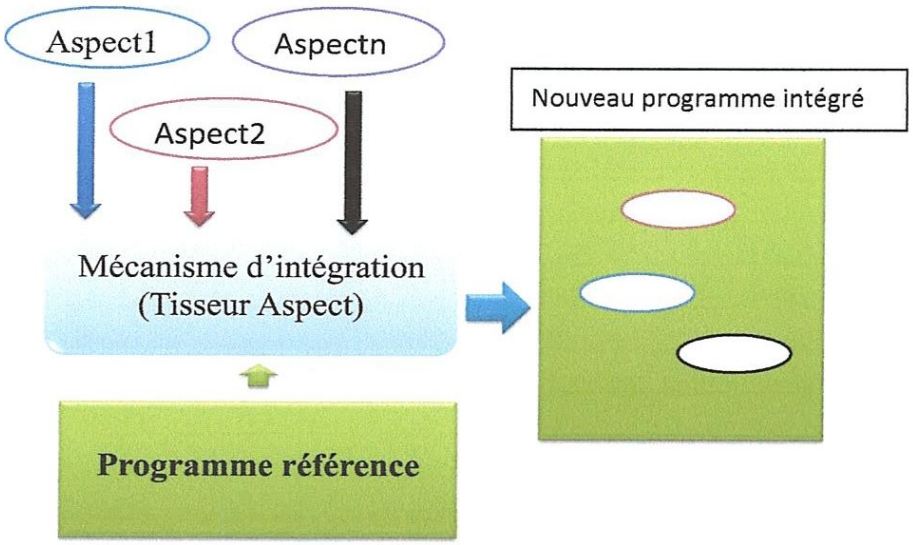


Figure 1.5. Modèle général de l'AOP

2.2. Fondements de l'AOP

D'une façon générale, la structuration d'une application dans le paradigme aspect repose sur sa décomposition en aspects encapsulant les préoccupations transverses, et en modules de base représentant les préoccupations métiers de l'application. Le système est ensuite construit par la composition de l'ensemble des aspects et les modules de base conformément aux *points de jointures*. Nous précisons dans ce qui suit les significations de chacun des termes *aspects* et *points de jointure* représentant les notions fondamentales de l'AOP.

2.2.1. Aspect :

C'est une unité de programmation permettant de modéliser et encapsuler une préoccupation transverse. Un aspect est un regroupement d'une ou de plusieurs définitions de *coupe*, d'une ou de plusieurs définitions de *conseil* et d'une ou de plusieurs associations de coupes à des conseils (Figure 1.6),

2.2.2. Coupe (pointcut) :

C'est un ensemble de *point de jointure* obtenu par une expression régulière,

2.2.3. Point de jointure (joinpoint) :

C'est un endroit précis dans l'exécution du programme définissent où l'aspect doit être intégré dans une application (un appel à une méthode, l'accès à un attribut, etc.),

2.2.4. Conseil (advice) :

C'est un bloc de code à insérer au niveau de points de jointure et qui implémente une préoccupation transverse. Un code advice peut être exécuté selon trois modes : avant, après, ou autour d'un point de jointure.

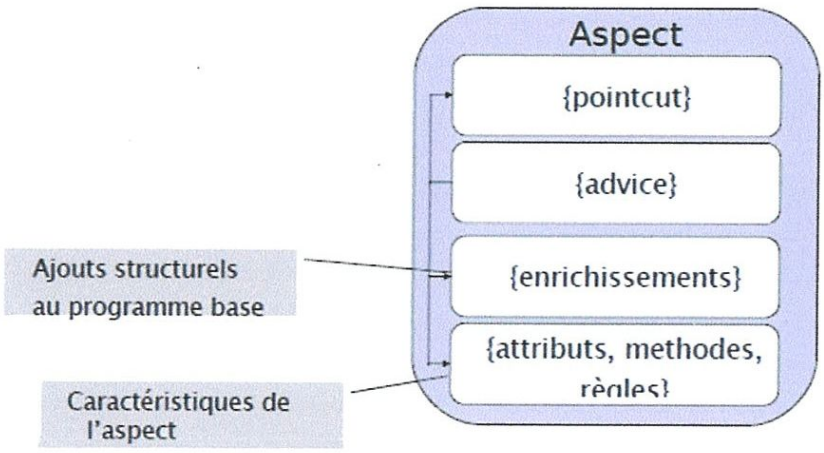


Figure 1.6. La structure d'un aspect

La Figure 5.1 montre un exemple simple de l'aspect logging en AspectJ[1] [Kiczales 2001]. Cet aspect définit un point decoupe, *loggableMethod* qui spécifie où la préoccupation de logging devrait joindre l'exécution de lafonctionnalité de base. Dans cet exemple, les points de jonction intéressants sont les appels (*call*) àtoutes les méthodes nommées *bar* indépendamment de la classe dans laquelle ils sont définis, de leurvaleur de retour, ou du nombre et du type des paramètres. L'objet qui exécute l'appel de méthodeest exposé au greffon par l'utilisation du mot-clé *this*.

[1] AspectJ est l'outil de référence implémentant la programmation par aspects en Java.

```
publicaspect Logging {

// où ?
pointcutloggableMethods(Object o): call (* bar(..)) &&this(o);

      // greffon (comportement à insérer) :
      before(Object o):loggableMethods(o) { // quand ?

            // quoi ?
            System.out.println("code appelé par l'objet " +
            o.toString());
      }
}
```

Figure 1.7. Un aspect de Logging en AspectJ

2.3. Le tissage d'aspects :

Un aspect défini des morceaux de code (les conseils) et les endroits de l'application où ils vont s'appliquer (les points de jointure). Un traitement automatique est donc nécessaire pour intégrer ces aspects dans l'application afin d'obtenir un programme fonctionnel fusionnant les modules de bases et les aspects. Cette opérationse nomme le tissage (weaving) et elleest réalisée par le *tisseur d'aspects*. Voici une liste de quelques tisseurs d'aspects existant pour différents langages :

- C : Aspect-C
- C++ : Aspect-C ++
- PHP : phpAspect
- En C#, VB.NET : AspectDNG
- Caml : Aspectual Caml
- Java : AspectJ, JAC, JBoss AOP

Le mécanisme de tissage d'aspects au sein d'un programme de base peut être établi statiquement ou dynamiquement :

2.3.1. Le tissage statique :

C'est le plus simple, il consiste à utiliser un compilateur qui génère le nouveau programme en insérant les aspects au programme de base. De cette manière, le nouveau programme contient les aspects intégrés qui sont exprimés en termes d'instructions additionnelles dans les méthodes du programme de base. Cette politique offre une exécution performante mais peu flexible. Elle peut réalisée en manipulant le code source et même en code binaire (bytecodes) du programme de référence,

2.3.2. Le tissage dynamique :

Consiste à utiliser un interpréteur pour ajouter dynamiquement, lors de l'exécution, les aspects au programme de base. Pour se faire, l'interpréteur doit instrumenter l'exécution du programme de référence afin de capturer les points où il faut ajouter les aspects et ensuite, exécuter ces aspects. Il est clair que le tissage dynamique est plus flexible que l'intégration statique puisque les aspects peuvent être ajoutés/enlevés à tout moment. Cependant, le temps d'exécution est plus important. L'intégration dynamique peut être appliquée au code binaire du programme de base.

Il faut noter ici, qu'il est possible qu'un tisseur puisse implémenter plus qu'un seul mode de tissage.

3. L'interférence des aspects

On peut trouver dans une même applications différentes préoccupations techniques ce qui implique le tissage de plusieurs aspects dans le même programme de base (un aspect pour chaque préoccupation technique). Le tissage de plusieurs aspects peut introduire des interactions entre les aspects dans le programme de base. Dans certains cas, ces interactions sont naturelles et souhaitables, mais dans des nombreux cas, les interactions sont indésirables parce qu'elles conduisent à des comportements inattendus du programme tissé. On appelle ce dernier type d'interactions *interférences*.

Comme exemples d'interactions entre aspects, considérer les deux aspects suivants dans un système téléphonique. Le premier aspect (A1) implémente la fonctionnalité de maitre des appels en attente; et le deuxième (A2)implémente le transfert d'appel si l'utilisateur est occupé.Si les deux aspects sont tissés (l'utilisateurs active les deux options) et

l'utilisateur est engagé dans un appel, alors qu'est-ce qui se passera quand il ya un autre appel en cours?

En effet, l'une des problèmes les plus difficiles de modèles d'implémentation de la programmation par aspects est de savoir comment tisser plusieurs aspects sans avoir des interférences.

Beaucoup de travaux sont consacrés pour détecter et résoudre les interférences entre les aspects. C'est différentes tentatives sont divisées selon [Hannousse2012] en deux catégories : *syntaxiques* et *sémantiques*.

3.1. Les approches syntaxiques :

Le principe de ces approches est d'analyser les aspects pour trouver si deux ou plusieurs aspects partagent des points de jointure communs. Cependant, il est montrer en [Hannousse et al2011] que le partage des points de jointure n'est toujours la source d'interférence,

3.2. Les approches sémantiques :

Ces approches prennent on considération le comportement (les actions) des aspects à tisser, elles ne se concentrent pas ni sur les points de jointure communs entre les aspects ni sur la manipulation des variables communes. Ces approches sont encore divisées dans [Hannousse2012] en deux sous-catégories : *approches modulaires* et *non modulaires*,

1. **Les approches modulaires** : les approches modulaires traitent les interactions sémantiques qui concernent le comportement des aspects indépendamment à tout programme de base,

2. **Les approches non modulaires :** les approches non modulaires traitent les interactions sémantiques relatifs à un programme de base donné.

Les vérifications syntaxiques et sémantiques sont deux approches complémentaires pour la détection des interférences. Une détection des interférences sémantique efficace doit inclure à la fois l'analyse de flux de contrôle et de données [Hannousse 2012]. Cela permet une analyse précise et complète des interactions entre les aspects.

Pour la résolution des interférences entre aspects, des stratégies de composition d'aspects générales et réutilisables doivent être fournies. En outre, le développeur doit être informé pourquoi et comment une interférence apparaît. Cela permet au développeur de choisir la bonne stratégie de composition qui résout cette interférence [Hannousse2012]. Les model checkers [Clarke et al1981] peuvent aider à la détection des interférences et donner suffisamment d'informations sur la violation des propriétés modélisant le comportement attendu des aspects.

4. Conclusion

Dans ce chapitre nous avons présenté le paradigme de programmation aspects, nous avons passé en revue sur ses principes, ses concepts et ses avantages comparés aux autres paradigmes de programmation. On conclut de cette présentation que ce paradigme, relativement nouveau, représente une évolution certaine dans le domaine de la modélisation et de la séparation avancée des préoccupations en général. Le problème majeur de ce paradigme est la détection et la résolution des interférences possibles entre les aspects a tissés dans le système de base.

Généralement, une interférence ne peut être résolue automatiquement, mais des informations sur l'interférence doivent être rapportées à l'utilisateur, ce qui lui permet de détecter la source du conflit et prendre la bonne décision. Cela peut être abouti à l'aide des outils d'aide à la décision. On peut citer parmi ces outils, des outils graphiques permettant de visualiser les différents aspects d'un système avec leurs points d'interactions dans le programme de base, cela indique les points critiques où les interférences peuvent apparaître dans le système après le tissage. En plus, en [Hanousse2012] il est prouver que les model checkers comme Uppaal [Larson1997] peuvent être utilisés afin de détecter les interférence sémantiques entre les différents aspects à partir de la description abstraite des aspects, donc, des outils de transformations des description abstraites des aspects vers des descriptions adéquates aux model checkers sontutiles pour l'analyse sémantique des interaction entre les aspects.

CHAPITRE II.

Programmation par Composants

Les développeurs de logiciels sont constamment appelés à construire des applications en moins de temps et avec moins de coût. Cela peut être établit en construisant des applications en *réutilisant* ce qui a été déjà développé. Au départ, les développeurs ont l'habitude de compter sur la méthode « copier-coller ». Mais, avec le temps, une approche modulaire pour le développement de logiciels est apparue, c'est la programmation par objets.

La programmation par objets depuis son apparence en 1967est considérée comme une évolution par rapport à la programmation procédurale ; elle permet de développer des systèmes évolutifs et maintenables. Maiselle a encore des limitations comme l'explosion rapide de nombre de classes utilisé, et une réutilisation limitée aux objets qui sont généralement des éléments de moindre taille. A cause de ces limites une nouvelle approche apparu, c'est *la programmation par composants.*

La programmation par composants met en avant la programmation par assemblage plutôt que par développement.L'idée derrière cette nouvelle approche est inspirée des autres domaines notammentl'électronique où les développeursessaye de trouver ou d'acheter des composants électroniques déjà existants et les rassemblés afin d'avoir la version finale de son système désiré ; les composants électroniques sont des éléments destinés à être assembler afin de réaliser des circuits électroniquesplus complexes comme par exemple les circuits intégrés. De façon similaire, la programmation par composants propose des composants logiciels offrants des services réutilisable, similaire au marché des composantes matériels et/ou électroniques et les intégrés rapidement dans nos applications.Dans cette partie, nous allons présenter le principe de la programmation par composant ainsi quelques modèles existant.

1. La programmation par composants :

La programmation par composants (CBSE : Component Based Software Engineering)[Szyperski 2002] permet la modularité des préoccupations fonctionnelles en termes d'entités logiciels séparées appelées composants. Chaque composant joue un rôle spécifique dans le système, il définit les services qu'il offre à d'autres composants et les services requis des autres composants. Les composants peuvent être combinés et assemblés afin de construire le système final. L'assemblage des composants se fait notamment en mettant en relation les interfacesoffertes d'un composant avec les interfaces requises d'autres composants.

La programmation par composants a des effets positifs sur la conception, plus particulièrement, elle assure les propriétés suivantes aux systèmes développés :

- *Modulaire* :les systèmes développés sont séparés en unités distinctes appelées composants, chaque composant encapsule une partie de système, ce qui facilite la mise à jour de ces systèmes et améliore leurs compréhension,
- *Facile a Maintenir* :si un composanttombe en panne on fait la réparation à ce composant et n'ont pas au système complet ; en plus,la modification d'un composant ne nécessite pas la recompilation de l'application complète,
- *Interopérable* :où il est possible d'assembler des composants développés dans des langages de programmation déférents,
- *Fiable :* les composants assemblés sont des composants testé et certifié,
- *Rapide à développé* :la construction des applications est faite par des composants préfabriqués.

Malgré tous ces atouts, actuellement, la programmation par composants connue certains limites :

- La modification d'une signature d'un service offert par un composant nécessite la modification dans tous les morceaux de code qui font l'appel à ce composant, dans ce cas l'avantage de l'indépendance des composants n'est plus réalisé,
- Seuls les développeurs connaitre l'implémentation des composants, à pour objectif de garder le contrôlede leurs projets,
- Construire un logiciel à base de composants nécessite un grand travail d'analyse.

2. Les éléments de base des systèmes à composants :

Dans cette section on va décrire les différents éléments architecturaux des systèmes à composants, cela inclus *les composants, les interfaces, les laissons,* et *les connecteurs.*

2.1. Composants :

Le composant c'est l'unité de base de composition de CBSE, il s'agit d'une boite noir ou grise, il cache sa logique et ses implémentations et expose ses services à travers un ensemble de ports appelés des interfaces. Un composant peut être vu comme un

programme informatique encapsulé dans une interface. Un composant logiciel possède deux structures *internes* et *externes*. Il existe deux structures internes possibles pour un composant selon son type :

- **Primitive :** encapsule un code exécutable, et on appelle aussi des composants atomique.

Figure 2.1. Composant primitif

- **Composite :** il encapsule un ensemble des composants soit primitive ou composite.

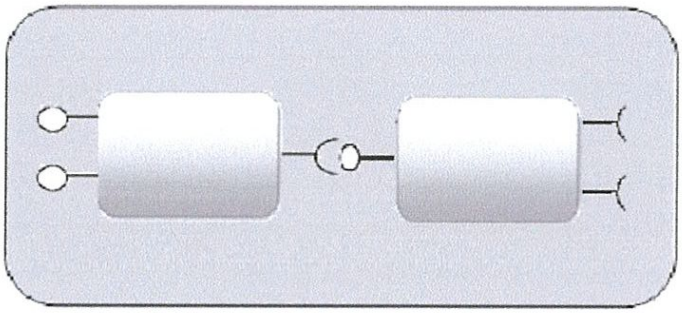


Figure 2.2. Composant composite

2.2. Interfaces :

Un composant peut contenir plusieurs interfaces qui représentent lesseuls points d'accès au composant. Les interfaces permettent à un composant d'exposer les moyens à utiliser pour communiquer avec son environnement. Les services fournispar un composantsont exposésdans les *interfaces offertes*, alors si le composant nécessite des services d'autres composants, il abstrait tous les servicesrequis dans des interfaces distinctes appelées *interfaces requises*.

- **Interface offert** : définit l'ensemble des services offerts par le composant.
- **Interface requise** : définit l'ensemble des services requis par le composant afin d'établir ses services offerts

2.3. Liaisons :

Est un mécanisme d'assemblage des composants. Une liaison relie une interface requise d'un composant par une ou plusieurs interfaces offertes des autres composants. Une liaison peut être directe (interface-à-interface) ou par le biais des médiateurs appelés *connecteurs*.

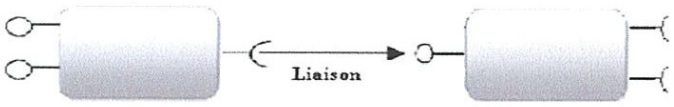


Figure 2.3. Liaison directe entre deux composants

Figure 2.4. Liaison par le biais d'un connecteur

2.4. Connecteurs :

C'est un élément permettant d'assembler des composants en utilisant leurs interfaces fournies et requises. Le connecteur est responsable de la liaison et de la communication entre composants ; il relie les interfaces des deux ou plusieurs composants et assure les échanges des messages entre ces composants. Il faut noter qu'un connecteur représente un concept abstrait, paramétrable et indépendant des composants spécifiques ; les attributs des connecteurs décrivent ses propriétés comportementales : type de communication (synchrone ou asynchrone) et la stratégie de communication (par message, par évènement, etc.).

3. Le processus de développement d'une application à base de composants :

Afin de développer un système à composants, le concepteur/développeur doit passer par quatre étapes intrinsèques :

1. **Construction :**cette étape consiste à créer des composants, c'est-à-dire à fournir la définition des composants et leurs interfaces offertes et requises.
2. **Livraison :** le composant créé, il doit être empaqueté et stocké dans des dépôts(*repository*) de composants pour être livré.
3. **Assemblage :**le client identifie et sélectionne, depuis les dépôts, les composants compatibles selon ses besoins. Il crée des nouvelles applications en assemblant des composants disponibles. Dans certains modèles, cette étape implique aussi laconfiguration des composants, laquelle consiste à donner des valeurs concrètes quiseront prises en compte au moment de l'instanciation des composants.
4. **Exécution :**si la composition des tous les composants est satisfaites, alors elle devient une application exécutable. À l'exécution, les composants sont instanciés en fonction del'architecture spécifiée.

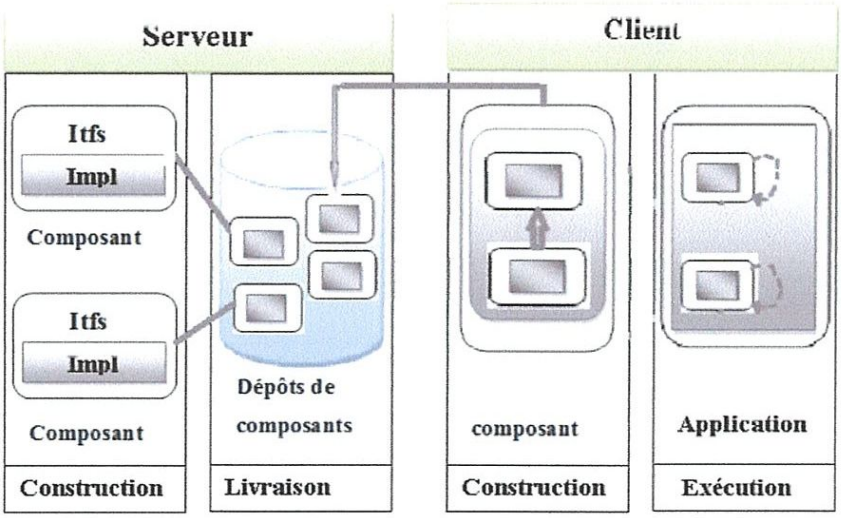


Figure 2.5. Processus de développement d'un système à composants

4. Les modèles à composants :

Un modèle à composant [Weinreich et al. 2001] définit un ensemble des standards pour l'implémentation des composants, et l'ensemble d'entités logicielles exécutables exigées pour soutenir l'exécution des composants qui se conforment au modèle.Ces standards concernent la composition, l'évolution, le déploiement, l'interopérabilité, et la personnalisation. Un modèle de composant définit également des standards pour une implémentation de modèle de composant associée. Il existe de nombreux modèles de composants pour différents domaines d'application comme EJB,CORBA/CCM,CAM/DAOP, FRACTAL, PRISMA, AspectLEDA.

5. Les modèles à composant et les préoccupations transverses :

La programmation par composants possède des avantages très importants. Ses principes de base comme : l'encapsulation, l'assemblage, et la modularité permettent d'améliorer la réutilisabilité et la maintenance des systèmes existantes.

Mais, certaines préoccupations non fonctionnelles telles que : la sécurité, la synchronisation, etc. ne parviennent pas à être encapsulées dans des composants uniques et distinctes, au contraire, elles se propagent sur plusieurs composants. Ces préoccupations particulières sont connues comme « *préoccupations transversales* ».

La solution consiste à *aspectualiser* les modèles à composants : c.à.d. de modéliser les préoccupations fonctionnelles sous forme des composants et de modéliser les préoccupations transverses sous forme des aspects à tisser aux composants.

6. Les catégories de modèles à composants :

Il y a différentes classifications des modèles à composants. Nous adaptons ici la classification proposée dans [Hannousse 2011] ; Cette classification est basée sur la manière adoptée par les différents modèles pour modéliser les préoccupations

transverses. Suivant [Hannousse 2011], les modèles à composants sont divisés en trois grandes catégories :

1. Les modèles de composants à conteneurs,
2. Les modèles à composants et aspects,
3. Les modèles à composants à base d'architecture logicielle.

6.1. Les modèles de composants et conteneurs :

Dans les modèles à composantsà base deconteneursles développeurs sont libérés de la mise en œuvre des préoccupations transverses, cesfonctionnalitéscomplexescomme la distribution,la synchronisation et la persistance sont prises en charge par les conteneurs. Le conteneur enveloppeun ou plusieurs composants et empêche toutaccès direct à leursservices. Chaqueappel d'un serviced'un composant est intercepté par le conteneur, celui-ci exécute les comportements des préoccupations transverses associées aux l'appel de service appelée puis décide de poursuivre l'appel (proceed) ou de simplement l'ignorer (skip).

6.1.1. Enterprise JavaBeans (EJB):

Le modèleEJB [Burke 2006] permet le développement etle déploiementde composants qui sontd'origine des objets Javaappelés*Enterprise Beans*.Les composants du modèle EJB s'exécutent au sein d'un support d'exécution appelé *Conteneur EJB* (EJB Container), lui même contenu dans un serveur EJB (EJB Server).

Chaque composant EJB est lui-même composé d'une *EJB Instance* (instance) et deux interfaces *home* et *remote* qui modélisent les interfaces d'accès au composant par un client :

- *EJB Instance :* est une instance d'une classe Java fournie par le développeur du composant. Cette classe implémente l'ensemble des services exposés par les interfaces du composant.
- *Interface Home :*expose les services qui gèrent le cycle de vie du bean : création, mise à jour, recherche et destruction. Elle est commune à toutes les instances de ce type de composant.

- *Interface Remote* :expose l'ensemble des services applicatifs du composant (business méthodes) qui sont fournis par le composant EJB.

Le client n'interagit jamais directement avec l'instance. Toutes ses requêtes passent par *home* ou *remote*, qui délèguent ensuite leur traitement soit vers l'instance, soit vers le conteneur. Ce mécanisme d'interposition permet d'exporter au client uniquement la partie qui le concerne et de cacher à celui-ci les services liés à la gestion du composant qui sont exploitées par le serveur. La figure 2.6montre l'architecture d'un composant EJB.

Le conteneur EJBenveloppeun ou plusieurs composants EJB etempêchent toutaccès direct à leursservices. A chaque réception d'un appel au service, le conteneur exécute une ou plusieurs préoccupations transverses associées au service appelé et relancer le service d'origine.

Le conteneur EJBprend en charge quatre préoccupations transverses :

- **Transaction :**décrit la possibilité d'une unité de travail à être exécutée en une seule pièce.
- **Sécurité :** est l'une des caractéristiques la plus importantes d'EJB. Il donne un jeu d'autorisations pour donner à chaque type d'utilisateur son propre droit d'accès à l'application.
- **La persistance :** c'est le protocole d'accès aux données pour transférer l'état de l'entité entre l'instance du bean et la base de données sous-jacente.
- **Concurrence :** à part les singletons, tous les autres types d'EJB sont placés dans des threads personnels (thread-safe) afin de gérer correctement, en parallèle, l'ensemble des requêtes des clients. Vous pouvez donc développer des applications parallèles sans vous soucier des problèmes liés aux threads.

Cespréoccupations sont prises en charge par le conteneur EJB et sont tissésaux entreprisesbeansà la demandede développeur. Pour cela, le développeur doit configurer d'une manière déclarative dans un fichier XML les préoccupations associées à chaque service de composant enveloppé par le conteneur.

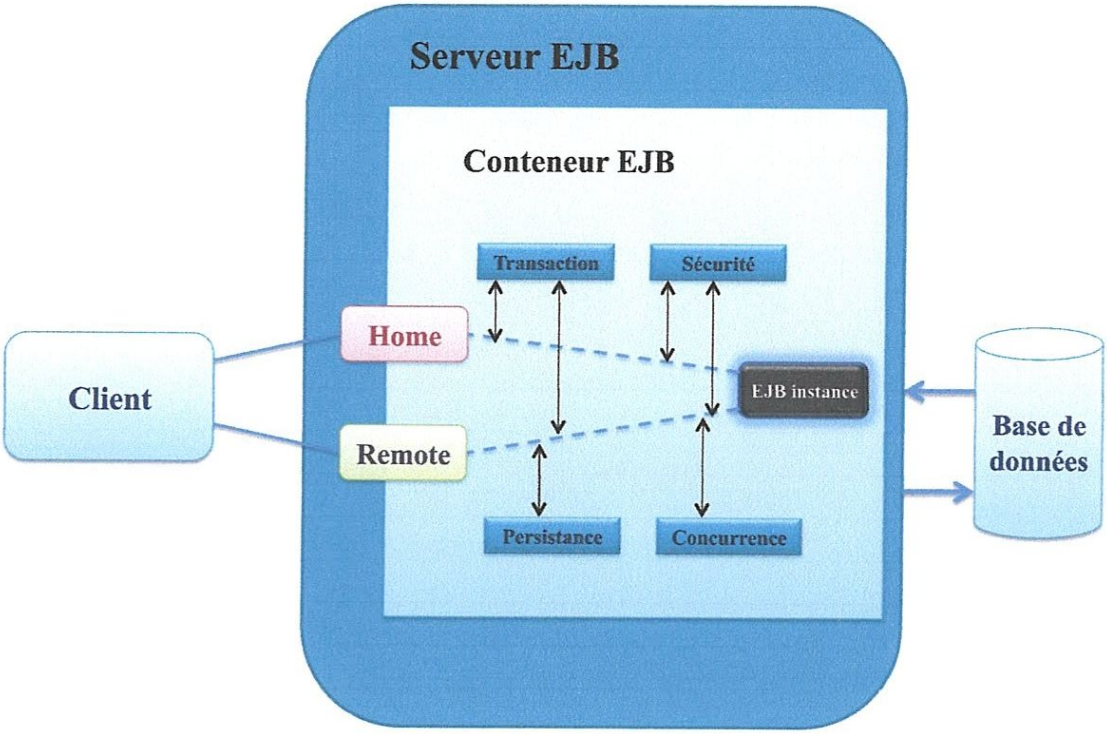


Figure 2.6. La structure d'un conteneur EJB

Le principal avantage de la technologie EJB est la portabilité. Autrement dit, une fois qu'un EJB est développé, il peut être exécuté dans n'importe quel conteneur EJB, tels que celles fournies par BEA, IBM ou serveurs GemStone.

6.1.2. CORBA/CCM :

CORBA [OMG 2000]est un Framework[2] client/serveur destinée aux applications réparties, ayant son propre langage et ses propres services, adaptée aussi à d'autres langages de programmation que le langage CORBA. Son objectif est de masquer l'hétérogénéité des langages de programmation et des machines pour construire des applications distribuées utilisant des composants reposant sur des plates-formes distincts et programmés dans des langages différents.

[2] Un Framework est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications. Il fournit suffisamment de briques logicielles qui sont organisés pour être utilisés en interaction les uns avec les autres pour pouvoir produire une application aboutie et facile à maintenir.

Le CCM[OMG 2006] est le modèle à composant associé à l'architecture distribuée CORBA.Ce modèle simplifie le développement des applications réparties en fournissant un niveau d'abstraction plus élevé. Un composant CCM possède quatre types d'interfaces qui sont appelés des *ports* : *facettes, réceptacles, source, et puits.* Les facettes et les réceptacles sont utilisés pour les communications synchrones, tandis que les sources d'événements et les puits sont utilisés pour les communications asynchrones.

Similaire à EJB, les composants CCM sont gérés par des conteneurs qui hébergent les instances des composants et facilitent leurs déploiements. Le conteneur des composants CCM pris en charge quatre préoccupations transverses : la persistance, la transaction, la sécurité, et la *notification.*

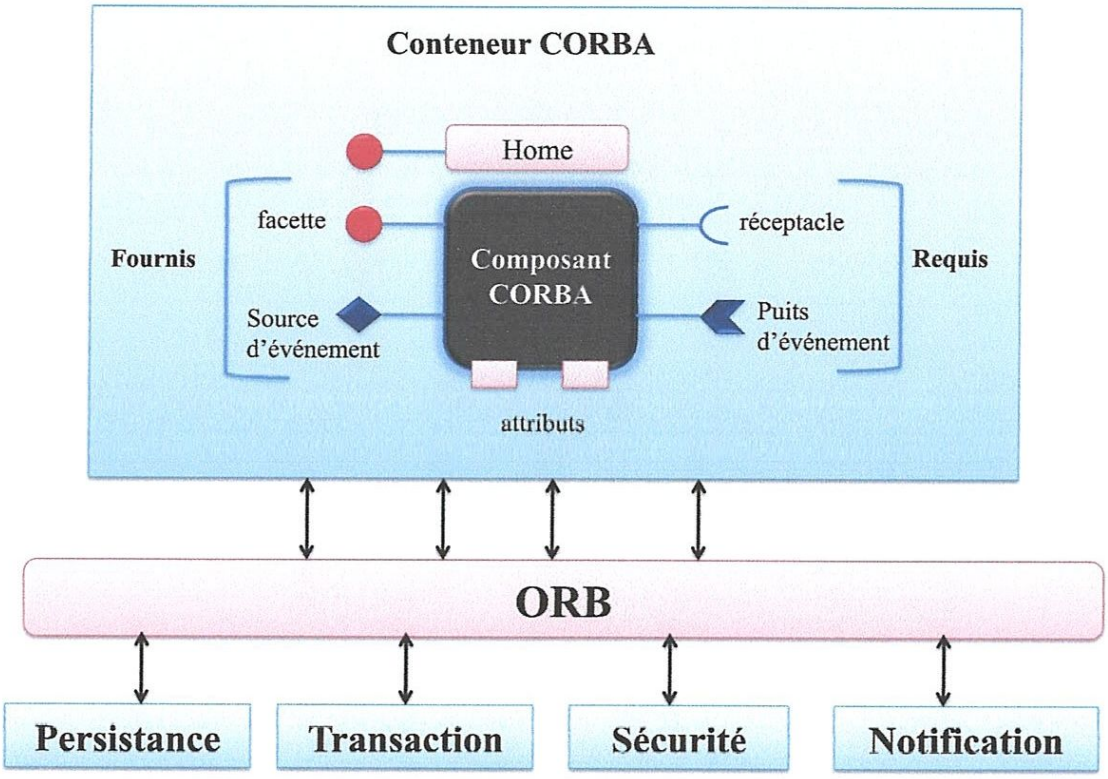


Figure 2.7. Composant CORBA

6.2. Les modèles de composants et aspects :

Les modèles à composants et aspects sont plus flexibles etextensibles par rapport aux modèles à conteneurs.Ils permettent à l'utilisateur de définirses propres préoccupations

transverses, contrairement aux modèles à conteneurs où l'utilisateur est limité par l'utilisation des préoccupations transverses prédéfinies et gérées par les conteneurs. Les préoccupations transverses dans cette nouvelle catégorie des modèles de composants sont modélisées par des aspects au sens AOP (voir Chapitre 1). Un mécanisme de tissage est alors défini par ces modèle afin d'assurer l'intégration des préoccupations transverses dans le système. Les différents modèles appartenant à cette catégorie diffèrent dans la manière de définition des aspects et le mécanisme de tissage associé. Dans ce qui suit, on va présenter deux modèles à composants et aspects on focalisant sur leurs façons de définitions des aspects et leurs mécanismes de tissages.

6.2.1. CAM/DAOP :

La technologie CAM/DAOP[Pinto et al. 2005] décrit les composants et les aspects par des entitésde première classe. Ces entitéssont ensuitetisséesdynamiquementà l'exécution.CAM/DAOPest conçu pour être une plateforme indépendante de tout langage de programmation,mais il est implémenté en Java seulement. Le modèle CAM/DAOP est divisé en deux parties complémentaires : CAM et DAOP.

- *CAM (Component Aspect Model)* :Le modèle CAMdéfinit à la fois les entitésprincipales des systèmes (composants et aspects) et les relations entreeux, cette description est exprimée en langage particulier basé sur XML appelé DAOP-ADL. DAOP-ADL décrit la structure des applications CAM en termes d'un ensemble de composants, aspects et contraintes de composition.
- *DAOP(DynamicAspect-Oriented Platform)* : C'est la plate-formeoù lescomposant etlesaspects définis par CAMsont dépoilés et exécutés.

Les informations surl'architecture d'applicationdécrites par DAOP-ADL sontchargées parla plate-formeDAOPlorsque l'applicationest lancée ; ensuite, la plate-formeDAOPconsulteau fur et a mesure ces informationspour effectuerla compositiondynamique de composantset aspects.

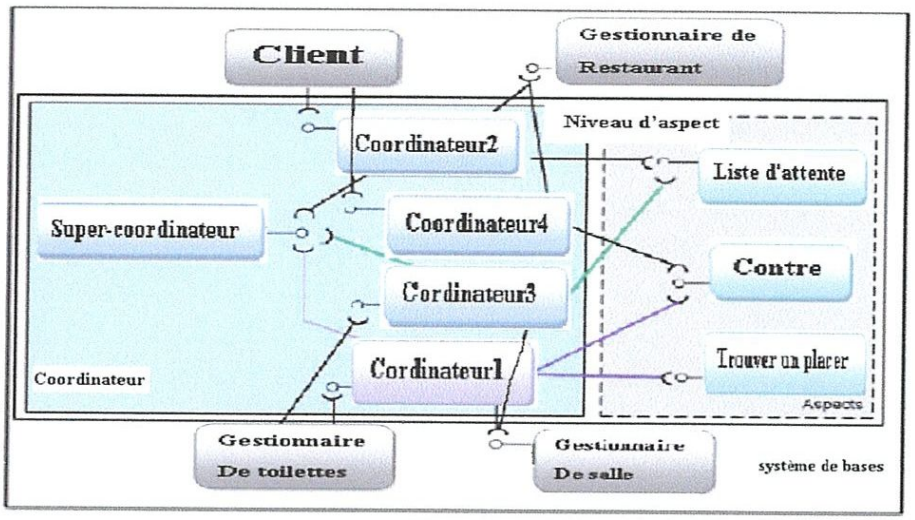


Figure 2.8.L'architecture CAM/DAOP

6.2.2. FRACTAL :

Fractal [Bruneton 2006]définit un Framework permettant de construire des applications à partir d'un modèle à composants hiérarchique, flexible et extensible développé par des chercheurs de l'I.N.R.I.A. et de FRANCE TELECOM. L'objectif de Fractal est de réduire les coûts de développement, de déploiement et de maintenance des systèmes logiciels en général, et des projets ObjectWeb en particulier. Fractal estun modèle de composantextensible quipeut êtrc utilisé avecdifférents langages de programmationpour concevoir, mettre en œuvre, déployer êtreconfigurer lessystèmeset les applications à base de composants.

Le modèle Fractal supporte à la fois les composants primitifs et composites grâce à sa hiérarchisation. Les composants primitifs se distinguent des composites par la définition de leurs contenus.Le contenu d'un composant primitif est défini par l'ensemble d'attributs et d'opérations implémentés par le composant, tandis que le contenu d'un composite est défini par l'ensemble de ses composants internes et leurs connexions.

UncomposantFractalest composé dedeuxparties un*contenu* etune*membrane* :

- *Le contenu* décrit l'ensemble des opérations et des attributs du composant pour le cas primitive ou l'ensemble des composants internes pour le cas composite,

- *La membrane*expose lesinterfaces fournies et requises d'un composant etencapsule un ensemble decontrôleursqui misent en œuvreles préoccupations transversesappliquées surle composant (voir Figue 2.9).

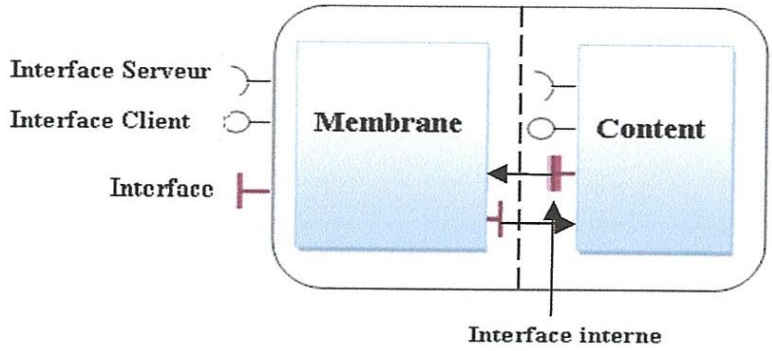


Figure 2.9.L'architecture d'un composant Fractal

Les composants sont reliés les uns aux autres par une liaisonqui relie une interface requise d'un composant à une interface offerte par un autre composant. En plus de la hiérarchisation de ses composants, Fractal accepte un ensemble de fonctionnalités intéressantes telles que : le *partage des composants* et *des objets contrôleurs* associés à la membrane de chaque composant.

- *Les composants partagés* : sont des composants qui peuvent appartenir à plusieurs composants composites en même temps ce qui permet de modéliser des ressources et leurs partages, tout en préservant l'encapsulation des composants.
- *Les objets contrôleurs* : chaque objet contrôleur modélise une préoccupation transverse associée au composant.

Fractal fournit également un langage de description d'architecture appelé *Fractal-ADL*pour décrire l'architecture des systèmes de composants.En plus, Fractal possède des capacités d'introspection et de reconfiguration pour permettre d'observer les systèmes et de reconfigurer les systèmes durant leurs exécution.

6.3. Les modèles de composants à base d'architecture logiciels :

La particularitédes modèlesdans cette catégorieest que les préoccupations transverses sont introduitesdans les phasesprécoces (i.e. analyse ou conception).

6.3.1. PRISMA :

PRISMA[Pérezet al. 2006]suit l'approche MDSOC (Multi-DimentionalSeparation Of Concerns) [Tarret al. 1999]qui considèreun système comme un ensemble des préoccupations ordinaires. La principale déférence est que la préoccupation transverse est la seule qui recoupe les déférentes autres préoccupations.

Une préoccupation transverse en PRISMA est définiepar un module séparéqui peut être importépar plusieurs autreséléments architecturaux (composants et connecteurs).

EnI'RISMA, les préoccupations transverses sont considérésa a partir de la phase de conceptioncomme des préoccupations régulières, qui peut recoupent d'autres préoccupations.

PRISMA propose une stratégie de composition des préoccupations transverses à l'aide d'un ensemble des mots clés, le développeur doit indiquer l'ordre d'exécutions explicites des préoccupations transverses pour chaque point de jointure dans les autres préoccupations. Le principal avantage de PRISMA, est son soutien de modèle formel de la spécification de protocole de préoccupations qui permet de faire une validation des propriétés des éléments architecturaux.

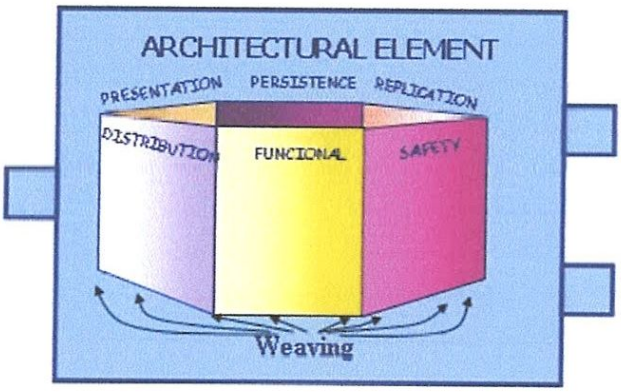


Figure 2.10.L'architecture d'un composant PRISMA

6.3.2. AspectLEDA :

Aspect LEDA [Navasa2009]définit les aspects au stade de la conception. Il est basé sur le modèle d'architecture logicielle orienté aspect. Cette approche comporte deux étapes : la définition d'un modèle initial de l'architecture c'est-à-dire décrivant l'architecture du système de base et l'ajout des aspects c'est-à-dire spécifiant les aspects à appliquer au système. Une fois les concepteurs extraire des informations sur les interactions avec les aspects de système de base. Ces informations comprennent, les points d'interaction de chaque aspect. Ainsi, l'ajout d'une nouvelle préoccupation revient à ajouter un nouvel aspect.

En AspectLEDA, les aspects sont d'écrits de la même manière que les composants. Toutefois, il introduit la notion de coordonnateur pour définir le processus de tissage qui synchronise les aspects et les composants et coordonne les deux niveaux.

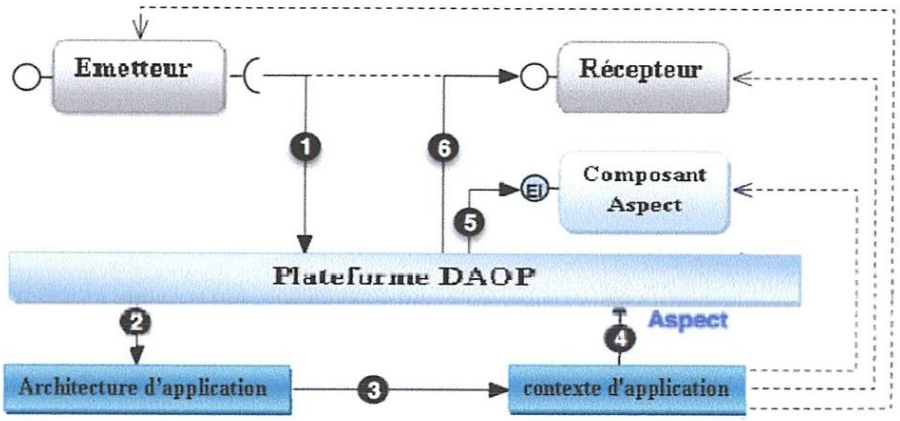


Figure 2.11.L'architecture d'un composant AspectLEDA

7. Conclusion :

Le paradigme composant est un paradigme efficace parce qu'ilrépond à des facteurs problématiques très intéressants au développement des logiciels comme la taille des logiciels, le coût, la vérification et l'évolution.Dans ce chapitre on a présenté un aperçu sur la programmation par composants avec ses bénéfices tel que l'amélioration de la réutilisabilité des entités logiciels et la modularité des systèmes ce qui facilite leur maintenance et évolution.Malheureusement, la présence des préoccupations transverses

dans les applications rendre les composants plus compliqués. Particulièrement, si ces préoccupations transverses ne sont pas traitées d'une manière adéquate elles réduisent la réutilisabilité des composants et compliquent la maintenance des systèmes logiciels.

Pour régler ce problème, les développeurs pensent à fournir un support explicite de l'approche AOP pour la modélisation des préoccupations transverses dans les systèmes à composants à travers la construction des modèles hybrides composants/aspects (voir la deuxième catégorie des modèles à composants).

Malheureusement, cette intégration des aspects au monde de composants manques certains outils qui rendre cette intégration efficace. Parmi les outils qui manquent, on note les outils graphiques qui permettent de visualiser le système avec ses différents types de préoccupations (régulières ou transverses) qui permet de détecter certains défailles dans les systèmes dans ses phase de conception, en plus des outils de vérification qui permettent d'assurer le bon fonctionnement des systèmes avec la présence de différents aspects qui se comporte différemment et en chevauchement.On va montrer dans le chapitre suivant notre participation à la résolution de ces problèmes.

CHAPITRE III.

Architecture des Systèmes à Composant et Aspects

Après une étude approfondie dans les chapitres précédents des systèmes à base des composants et des aspects, nous allons décrire dans ce chapitre la conception et l'architectures de l'outil graphique visualisants les systèmes à composants et aspects.

Notre outil consiste au développement d'un éditeur graphique pour la modélisation des systèmes à base des composants et des aspects. Nous allons utiliser les outils d'Eclipse que nous verrons plus tard pour maitre en œuvre une approche MDA pour la modélisation des systèmes à base de composants et aspects avec ses descriptions ADL adéquates. On commance ce chapitre par la description des outils utilisés pour le développement de notre application.

1. Model Driven Architecture (MDA)

MDA [OMG 2003] est une approche de développement du logiciel, proposée par l'OMG[3] (Object Management Group) depuis 2001, qui se base sur les modèles et s'appuyant sur le standard UML. L'idée fondamentale est que les fonctionnalités du système à développer sont définies dans un modèle indépendant de la plate-forme PIM (Platform Independent Model). Le PIM est ensuite traduit dans un ou plusieurs modèles spécifiques à la plate-forme PSM (Platform Specific Model). Enfin, le PSM est utilisé pour générer le code source de la plate-forme ciblée CIM (Computational Independent Model).

1.1. Le principe de l'MDA

OMG propose la façon de penser une application. Cette révolution s'appuie sur desnormes éprouvées, regroupés au sein du MDA. La démarche MDA fait une changement importante dans la conception des applications et fait une séparation entre le logique métier de l'entreprise et la logique d'implémentation. Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application.

1.2. L'architecture du MDA :

L'MDA se compose de quatre couches. La première couche c'est le noyau qui est composé par trois technologies (UML, MOF, CWM) spécifiées par l'OMG pour modéliser la logique métier de l'application. Dans la deuxième couche se trouve le standard XMI/XML qui permet le dialogue entre les middlewares (e.g., Java, CORBA, .NET et Web-services). La troisième couche contient les services qui permettent de gérer les évènements, la sécurité, les répertoires et les transactions. Finalement, la quatrième couche propose des Frameworks spécifiques au domaine d'application (e.g., Finance, Télécommunication, Transport, Espace, médecine, commerce électronique, manufacture,...).

[3] OMG " Object Management Group" est une association américaine à but non-lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

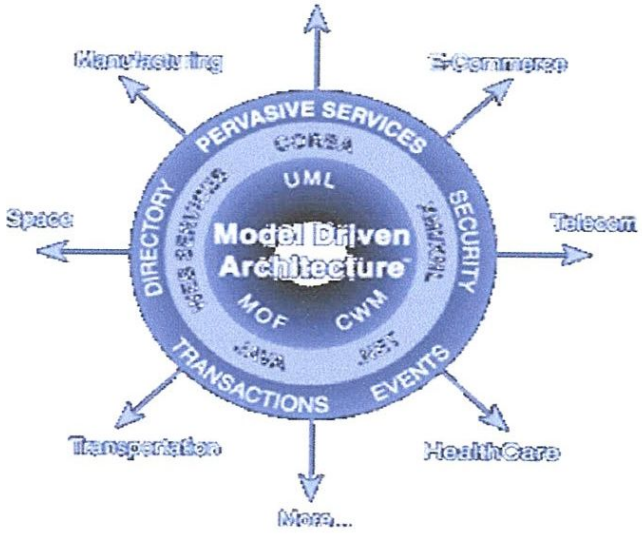


Figure 3.1 : L'architecture du MDA

2. La modélisation en UML :

La modélisation c'est une étape préalable. Elle se fait en utilisant les outils conceptuels d'UML. Elle aboutit à un ensemble de modèles de la future application, représentée par des diagrammes de classes. La modélisation consiste à créer une représentation simplifiée d'un problème nommé *modèle*. Ce dernier constitue ainsi à une simplification de la réalité qui permet de mieux comprendre le système à développer tel qu'il ne représente pas toute la réalité mais qu'un point de vue particulier.

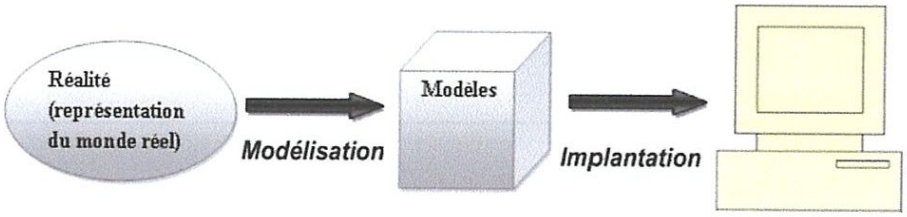


Figure 3.2 : Les étapes de résolution d'un problème

Un métamodèle permet de définir précisément les concepts manipulés dans les modèles ainsi que les relations entre ces concepts. En terme plus simple, un métamodèle c'est le modèle exprimé dans son propre langage (décrit en UML par exemple).

3. Les systèmes à composants et aspects par MDA :

Dans notre projet le cœur de l'application est le métamodèle décrivant les différents concepts des systèmes à base de composant et des aspects. Notre métamodèle peut être déviser en deux partie complémentaires : *structurelle* et *comportementale* :

3.1. La partie Strucrturelle :

Cette partie décrite les différents élemements architecturals des systèmes à composants et aspects, cela inclus : les composants, les aspects, les interfaces, les pointcuts, etc. Dans ce qui suit on va décrire comment ces différents élements sont modélisés dans notre métamodèle.

System : c'est la classe racine modélisant le point d'entré d'un système,

Node : c'est la classe que tous les autres classes possèdant l'attribut *Name* héritent elle,

Component : possède deux type : *primitive* ou *composite* ces deux classes hérite le contenu de la classe component,

Primitive : c'est une classe atomique qui modèlise un composant primitive dans un système,

Composite : elle modèlise un composant composite dans un système qui se compose de différents autres composants primitives et/ou composites,

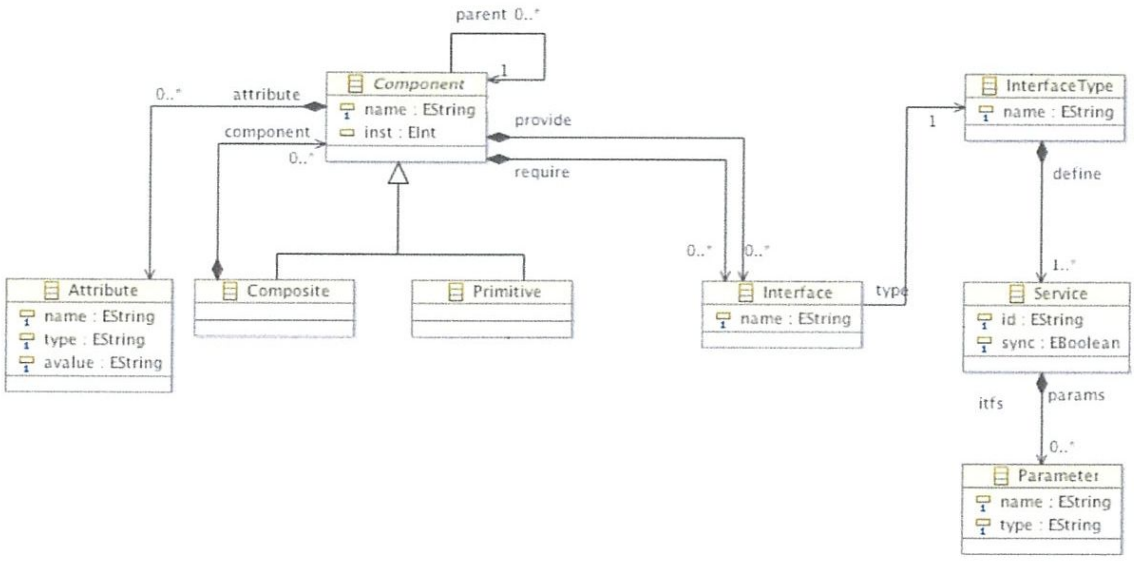


Figure 3.3 : Modélisation des composants et ses éléments

Chaque composant possède un ensemble des interface décrivant les différentes services offertes et requises par un composant. Dans notre modélisation, nous avant adopter les classes suivantes pour décrire l'architecture interne d'un composant :

Attribut : modélise un attribut d'un composant ou d'un aspect définie comme une variable dans un langage de programmation classique par un identificateur et un type,

Interface : modélise une interface d'un composant,

InterfaceType : dans un système à composants on peut avoir différents composants offerts ou requies des interfaces implémentant identiquement ou différement un ensemblme de services, pour cela nous avant introduit la classe *InterfaceType* qui décrit un ensemble de signature de services qui peuvent être implémenter par différentes interfaces.

Service : modélise un service définie par une interface d'un composant ; chaque service est définie par un identificateur unique *id* et un booléan *sync* indiquant le mode de service (synchrone ou asynchrone).

Paramétre : modélise une paramétre d'un service qui possède une nom et un type,

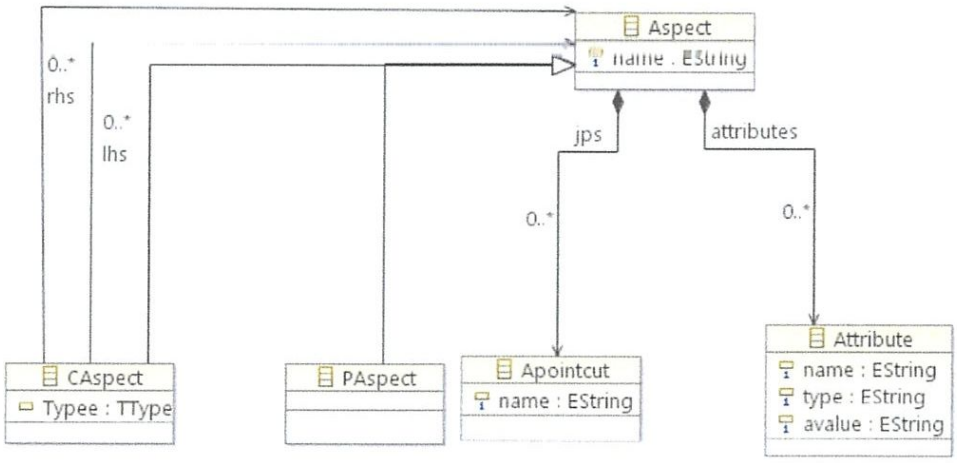


Figure 3.4 : Modélisation des aspects

Aspect : cette classe abstraite modélise un aspect dans le système qui se compose d'un ou plusieurs points de coupures (pointcuts, voir Chapitre I),

APointcut : modélise un point de coupure d'un aspect décrit par une chaine de caractère,

PAspect : hérite de la classe aspect et modèlise un aspect atomique dans le système,

CAspect : hérite de la classe aspect et modélise la composition des deux aspects en utilidsant une des opérateurs de composition offerts par notre plateforme (parallèle, séquentiel , Altérnatitive ou Conditionelle), les deux aspects à composer sont désigner par les deux connecteurs lhs (left-hand-side) et rhs (right-hand-side). Il faut noter ici que la compsotion n'est des aspects n'est pas commutative.

3.2. La partie Comportemantale :

La patrie comportementale décrite les différents élements utiles pour la vérification formelle des intéractions composant-composant, composant-aspect, et aspect-aspect dans un système à composant et aspects. La vérification formelle est nécessaire pour détecter la possiblité des inteférence entre les aspects dans le système (Voir Chapitre II). Dans ce qui suit, on va décrire la modélisation adoptés des éléments comportementales des systèmes à composants et aspects.

Pour la vérification formmelle chaque service est définie par une *précondition* décrivant une condition déclanchante du srevice, une ou plusieurs *postconditions* décrivant les actions suit l'exécution du service, et potentiellement une *contrainte temps réel* décrivant le temps min et max nécessaire à l'exécution du service ; tous ces éléments sont modélisés par les classes suivantes :

Pré : modélise une expression régulière décrivant une condition booléenne,

Post : modélise une action à exécuter après l'exécution du service, cela inclus la misc à jour d'un attribut d'un composant,

RTConstraint : modélise une contrainte temps réel exprimer en terme d'un interval du temps définie par ses deux bones inférieurs (lower) et supérieur (upper),

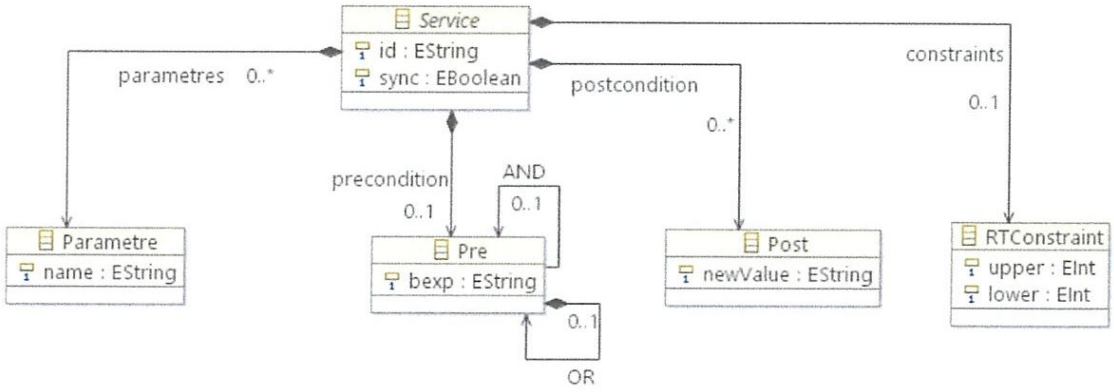


Figure 3.5 : Modélisation de comportement d'un service

On plus, le comportement abstraite des composants et des aspects est décrite par un protocol décrivant les différentes traces d'exécution possibles de chaque élément. Ce protocol est modélisé dans notre métamodèle par les classes suivantes :

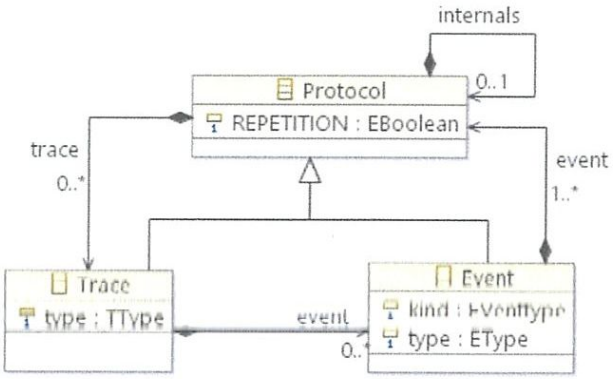


Figure 3.6 : Modélisation de protocole

Protocol : c'est une classe abstratie décrivant le comportement global d'un aspect ou d'un composant, cette comportement peut être répéter indifiniment ; cela est décrit par le booléen *repetition*,

Event : modélise un cas particulier d'un comportement ou un composant ou un aspect reçoit ou déclanche une demande d'exécution d'un service. Un evénement *event* est définie par : (1) *kind* qui spécifie la nature de l'évenement (appeller, annuler, ou conituer

l'exécution d'un service); et (2) *type* qui indique si l'evénement est réçu ou déclanché par le cmposant ou l'aspect,

Trace : modélise touts les traçes d'exécution possibles d'un composant ou d'un aspect on décrivants les différents evénements et leurs relation modéllisée par *ttype* (séquence - Seq, parallél - Par, if-then-else - Cond, ou internes - Int). Voici un exemple de trace où E_1,..., E_5 sont des événement :

$$\text{SEQ}\ (E_1,\ \text{PAR}\ (E_2,\ \text{INT}\ E_3\ \{\text{SEQ}\ (E_4,\ E_5\})$$

Ce trace décrit l'exécution de l'evénement E_1 suivi par l'exécution de l'exécution paralléle des évenements E_2 et E_3 où l'exécution de ce dernier fait appel à l'exécution séquentielle des deux autres évènements E_4 et E_5.

4. Discussion

La Figure 3.7 présente notre première tentative de modélisatoin des systèmes à composants et aspects. Dans cette proposition on a trouvé trois problèmes :

1. La modélisation des interfaces offertes et requises par deux relations relionts la classe *composant* par la classe *Interface* ne nous permettra pas d'avoir, dans la représentation graphique correspondante, les interfaces comme des objets graphiques à intégrés dans le composant,
2. La classe *WeavingRule* décrivant à la fois la régle de tissage et de composation des aspects ne nous permettra pas d'avoir des objects graphiques modélisants les opérateurs de compositions des aspects,
3. La définition d'un protocole par une composition d'événements ou des traces ne nous permettra pas de choisir l'un des deux cas mais les deux à la fois,

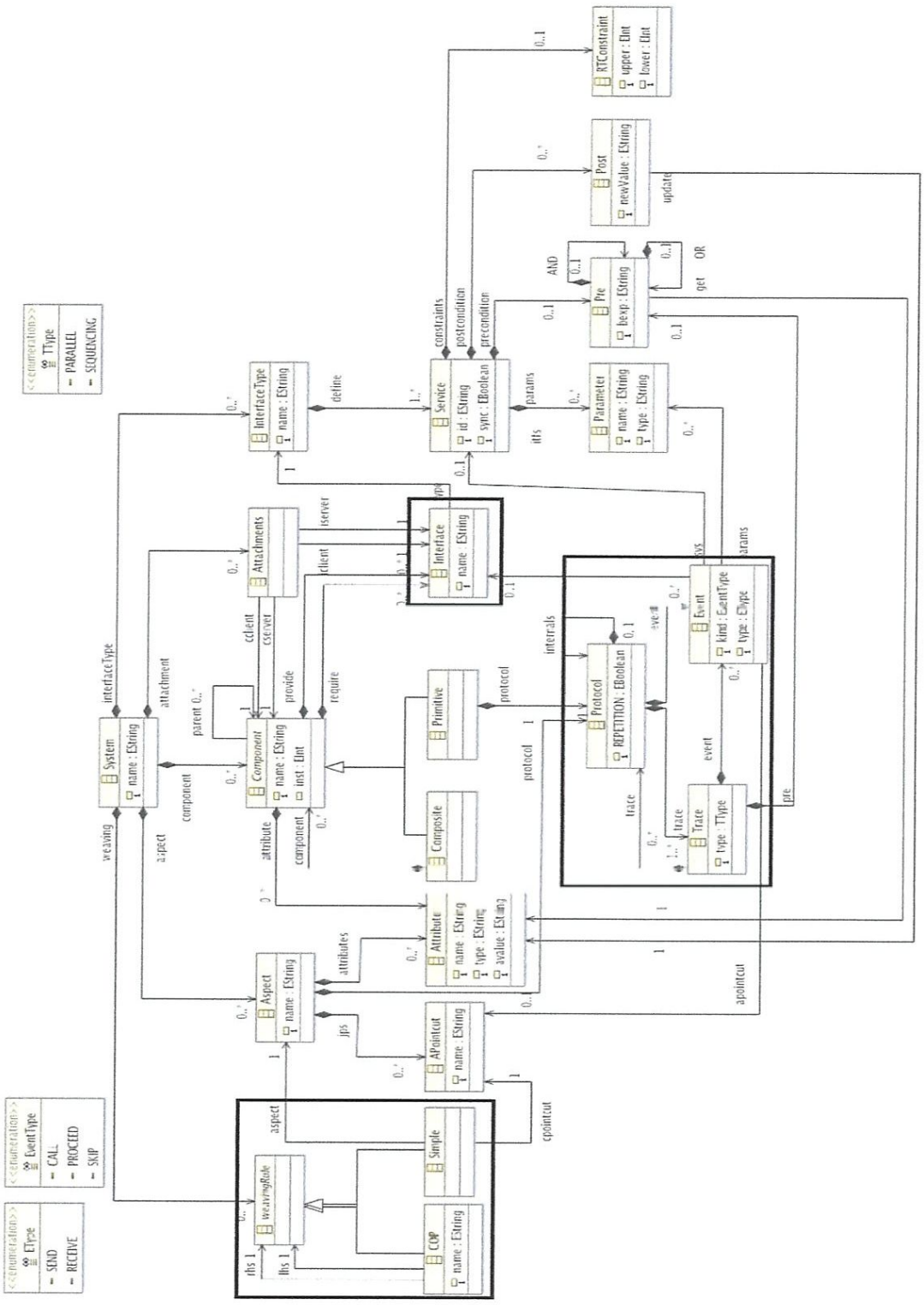


Figure 3.7 : Première tentative pour le méta modèle

Afin de pallier ces trois problèmes nous avons changer le métamodèle précedant par celui présenté dans Figure 3.8.

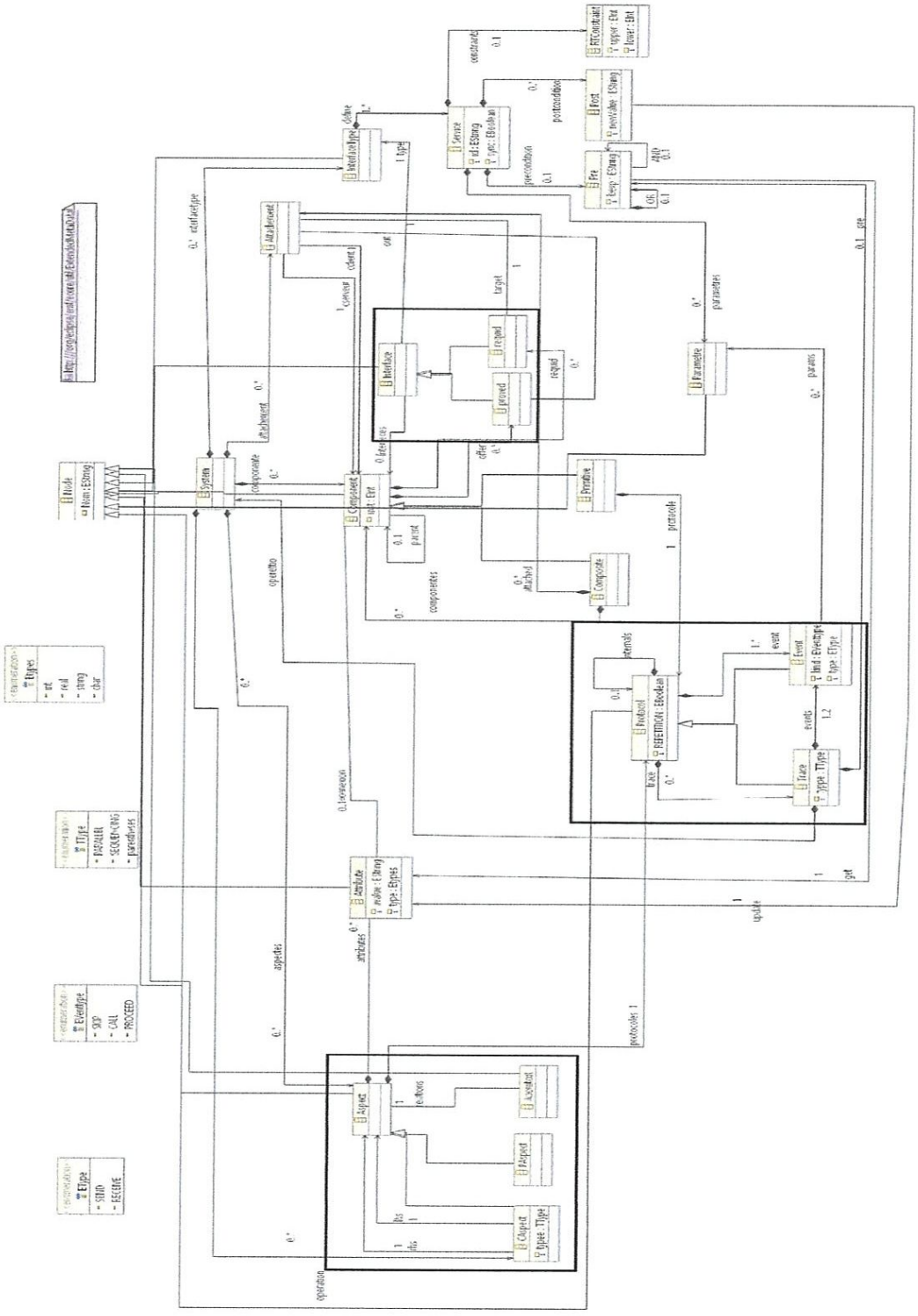


Figure 3.8 : Deuxième tentative pour méta modèle

Dans cette version, nous avans ajouté les deux classes *Provided* et *Required* héritant de la classe *Interface* modélisant, repectivement, les interfaces offertes et requises. Nous avons aussi remplacer la classe *WeavingRule* par une relation de composition entre aspects. Finalement, nous avons remplacer le relation de composition entre les classes *protocole*, *event*, et *trace* par une relation d'héritage.

La version finale de notre métamodèle adoptée pour les systèmes à composant et aspects est présentée dan la Figure 3.9. Dans cette version finale nous avons rajouté une liaison aspect-composant qui modélise l'opération de tissage d'aspects.

5. Conclusion

Nous avons présenté dans ce chapitre l'approche MDA pour la conception de l'architecture des logiciels qui utilse le formalisme UML qui est un standard de modélisation largement utilisé par les industriels et les chercheurs. Nous avons aussi prsésenté le métamodèle adopté pour les systèmes à composants et à aspects avec les différents difficultés rencontré durant la conception. Dans le chapitre suivant on monter comment ce méta modèle est utilisé pour le dévelopement de l'outil graphique attendus.

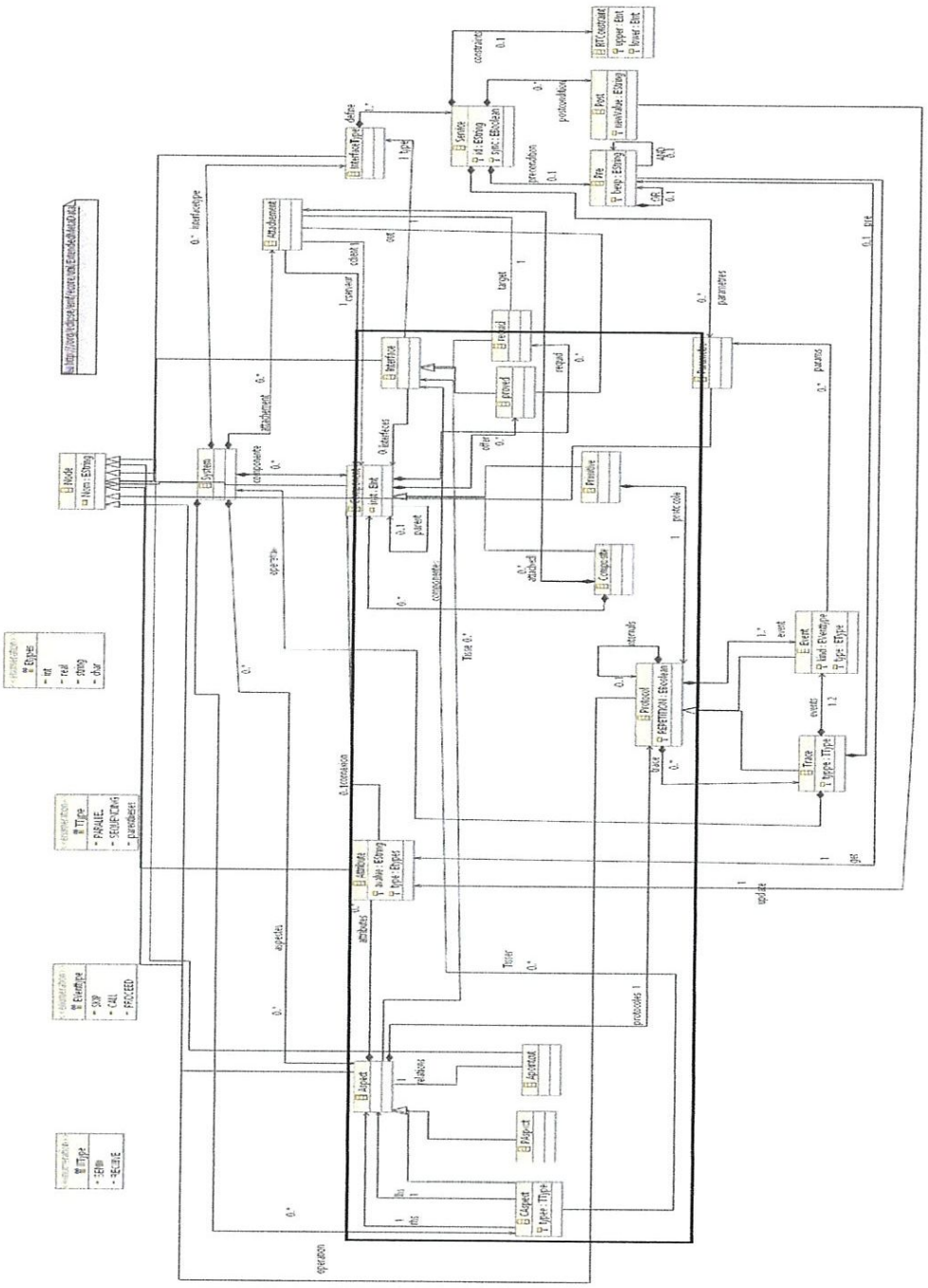


Figure 3.9: Le métamodèle modélisant les systèmes à composants et aspects

CHAPITRE IV.

Outil Graphique pour les Systèmes à Composants et Aspects - Implémentation

Pour bien concevoir et réaliser notre outil graphique, nous avons choisi de travailler avec l'environnement Eclipse. En effet, c'est un éditeur qui s'adapte le mieux à notre contexte de développement puisqu'il offre des plug-ins facilitant et rendant plus rapide la réalisation d'un éditeur graphique.

Dans cette section nous présentons Eclipse (sa plateforme, son architecture, etc.), puis nous définissons la notion d'un plug-in et les étapes de son développement et nous terminons par la présentation des différents plug-ins utilisés lors de l'implantation de notre outil.

1. Plateforme d'Eclipse :

Eclipse est une plate-forme universelle pour des environnements de développement intégrés fondée sur une architecture générique, ouverte et extensible. Elle s'agit d'une infrastructure de programmation portable. Elle présente un ensemble de Framework et de services pour assurer une interopérabilité des plug-ins qui se rajoutent par-dessus. C'est l'équivalent d'un noyau pour un système d'exploitation. En effet, la plateforme d'Eclipse est composée de :

1. **Noyau de la plateforme (Platform RunTime):** il constitue de l'infrastructure modulaire. Il prend la charge de détecter et de charger les plug-ins disponibles à chaque démarrage,
2. **Workspace :** c'est un espace dans lequel les ressources manipulées avec Eclipse (.java, .class, .xml) sont stockés,
3. **SWT (Standard Widget Toolkit) :** une bibliothèque graphique portable offrant les fonctions d'affichage graphique et définit un ensemble de composants d'interface standard (Widgets),
4. **JFace :** un ensemble de composants d'interface utilisateur positionné sur le Kit SWT permettant la gestion de fonctions d'interfaces évoluées,
5. **Workbench** : un module fournissant l'interface utilisateur. Le Workbench sert à manipuler les ressources, éditer et déboguer le code source, etc. Il contient les bibliothèques graphiques SWT et JFace.

1.1. Les plugins Eclipse utilisés :

La création d'un éditeur graphique sous Eclipse exige l'utilisation des frameworks EMF (Eclipse Modeling Framework), GEF (Graphical Editing Framework) et GMF (Graphical Modeling Framework). La figure suivante fait la synthèse des dépendances entre l'éditeur graphique généré, GMF, EMF, GEF et la plateforme d'Eclipse. GMF permet la réalisation et la génération d'un éditeur graphique. Il utilise donc EMF permettant de créer le méta-modèle et GEF gérant la côté graphique.

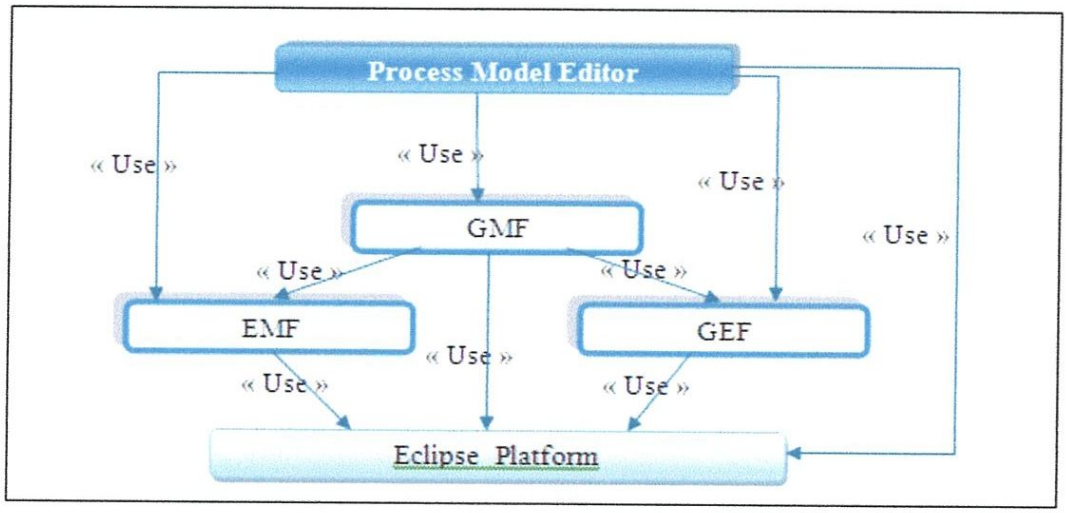


Figure 4.1. Dépendances du plugin GMF

2. Développement de l'éditeur graphique :

Après avoir présenté l'environnement de développement utilisé pour l'implémentation de notre éditeur graphique, cette section est consacrée à la réalisation des éditeurs graphique :

2.1. Définir la palette :

La palette représente les outils utilisés pour créer les nœuds et les connexions dans la zone de dessin de l'éditeur sont définies dans le fichier de définition d'outils. Pour cela nous avons créé le fichier « Systemeca.gmftool ».

Notre palette composée de 4 parties :

1. **Attribut** : commun entre les parties Fonctionnel et Non Fonctionnel,
2. **Fonctionnel** : contient les deux types de composant (primitive, composite), les interfaces *provid* et *requid* et un attachement,
3. **Non-Fonctionnel** : contient les *aspects*, *pointcut*, les liens *lhs* et *rhs* et les *opérations*,
4. **Protocol** : c'est une partie commun entre les deux parties : fonctionnelle et Non fonctionnelle, contient *Protocol*, les *events* et les *traces*, le nœud Protocol indique le nom de Protocol et s'il existe une répétition du trace ou non.

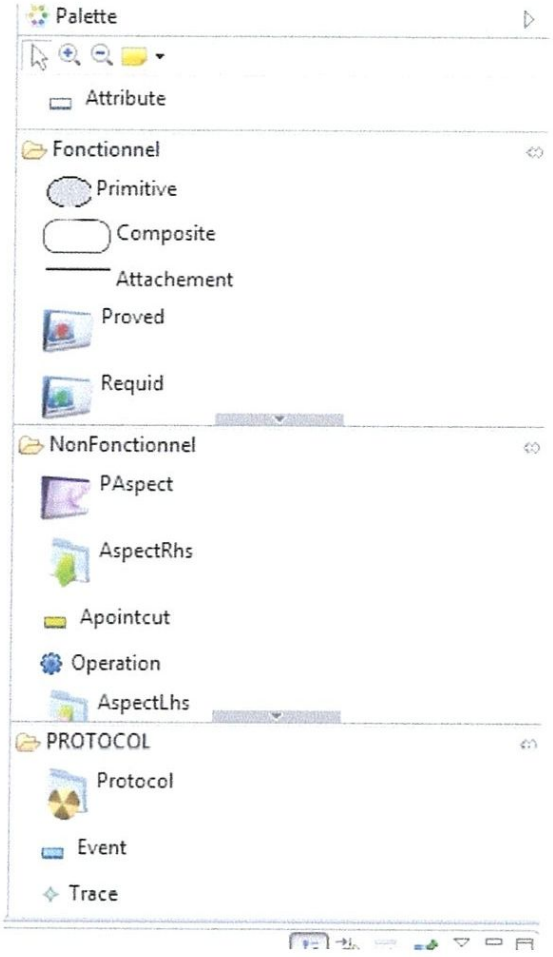


Figure 4.2. La palette de l'éditeur graphique

Composant Primitive :

Un composant primitive porte un *nom* et *instance* et contient des interfaces *provid* et *requid*, des *attributs* et un seul *Protocol*.

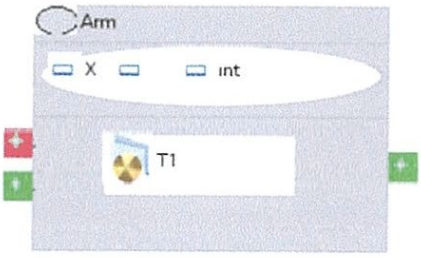


Figure 4.3. Composant Primitive

Composant composite :

La figure suivante représente un composant composite porte un nom et composé des composants primitive, des interfaces provid et requid et des attachements.

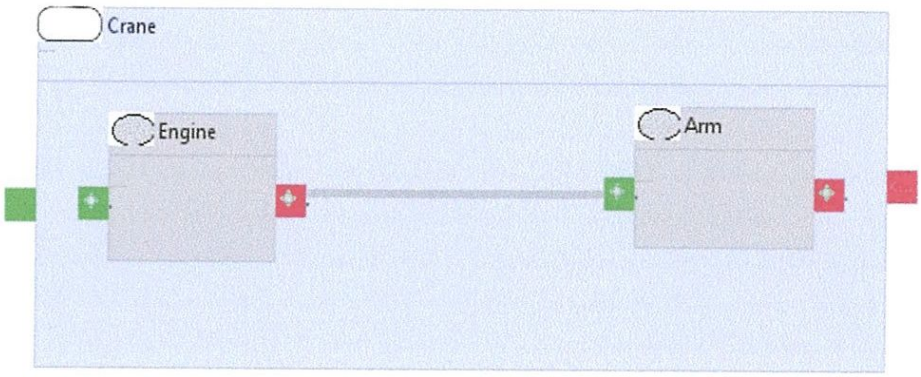


Figure 4.4. Composant Composite

Aspect primitif :

La figure suivante montre un aspect primitive porte un nom et possède des pointcuts, des attributs et un Protocol.

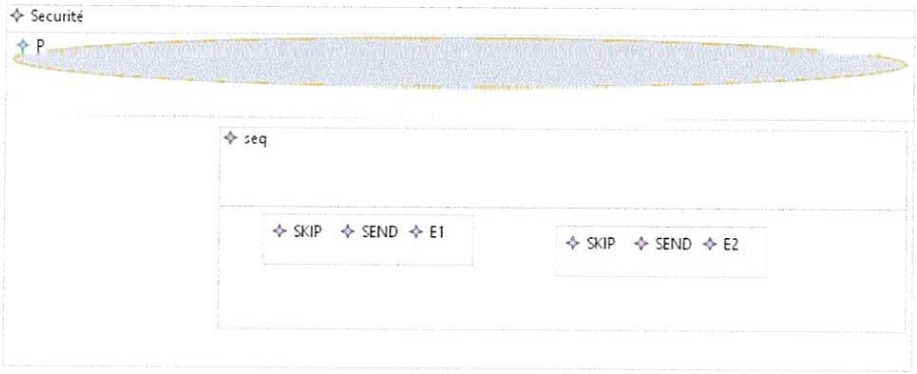


Figure 4.5. Aspect Simple

Aspect Composite :

L'aspect composite relie deux aspects primitifs par un opérateur de composition.

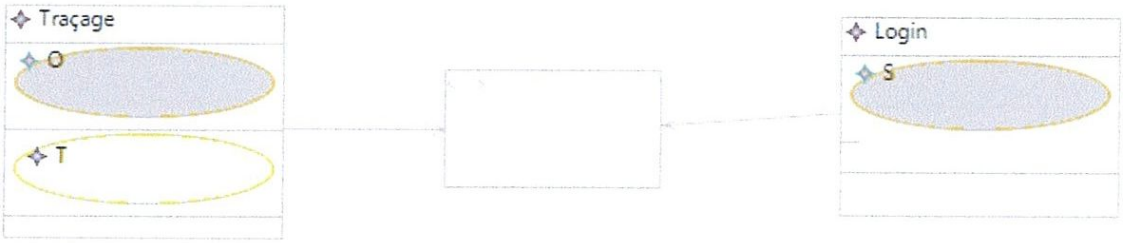


Figure 4.6. Aspect Composite

3. Langages de Description d'architecture (ADL)

Nous avons fait aussi une transformation de modèle en texte on a utilisé langages de Description d'architecture (ADL).

ADL : C'est un langage informatique décrivant l'architecture logicielle et matérielle d'un système. La description peut couvrir des fonctions logicielles telles que les processus, threads, données et sous-programmes ainsi que composant matériel tels que processeurs, périphériques, autobus et mémoire.

L'architecture logicielle d'un programme ou d'un système informatique est la structure ou les structures du système, qui comprennent des éléments de logiciel, les propriétés visibles de l'extérieur de ces éléments et les relations entre eux.

3.1. Description d'ADL

L'ADL proposé par le Docteur Hannousse enrichit ADL(s) avec les informations nécessaires pour détecter et résoudre les interférences entre les aspects. Le comportement des éléments primitifs et aspects est unifiés et explicitement spécifié. En plus un ensemble de règles de tissage sont fournis pour spécifier où et comment les aspects sont tissés et composé pour le système de base. Le tableau suivant montre la grammaire de notre ADL.

Dans le tableau **cId**, **itfId**, et **svId**, sont référer à les identificateurs de : composant, interface, service, en plus **asId**, **pcId**, **opId**, sont référer à les identificateurs de : aspect, apointcut. Et le **t** pour le data type.

3.2. Grammaire de l'ADL :

Voici la syntaxe de l'ADL en format BNF, où les mots en gras sont des mots-clés réservés au langage.

```
Architecture ::= system id <Interfaces><Components><Attachments> [<Aspects] [<Ops>]
Interfaces ::= (interface id {(@(syn |asyn) [@rtc "["int,int"]"] svId)+})+
Components ::= Primitive | Composite | <Components>; <Components>
Primitive ::= primitive id [(n :>= 2)] {
<Template> Computation <Behavior>
}
Composite ::= composite id [(n :>=2)] {
<Template> internals cId+
}
Attachments ::= binding (client=cId1.itfId1 server=cId2.itfId2 )
Template ::= [attributes (t id;)+] [provides (itfId id;)+] [requires (itfId id;)+]
Behavior ::= process id

Aspects ::= (aspect id <Behavior>;)+
Ops ::= (operator id <Behavior>;)+
```

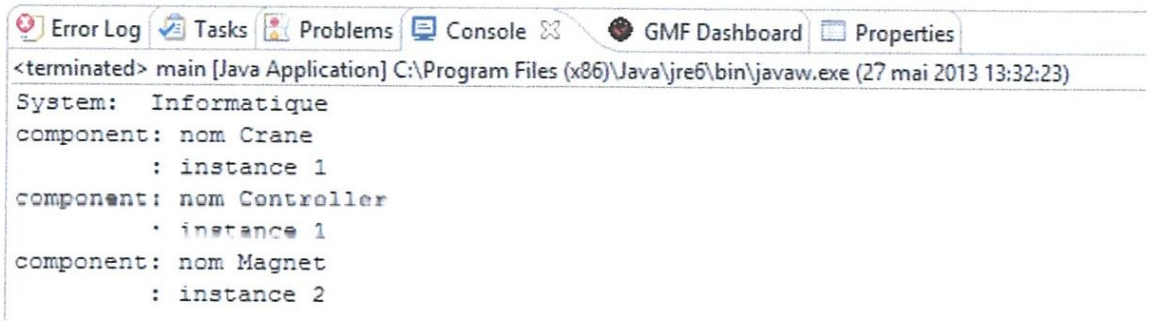


Figure 4.7. La transformation d'un modèle en texte

4. Conclusion

Dans cette partie, nous avons défini l'architecture et l'interface de notre outil graphique adopté à la spécification et la modélisation des systèmes à composants et aspects. On plus nous avons présenté l'architecture de l'environnement Eclipse utilisé dans notre projet.

Conclusion et Perspectives

En conclusion, nous avons proposé, dans le cadre de notre projet de mastère, une approche graphique de modélisation des systèmes à base de composants et des aspects. On a utilisé les outils d'Eclipse pour maître en œuvre l'approche MDA pour modéliser ces systèmes. Comme première étape on a défini une manière simple et efficace pour modéliser les systèmes à base de composants et des aspects dans un haut niveau par l'outil de modélisation GMF. Ensuite, on a généré une description ADL. L'outil proposé d'écrit une première étape dans la conception et la réalisation d'une approche efficace pour composants et aspects.

Dans notre projet, le tissage des aspects aux différents composants d'un système se fait manuellement par l'utilisateur en indiquant les différents points de jointures point par point en ajoutant des relation entre l'aspects et les interfaces des composants à aspectualizer, cette étape peut être automatiser en implémentant le langage formel VIL introduit dans [Hannousse 2010]. De cette façon, le tissage des aspects se fait automatiquement et d'une manière déclarative (pas besoin de définir les points de jointures point par point) ce qui facilite considérablement la conception des systèmes hybrides composants et aspects.

Bibliographie

J. Baltus, "La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué", Mini-Workshop: Systèmes Coopératifs. Matière Approfondie, Institut d"informatique, Namur, 2001.

[Bruneton 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Qu´ema and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. Software-Practice and Experience, vol. 36, no. 11-12, pages 1257–1284, 2006.

[Burke 2006] Bill Burke and Richard Monson-Haefel. Enterprise javabeans 3.0 (5th edition). O'Reilly Media, Inc., 2006.

[Kiczales 2001b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G.Griswold. An Overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[Kiczales 2001a] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9, page 313. ACM, 2001.

[Gabsi, 2011] Wafa GABSI. Modélisation @Runtime des systèmes à base de composants, Université de Sfax École Nationale d'Ingénieurs de Sfax,page 16.

[Hannousse 2010] Gilles Ardourel, and Rémi Douence. Views for Aspectualizing Component Models. In *the 9th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2010)*, St-Malo, France, pages acp4is10, March 2010.

[Hanousse 2011] Aspectualizing Component Models : implementation and Interferences Analysis.

[Hannousse 2012] A.H. Hannousse, Aspectualizing Component Models: Implementation and Interferences Analysis, LAP LAMBERT Academic publishing, 1st edition, Saarbrücken, Germany, 2012,

[Hannousse et al. ,2011] *the 10th International Conference on Generative Programming and Component Engineering (GPCE'11)*, Portland, Oregon : États-Unis (2011)

[OMG 2006] OMG. Corba Component Model Specification 4.0. Online: http://www.omg.org/cgibin/apps/doc?formal/06-04-01.pdf, 2006.

[Navasa 2009] Amparo Navasa, Miguel A. P´erez-Toledano and Juan M. Murillo. An ADL dealing with aspects at software architecture stage. Information and Software Technology, vol. 51, no. 2, pages 306–324, 2009.

[Pérez 2006] Jennifer P´ erez, Nour Ali, Jose Cars´ı and Isidro Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In CBSE: Component-Based Software Engineering, volume 4063 of LNCS, pages 123–138. Springer Berlin / Heidelberg, 2006.

[Szyperski 2002] Clemens Szyperski, Dominik Gruntz and Stephan Murer. Component software: Beyond object-oriented programming. Component Software Series. ACM Press and Addison-Wesley,New York, NY, 2nd ´ edition, 2002.

[Tarr 1999] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In Proceedings of the 21st international conference on Software engineering, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[Weinreich et al.2001] Rainer Weinreich, et Johannes Sametinger . modeles-de-composants hada howa ref ta3 la page 28