Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mentouri de Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

N° d'ordre:

Série:

THESE

Présentée en vue de l'obtention du diplôme de **DOCTORAT**

De l'université Mentouri-Constantine

Sélection sémantique de composants logiciels basée sur des critères de qualité de service

Option
Informatique

Par
Lamia YESSAD

Devant le Jury

Professeur Zaidi SAHNOUN	Université de Constantine	Président
Professeur Zizette BOUFAIDA	Université de Constantine	Directrice de thèse
Professeur Mohamed Tayeb LASKRI	Université d'Annaba	Examinateur
Docteur Abdelkamel TARI	Université de Béjaia	Examinateur
Docteur Faiza BELALA	Université de Constantine	Examinatrice

Année 2011 -2012

Remerciements

Je tiens tout d’abord à remercier ma directrice de thèse Professeur Zizette Boufaida pour tout ce qu’elle m’a apporté : idées, orientations et encouragements.
Je remercie également Professeur Mahmoud Boufaida de m’avoir accueillie dans son équipe SIBC au laboratoire LIRE de Constantine.
Je remercie les membres du jury d’avoir accepté d’évaluer ce travail de thèse. D’abord, je remercie Professeur Zaidi Sahnoun d’avoir accepté de présider ce jury. Aussi, je remercie Docteur Faiza Belala, Professeur Mohamed Taib Laskri et Docteur Mohamed el Kamel Tari d’avoir accepté d’examiner mon manuscrit de thèse. Je suis honorée par leur présence dans mon jury de soutenance.
Je tiens à remercier Professeur Michel Riveill de m’avoir accueillie dans son équipe à l’université de Nice-Sophia Antipolis.
Je n’oublie pas de remercier Docteur Zahia Guessoum de m’avoir si gentiment reçue au LIP6 de Paris.
Je ne remercierai jamais assez ma très chère sœur Amel pour son aide précieuse et ses encouragements.
Enfin, un grand merci à mon très cher époux Zineeddine pour son soutien et sa patience.

ملخص

في هندسة البرمجيات القائمة على عنصر، مرحلة الاختيار مهمة جدا و لكن صعبة التنفيذ. هذا العمل يتعلق باستكشاف الوسائل التقنية لاختيار العناصر الموجودة في المستودعات من أجل تطوير البرمجيات و بالأخص نعمل في إطار مرحلتي التعرف و التقييم.

في المرحلة الأولى هدفنا هو التعرف على معايير نوعية الجودة ونستعمل من أجل ذلك الأنطولوجيا التي سميناها "QoSOntoCS ".

أما في المرحلة الثانية فنبحث على العناصر معتمدين على المعايير السابقة.

في الأخير، نحلل نتائج التقييم باستخدام الطريقة"MCDA ". هذه الطريقة مفيدة في اختيار أحسن عنصر.

من خلال هذا العمل حددنا الإطار الهندسي لاحتواء المراحل المذكورة أعلاه.

Résumé

La réutilisation des composants logiciels en génie logiciel est un problème complexe et difficile à résoudre à cause de la difficulté qu'on a à évaluer la pertinence d'un composant pour un développement donné. Notre travail de thèse apporte une solution au problème de la sélection de composants logiciels, stockés dans des référentiels distants, en vue de leur réutilisation lors de la phase de développement du logiciel. Plus particulièrement, nous proposons une solution pour les étapes d'identification et d'évaluation de composants logiciels.

Dans l'étape d'identification, notre objectif est de déterminer les critères de qualité de service qui permettent d'amorcer l'étape de recherche de composants. Ces critères sont hiérarchisés et décrits dans une ontologie (QoSOntoCS) modélisant la QoS comme une hiérarchie de caractéristiques, de sous caractéristiques et d'attributs, conformément au standard "ISO/IEC 9126". Les attributs constituent le niveau le plus bas de la hiérarchie et sont mesurables par des métriques.

Dans l'étape d'évaluation des composants recherchés, notre technique vise à rechercher les composants qui satisfont les critères de qualité spécifiés par le développeur. Les niveaux d'appariement (matching) proposés résultent des liens taxinomiques définis dans notre ontologie "QoSOntoCS " et considèrent les aspects spécifiques aux composants logiciels.

Afin de sélectionner le meilleur composant à proposer au développeur, nous analysons les résultats d'évaluation. Pour cela, nous appliquons la technique "MCDA "qui permet de classer les composants retenus à l'issue de l'étape d'évaluation en fonction des préférences du développeur pour certains critères plutôt que d'autres. En somme, nous proposons un système de sélection de composants logiciels dans lequel nous utilisons (1) des modèles issus du domaine du Web sémantique pour décrire et rechercher sémantiquement des composants logiciels et (2) la technique de décision multi-critères "MCDA " pour sélectionner, in fine, le meilleur composant. La faisabilité de notre approche est démontrée en implémentant l'algorithme de matching proposé. Par ailleurs, des tests ont été réalisés pour déterminer les performances de notre solution.

Abstract

Reuse of software components in the software engineering is a complex and a hard problem because the issue of evaluating the component relevance for one development is not yet solved. Our thesis proposes a solution to the component selection problem where components are stored in remote repositories, in order to reuse them in the software development. Mainly, we propose a solution to the steps of identifying and selecting software components. In the identification step, our goal is to determine the criteria for quality of service helping the components research phase. These criteria are hierarchical and semantically represented by an ontology that describes the characteristics and the attributes of quality of service in accordance with the "ISO/IEC 9126" model. In the evaluation (or assessment) phase of the component selection, our technique allows to select components that meet the quality criteria specified by the developer. First, the matching levels are based on both the taxonomic relationships defined in our ontology and the component-specific aspects. Then, in order to select the best component to propose to the developer, we analyze the evaluation results by applying the "MCDA" technique. This latter classifies the retrieved components according to the developer preferences. In sum, we propose a selection process of software components by using (1) models from the field of Semantic Web to semantically describe and search software components and (2) "MCDA" technique to select, in fine, the best component. The feasibility of our approach is shown through the implementation of the proposed matching algorithm and tests are conducted in order to determine the performance of our solution.

Table des matières

Liste des figures

Liste des tableaux

Introduction Générale

Problématique et cadre théorique de la recherche

Les processus de génie logiciel traditionnels sont coûteux en termes de temps et d'effort de développement. Une des raisons à cette situation est que la réutilisation de composants logiciels existants n'est souvent pas la préoccupation centrale de ces processus. Quand bien même la réutilisation est considérée dans certains cas, souvent seules les fonctionnalités d'un composant sont prises en compte au détriment des aspects non fonctionnels relatifs à la qualité de service (QoS[1], pour Quality Of Service en anglais) tels que la fiabilité du composant, sa disponibilité et sa sécurité.

Partant de ce constat, la problématique centrale de notre travail est la réutilisation de composants logiciels en prenant en compte la QoS requise ou rendue par ces composants. Dans le cadre de cette thèse, nous apportons une solution à la question de recherche suivante: Comment aider les développeurs à sélectionner des composants logiciels répondant à une certaine QoS et cela dans un domaine où complexité et hétérogénéité sont les mots d'ordre? Il s'agit de réutiliser des composants logiciels pré-existants et d'éviter de tout développer à partir de zéro (from scratch).

Nous apportons deux éléments de réponse à cette question de recherche : En premier lieu, nous décrivons sémantiquement la QoS d'un composant logiciel en se basant sur un formalisme ontologique et, en second lieu, nous proposons un mécanisme de sélection de composants logiciels répondant aux besoins de QoS des développeurs. L'objectif de notre travail est d'automatiser au maximum le processus de sélection de composants logiciels afin d'économiser, lors du développement, l'effort, le coût et le temps de réalisation.

Pour atteindre cet objectif, nous avons fait trois choix conceptuels et techniques :
D'abord, nous réutilisons les composants logiciels existants dans différents référentiels ou COTS (Component On-The-Shelf), c'est-à-dire nous nous intéressons aux composants sur étagère. Ce choix permet d'économiser le temps et l'effort nécessaires au développement d'un logiciel. Toutefois, cette économie n'est possible que si les moyens qui favorisent la

[1] Dans le reste de l'introduction, nous utilisons l'acronyme QoS pour désigner les termes qualité de service

réutilisation des composants logiciels sont déjà mis en œuvre. Pour cela, nous proposons une ontologie qui permet de formaliser la description des composants logiciels et en particulier la description de leurs caractéristiques de QoS. Les descriptions ontologiques (ou annotations sémantiques) produites sont stockées dans une base de connaissances locale afin de permettre la sélection automatique des composants logiciels.

Ensuite, nous adoptons le processus de sélection GCS (pour **G**eneral **C**OTS **S**election) (Mohamed et al., 2007) pour le choix de composants logiciels pertinents, c'est-à-dire ceux dont la QoS correspond à celle spécifiée par le développeur. Notons qu'un processus de sélection COTS comprend les étapes suivantes :

- Etape 1 : Définir les critères d'évaluation basés sur les exigences et les contraintes du développeur;
- Etape 2 : Rechercher les composants à partir des référentiels ;
- Etape 3 : Filtrer les résultats de recherche basée sur les besoins fonctionnels du développeur. Ce qui permet la définition d'une liste restreinte de composants prometteurs qui doivent être évalués de manière plus détaillée ;
- Etape 4 : Évaluer les composants candidats de la liste restreinte ;
- Etape 5 : Analyser les données d'évaluation et sélectionner le composant qui possède la meilleure correspondance avec les critères du développeur.

Le choix de ce processus est justifié par le fait que les auteurs (Mohamed et al., 2007) cadrent la démarche de sélection et ne préconisent aucune méthode pour la mettre en œuvre. Ce processus nous permet donc de suivre des étapes précises sans pour autant se limiter à un formalisme donné ou à une technique particulière. Aussi, nous focalisons notre recherche sur les étapes 1, 4 et 5, à savoir une identification des critères de QoS, une évaluation des composants par rapport à ces critères et une analyse qui a pour but de classer, par ordre de pertinence, les composants retenus lors de l'évaluation.

Enfin, nous proposons une architecture de sélection de composants logiciels réutilisables que nous baptisons « QoS-CSA » (**QoS**-based **C**omponent **S**election **A**rchitecture) où nous exploitons : (1) les modèles et technologies du Web sémantique pour les étapes d'identification et d'évaluation, et (2) la méthode de prise de décision multi-critères MCDA (**M**ulti-**C**riteria **D**ecision **A**id) pour l'étape d'analyse de l'évaluation.

Notre travail s'inscrit dans le domaine du génie logiciel orienté composant (CBSE, **C**omponent-**B**ased **S**oftware **E**ngineering en anglais) et plus précisément le domaine de l'ingénierie par réutilisation (« design by reuse ») où l'activité de sélection de composants logiciels est importante et multi-critères. Notre travail s'intéresse aux critères de QoS pour la sélection de composants. Généralement, deux scénarii sont possibles pour la sélection. Le premier scénario concerne la sélection de composants primitifs alors que le second prend en charge les composants composites, c'est-à-dire des composants contenant d'autres composants appelés sous-composants. Un composant composite possède une structure hiérarchique de sous-composants et peut être placé dans des référentiels. Cependant, la plupart des modèles de composants n'offrent pas de référentiels pour ce type de composants. De plus, la méthode de sélection se complexifie lorsqu'il s'agit de composants composites. Par conséquent, nous n'allons pas traiter cette problématique qui peut constituer un sujet de thèse à part entière.

Par ailleurs, un des problèmes ouverts dans le domaine des composants logiciels concerne l'assurance de la QoS. L'évolution de ces composants en un assemblage ne garantit pas forcément la QoS exigée par le développeur. Cette problématique appartient au domaine de l'intégration de composants logiciels qui n'est pas du ressort de ce projet.

Cadre pratique de la recherche

Nous nous plaçons à posteriori de l'étape 3 du processus GCS (sélection fonctionnelle) et nous considérons que les composants sont déjà pertinents du point de vue fonctionnel. L'étape de sélection fonctionnelle est en dehors du cadre de notre travail. Nous nous intéressons uniquement à la sélection de composants en fonction de leur QoS.

Nous proposons donc l'architecture « QoS-CSA » qui permet, à des développeurs, de spécifier leurs besoins en termes de QoS. Une fois les critères spécifiés, le système permet de sélectionner le composant qui répond le mieux aux attentes du développeur. Pour ce faire, nous avons choisi de décrire les connaissances sur le composant recherché et les connaissances sur les composants candidats dans un langage formel et partagé. D'une part, la formalité du langage a pour but de faciliter l'automatisation du processus de sélection initié par le développeur sachant que la description commune des connaissances sur les composants améliorent l'interopérabilité et réduisent l'hétérogénéité du système. Ce langage est une ontologie qui permet une explicitation formelle et partagée des connaissances sur la QoS.

Nous avons baptisé cette ontologie « QoSOntoCS » (**QoS Onto**logy for **C**omponent **S**election). Ainsi, elle intègre deux niveaux d'abstraction:

1. Modèle générique : représente des connaissances génériques sur les composants logiciels. Il s'agit de décrire les caractéristiques externes des composants et de capitaliser les requêtes déjà exécutées (sélections antérieures). Les connaissances sur les composants, les acteurs (indexeurs et sélectionneurs) et les sélections antérieures sont stockés dans la base de connaissances. Cela permettra une meilleure réutilisation au sein de l'architecture QoS-CSA.
2. Modèle spécifique au domaine (ou MSD) : permet de représenter les connaissances sur la QoS des composants logiciels. Ce niveau est primordial pour représenter des concepts concrets de la QoS. Notre choix s'est porté sur CQM (Alvaro et al., 2006), un modèle de QoS conforme au standard "ISO/IEC 9126" et adapté aux composants logiciels réutilisables.

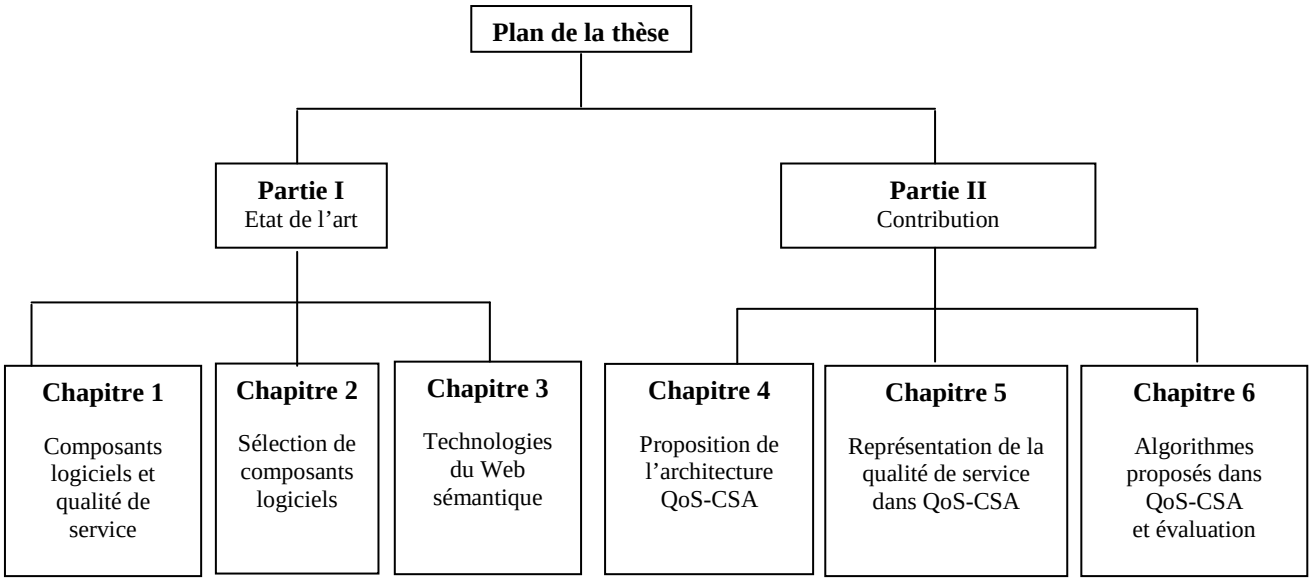
La base de connaissances, générée par cette ontologie, constitue le module principal de notre architecture. Cette dernière englobe également d'autres modules de traitements (par exemple, ComponentAnnotation et Qo2Selection). L'API Jena, combiné au raisonneur Pellet, permet de manipuler les connaissances du système. D'un côté, la requête spécifiée par le développeur représente l'annotation du composant recherché et de l'autre côté la base de connaissances comprend un ensemble d'annotations de composants candidats (disponibles dans des référentiels tels que ComponentSource[2]). Nous proposons le calcul d'un score de pertinence pour chaque composant candidat relativement à chaque critère de qualité spécifié dans le composant recherché. Nous obtenons donc, pour chaque composant, un ensemble de scores locaux. Ces derniers permettent de calculer un score de pertinence global, pour chaque composant, en utilisant la technique MCDA. Ces scores se basent sur les relations de subsomption existant entre les différents critères de qualité ainsi que la vérification de la conformité entre les valeurs d'une métrique de qualité.

Nous implémentons enfin notre approche de sélection de composants au sein de l'architecture QoS-CSA afin de tester sa faisabilité et sa performance. Pour cela, nous reprenons l'exemple (George, 2007) du composant de téléchargement FTP (File Transfer Protocol) du référentiel ComponentSource. Dans la section « Communication Internet » de ComponentSource,

[2] http://www.componentsource.com/index.html

plusieurs composants traitant du protocole FTP offrent un service de téléchargement. Nous focalisons sur les cinq attributs de qualité suivants: Data Encryption, Memory Usage, Disk Usage, Mobility et Cost. Evidemment, ces attributs sont déjà définis par le MSD.

Plan du mémoire



Cette thèse est composée de deux parties.

- La première partie présente un état de l'art relatif à notre problématique de recherche. Elle est structurée en trois chapitres :

  - ✓ Au premier chapitre, nous présentons d'abord la notion de composant logiciel ainsi que son évolution depuis la notion d'objet à la notion de service aujourd'hui. Nous expliquons ensuite le concept de QoS pour les composants logiciels au travers de quelques modèles et standards de qualité. Enfin, nous pointons le doigt sur certaines problématiques du génie logiciel orienté composants.

  - ✓ Le second chapitre porte sur la problématique de sélection de composants logiciels. Nous présentons quelques techniques de recherche et de sélection de composants qui peuvent être utilisées dans le processus GCS. Nous optons pour ce processus après l'étude de différentes méthodes de sélection existantes.

  - ✓ Au troisième chapitre, nous décrivons des modèles et des techniques du domaine du Web sémantique et nous montrons l'apport du raisonnement et de l'appariement, issus de ce domaine, dans les processus de sélection des composants logiciels.

- La seconde partie présente, de manière détaillée, le cœur de notre travail de thèse. Cette partie est composée de trois chapitres.

- ✓ Nous détaillons, dans le quatrième chapitre, l'architecture de sélection « QoS-CSA » que nous avons proposée. Nous commençons par décrire les deux cas d'utilisation relatifs au processus de sélection, à savoir l'annotation de composants et la sélection d'un composant. Nous définissons par la suite les composants constituant QoS-CSA.

- ✓ Nous décrivons, dans le cinquième chapitre, l'ontologie QoSOntoCS que nous avons construite pour la représentation des connaissances sur les composants logiciels et en particulier leur QoS. Nous expliquons comment ces connaissances sont exploitées dans le processus de sélection.

- ✓ Au dernier chapitre, nous discutons les deux principaux algorithmes proposés dans le cadre de ce projet, à savoir le matching de QoS et le ranking de composants. Aussi, nous pratiquons une évaluation afin de valider notre approche de sélection sémantique.

Enfin, nous terminons ce travail par une conclusion générale incluant ses améliorations et ses perspectives.

Partie I

Etat de l'art

Chapitre 1

Composants logiciels et qualité de service

L'approche orientée composant apporte une nouvelle vision au domaine du génie logiciel où la construction d'applications se base essentiellement sur des entités existantes appelées composants logiciels. Cette vision a pour but d'améliorer l'efficacité du développement de logiciels et répondre ainsi à la première vocation du génie logiciel. L'approche orientée composant n'a pas encore atteint la maturité de l'approche objet et plusieurs problématiques de recherche restent encore à traiter. Dans le cadre de cette thèse, nous nous intéressons au problème de la qualité de service des applications à base de composants logiciels. Ce chapitre est organisé comme suit :

D'abord, nous présentons les origines historiques de l'approche orientée composant. Ensuite, nous abordons les points essentiels qui nous permettent d'expliquer cette approche en général et la notion de qualité de service (QoS) en particulier. Enfin, nous décrivons les deux cycles de développement qui caractérisent le génie logiciel orienté composant afin d'aborder quelques problématiques existantes dans ce domaine.

1. Des objets aux services

1.1. Origines historiques

La notion de composant logiciel n'a pas encore reçu une définition aussi complète et communément acceptée que celle d'objet (Macedo, 2004). Cette notion est apparue en 1968 lors du workshop NATO Software Engineering lors de la présentation de M. McIllroy intitulé «Mass Produced Software Component » (McIllroy, 1968). Le but ultime est de pouvoir construire des applications par assemblage de composants logiciels (Magee et al., 1995) par analogie aux composants matériels à l'image, par exemple, des pièces détachées dans l'industrie automobile. Cependant, la réutilisation d'un composant logiciel implique nécessairement la connaissance du rôle de ce composant au sein d'un assemblage, c'est-à-dire la façon dont il va interagir avec son environnement et vice versa. Ainsi, pour réutiliser un composant logiciel, il est donc nécessaire de lui associer des spécifications précises qui décrivent son fonctionnement ainsi que ses dépendances vis-à-vis de son environnement.

Dans la pratique, il existe trois approches qui favorisent la réutilisation des composants logiciels dans le but d'améliorer le développement et la maintenance des logiciels : (1) l'approche orientée objet, élaborée par Alan Kay dans les années 1970 (2) l'approche orientée composant, apparue ultérieurement dans les années 90 et (3) l'approche orientée service, arrivée quant à elle à la fin des années 90.

L'approche orientée objet a apporté essentiellement les principes d'encapsulation et d'abstraction. L'idée est de cacher les détails d'implémentation aux utilisateurs qui ne manipulent les objets qu'à travers leurs interfaces. Objets et composants partagent un certain nombre de motivations communes (Le Meur et al., 2007) : ils possèdent tous les deux des propriétés encapsulées, sont accessibles par des interfaces bien spécifiées et améliorent la réutilisation du logiciel. De plus, les composants comme les objets se déploient et s'exécutent dans des contextes qui leur sont spécifiques. Cependant, le composant permet, par le biais de ces contextes, de mettre en œuvre des paramètres non fonctionnels tels que la sécurité, la persistance, les mécanismes transactionnels, etc.

Le service quant à lui prône une indépendance vis-à-vis des aspects technologiques permettant ainsi un couplage faible (Le Meur et al., 2007), c'est-à-dire un service est indépendant du contexte d'utilisation ainsi que de l'état des autres services ; et inter-opère avec eux via des protocoles standardisés. Un service définit donc une entité logicielle (par

exemple ressource, application, module, composant logiciel, etc.) qui communique avec d'autres services par échanges de messages et qui expose un contrat d'utilisation (Mahesh et Dodani, 2004). Relativement aux approches à objet ou à composant, l'approche orientée service cherche à fournir un niveau d'abstraction supérieur, en encapsulant des fonctionnalités et en permettant la réutilisation de services existants.

1.2. Composant vs service

Les approches orientée composant et orientée service sont similaires du point de vue de la pratique. Elles partagent les activités d'identification d'entités réutilisables (composant ou service) dans le but de répondre aux besoins des activités de composition de ces entités. L'objectif principal est de réaliser l'application globale en exploitant les notions de composition d'entités réutilisables. Cependant, les théories derrière ces deux approches sont différentes (Hock-koon and Oussalah, 2012).

Alors qu'un composant est disponible pour son utilisateur, un service est centré sur l'utilisation d'une fonction fournie par un tiers (Stojanovic and Dahanayake, 2005). Pour illustrer cette première distinction, dans (Hock-koon and Oussalah, 2012), on donne un exemple de l'industrie des jeux vidéo sur PC. Cette industrie exploite deux modèles de distribution de contenus : (1) le modèle classique où le joueur achète une copie de son jeu. Il est ensuite responsable du déploiement de cette copie sur sa propre machine et (2) le modèle Cloud gaming où le joueur paie le droit d'accéder à un jeu qui est exécuté sur une plate-forme à distance sous la responsabilité du fournisseur. Il a uniquement besoin d'une interface et d'une connexion pour accéder à cette plate-forme. Le premier modèle correspond à une approche orientée composant. Le jeu (composant) doit donc être accompagné d'une documentation qui définit les contraintes côté client (ex. configuration système minimale requise, instructions d'installation, …) afin de pouvoir exécuter ce jeu. Tandis que dans le second modèle où la notion de service est illustrée, le joueur est déchargé des contraintes précédentes. Dans ce cas le fournisseur est responsable du déploiement, de l'exécution et de la gestion du service. La localisation physique du service et ses évolutions sont transparentes pour le client.

De plus, le service exécuté à distance doit être capable de gérer de multiples connexions parallèles. Ce principe n'est pas nécessaire à un composant. En effet, bien que pouvant appartenir à de multiples compositions au runtime, différentes instances du composant sont créées et chacune d'elles est déléguée à un client.

Contrairement au composant qui doit suivre un modèle particulier parmi la panoplie de modèles existants (Crnkovic et al., 2007), le service prône un unique modèle homogène de services (Erickson and Siau, 2008) qui doit être standardisé et utilisé par tous. La collaboration entre modèles de composants différents est une tâche très difficile (Crnkovic et al., 2006) (par exemple, les ponts entre COM et CORBA).

Même si le principe d'automatisation est recherché dans l'approche orientée composant, il n'est pas explicité dans la définition des modèles de composants. Cependant, il est présent dans la définition du service (MacKenzie et al., 2006). En effet, la grande majorité des travaux cherchent à automatiser les mécanismes liés aux services tels que la publication de services, la découverte et la sélection de services, la composition de services, etc. Ce principe d'automatisation est poussé à son extrême par le concept d'auto-adaptation (Nitto et al., 2008) qui cherche à coordonner l'ensemble des mécanismes liés au service afin d'assurer des adaptations contextuelles réactives ou proactives.

Toutes les caractéristiques abordées soulignent le caractère dynamicité des services. Ainsi, les auteurs de (Hock-koon and Oussalah, 2012) ont rajouté le mot clé « dynamicité » pour distinguer l'approche orientée service de l'approche orientée composant, définit dans (Crnkovic et al., 2007) en utilisant les deux mots clés : réutilisabilité et composabilité.

2. Approche orientée composant

2.1. Définition d'un composant logiciel

Afin de présenter et d'expliquer le concept de composant, nous avons cité plusieurs définitions issues de la littérature. En dépit de l'hétérogénéité de ces définitions (qui traite chacune d'un point de vue différent), elles ont un point de convergence : la réutilisabilité. La définition qui nous servira de guide dans la suite de cette thèse est extraite de (Szyperski, 2002) : « Un composant logiciel est une unité de composition ayant des interfaces contractualisées ainsi que des dépendances contextuelles. Ces dépendances explicitent les interfaces exigées ainsi que les plate-formes d'exécutions possibles. Un composant peut être déployé indépendamment, et être sujet à la composition». Un composant est donc une boite noire avec des interfaces bien définies pour les services qu'il fournit ou requiert. Les applications sont ensuite construites par assemblage (ou composition) de composants en fonction de leurs interfaces fournies et requises.

2.2. Caractéristiques fonctionnelles et non fonctionnelles

En s'appuyant sur la définition de (Szyperski, 2002), nous caractérisons un composant logiciel par un ensemble d'interfaces fournies et requises. Les interfaces fournies regroupent l'ensemble des services offerts par un composant aux autres composants. Les interfaces requises, quant à elles, contiennent ce dont un composant a besoin pour fonctionner. Dans les deux cas l'interface d'un composant contient un ensemble de propriétés fonctionnelles et non-fonctionnelles qui sont visibles depuis son environnement (George, 2007).

Les propriétés fonctionnelles des interfaces prennent la forme d'opérations dont on donne la signature et parfois la sémantique. La signature d'une opération est de la forme *nomOp = (param1, param2,…, paramN)* → *result, où nomOp* est le nom de l'opération ; *(param1, param2,…, paramN)* est l'ensemble de ses paramètres ou arguments ; et *result* est son résultat. Les aspects sémantiques prennent la forme de prédicats (pré-conditions, post-conditions et invariants) associés aux interfaces et aux opérations. Ces prédicats représentent des spécifications comportementales.

Les propriétés fonctionnelles décrivent les aspects syntaxiques et sémantiques des services fournis et requis par un composant, alors que les propriétés non fonctionnelles représentent la manière de rendre ou d'attendre ces services. Dans le cadre de cette thèse, nous considérons de manière interchangeable les propriétés non fonctionnelles et la QoS. Par exemple, dans (ITU, 1994), on définit la QoS comme étant « l'effet combiné des performances d'un service qui déterminent le degré de satisfaction de l'utilisateur de ce service ». Dans cette définition, on considère que la QoS est un sous ensemble de propriétés non fonctionnelles appelé Performance (Efficiency en anglais). Néanmoins, nous nous basons sur le standard ISO/IEC 9126 (ISO, 2001) qui définit un large éventail de propriétés non fonctionnelles appelées caractéristiques (voir la section 3.2.1). Aussi, dans le standard ISO/IEC 9309 (OMG, 1995), on définit la QoS comme étant « un ensemble de qualités liées au comportement collectif d'un ou plusieurs objets ». Cette définition stipule l'existence de caractéristiques dynamiques. En effet, nous distinguons deux types de caractéristiques :

- Caractéristiques statiques : leur quantification est déterminée lors du développement et ne change pas à l'exécution, par exemple les besoins d'installation.
- Caractéristiques dynamiques : leur quantification n'est pas stable et peut changer à l'exécution, par exemple le débit ou le temps de réponse.

Enfin, il est communément admis que la QoS d'une application dépend étroitement de la QoS des composants qui l'intègrent (Bass et al., 2003 ; Simao and Belchior, 2003).

2.3. Architectures à base de composants

Une architecture à base de composants est définie comme une description abstraite de l'application, de sa décomposition en composants, de l'interface de ces composants et de leurs interactions.

2.3.1 Langages de description d'architectures

Un langage de description d'architectures (ADL, pour Architecture Description Language en anglais) (ACCORD, 2002) permet la définition d'un vocabulaire précis et commun à destination des acteurs devant travailler autour de la spécification liée à l'architecture (architectes, concepteurs, développeurs, intégrateurs et testeurs). Son objectif est triple : (1) Il spécifie les composants de l'architecture de manière abstraite sans entrer dans les détails d'implémentation, (2) il définit de manière explicite les interactions entre composants d'un système et (3) il fournit un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments.

Les éléments de base qui forment la plupart des ADLs sont : le composant, le connecteur et la configuration. Le *connecteur* correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants. Il définit les règles qui régissent les interactions, par exemple un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux. La *configuration*, quant à elle, définit la structure et le comportement d'une application formée de composants et de connecteurs. Elle détermine les composants et les connecteurs appropriés à l'application et vérifie la correspondance entre les interfaces des composants et des connecteurs. La configuration est comportementale si elle modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs ; et l'évolution des propriétés de la QoS. Elle définit également le schéma d'instanciation des composants au moment de l'initialisation de l'application, ainsi que le placement des composants sur les sites au moment du démarrage du système et leur évolution pendant le cycle de vie de l'application.

2.3.2 Contractualisation de composants

La contractualisation est une approche qui permet de préciser les types de données en définissant une partie de leur mise en œuvre ou simplement en faisant apparaître des contraintes masquées liées à cette mise en œuvre (Barais et al., 2003). Il est donc nécessaire de préciser pour chaque composant un ensemble de contrats correspondant aux niveaux identifiés par (Beugnard et al., 1999) :

- Contrat basique, relatif aux propriétés syntaxiques (noms des méthodes et types des paramètres) et les propriétés sémantiques simples (exprimées par un langage de définition d'interfaces) ;
- Contrat comportemental, relatif aux propriétés qui peuvent être spécifiées avec des pré-conditions, des post-conditions et des invariants, en incluant les contraintes sur l'historique. Ces propriétés sont liées à une méthode, vue comme une unité atomique ;
- Contrat de synchronisation, relatif aux propriétés qui concernent les interactions entre composants. Cela correspond à la prise en compte des interactions du composant avec l'extérieur, mais ayant une influence sur son comportement interne. On peut décrire leur exécution sous forme d'une suite d'appels de méthodes selon un motif spécifié ;
- Contrat qualitatif, relatif aux propriétés non fonctionnelles, telles que la fiabilité, la disponibilité et la sécurité.

L'intérêt d'enrichir la description des interfaces avec des descriptions précises est de faciliter la réutilisation de composants.

2.4. Modèles de composants

Les auteurs du livre « Component-Based Software Engineering : Putting the Pieces Together », ont donné la définition suivante (Heineman et Councill, 2001) :

« Un composant est un élément logiciel conforme à un modèle appelé modèle de composants qui définit des standards de composition et d'interactions ».

Cette définition focalise sur l'existence de plusieurs modèles de composants et que chaque modèle définit des règles pour l'assemblage de composants. Cependant, la réalité révèle que les modèles de composants les plus industrialisés sont des modèles plats qui ne proposent qu'une unique dimension de configuration pour une application.

Nous présentons dans ce qui suit quelques modèles de composants appartenant à deux catégories : les modèles plats et les modèles hiérarchiques.

2.4.1 Modèles plats

La plupart des modèles industriels appartiennent à cette catégorie (CCM, EJB,.NET, …). La composition des composants (boites noires) s'effectue sur un unique niveau. Nous présentons, dans ce qui suit, un exemple de ces modèles.

Modèle CCM

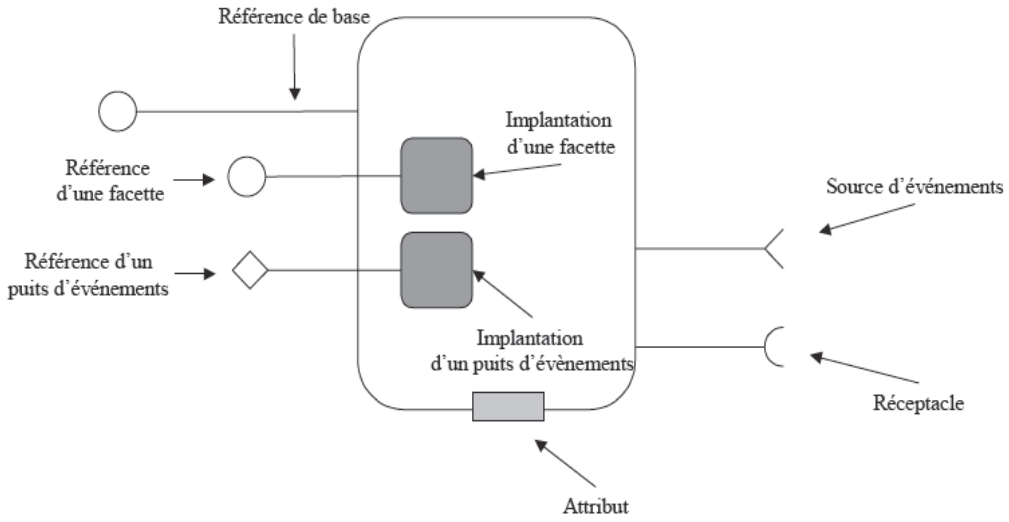


Figure 1.1 : Modèle de composants CCM

Un composant CCM est constitué de :

- Une référence de base ;
- Des interfaces : 4 types de port (la facette, le réceptacle, la source et le puits d'événements) ;
- Des attributs.

A ces éléments constituant un composant CCM est associé une fabrique (Home) qui constitue un gestionnaire pour les instances de composants, permet de créer à l'exécution des instances de même type et permet également la recherche d'une instance en utilisant des clés (OMG, 2002).

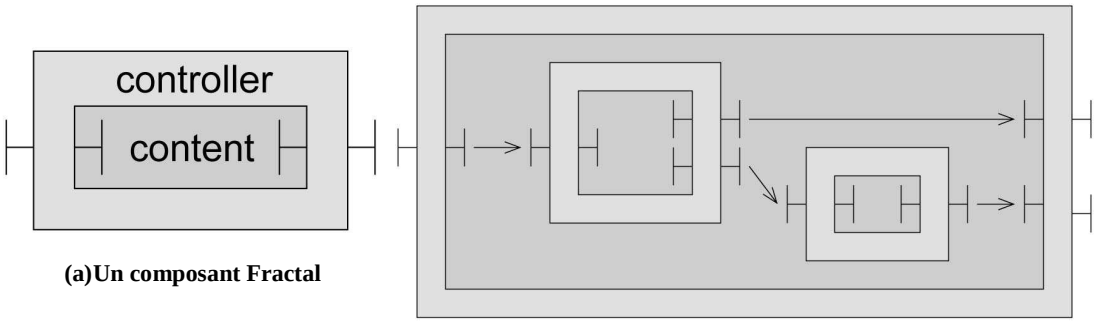
2.4.2 Modèles hiérarchiques

Un composant composite est vu comme étant un graphe hiérarchique d'autres composants (appelés sous-composants) avec des composants primitifs comme feuilles. Contrairement au composant composite, un composant primitif contient le code métier. Les sous-composants ne

peuvent être reliés que dans un même niveau d'imbrication et leurs interfaces peuvent être déléguées respectivement à leurs composants composites. Les modèles hiérarchiques permettent la création de composants composites (appelés boites grises). Nous présentons dans ce qui suit un exemple de ces modèles.

Modèle Fractal

Fractal (Bruneton, 2004) est un framework de composition qui définit un modèle de composants modulaire extensible et générique. Ce framework peut être utilisé avec différents langages de programmation pour concevoir, implanter, déployer et reconfigurer les systèmes de toute nature : systèmes d'exploitation, intergiciels ou interfaces graphiques. Le modèle de composants Fractal adopte le principe de séparation des préoccupations (Yessad, 2007) dans la conception des composants et des systèmes d'information à base de composants. Ce principe préconise la séparation des aspects et des préoccupations d'un système d'information en plusieurs entités distinctes : par exemple, implémenter les services fonctionnels fournis par le système d'information dans des entités différentes de celles qui assurent les aspects non fonctionnels comme la configuration, la sécurité et la disponibilité, etc.



(a)Un composant Fractal

(b) Composition de composants dans Fractal

Figure 1.2 : Modèle de composants Fractal

3. Représentation de la QoS

Dans l'approche orientée composant, la QoS correspond au niveau quatre de (Beugnard et al., 1999). L'idée serait d'associer à un composant une description de QoS conforme à un modèle donné. Toutefois, des composants ayant la même fonctionnalité peuvent avoir différentes

descriptions de QoS d'où l'intérêt de spécifier individuellement la QoS de chaque composant. Malheureusement, il n'existe pas de modèle formel et commun pour décrire la QoS des composants.

Le développeur d'un composant devrait disposer d'un formalisme adapté pour exprimer toutes les contraintes liées à la qualité de service fournie et requise par un composant. Dans ce qui suit, nous verrons comment les travaux existants ont pu représenter la qualité de service, ce qui nous a éclairés pour choisir notre propre formalisme de représentation.

3.1. Langages de spécification de la QoS

3.1.1 QML (QoS Modeling Language)

QML (Frolund and Koistinen, 1998) est un langage de spécification de contraintes de qualité de services. Les qualités de service visées par QML sont très générales, regroupées en catégories comme la fiabilité, la sécurité et la performance. Chaque catégorie définit une ou plusieurs dimensions (avec une unité) qui mesure un aspect de la catégorie.

Les métriques (ou mesures) sont numériques pour les aspects quantitatifs (quantités, pourcentages, seuils, etc.) ou ensemblistes pour les aspects qualitatifs (ensembles de valeurs ordonnées). Des précautions sont prises pour pouvoir comparer des métriques et déterminer si une valeur est préférable à une autre (stronger than, en aglais), afin de déterminer des compatibilités (conformance) ou de permettre des négociations lors du processus de sélection de composants.

L'approche est contractuelle et applique des principes de séparation des préoccupations. Les contrats sont des instances personnalisées et typées. Le lien entre les contrats non fonctionnels et les aspects fonctionnels (interfaces) est établi dans une construction appelée *profile*. Dans une application répartie, les clients peuvent personnaliser des contrats non fonctionnels requis. Les serveurs proposent des contrats fournis qui peuvent être différents, mais doivent être conformes ou préférables. Par exemple, un délai d'attente fourni delay < 10 ms est préférable à un délai requis delay < 20 ms, donc est conforme. Inversement, un débit throughput >100 mb/sec est préférable à un débit throughput >10 mb/sec, donc est conforme. QML se distingue par sa logique de mise en œuvre de métriques qui a pour intérêt de rendre comparable des spécifications de manière statique. Compatibilité et substituabilité des entités spécifiées peuvent ainsi être vérifiées avant exécution.

Le langage QML possède de nombreuses qualités (par exemple : généralité d'utilisation, lisibilité, séparation des préoccupations, réutilisabilité). Cependant, il présente certaines lacunes :

- Les dimensions sont déclarées dans les contrats mais pourraient être avantageusement définies dans des classes séparées afin d'être mieux réutilisables.
- Les contraintes exprimables sont relativement pauvres et non paramétrées par des arguments;
- Les profils établissent un couplage imparfait entre les contraintes de qualité et les aspects fonctionnels.

3.1.2 CQML (Component QML)

CQML (Aagedal, 2001) reprend les concepts principaux de QML en les développant. Il s'articule autour de quatre concepts : la métrique, le contrat, l'association avec les entités d'implémentation et l'utilisation de catégories pour organiser les trois concepts précédents.

Une caractéristique de QoS correspond au minimum à une grandeur dont le domaine est un intervalle, un ensemble ou une énumération. A l'instar de QML, CQML propose la dérivation qui consiste à définir une grandeur en fonction d'une autre. Les dérivations fournies sont essentiellement statistiques et sont plus nombreuses que dans QML. Les caractéristiques acceptent des paramètres et peuvent inclure dans leur définition des invariants, exprimés à l'aide de prédicats OCL[3]. De plus, les valeurs possibles d'une grandeur peuvent être exprimées par une formule OCL en fonction des paramètres de la caractéristique.

[3] Object Constraints Language

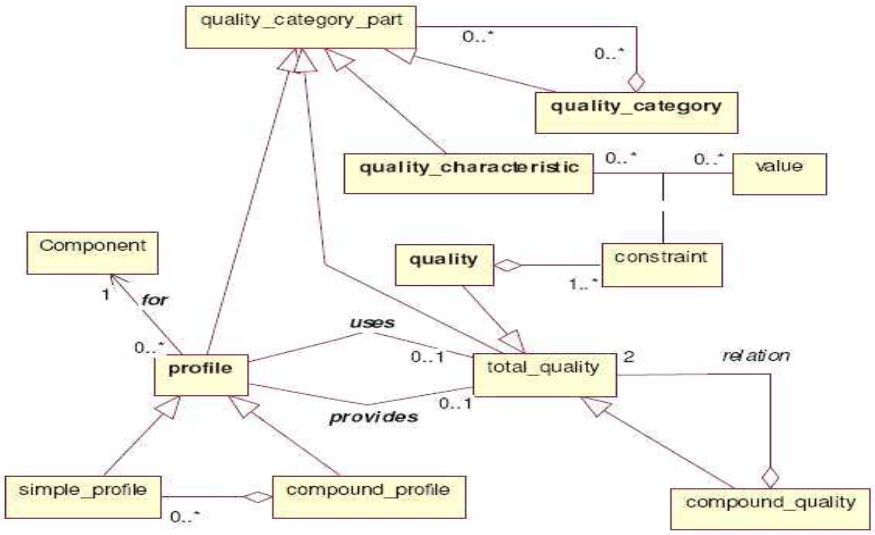


Figure 1.3 : Modèle de description de la QoS dans CQML (Aagedal, 2001)

La composition de composants peut être prise en compte du point de vue de la QoS. CQML distingue la relation entre un composant utilisé par un autre et la relation entre plusieurs composants eux-mêmes utilisés par un autre. Dans le premier cas, la composition est modélisée par l'opérateur "sequential" indiquant la dépendance d'un composant par rapport à un autre. Dans le second cas, les composants sont indépendants. Néanmoins, vis-à-vis de leur utilisateur, ils sont susceptibles d'être utilisés conjointement ou non.

Les expressions de QoS regroupent une liste de contraintes sur des caractéristiques de QoS, à la manière des contrats de QML. Le corps de chaque contrainte est une expression OCL. Sa validité peut être modulée dans le cas de contraintes statistiques qui requièrent de manière globale plusieurs mesures avant de pouvoir être décidées.

Les profils QML et CQML ont en commun le principe d'attacher un contrat à une entité d'implémentation. Le profil CQML définit les spécifications de qualité de service fournies et requises par un composant. Cependant, la spécification est moins fine que dans QML : CQML ne descend pas jusqu'au niveau de la méthode. Mais comme pour QML, il est possible d'attacher plusieurs contrats à une même entité d'implémentation permettant ainsi la séparation des préoccupations. CQML introduit les négociations de QoS en phase de configuration ainsi que la gestion dynamique de celle-ci. Pour la négociation, les profils sont comparables à l'aide d'une relation de conformité, semblable à celle de QML, et sont ordonnés. Enfin, en phase d'exécution, la transition d'un profil à un autre peut être associée à

des réactions du système (à travers des callbacks). Du point de vue de la composition, dans (Aagedal, 2001) on explique que les caractéristiques de QoS d'un assemblage peuvent se déduire de celles de ses composants mais placent ces travaux en dehors du champ d'étude de CQML.

3.2. Modèles de QoS

3.2.1 ISO/IEC 9126

Le modèle de qualité défini dans la norme ISO/IEC 9126-1 (ISO, 2001), mis à jour dans la norme ISO/IEC 25010 (ISO, 2010), est un framework qui doit être adapté au domaine d'application cible. Cependant, rares sont les travaux (Alvaro et al., 2006) qui fournissent cette adaptation ce qui rend son utilisation délicate.

Tableau 1.1: ISO 9126 : caractéristiques et sous-caractéristiques de qualité (ISO, 2001)

Characteristics	Subcharacteristics
Functionality	*Suitability*
	Accuracy
	Interoperability
	Compliance
	Security
Reliability	*Maturity*
	Recoverability
	Fault Tolerance
Usability	*Learnability*
	Understandability
	Operability
Efficiency	*Time behavior*
	Resource behavior
Maintainability	*Stability*
	Analyzability
	Testability
	Changeability
Portability	*Installability*
	Conformance
	Adaptability
	Replaceability

Le tableau 1.1 montre les caractéristiques et les sous-caractéristiques de qualité du standard ISO 9126, qui sont définies comme suit :

- Capacité fonctionnelle (ou Functionality) : C'est la capacité d'un logiciel à répondre aux besoins fonctionnels. Cela inclut la capacité à se plier aux exigences de

l'utilisateur (Compliance), la couverture du besoin (Suitability), le degré de précision de sa satisfaction (Accuracy), la capacité à répondre aux besoins de sécurité (Security) et la capacité à répondre aux besoins d'interopérabilité (Interoperability) ;

- Fiabilité (Reliability) : C'est la capacité d'un logiciel à fournir ses services dans des conditions précises et pendant une période déterminée. Cela inclut sa maturité (Maturity), sa capacité à se rétablir d'éventuelles pannes (Recoverability), et sa tolérance aux erreurs (Fault tolerance) ;

- Utilisabilité (Usability) : C'est la capacité d'un logiciel à être appris rapidement (Learnability), facile à comprendre (Understandability) et opérable (Operability) ;

- Performance (Efficiency) : C'est la capacité d'un logiciel à fournir ses services de manière efficace en termes de temps d'exécution (Time behavior) et de consommation des ressources système (Resource behavior) ;

- Maintenabilité (*Maintainability*) : C'est la capacité d'un logiciel à être maintenu à jour. Cela inclut la stabilité (*Stability*), ainsi que la facilité avec laquelle il peut être analysé (*Analyzability*), testé (*Testability*) et changé (*Changeability*) ;

- Portabilité (*Portability*) : C'est la capacité d'un logiciel à être transféré d'un environnement à un autre. Cela inclut la facilité avec laquelle il peut être installé (*Installability*), intégré en harmonie avec le reste de l'application (*Conformance*), adapté (*Adaptability*) et remplacé (*Replaceability*).

Ce standard a servi en tant que référence pour certains modèles de qualité de service spécifiés dans le cadre de l'approche orientée composant, comme par exemple CQM. Dans ce qui suit, nous montrons comment le modèle QCM a effectué l'adaptation d'ISO 9126 afin de définir la qualité liée aux composants logiciels.

3.2.2 CQM (Component Quality Model)

Ce modèle (Alvaro et al., 2006) constitue une adaptation d'ISO 9126 aux composants. La première différence notable est que CQM supprime des sous-caractéristiques initiales d'ISO 9126. Les sous-caractéristiques qui disparaissent sont Analyzability et Conformance. En effet, selon (Alvaro et al., 2006), les composants disposent de méthodes d'auto-analyse au déploiement. Une autre sous-caractéristique, Installability, change de nom et devient

Deployability. En effet, une fois que les composants sont acquis, ils sont déployés et non installés. En outre, CQM ajoute d'autres sous-caractéristiques :

- Self-Contained : C'est la capacité d'un composant à exécuter lui-même les services qu'il offre, sans l'aide d'autres composants ;
- Configurability : C'est la capacité d'un composant à être configuré. Cela inclut le mode de configuration (fichier XML ou texte), le nombre de paramètres, etc...
- Scalability : C'est la capacité d'un composant à s'accommoder d'un nombre important de données sans changer son implémentation ;
- Reusability : C'est la capacité d'un composant à être réutilisé ;
- Marketability : C'est la capacité d'un composant à être commercialisé. Les sous-caractéristiques qui la composent sont le temps de développement du composant (Development time), son coût (Cost), le temps qu'il faut pour le mettre sur le marché (Time to market), le type du marché visé (Targeted market) et la capacité du composant à être abordable pour ce type de marchés (Affordability).

CQM décompose ses sous-caractéristiques en attributs de qualité, auxquels sont associés des métriques. Trois types de métriques sont disponibles :

- *Presence* mesure la présence d'un attribut qualité par le biais d'un booléen ;
- *IValues* mesure la valeur d'un attribut par le biais d'un entier. Notons que l'unité de mesure est représentée par une chaîne de caractères ;
- *TypeRatio* est utilisé pour les pourcentages.

Chaque sous-caractéristique de qualité peut être décomposée en un ou plusieurs attributs, et à chaque attribut est associée une métrique. Par exemple, la sous-caractéristique Time behavior, mesurable à l'exécution, est décomposée en trois attributs, tous mesurés par une métrique IValues. L'avantage de CQM est qu'il est complet à tous les niveaux : caractéristiques, sous-caractéristiques, attributs et métriques (voir l'annexe A). Il est également spécifique aux composants et les métriques sont simples d'emploi. Cependant, dans (Alvaro et al., 2006), on ne propose aucun mode de comparaison entre deux composants utilisant ce modèle de qualité.

3.3. Ontologies de QoS

3.3.1 QoSOnt ontology

Dans (Dobson et al., 2005), on a développé QoSOnt, une ontologie basée sur les taxonomies et les modèles de QoS existants. L'ontologie QoSOnt comprend plusieurs ontologies qui sont organisées en trois couches: la couche de base (voir figure 1.4), la couche des attributs et la couche spécifique au domaine.

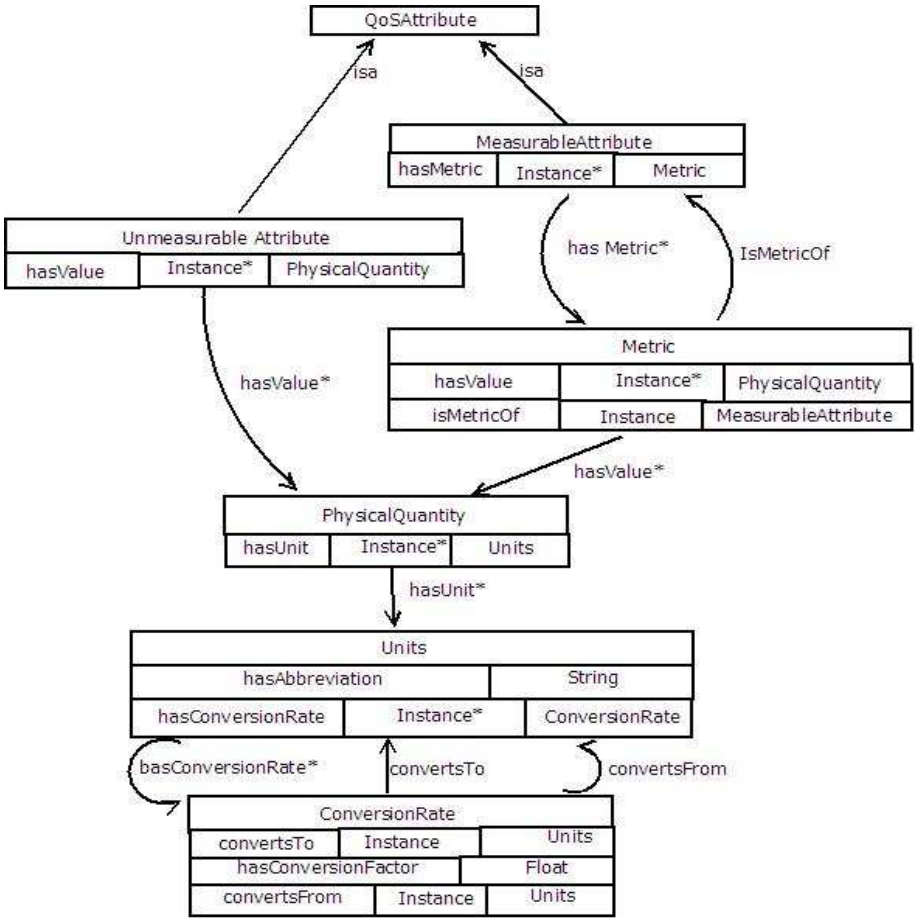


Figure 1.4 : L'ontologie de QoS de base (Dobson et al., 2005)

La couche de base comprend une ontologie de base contenant des concepts de QoS génériques et des ontologies unitaires, comme par exemple l'ontologie du temps. Elle représente un ensemble minimal des concepts génériques. Le concept central est *QoSAttribute* qui est spécialisé en *UnmeasurableAttribute* et *MeasurableAttribute*. Un attribut mesurable peut avoir une ou plusieurs métriques associées. Le concept *PhysicalQuantity* représente la valeur d'un attribut de QoS. Dans le cas d'un attribut mesurable, PhysicalQuantity peut avoir

une ou plusieurs unités. Une unité peut être convertie vers une autre grâce à la notion *ConversionRate* qui précise les règles de conversion entre deux unités.

La couche attribut contient des ontologies qui définissent certains attributs de QoS et leurs métriques. Par exemple, dans (Laprie et al., 2004), deux ontologies (ontologie de fiabilité et ontologie de performance) sont spécifiées en définissant un ensemble d'attributs de QoS communs.

La couche spécifique au domaine relie les couches inférieures à des types spécifiques de systèmes, comme par exemple les réseaux ou les Web services. Pour les Web services, cette couche fournit des concepts pour le raccordement des concepts de QoS des couches inférieures avec des profils de services OWL-S.

3.3.2 onQoS ontology

L'ontologie onQoS (Giallonardo et Zimeo, 2007) est développée pour spécifier les exigences de QoS par les utilisateurs et la publication de QoS par les fournisseurs de services. Cette ontologie est composée de trois couches extensibles et complémentaires : «upper », « middle » et « low ».

L'ontologie « upper » fournit les concepts suivants: *QoSParameter* pour caractériser des propriétés de QoS, *QoSMetric* pour mesurer les instances de *QoSParameter*; et *MeasurementProcess,* le processus par le biais duquel des chiffres ou des symboles sont affectés aux paramètres de QoS selon des règles bien définies. *Scale* définit les catégories ou les ensembles des données dans lesquels un paramètre de QoS peut être placé par rapport à une propriété de QoS particulière. *ScaleValue* est la valeur scalaire (un nombre ou un symbole) d'un paramètre de QoS. *QoSMetric* est également défini comme un paramètre de service qui peut être inclut dans un profil de service afin de connecter une spécification de la QoS à une description du service.

L'ontologie « middle » s'étend et affine les concepts de l'ontologie « upper » afin de spécifier les paramètres de QoS indépendants du domaine. En premier lieu, les paramètres de QoS définis dans cette ontologie sont similaires à ceux spécifiés dans (Papaioannou et al., 2006). En second lieu, QoSMetrics sont classés en fonction de l'échelle (scale) utilisée (nominale, ordinale, ratio ...) du nombre de paramètres mesurés, ou de l'information exploitée dans le processus de mesure. Ces catégories sont *ElementaryMetric*, *DerivedMetric*, *InternalMetric*,

ExternalMetric, *NominalMetric*, *OrdinalMetric*, *NumericMetric*, *RatioMetric*, et *IntervalMetric*. En troisième lieu, *QoSMetricFunctions* sont des fonctions utilisées pour agréger des métriques de la QoS élémentaires, dérivées, internes ou externes afin de définir une métrique dérivée. *QoSMetricFunctions* sont divisées en *NominalMetricFunction*, *NumericMetricFunction*, et *OrdinalMetricFunction* ; et elles peuvent être *BinaryFunctions* (avec deux arguments) ou *UnaryFunctions* (avec un argument). L'ontologie « low », quant à elle, définit les concepts, les propriétés et les contraintes pour des informations de la QoS propres à un domaine. Par exemple, *QoSNetJitter*, *QoSNetLatency*, *QoSNetRTT* sont des concepts spécifiques pour la QoS dans le domaine réseau. Cette approche donne une spécification bien définie pour les métriques, les types de valeur, la transformation de métriques ainsi que la description d'un ensemble de propriétés de QoS communes. Cependant, on ne mentionne rien sur les caractéristiques de nombreuses propriétés de QoS telles que : unité, direction de l'impact, dynamisme, relations de QoS, etc. Le support d'utilisation de la QoS est également très limité (Tran and Tsuji, 2009).

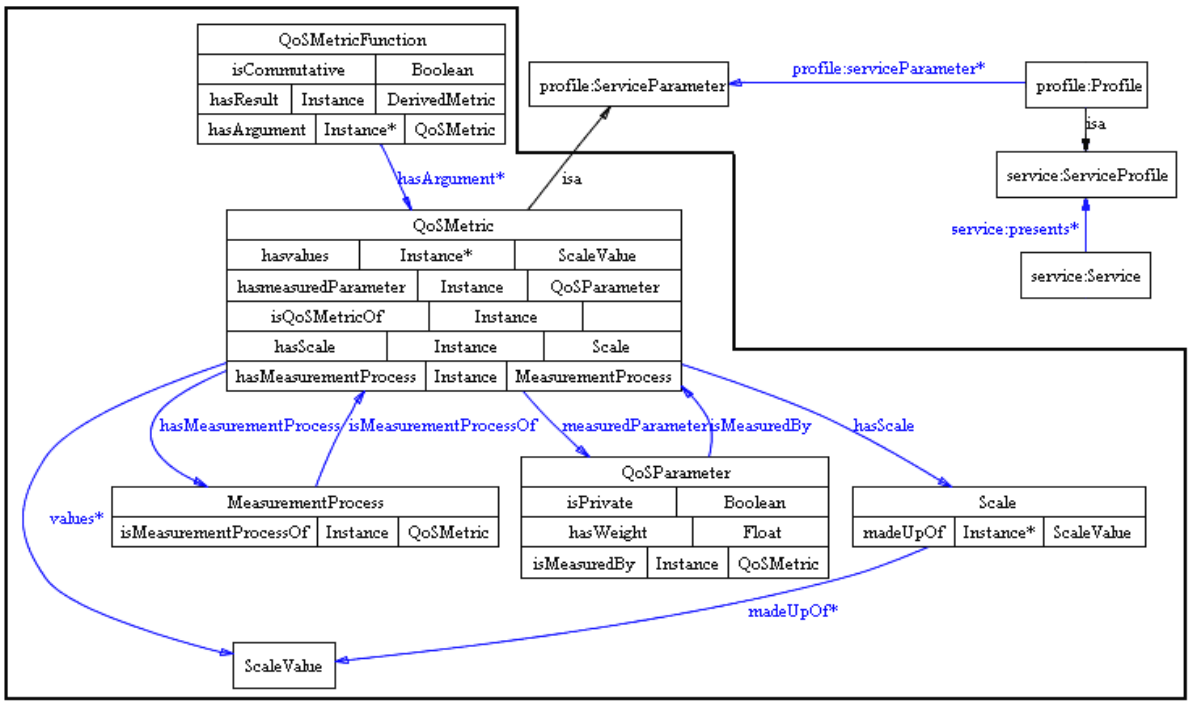


Figure 1.5 : L'ontologie OnQoS (Giallonardo et Zimeo, 2007)

4. Génie logiciel orienté composant (CBSE)

4.1. Objectifs

L'objectif principal de l'approche orientée composant est de construire les applications par assemblage de composants logiciels pré-existants. De ce fait, le CBSE distingue deux cycles de vie : un pour le développement du composant (Design for reuse) et un autre pour le développement d'un système à base de composants (Design by reuse). Ces deux démarches sont intrinsèquement différentes :

- Le cycle de vie d'un système à base de composants se focalise sur l'identification des composants réutilisables et des relations qui les relient. Un nouveau processus s'intègre dans cette démarche afin de répondre aux besoins d'identification, de sélection et d'adaptation des composants déjà développés. Afin de réduire l'effort d'adaptation, la sélection de composants doit être pertinente, c'est-à-dire les composants sélectionnés doivent être très proches des composants qu'on recherche.
- Le cycle de vie d'un composant se focalise sur l'enjeu majeure de la réutilisabilité : le composant est créé afin d'être réutilisé dans différentes applications existantes ou futures ; il doit être spécifié précisément et formellement, facile à comprendre, assez générique, facile à adapter, à délivrer, à déployer et à remplacer.

4.1. Cycles de vie

Un cycle de vie du CBSE utilise les méthodes, outils et principes de l'ingénierie logicielle classique. Il s'étend de la phase d'analyse et de spécification des besoins à la phase d'opérationnalisation et de maintenance d'un système logiciel.

4.1.1 Design by reuse

Ce cycle de vie comprend un nouveau processus qui vient s'ajouter à la fois à la phase de conception du système et à la phase de maintenance. Nous montrons dans la figure 1.6 comment le cycle en V est adapté afin de répondre aux besoins du CBSE.

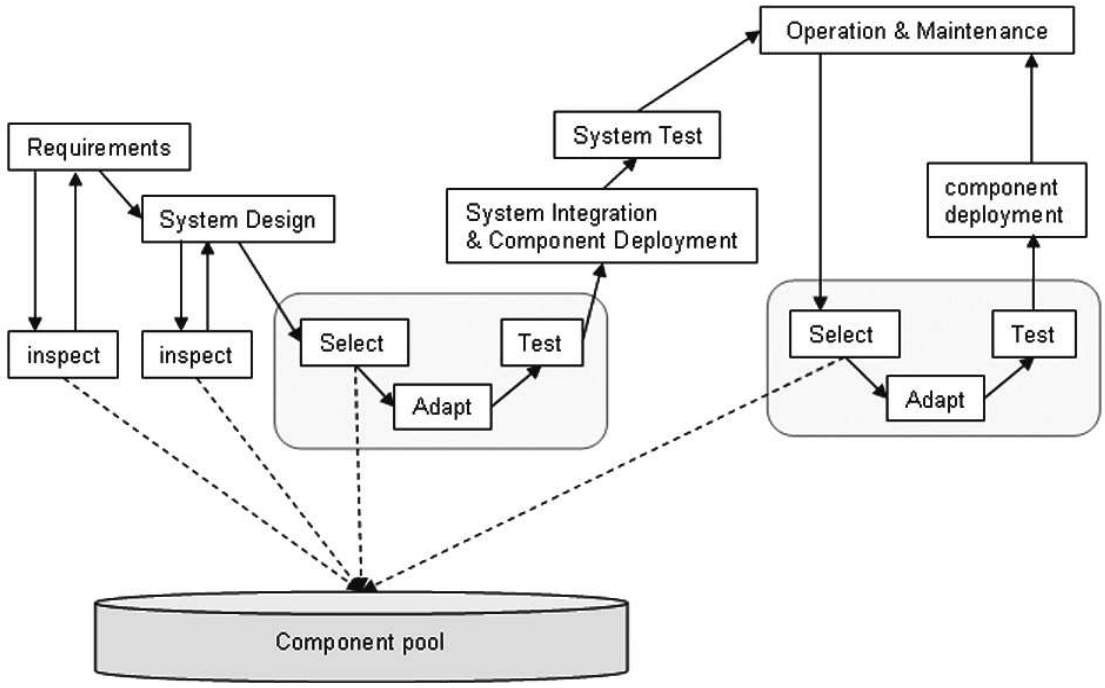


Figure 1.6 : Le cycle V en CBSE (Crnkovic et al., 2006)

Le but de cette section n'est pas de détailler toutes les phases du cycle de vie mais de mettre l'accent sur les activités spécifiques au cycle « Design by reuse » du CBSE. Ces activités sont décrites comme suit :

- ✓ *Inspection de composants.* Le processus d'analyse et de conception doit prendre en compte les besoins du système qui doivent être compatibles avec les besoins des composants disponibles. Par exemple, la sélection d'une technologie particulière doit être prise en considération : une technologie à composant peut requérir une implémentation particulière et inclure des services spécifiques tels que la communication inter-composants.
- ✓ *Sélection de composants*. Pour réussir cette étape, il faut un nombre significatif de candidats dans les référentiels ainsi que des outils pour les identifier et les évaluer. La sélection est effectuée en se basant soit uniquement sur les caractéristiques fonctionnelles, soit sur les caractéristiques fonctionnelles et non fonctionnelles.
- ✓ *Adaptation de composants*. Certains composants peuvent être intégrés directement dans le système, d'autres ont besoin d'adaptation, soit par un processus de paramétrage ou par ajout du code qui permet une meilleure composition. Dans certains cas, il n'est pas

possible de réutiliser le composant lui-même, mais seulement son interface qui doit être implémentée à nouveau.

- ✓ *Test de composants.* Il s'agit d'associer les besoins du composant avec les besoins du système et vérifier les propriétés du système à partir des caractéristiques fonctionnelles et/ou non fonctionnelles des composants.

4.1.2 Design for reuse

Ce cycle est similaire au cycle « Design by reuse » : les besoins doivent être analysés et définis ; et le composant doit être conçu, implémenté, vérifié et livré. Lors de la construction d'un nouveau composant, les développeurs peuvent réutiliser d'autres composants et exploiter les procédures d'évaluation semblables à celles du développement par réutilisation. Cependant, un composant construit pour être réutilisé doit être générique et flexible. Ainsi, la tâche du développement se complexifie et doit répondre aux besoins suivants :

- Plus d'effort dans la conception et l'implémentation du composant ;
- Plus d'effort dans le test du composant (le test s'effectue en isolation ou dans différentes configurations) ;
- Plus de qualification de la part des développeurs ;
- Plus de documentation pour la spécification du composant.

Une fois le composant testé, spécifié, stocké dans un référentiel de composants, la prochaine étape du cycle de vie du composant est la phase de déploiement dans le système. Ce déploiement doit s'effectuer de manière automatique et sans effort d'adaptation. Notons qu'une technologie à composant fournit un ensemble de services techniques qui permettent de connecter un composant dans un système.

5. Synthèse

Le principe de réutilisabilité, qui est le point central de notre travail, représente un intérêt majeur pour les approches objet, composant et service. Cependant, l'approche orientée objet a atteint ses limites dans ce domaine et a laissé la place à l'approche composant où la réutilisabilité est au cœur de plusieurs travaux de recherche. L'approche orientée service tente de suivre le même chemin en mettant en œuvre un couplage faible entre les services pour favoriser leur réutilisabilité. Les approches composant et service ne sont pas antagonistes car elles opèrent pour un objectif commun. Les différences entre un composant et un service sont

principalement d'ordre technique. Un service peut être considéré comme une façade qui se positionne devant un composant afin de le standardiser (Barais et al., 2003). La principale lacune d'un service par rapport au composant réside dans le manque d'expression de ses dépendances avec d'autres services. Celles-ci sont exprimées par des interfaces requises dans l'approche orientée composant.

L'hétérogénéité des modèles de composants réduit leur interopérabilité. Une panoplie de modèles a été proposée et classifiée selon différents critères. Dans notre travail, nous avons présenté la classification selon la dimension de configuration. Aussi, nous aurions pu adopter le choix de (Lau and Wang, 2007) où on présente une classification des modèles de composants basée sur la possibilité d'existence d'un référentiel de composants lors du développement. On a conclut que rares sont les modèles hiérarchiques qui s'appuient sur des référentiels existants alors que la plupart des modèles plats exploitent leurs propres référentiels. La sélection de composants à partir de ces référentiels est une tâche difficile à mettre en œuvre et on prend très peu en compte les caractéristiques de QoS comme paramètres de recherche discriminants guidant le développeur dans le choix d'un composant. Dans (Frolund and Koistinen, 1998), on montre qu'il est essentiel de prendre en compte cette QoS au design-time. Ainsi, il ne s'agit plus de satisfaire uniquement les caractéristiques fonctionnelles d'un composant mais aussi ses caractéristiques de QoS. Les trois premiers niveaux de contrat de (Beugnard et al., 1999) définissent l'interface d'un composant d'un point de vue fonctionnel. Cependant, d'autres propriétés plus quantitatives (niveau 4) peuvent permettre de caractériser un composant et son fonctionnement au sein d'un assemblage. Par exemple, si on prend un composant qui distribue du contenu multimédia alors la latence, la qualité du résultat ou la synchronisation entre l'image et le son sont des caractéristiques de QoS qui affectent le degré de satisfaction de l'utilisateur de ce service.

Plusieurs formalismes sont proposés afin de décrire la QoS des applications. D'un côté, dans le domaine du CBSE, nous trouvons différents langages et modèles à partir desquels nous nous sommes inspirés pour construire notre propre modèle. Malheureusement, ces outils restent restreints du fait qu'ils sont liés à des contextes particuliers. De l'autre côté, dans le domaine SOA (Service Oriented Architectures), l'approche adoptée se base sur le modèle ontologique du Web sémantique. L'ontologie présente plusieurs avantages car elle fournit un langage commun, facilite l'interopérabilité, et permet le raisonnement par machine.

Pour ces raisons, nous avons choisi de décrire la QoS par le biais d'une ontologie. Cela permet de représenter n'importe quel composant en faisant abstraction de ses choix techniques (modèle, langage de spécification de la QoS,…). En résumé, nous traitons dans le cadre de cette thèse la sélection de composants primitifs, indépendamment de leurs modèles, destinés aux architectures les intégrant. Nous prenons en compte des critères de QoS qui permettent de retourner au développeur un meilleur résultat, c'est-à-dire le meilleur composant réutilisable satisfaisant la requête du développeur.

5. Conclusion

Dans ce chapitre, nous avons abordé quelques points de l'approche composant qui nous ont permis de capturer les éléments nécessaires pour l'élaboration du framework de notre travail de thèse. D'abord, l'entité réutilisable qui nous intéresse est le composant qui expose ses interfaces fournies et requises. Ensuite, nous devons décrire les caractéristiques non fonctionnelles des composants en se focalisant sur la spécification des caractéristiques de qualité de service. Cette spécification doit être indépendante des modèles de composants mais destinée aux architectures à base de composants. Enfin, nous choisissons de résoudre le problème de sélection de composants logiciels qui est la clé de réussite des cycles « Design for reuse » et « Design by reuse » du CBSE. Cette problématique sera abordée en détail lors du prochain chapitre.

Chapitre 2

Sélection de composants

2. Sélection de composants 41

Le génie logiciel orienté composant met l'accent sur la réutilisation de composants logiciels appelés aussi « composants sur étagère » (ou COTS, pour Components Off The Shelf en englais). Dans le chapitre précédent, nous avons présenté le processus qui permet de différencier un cycle de vie CBSE d'un cycle de vie classique (par exemple le cycle de vie en V). Ce processus est celui de la sélection de composants logiciels. Il existe une panoplie de méthodes de sélection[4] qui modélisent le processus de sélection des COTS afin de pallier les difficultés liés à sa complexité. Ces méthodes sont d'une efficacité variable et pourraient être adaptées à différents contextes.

Dans le reste de ce chapitre, nous définissons d'abord et de manière générale le problème de sélection. Ensuite, nous présentons plusieurs techniques de recherche et de sélection de composants logiciels pouvant d'être réutilisés dans un processus de sélection. Enfin, nous analysons quelques méthodes de sélection existantes pour extraire les avantages et les

[4] Dans le reste de ce chapitre, nous utilisons le terme « sélection » pour désigner les termes « sélection de composants sur étagère »

inconvénients de chacune ; et choisir, in fine, la méthode la mieux adaptée à notre contexte de travail.

1. Définition du problème

Les composants sur étagère sont des composants logiciels développés par des tiers et stockés dans des référentiels (repositories) pour être utilisés ultérieurement dans la construction d'applications. A cet effet, il existe un besoin grandissant pour la sélection de composants pertinents répondant aux exigences des développeurs. Cependant, le succès de la sélection est problématique pour un certain nombre de raisons:

- Absence de processus bien définis : la plupart des organisations appliquent le processus de sélection de manière ad-hoc, ce qui rend la planification difficile, les méthodes d'évaluation et les outils appropriés inutilisés. De plus, les leçons tirées des expériences précédentes ne sont pas capitalisées;

- Critères d'évaluation : Parfois, on définit des attributs inappropriés et on néglige des attributs discriminants ;

- Nature boite noire des composants : Le fait de ne pas pouvoir accéder au code source rend restrictive la compréhension des composants et par conséquent rend l'évaluation difficile;

- Mises à jour continues des composants : La fréquence élevée des modifications dans les référentiels de composants rend l'évaluation difficile. Par exemple, une nouvelle version du composant peut avoir une fonctionnalité qui n'est pas disponible dans le composant en cours d'évaluation.

Il existe plusieurs méthodes de sélection (Mohamed et al., 2007 ; Wanyama, 2008) qui tentent de réduire les risques liés aux problèmes précédents. Chaque méthode utilise un ensemble de techniques pour la **recherche** et la **sélection** des composants. Le choix de ces techniques influence la performance de la sélection : Souvent, le composant sélectionné doit être adapté afin de répondre exactement aux exigences et aux contraintes du développeur.

2. Techniques de recherche de composants

La recherche de composants logiciels est un problème non trivial. Après avoir déterminé les critères d'évaluation, il faut être capable de retrouver le composant recherché, s'il existe, parmi un ensemble grand de composants logiciels. Dans le cas échéant, il faut trouver le composant le plus proche de celui recherché. Dans la littérature, il existe plusieurs techniques de recherche d'information dont le but est d'avoir un résultat performant.

2.1. Techniques classiques de recherche d'information

Un composant peut être considéré comme un type particulier de documents. Ainsi, plusieurs recherches ont tenté d'appliquer les techniques classiques de recherche d'information pour rechercher des composants.

Dans (Frakes and Nejmeh, 1987), on applique la technique de recherche d'information textuelle sur un référentiel d'artefacts logiciels (des codes sources en langage C). Les artefacts sont indexés par des termes extraits des entêtes et des commentaires se trouvant dans les codes sources. L'opération d'indexation est entièrement automatisée pour garantir une uniformité des termes d'indexation. Cette approche d'indexation dépend fortement des habitudes individuelles des programmeurs pour commenter le code source. De plus, la méthode n'utilise pas un vocabulaire contrôlé, ce qui demande un effort supplémentaire des programmeurs pour trouver les bons termes et des utilisateurs du système pour trouver ces mêmes termes.

Les techniques textuelles de recherche d'information sont appliquées sur les codes sources dans le domaine de la réingénierie des systèmes d'information. Elles aident l'ingénieur à comprendre la structure du code source et lui permettent de retrouver des constructions spécifiques (des patterns). Le système ESCAPE (Paul and Prakash, 1994) analyse les codes sources pour extraire des représentations sous forme de graphes syntaxiques abstraits. Un graphe syntaxique abstrait est une représentation du code source où les nœuds représentent du code source (while-statement, relational-expression, statement-list, etc.). L'ingénieur utilise une interface graphique pour exprimer sa requête qui sera traduite en une expression algébrique conformément au modèle algébrique du code source, défini par les auteurs. Enfin, le système optimise la requête et la compare aux graphes syntaxiques des composants.

Dans (Mishne, 2004), on utilise une approche similaire à celle utilisée dans le système ESCAPE. Cette approche utilise le formalisme des graphes conceptuels pour décrire à la fois les codes sources et les requêtes. L'utilisateur soumet sa requête sous la forme d'un code

source. Le système extrait une représentation sous forme d'un graphe conceptuel, puis la compare aux graphes conceptuels qui indexent les composants (codes sources) du référentiel.

2.2. Techniques de classification externe

Le principe de ces techniques est d'indexer les composants à partir de leurs représentations.

2.2.1 Approche par mots-clés

Dans (Matsumoto, 1987), on décrit les composants par un ensemble de mots-clés. L'indexation des composants dans le référentiel est une simple classification par mots-clés.

Dans (Karlsson, 1995), on étend la technique de classification par mots-clés en associant un poids à chaque mot-clé. On utilise donc des relations floues pour associer les termes d'indexation aux composants.

En résumé de tous ces travaux, la recherche de composants par mots-clés a montré rapidement ses limites

2.2.2 Approche par langage naturel

Dans (Maarek et al., 1991), le système de recherche de composants utilise des techniques textuelles de recherche d'information sur des descriptions de composants en langage naturel. L'interrogation du référentiel de composants se fait à travers des requêtes en langage naturel. Chaque description textuelle de composant est analysée dans le but d'extraire des termes d'indexation. Ces termes représentent le descripteur qui est utilisé pour la correspondance avec les requêtes utilisateurs. L'implantation de ce système s'appelle GURU.

L'évaluation des performances du système GURU a été réalisée avec une collection de documents (composants) et de requêtes de test en évaluant l'effort de l'utilisateur, l'effort de maintenance, la précision et le rappel.

Dans (Helm and Maarek, 1991), on fait évoluer l'approche utilisée dans GURU pour indexer des bibliothèques de classes orientées objet en exploitant les informations supplémentaires fournies par la relation d'héritage. Cette technique peut être classée dans la catégorie des techniques de classification structurelle, mais nous la présentons dans cette section car elle s'appuie sur l'extraction de la structure des composants à travers l'analyse des descriptions textuelles des composants et non pas des composants eux-mêmes.

Dans (Devanbu et Brachman, 1991), on propose un système de recherche de composants appelé Lassie (Large Software System Information Environment). Lassie est composé d'une base de connaissances, d'un algorithme de recherche sémantique basé sur des inférences formelles et d'une interface utilisateur avancée. La base de connaissances de Lassie est construite en utilisant un langage de représentation de connaissances basé sur la classification. Les représentations sont construites grâce à l'analyse des descriptions en langage naturel des composants. La base de connaissances est utilisée comme un index pour la recherche de composants réutilisables. Elle est structurée sous la forme d'une taxinomie de nœuds de quatre types (action, objet, acteur et état) reliés par le lien IsA et d'autres liens exprimant des relations de niveau implantation telles que function-callsfunction, sourcefile-includes-headerfile, etc. Une requête Lassie est la description d'une action avec un certain nombre de paramètres. Lassie évalue la requête en recherchant dans la base de connaissances une instance vérifiant ses paramètres. Cette technique peut être classée dans la catégorie des techniques de recherche sémantique (voir la section 2.5) mais nous la décrivons dans cette section car elle s'appuie sur une représentation des connaissances extraites à partir de descriptions textuelles de composants.

2.2.3 Approche par facettes

Afin d'améliorer la recherche par mots-clés, dans (Prieto-Diaz, 1991), on a présenté une approche à base de facettes pour classer les composants logiciels. Cette approche a été reprise dans d'autres travaux (Isakowitz and Kauffman, 1996 ; Zhang et al., 2000). Une facette représente une information particulière qui permet d'identifier et de caractériser un composant. Elle est définie par son nom et son vocabulaire, c'est-à-dire l'ensemble des mots-

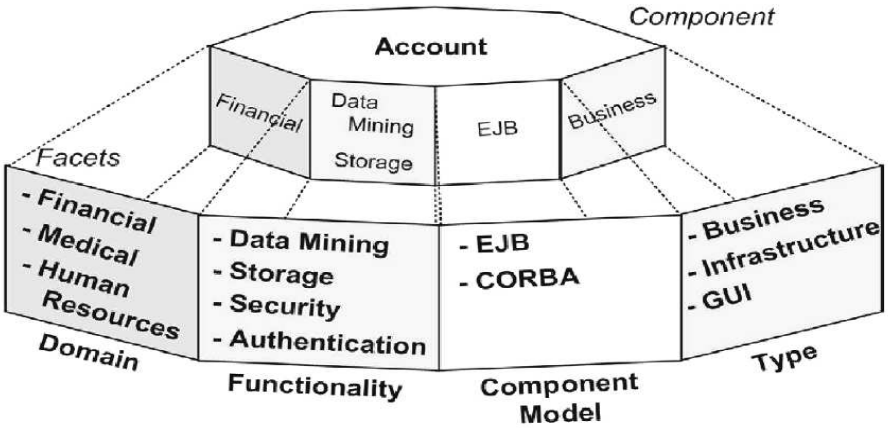


Figure 2.1 : Classification par facettes (Almeida et al., 2007)

clés qui permettent de la décrire. Pour spécifier un composant, un ou plusieurs mots-clés doivent être choisis dans le vocabulaire de chaque facette.

La figure 2.1 montre un exemple de description de composant au moyen d'un ensemble de quatre facettes : Domain, Functionality, Component Model et Component Type. Les mots-clés EJB et CORBA du vocabulaire de la facette Component Model correspondent aux deux modèles de composants disponibles. De même, le domaine d'un composant peut être financier (Financial), médical, ou concerne les ressources humaines (Human Resources). La fonction (Functionality) d'un composant peut être : Data Mining, Storage, Security ou Authentification.

Enfin, le Type d'un composant peut être : Business, Infrastructure ou GUI. Pour spécifier le composant Account représentant un compte bancaire (voir la figure 2.1), Financial est le mot-clé choisi pour caractériser sa facette Domain, Data Mining et Storage ont été choisis pour la facette Functionality, EJB pour la facette Component Model et Business pour la facette Type.

Notons que le succès de la recherche par facettes dépend du choix judicieux des différentes facettes.

2.3. Techniques de recherche structurelle

Alors que les techniques précédentes se basent sur les descriptions de composants, les techniques de recherche structurelle concernent l'aspect structurel des composants eux-mêmes. Cette catégorie se divise en deux sous-catégories : les techniques d'appariement (ou matching en anglais) de signatures et les techniques d'appariement de spécifications. Les approches d'appariement de signatures sont en réalité un sous-ensemble des approches d'appariement de spécifications. En plus de l'aspect signature, une approche par spécification s'intéresse à la transformation effectuée sur les données et à la conception interne du composant. Nous faisons cette distinction entre les deux catégories car les approches de classification par signatures sont exploitables avec les composants de type boîte noire, boîte en verre et boîte blanche. Les approches de classification par spécification ne sont par contre applicables qu'avec les composants de type boîte blanche et boîte en verre.

2.3.1 Appariement de signatures

Un composant logiciel offre généralement ses services à travers des opérations regroupées dans des interfaces. La signature d'un composant est l'union des signatures de toutes ses

interfaces. De même, la signature d'une interface est l'union des signatures des opérations qu'elle déclare.

Dans (Ritti, 1989), on propose une technique d'appariement de signatures. Cette technique est appliquée sur une collection de composants écrits dans un langage fonctionnel. L'algorithme d'appariement (mesure de la distance) adopté utilise un système de type polymorphique et garantit une invariance vis-à-vis de l'ordre des déclarations dans un type.

Dans (Gaudel and Moineau, 1991), on introduit la théorie de la réutilisabilité logicielle sur des spécifications algébriques (Wirsing, 1991) de composants logiciels. Les composants sont spécifiés avec le langage PLUSS4 (Bidoit, 1989). Une spécification décrit la signature d'un composant et un ensemble d'axiomes décrivant les interactions entre les méthodes. Dans cette approche, on a défini la relation d'ordre «réutilisabilité » entre les spécifications. Le lien de réutilisabilité entre deux spécifications indique que moyennant un enrichissement, la spécification source peut devenir équivalente à la spécification cible. En variant la définition du critère d'équivalence et la façon dont une spécification peut être enrichie. On a aussi définit un large éventail de critères d'appariement. Ces critères tiennent compte du renommage de méthodes et de la relaxation du critère d'équivalence entre spécifications. Les travaux de (Moineau and Gaudel, 1991) sont validés avec le référentiel de composants ReuSig qui utilise uniquement une technique d'appariement de signatures. On a également proposé une version spécialisée du référentiel de composants ReuSig baptisée ROSE-Ada (Badaro and Moineau, 1991) qui gère des collections de composants Ada.

Dans (Zaremski and Wing, 1993 ; Zaremski and Wing, 1995(a)), on propose une technique de classification et de recherche de composants basée sur les signatures. Le processus de recherche consiste à parcourir le référentiel de composants pour rechercher les composants dont les signatures sont compatibles avec celles de la requête. Deux signatures sont considérées compatibles si elles sont identiques modulo un renommage et une permutation des noms et des paramètres de méthodes. Une relaxation des critères d'appariement (équivalence) permet d'améliorer le rappel de cette approche. Les auteurs proposent deux types de relaxations : par changement du critère d'appariement ou par transformation des signatures de la requête et des composants. Le critère d'appariement par équivalence peut être remplacé par d'autres critères comme l'appariement par généralisation et l'appariement par spécialisation. Le critère d'appariement par généralisation (respectivement par spécialisation)

sélectionne les composants dont les signatures sont identiques ou généralisent (respectivement spécialisent) la signature de la requête. Les processus de recherche des composants peuvent transformer la spécification de la requête et celles des composants pour arriver à les apparier. On propose deux types de transformation : enrichissement et appauvrissement. L'enrichissement (respectivement appauvrissement) consiste en l'ajout (respectivement la suppression) de méthodes ou de paramètres. Grâce à ces transformations, le système peut retrouver des composants qui n'ont pas exactement le même nombre de paramètres ou de méthodes que la requête mais qui ont un certain degré de ressemblance. Ces deux types de relaxation peuvent être combinés pour produire d'autres critères d'appariement.

2.3.2 Appariement de spécifications

Les techniques de recherche de composants par appariement de spécifications tentent de retrouver les composants dont les spécifications correspondent à la spécification de la requête. Ces techniques peuvent être classées selon les types de composants auxquels elles sont applicables : les composants logiciels et les composants conceptuels. Nous nous intéressons dans ce qui suit au premier type de composants.

Les techniques de recherche de composants par appariement de spécifications représentent généralement les composants logiciels selon le modèle de la figure 2.2.

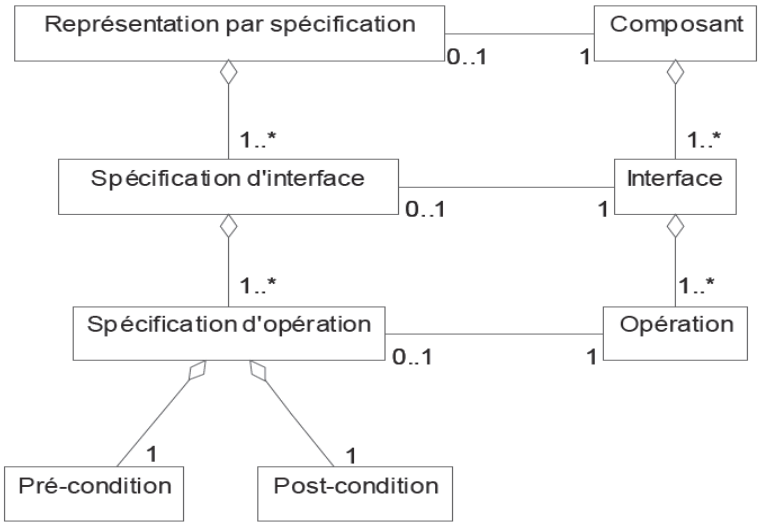


Figure 2.2 : Modèle de représentation par spécifications (Zaremski and Wing, 1995(a))

Dans (Zaremski and Wing, 1995(a)), on a étendu des travaux antérieurs sur les signatures pour proposer une approche d'appariement de spécifications. Les requêtes et les composants sont représentés par un ensemble de paires de pré et post conditions (une paire par méthode). On définit aussi un critère général d'appariement grâce à la formule :

$$Match(S,Q) = (Q_{Pré} R_1 S_{Pré}) R_2 (S_{Post} R_3 Q_{Post})$$

où $S = (S_{Pré}, S_{Post})$ est la spécification d'un composant,

$Q = (Q_{Pré}, Q_{Post})$ est la spécification d'une requête,

R_1, R_2, R_3 sont trois relations logiquement connectives.

En variant les relations R1, R2 et R3, on obtient une hiérarchie de critères d'appariement.

Dans (Rollins and Wing, 1991), on présente un référentiel de composants utilisant une approche de classification par spécifications. Les composants sont spécifiés dans le langage λ-Prolog. Cette technique exploite les mécanismes d'inférence du langage λ-Prolog pour l'appariement de spécifications. Les composants dont les spécifications sont pertinentes par rapport à la spécification de la requête sont donc sélectionnés.

Le système CAPES (Steigerwald, 1992) est un environnement de prototypage de systèmes temps réel embarqués. L'environnement CAPES comprend : un système de support d'exécution, un éditeur dirigé par la syntaxe, un référentiel de composants et un environnement de conception. Les requêtes utilisateur et les composants du référentiel sont spécifiés en utilisant un langage de spécification augmenté avec le langage OBJ[5]. Le processus de recherche de composants utilise la technique d'interrogation par consistance. En exploitant les capacités d'inférence offertes par OBJ, le système sélectionne uniquement les composants consistants par rapport à la spécification de la requête.

Enfin, dans (Hemer and Lindsay, 2001), on propose un référentiel de modules. Un module est un ensemble d'unités. Une unité peut être une procédure, une classe ou une structure de données. Si un développeur a besoin d'un module qui implante des opérations sur les listes chaînées alors il doit spécifier toutes les primitives dont il a besoin sous la forme d'unités, puis chercher dans le référentiel le module qui contient ces unités. On propose également trois stratégies de recherche de modules : ALL-match, SOME-match et ONE-match. Le critère

[5] OBJ n'est pas un acronyme mais une mnémonique du terme « Object »

ALL-match vérifie que toutes les unités de la requête correspondent à des unités distinctes du composant. Le critère SOME-match relaxe le critère ALL-match en se contentant de vérifier qu'au moins un sous-ensemble non vide d'unités de la requête correspond à un sous-ensemble non vide d'unités du composant. Le critère ONE-match vérifie qu'au moins une des unités de la requête correspond à une des unités du composant.

2.4. Techniques de recherche comportementale

Les techniques de recherche comportementale s'intéressent à l'aspect dynamique des composants en analysant leur comportement lors de l'exécution.

2.4.1 Approche par analyse des traces d'exécutions

Dans (Podgursky and Pierce, 1993), on déduit après une observation statistique qu'un composant appartenant à un référentiel de composants peut être identifié en se basant uniquement sur son comportement avec des paramètres d'entrée aléatoires. En se basant sur cette observation, on a construit une base de composants qui a les propriétés suivantes :

- Un composant est représenté par son code exécutable et la description de ses paramètres d'entrée. Un paramètre d'entrée est défini par son type ainsi qu'une distribution probabiliste des valeurs qu'il peut prendre par rapport à son domaine de définition. La distribution probabiliste permet d'estimer la pertinence du choix d'une valeur par rapport à son occurrence lors de l'utilisation du composant.

- La requête se compose de deux parties : l'espace des valeurs d'entrée désirées et une condition qui détermine si la réponse d'un composant est satisfaisante par rapport aux besoins de l'utilisateur.

- L'opération de comparaison de la requête et d'un composant de la base se fait en trois étapes : sélection aléatoire d'un certain nombre de valeurs d'entrée en se basant sur les paramètres de la requête et la description du composant dans la base, application des valeurs d'entrée sur tous les composants de la base, sélection des seuls composants qui vérifient les paramètres de la requête. On constate que la probabilité qu'un composant de la base soit non pertinent par rapport à une requête bien qu'il satisfasse sa condition après son exécution avec n valeurs d'entrées aléatoires, décroît exponentiellement en fonction de n. On peut donc atteindre une très grande précision avec des valeurs petites de n. Malgré une bonne précision, Cette méthode fournit un rappel très faible.

Dans (Hall, 1993), on propose une approche de recherche comportementale généralisée des composants. On critique certains aspects de l'approche précédente et on propose des améliorations parmi lesquelles :

- La possibilité de définir des paramètres optionnels pour les fonctions, ce qui a tendance à améliorer le rappel ;

- La possibilité que l'utilisateur choisisse, lui-même, les données d'entrée par rapport auxquelles les composants sont testés. On affirme que le fait de choisir aléatoirement les paramètres d'entrée n'améliore pas forcément la précision et le rappel, mais au contraire affecte fortement les performances du système ;

- La relaxation de la fonction de mesure de similarité entre les résultats d'exécution et les conditions de sélection précisées dans la requête utilisateur afin d'améliorer le rappel.

Enfin, dans (Park and Bais, 1997), on propose une autre approche pour la recherche comportementale qui représente une continuité des travaux de (Podgursky and Pierce, 1992 ; Hall, 1993). Au lieu de générer statistiquement les valeurs d'entrée (Podgursky and Pierce, 1992) ou de demander à l'utilisateur de les spécifier (Hall, 1993), on utilise une méthode inductive. L'idée consiste à faire une induction sur les structures d'entrée en utilisant une base de cas. Les entrées obtenues sont les cas sélectionnés dans la base de cas et les cas induits.

2.4.2 Approche par spécification comportementale

Dans (Chou et al., 1996), on présente un référentiel de composants orientés objet utilisant une technique comportementale de sélection de composants. Un composant est représenté dans son référentiel non plus par son code exécutable, mais par des réseaux sémantiques décrivant son comportement. La requête est comparée aux composants pour sélectionner ceux qui maximisent la fonction de similarité qui est égale au rapport de la cardinalité des comportements communs sur la cardinalité totale des comportements de la requête (coefficient de Jaccard's). L'avantage de cette technique est qu'elle permet non seulement de retrouver les composants simples (une classe), mais également les composants complexes (un graphe de classes). De plus, la technique est outillée par un éditeur de spécifications, un outil de classification de spécifications, un gestionnaire de référentiels de composants et un outil de recherche de spécifications.

2.5. Techniques de recherche sémantique

Les techniques de recherche sémantique s'intéressent à l'aspect sémantique des composants en analysant les axiomes formels qui représentent des informations supplémentaires sur les concepts et leurs relations ainsi que des restrictions relatives aux valeurs de propriétés et de concepts. Par conséquent, les axiomes jouent un rôle important dans le domaine sémantique et peuvent être pertinents pour la recherche de composants (Noy and Hafner, 1997).

2.5.2 Approche par extension de mots clés

Les techniques de cette catégorie étendent les techniques de recherche par mots-clés. Les mots-clés ajoutés possèdent un lien sémantique avec les mots-clés extraits de la requête.

Durant les décennies 60 à 80, les techniques d'expansion de requêtes se basaient sur le critère de co-occurrence des termes d'indexation et n'ont pas eu beaucoup de succès (Salton, 1986). Néanmoins, le travail de (Peat and Willett, 1991) conclut que les requêtes étendues avec ces modèles ne sont pas meilleures que les requêtes d'origine et que des requêtes étendues avec des mots choisis aléatoirement donnent des résultats parfois meilleurs. Ces constats soulignent la nécessité d'avoir recours à d'autres ressources pour l'expansion. Des expériences sur l'expansion des requêtes par des termes reliés sémantiquement ont déjà été effectuées et ont abouti à des conclusions différentes. Dans (Voorhees, 1994 ; Moldovan and Mihalcea, 2000 ; Baziz, 2005), on a utilisé WordNet[6] pour l'expansion des requêtes en ajoutant des termes reliés sémantiquement aux termes de la requête initiale. La relation sémantique de base utilisée est la synonymie. Cette technique nécessite de désambiguïser les termes dans les requêtes initiales. Il a été reporté que les requêtes longues deviennent vite bruitées et dégradent la précision après une expansion non contrôlée qualifiée d'"agressive" (Voorhees, 1994). Cependant, ces auteurs s'accordent à dire que cette méthode peut être intéressante si la désambiguïsation s'avère performante.

2.5.2 Approche basée sur les ontologies

Les moteurs de recherche sémantiques reposent sur certaines structures d'ontologies. Les ontologies sont généralement construites à partir de concepts, de propriétés, de contraintes et éventuellement d'axiomes.

[6] http://wordnet.princeton.edu/

Les ontologies peuvent servir à calculer la similarité entre la représentation de la requête et la représentation des documents dans le cas où les deux représentations sont construites à partir des concepts d'une même ontologie. Cette approche est utilisée dans (Andreasen, 2003) où documents et requêtes sont décrits en utilisant le langage et l'ontologie Ontologu. Cette ontologie est un graphe orienté contenant un ensemble de concepts reliés par des relations dont la relation de subsomption. L'avantage du calcul de la similarité est de classer les documents en fonction de leur similarité à la requête, cette similarité dépend de l'organisation des concepts dans l'ontologie.

Cette approche est originale car elle calcule la similarité entre les concepts des documents et ceux de la requête (Ralalason, 2010). Cependant, on ne donne aucune indication sur la combinaison des différents facteurs de la mesure. De plus, les auteurs ne considèrent pas le cas de figure où plusieurs concepts sont retrouvés à la fois dans la requête et les documents ni comment les différentes similarités sont combinées. Enfin, aucune évaluation n'est proposée.

Dans (Guarino, 1999), on présente le système OntoSeek qui permet d'améliorer l'accès aux pages jaunes à partir d'un mécanisme reposant sur WordNet. Les documents et les requêtes sont représentés par des graphes conceptuels formés de nœuds et d'arcs dont les labels sont issus de WordNet. Une interface aide l'utilisateur dans la conception de ces graphes. Pour étiqueter les nœuds, l'utilisateur peut soit proposer des mots qui sont ensuite désambiguïsés à partir de WordNet, soit directement naviguer dans WordNet pour sélectionner les concepts qui l'intéressent.

De même, les arcs sont étiquetés soit à partir d'une liste proposée par le système, soit à partir de termes proposés par l'utilisateur. Un procédé d'appariement entre le graphe de la requête et l'ensemble des graphes représentant les documents est ensuite mis en place. Le système recherche les graphes de documents qui subsument (ou qui spécialisent) le graphe de la requête. Les résultats sont présentés à l'utilisateur à travers une interface HTML.

Dans (Alnusair and Zhao, 2010), on utilise un algorithme de similarité fondé sur un arbre sémantique construit à partir d'une ontologie de domaine OWTS (Ontology-based Weighted semantic Tree Similarity algorithm) en se basant sur la représentation sémantique des requêtes et des titres de documents.

Dans (Aufaure et al., 2007), on recherche les informations pertinentes en appliquant les techniques du modèle vectoriel et le cosinus pour calculer la similarité entre concepts, après avoir reformulé les requêtes.

Dans (Xiaomeng and Atle, 2006), on propose une méthode heuristique semi-automatique qui permet de faire correspondre deux ontologies. La mise en correspondance entre ontologies est focalisée sur la mise en correspondance des concepts et relations où la distance sémantique est enrichie à l'aide de la lemmatisation, la recherche grammaticale des fonctions, des mots et des types (nom, verbe, adjectif…) de mots moyennant Wordnet.

Dans (Gligorov et al., 2007), on propose une méthode de mise en correspondance approximative de concepts basée sur la mesure de similarité utilisée par Google. Cette mise en correspondance approximative est surtout nécessaire quand les concepts sont vagues, flous ou mal définis. L'appariement d'ontologie est souvent la recherche d'équivalence entre deux concepts A et B de deux hiérarchies différentes. L'équivalence est traitée comme une subsomption mutuelle entre A et B. On dit qu'un concept A subsume un concept B si ce dernier est un sous-concept de A.
Ces fonctions de similarité sémantique conceptuelle peuvent restituer des documents potentiellement pertinents pour les requêtes des utilisateurs.

2.7. Discussion

Mise à part la recherche sémantique, la classification que nous avons présentée est celle décrite dans (mili et al., 2001). Ces catégories traditionnelles sont limitées parce qu'elles ne tiennent pas compte (ou très peu) des relations sémantiques entre composants ou des informations pertinentes liées à un domaine spécifique lors du traitement des requêtes. Cependant, la recherche sémantique propose une approche fondée sur des ontologies et des modèles de domaine dans le but d'accroître l'efficacité et la pertinence de la recherche et de fournir des informations sur les composants sélectionnés. Dans ce domaine, on propose beaucoup de mesures de similarité qui permettent d'effectuer le rapprochement entre le composant recherché (requête) et les composants candidats pré-existants dans différents référentiels. Le calcul de similarité s'appuie sur les concepts et leurs relations ainsi que les axiomes formels s'ils existent.

3. Techniques de sélection de composants

Le problème de la prise de décision multi-critères, qui consiste à trouver le meilleur candidat (ou à classer les candidats disponibles) selon différents critères, a été introduit par (Koopmans, 1951) et (Kuhn and Tucker, 1951). Depuis, la prise de décision multi-critères s'est développée en tant que domaine de recherche avec des techniques qui s'appliquent à de nombreux domaines, tels que l'intelligence artificielle et le génie logiciel orienté composant (George, 2007) où on les a utilisées dans le processus de sélection de composants. En effet, l'objectif ultime du processus de sélection est de choisir le meilleur candidat pour une application donnée. Ce choix est le résultat d'une décision, qui est prise après avoir évalué tous les candidats en se basant sur les mêmes critères. Cette étape doit donc être réalisée au moyen d'une technique capable d'analyser et de classer les candidats selon plusieurs critères d'évaluation préalablement définis et pondérés.

3.1. MCDA (Multi-Criteria Decision Aid)

L'aide à la décision multi-critères MCDA (Roy, 1996) consiste à :

1. identifier les critères qu'un composant devrait remplir ;
2. assigner à chacun de ces critères un nombre réel qui représente son poids, c'est-à-dire son importance dans la décision ;
3. assigner aux candidats des valeurs réelles indiquant leurs scores de satisfaction de ces critères (il y a un score par critère pour chaque candidat, le score maximal étant égal au poids du critère) ;
4. additionner ces scores pour chaque candidat afin d'obtenir son score total.

Le meilleur candidat est celui qui a le score le plus élevé. L'évaluation des candidats, quant à elle, se fait d'abord localement pour chaque critère, puis globalement en agrégeant les scores locaux.

Le tableau 2.1 montre un exemple d'application de la technique MCDA à la sélection de composants. L'utilisateur a défini 6 critères pour l'évaluation des candidats (première colonne sur le tableau). Il a ensuite estimé manuellement l'importance de chaque critère en lui attribuant un poids (Weight sur la seconde colonne). Puis il a évalué manuellement le score de chaque candidat pour chaque critère (sur les autres colonnes).

Tableau 2.1: Exemple d'application de MCDA (George, 2007)

Evaluation criteria	Weight	Score Comp. A	Score Comp. B	Score Comp. C
Usability	25.0	23.0	15.0	20.0
Compatibility	10.0	7.0	2.0	1.0
Cost	15.0	10.0	5.00	4.0
Functionality	20.0	15.0	4.0	12.0
Security	20.0	15.0	12.0	17.0
Technical support	10.0	7.0	5.0	5.0
Total score	100.0	**77.0**	**45.0**	**59.0**

Le poids représente également le score maximal qu'un candidat peut avoir pour ce critère, selon l'utilisateur. Il suffit ensuite d'additionner les scores du candidat afin d'obtenir son score total, par exemple celui du candidat C est égal à : 20 + 1 + 4 + 12 + 17 + 5 = 59. Afin de garder des valeurs homogènes, la somme des poids est égale à 100. De ce fait, le score final de chaque candidat représente son pourcentage total de satisfaction des critères d'évaluation. On constate donc que le candidat A est de loin le meilleur avec un score total de 77, loin devant les candidats B (avec un score total de 45) et C (avec un score total de 59). A est donc le meilleur candidat et doit être sélectionné.

3.2. WSM (Weighted Scoring Method)

La méthode intitulée WSM (ou WAS, pour Weighted Average Sum en anglais) est la technique d'agrégation la plus utilisée dans les situations de prise de décision (Ncube and Dean, 2003). Le principe général tel qu'il est décrit dans (Kontio et al., 1996) est le suivant : « Des critères sont définis et à chaque critère on assigne un poids ». WSM présente quelques points communs avec MCDA : (1) à chaque critère d'évaluation on associe un poids qui représente son importance, (2) à chaque couple critère-candidat on associe un score qui représente la capacité du candidat à remplir le critère et (3) le candidat qui possède le score total le plus élevé est considéré comme le meilleur et doit être sélectionné. Cependant, contrairement à MCDA, le poids ne représente pas le score maximal pour un critère et influe sur le score du candidat. L'avantage est de pouvoir assigner des poids indépendamment des scores locaux. L'inconvénient est que les poids ne sont plus bornés. Pour calculer le score total d'un candidat, on multiplie pour chaque critère d'évaluation le score correspondant par le poids du critère, et on agrège les produits ainsi obtenus.

La formule WSM la plus commune est celle donnée dans (Kontio et al., 1996) et qui utilise l'addition comme fonction d'agrégation :

$$score_c = \Sigma_{j=1}^{n}(weight_j * score_{cj})$$

Le poids $weight_j$ représente l'importance du $j^{ème}$ critère par rapport aux n-1 autres critères d'évaluation. Le score local $score_{c\,j}$ évalue le degré de satisfaction du critère numéro j par le candidat c. Le score total $score_c$ représente la valeur globale d'évaluation de c.

Le tableau 2.2 donne un exemple d'application de la méthode WSM sur 3 composants candidats *A*, *B* et *C*. L'utilisateur a défini 6 critères pour l'évaluation de ces candidats (première colonne sur le tableau), puis a estimé l'importance de chaque critère en lui attribuant un poids (la seconde colonne) avant d'évaluer le score de chaque candidat pour chaque critère (les autres colonnes). Les critères sont les mêmes que pour le tableau 2.1. Cependant, les valeurs pour les scores sont indépendantes des poids. Afin de préserver malgré tout l'homogénéité des valeurs, chaque poids et chaque score sont compris entre 1 (la mauvaise valeur) et 5 (la meilleure valeur). Par exemple, pour le critère sécurité, l'utilisateur a considéré que le candidat *A* n'était pas du tout sûr, que le candidat *B* l'était peu, et que le candidat *C* avait le niveau maximal.

Tableau 2.2 : Exemple d'application de WSM (Ncube and Dean, 2002)

Evaluation criteria	Weight	Score Comp. A	Score Comp. B	Score Comp. C
Usability	2.0	3.0	3.0	3.0
Compatibility	4.0	1.0	5.0	2.0
Cost	3.0	3.0	5.0	1.0
Functionality	5.0	4.0	4.0	3.0
Security	4.0	1.0	2.0	5.0
Technical support	5.0	2.0	5.0	3.0
Total score	XXXX	**53.0**	**94.0**	**67.0**

Ensuite, on multiplie ces scores avec le poids donnée plus haut et on obtient les scores totaux de chaque candidat : par exemple le score total du candidat *C* est égal à : 2 * 3 + 4 * 2 + 3 * 1 + 5 * 3 + 4 * 5 + 5 * 3 = 67. On s'aperçoit donc que le candidat *B* possède le meilleur score et doit ainsi être sélectionné.

3.3. AHP (Analytic hierarchical process)

AHP est une technique de prise de décision multi-critères mise au point par (Saaty, 1992). Elle est utilisée avec succès dans de nombreux domaines dont celui du génie logiciel (Finnie et al., 1995 ; Min, 1992). Par rapport aux techniques précédentes, AHP aide à décrire les besoins de manière organisée.

Tout d'abord, il faut définir l'objectif principal à atteindre pour prendre une décision sur la sélection d'un produit candidat. Ensuite, on décompose cet objectif en un arbre hiérarchique de critères et de sous critères d'évaluation dont les feuilles sont les candidats à évaluer. Plus on descend dans l'arbre, plus les critères d'évaluation se précisent. La figure 2.3 montre un exemple d'application de cette décomposition. Si l'objectif principal est d'acheter une voiture, les critères principaux d'évaluation d'une voiture sont sa consommation, son prix, la sécurité qu'elle offre, etc... Les critères se décomposent ensuite en sous-critères, par exemple la sécurité consiste en la présence d'ABS, d'airbags... Enfin, on relie les voitures candidates (*Volvo, Mercedes*...) aux critères et aux sous-critères qui vont être utilisés pour les évaluer.

De plus, à l'intérieur de chaque nœud critère, on détermine l'importance de chaque souscrit par rapport aux autres. Pour ce faire, on assigne un poids à chaque nœud de l'arbre par rapport aux autres nœuds ayant le même parent. Par exemple, supposons que l'achat d'une voiture ne se décide qu'en fonction des trois critères montrés sur la figure 2.3 : La consommation, le prix et la sécurité. On peut pondérer ces trois critères de telle sorte que la consommation est 2 fois plus importante que la sécurité et 6 fois moins importante que le prix. A ce stade, on peut utiliser une méthode comme WSM pour évaluer chaque candidat *c* en additionnant les scores locaux à l'intérieur de chaque nœud et en propageant les sommes obtenues jusqu'à la racine de l'arbre pour obtenir le score total du candidat.

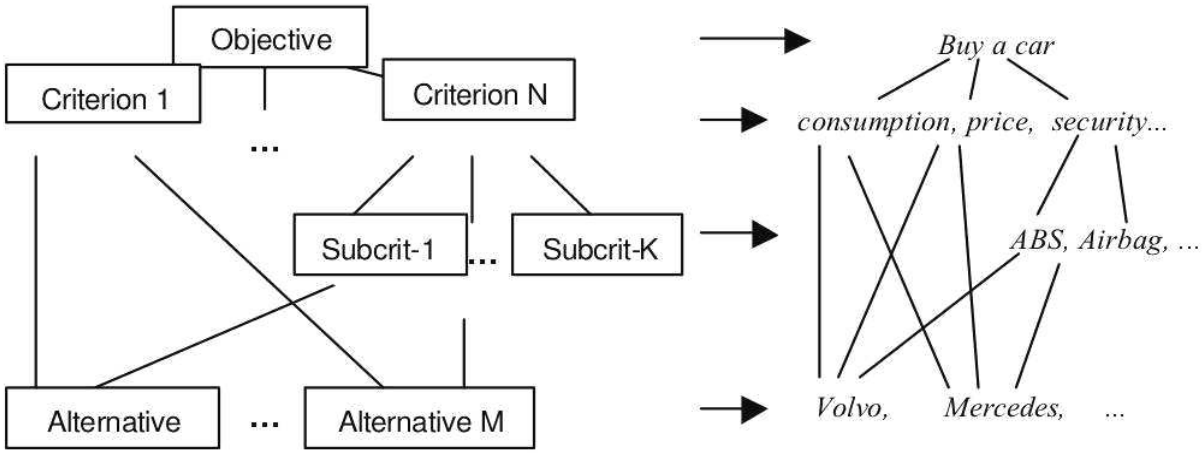


Figure 2.3 : Hiérarchisation de critères dans AHP (Lozano and Gomez, 2002)

AHP est donc un moyen de définir et de hiérarchiser les différents critères d'évaluation avec précision. Cela permet d'utiliser WSM en assignant des poids cohérents. En effet, à l'intérieur de chaque nœud, les poids des nœuds fils sont bornés les uns par rapport aux autres. Cependant, comme il est décrit dans (Ncube and Dean, 2002), il est difficile de l'employer tel quel pour des systèmes logiciels complexes, qui comporteraient un nombre élevé de critères et de sous-critères et pour lesquels il y aurait un grand nombre de candidats à évaluer.

3.4. Discussion

Face à un nombre élevé de composants et de critères, les techniques d'évaluation deviennent fastidieuses. En effet, toutes les attributions de poids et de scores locaux, pour chaque candidat et chaque critère, sont manuelles.

La technique AHP peut utiliser WSM pour évaluer chaque candidat en additionnant les scores locaux à l'intérieur de chaque nœud et en propageant les sommes obtenues jusqu'à la racine de l'arbre pour obtenir le score total du candidat.

La technique AHP rend le processus d'évaluation plus pertinent grâce au raffinement des différents critères. Le recouvrement des critères est moins pénalisant que dans les techniques MCDA et WSM. Cependant, l'application de AHP est beaucoup plus complexe. De ce fait, nous utilisons la technique MCDA pour des raisons de simplicité.

4. Etude de méthodes de sélection

Dans (Maiden et al., 1997), on a constaté que les méthodes classiques de spécification des besoins (*Requirements Engineering*) étaient inappropriées pour des applications construites par assemblage de composants sélectionnés à partir de référentiels. Pour réussir la sélection de composants, il est donc primordial que les besoins soient spécifiés avec précision (Ncube, 2000). A cet effet, il existe de nombreux travaux qui permettent à l'utilisateur de spécifier correctement ses besoins en tenant compte de l'offre qui existe sur le marché des composants.

4.1. Description des méthodes

4.1.1 OTSO (Off The Shelf Option)

OTSO est un processus de sélection mis au point par J. Kontio dans (Kontio et al., 1995). A l'origine, il s'agissait d'un mécanisme de recherche de composants dans les référentiels et il est finalement devenu l'un des premiers processus pionniers de la sélection.

Vu que l'objectif principal est d'intégrer un ou plusieurs composants dans une application, OTSO propose de :

1. donner une définition claire des tâches à mener dans le processus de sélection,
2. donner une définition incrémentale, hiérarchique et détaillée des critères, et
3. utiliser de manière appropriée les méthodes existantes de prise de décision (MCDA) pour analyser les résultats de l'évaluation.

Pour chacun de ces objectifs, OTSO recommande un certain nombre de méthodes. La figure 2.4 montre les principales phases (représentées par des cercles) du processus OTSO ainsi que les paramètres et données utilisées (encadrées par deux barres).

Une première phase consiste à définir les critères d'évaluation selon quatre paramètres liés à l'application à composer. Le premier paramètre concerne les besoins de l'application (*Requirement specification*). Cela inclut en particulier les besoins fonctionnels. Le second paramètre concerne l'architecture globale de l'application (*Design specification*). Quant aux troisième et quatrième paramètres, il s'agit de contraintes d'organisation (*Organisational characteristics*) et de contraintes de projet (*Project plan*) relatifs à son développement. Dans cette phase, OTSO propose ses propres critères, détaillés dans (Kontio et al., 1996).

Dans la phase de recherche (*Search*), le but est d'identifier les candidats potentiels ce qui peut conduire à modifier la spécification des besoins et à influencer la phase de définition des critères d'évaluation.

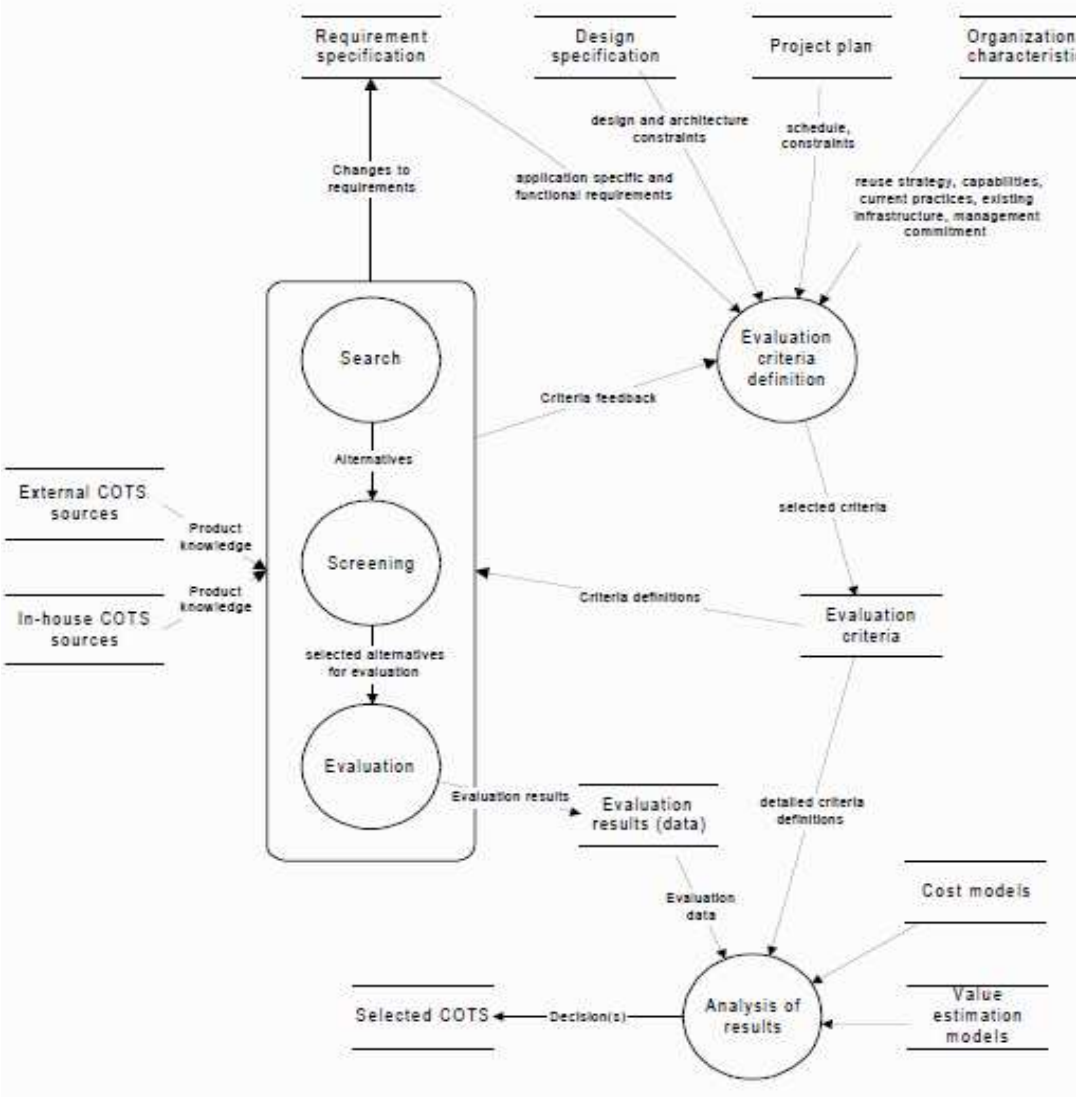


Figure 2.4 : La méthode OTSO (Kontio, 1996)

La phase de pré-sélection (*Screening*) consiste alors à filtrer les candidats les plus pertinents au moyen de critères de base et de certaines informations sur ces candidats (*COTS Sources*). On peut alors passer à la phase suivante, qui consiste à évaluer en détail chaque candidat retenu par le biais de critères plus précis. La phase finale consiste à analyser les résultats de ces évaluations en fonction des critères précédemment définis, de méthodes d'estimation de coût et d'approches multi-critères telles que WSM (voir la section 3.2) et AHP (voir la section 3.3). Cela permet de classer les candidats et de sélectionner le meilleur d'entre eux.

4.1.2 PORE (Procurement-Oriented Requirements Engineering)

La méthode PORE est proposée (Maiden and Ncube, 1998) dans le but d'aider l'utilisateur à bien choisir son composant en fonction de l'existant. Cette méthode entrelace la spécification des besoins et la sélection de candidats, comme le montre la figure 2.5. D'une part, les besoins de l'utilisateur (*customer requirements*) permettent la sélection des candidats pertinents (*product selection*) parmi ceux qui sont disponibles. D'autre part, l'examen des candidats ainsi filtrés (*candidate products*) informe l'utilisateur et l'incite à définir de nouveaux besoins ou à détailler ceux qui sont toujours valables.

La précision de la spécification des besoins s'accroît au fur et à mesure que le nombre de candidats décroît.

Initialement, PORE définit les trois types de besoins suivants :

1. les besoins fonctionnels principaux,
2. les besoins fonctionnels secondaires, et
3. les besoins non-fonctionnels.

Afin de satisfaire ces trois besoins, PORE utilise un processus itératif, décomposé en cinq processus génériques illustrés sur la figure 2.5. Le premier processus (*Identify Product*) consiste à repérer les candidats potentiels au moyen d'une étude de marché ou d'une expertise. Le second processus (*Acquire Information*) consiste à obtenir des informations à la fois sur les besoins du client (*customer requirements*) et sur les candidats précédemment identifiés (*product features*).

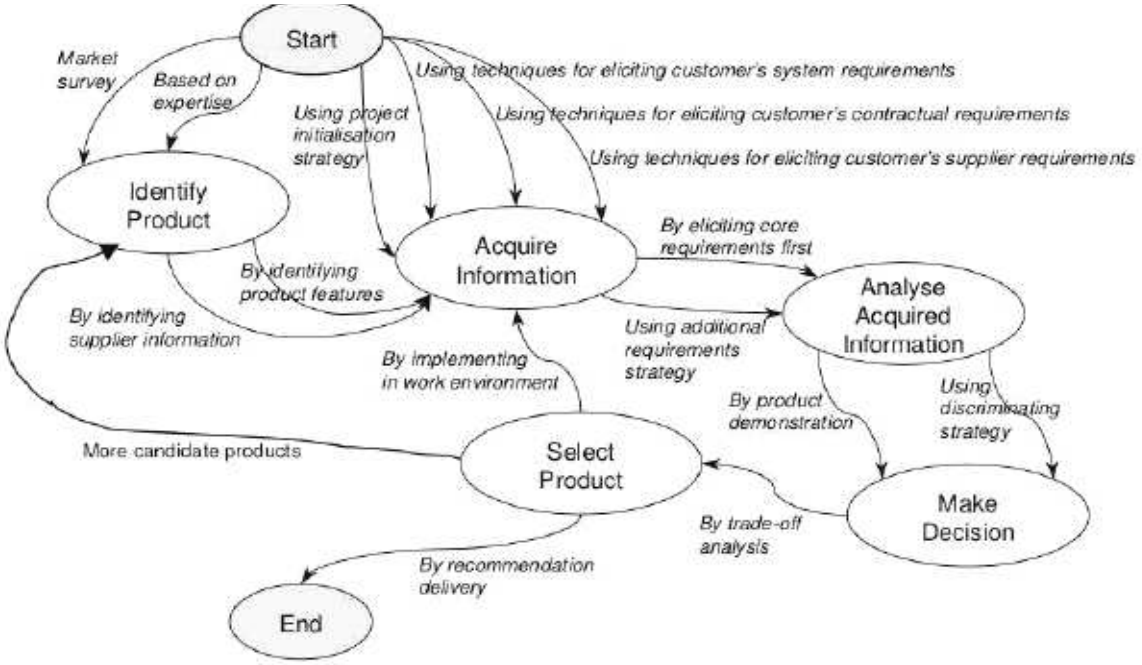


Figure 2.5 : Les cinq processus génériques de PORE (Ncube, 2000)

Le troisième processus (*Analyse Acquired Information*) consiste à analyser l'information ainsi acquise pour produire des données exploitables dans le quatrième processus (*Make Decision*). Ce dernier consiste, à partir de données et de techniques de prise de décision multi-critères, à déterminer si les candidats satisfont les besoins ou non. Quant au cinquième processus (*Select Product*), il consiste à rejeter les candidats qui ont été évalués comme non satisfaisants par le processus précédent. Enfin, une fois que la sélection est effectuée, deux choix sont possibles : (1) Soit on recommence l'opération en retournant au premier ou au second processus pour détailler les besoins et réduire la liste des candidats, (2) soit on arrête la sélection (*End* sur la figure 2.5) quand on estime que la requête est satisfaite. Chaque processus peut être répété plusieurs fois dans le cas où l'information est insuffisante.

Afin d'exécuter ces processus génériques, PORE se base sur les méthodes existantes. Par exemple, l'acquisition d'informations dans le second processus se fait au moyen de cas d'utilisation et de techniques d'ingénierie des connaissances. De plus, AHP (voir section 3.3) est utilisé dans le quatrième processus afin de choisir les critères de sélection des candidats.

La méthode PORE est composé de trois méta-modèles qui décrivent : (1) les candidats, (2) les besoins fonctionnels et non-fonctionnels et (3) le score de satisfaction des besoins par un candidat. Cependant, ce score est estimé manuellement (Ncube, 2000).

4.1.3 CRE (COTS-based Requirements Engineering)

CRE (Alves et Castro, 2001) possède quatre phases itératives : *Identification*, *Description*, *Evaluation* et *Acceptation*.

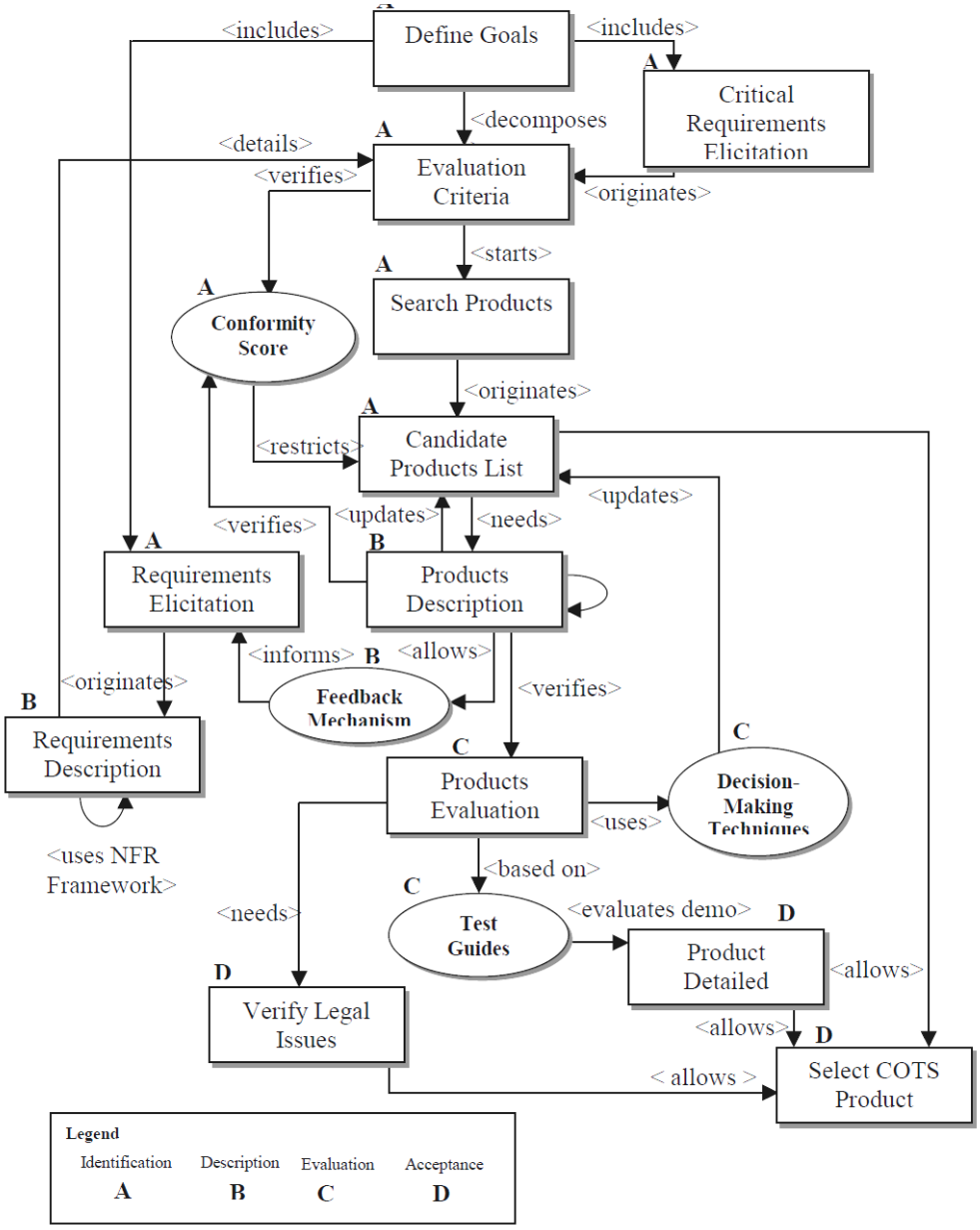


Figure 2.6 : Etapes de la méthode CRE (Alves et Castro, 2001)

La phase d'identification permet de définir les objectifs en se basant sur certains facteurs : (1) besoins des utilisateurs, (2) architecture de l'application, (3) objectifs et restrictions du projet, (4) disponibilité des composants et (5) infrastructure de l'organisation.

Les critères d'évaluation devraient être élaborés en considérant tous ces facteurs. A ce stade, les besoins et particulièrement les besoins non fonctionnels (NFR) sont mal définis car ils sont mal compris par les clients. De ce fait, ces besoins doivent être modélisés et affinés dans la phase de description avant d'effectuer l'évaluation technique.

Dans la phase de description, les critères d'évaluation doivent être élaborés en détail en insistant sur le rôle des NFR pour la discrimination de produits (composants) similaires. En général, les clients ne sont pas en mesure d'apporter une description complète des aspects de qualité (stabilité, flexibilité, performance, etc.). Cependant, le traitement des NFR peut être indispensable pour le succès d'un système de sélection. La méthode CRE utilise le NFR Framework (Chung et al., 2000) pour la représentation et l'analyse des besoins non-fonctionnels. Ce framework est une approche orientée processus où les NFR sont explicitement représentés comme des objectifs à atteindre. Chaque objectif sera ensuite décomposé en sous-objectifs représentés par une structure de graphe inspiré des arbres ET/OU utilisés dans le domaine de la résolution de problèmes. Par exemple, la figure 2.7 montre une décomposition des besoins non-fonctionnels en utilisant le NFR framework.

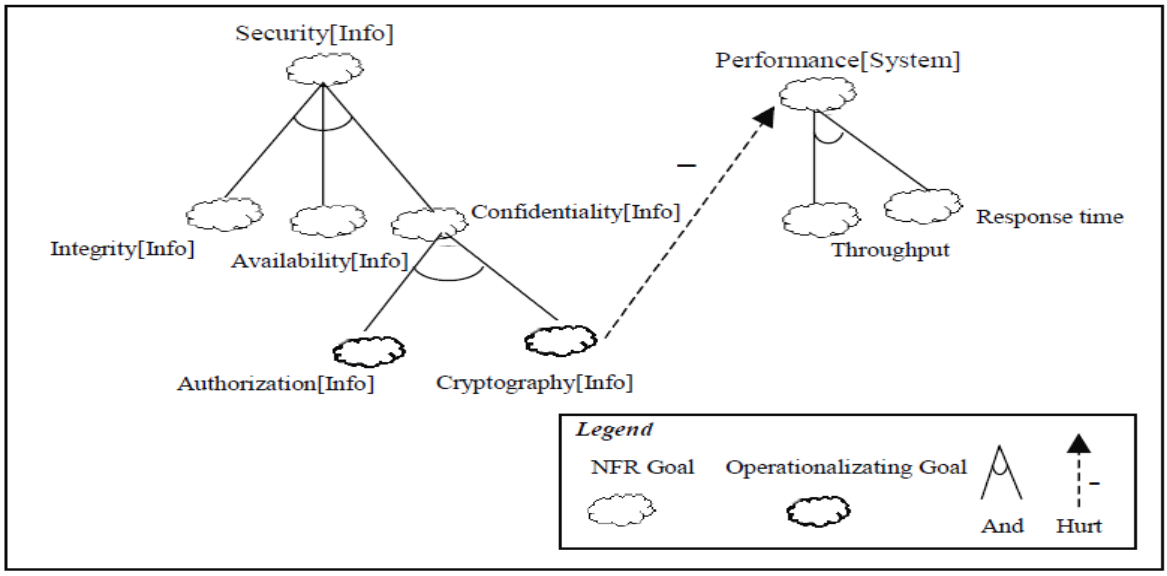


Figure 2.7 : Décomposition des besoins non fonctionnels utilisant le NFR Framework (Alves et Castro, 2001)

Par exemple, dans la Figure 2.7, l'objectif Security[Info] est décomposé en plusieurs sous-objectifs : Integrity[Info], Availability[Info] et Confidentiality[Info]. Ces derniers doivent être satisfaits afin que l'objectif global soit atteint. Notons aussi que les interactions entre les

différents types de NFR peuvent être spécifiées. Par exemple (voir la figure 2.7), la cryptographie influe négativement les performances du système.

Dans la phase d'évaluation, la décision de sélection d'un produit particulier COTS est basé sur le coût estimatif par rapport à l'analyse des avantages de chaque solution de rechange COTS. Le processus CRE suggère l'utilisation de modèles de coûts tels que les COCOTS (constructive COTS) (Boehm, 1998).

4.2. Etude analytique des méthodes

Même s'il n'existe pas de méthode communément acceptée pour la sélection de composants (Ruhe, 2003), les méthodes existantes partagent certaines étapes clés qui se chevauchent et peuvent être itératives. Ces étapes, présentées dans (Mohamed, 2007) sous l'appellation de « processus de sélection général » (GCS pour General COTS Selection), sont les suivants :

- *Etape 1*: Définir les critères d'évaluation basés sur les exigences et les contraintes du développeur;
- *Etape 2*: Rechercher les composants à partir des référentiels ;
- *Etape 3*: Filtrer les résultats de recherche basée sur les exigences. Cela permet la définition d'une liste des composants candidats prometteurs qui doivent être évalués de manière plus détaillée ;
- *Étape 4*: Évaluer les composants candidats de la liste restreinte ;
- *Étape 5*: Analyser les données d'évaluation (à savoir la sortie de l'étape 4) et sélectionner le composant qui possède la meilleure correspondance avec les critères. Généralement, les techniques de prise de décision, comme par exemple le processus hiérarchique analytique (AHP) (voir section 3.3), sont utilisées pour effectuer la sélection.

Après l'*étape 5*, le composant sélectionné est généralement adapté pour répondre exactement aux exigences et aux contraintes de l'utilisateur.

4.3. Problèmes liés aux méthodes de sélection actuelles

La majorité des méthodes de sélection n'ont pas traités suffisamment les besoins non-fonctionnels. Comme nous l'avons déjà mentionné, les critères non fonctionnels peuvent être décisifs dans le choix des composants candidats. Certaines méthodes comme OTSO ne proposent rien pour spécifier ces besoins. Un autre inconvénient est que ces méthodes sont

basées sur des techniques de prise de décision multi-critères où les scores sont toujours attribués manuellement.
De ce fait, nous optons pour un examen attentif des besoins non fonctionnels en hiérarchisant et en affinant ces besoins. Aussi, nous prenons en compte les besoins non fonctionnels liés aux spécificités du domaine de la réutilisation.
Finalement, nous modélisons les besoins non fonctionnels par des modèles ontologiques et non numériques.

5. Conclusion

Dans ce chapitre, nous avons décrit les différentes techniques utilisées pour la recherche des composants et leur sélection Le but étant de pouvoir choisir parmi la panoplie de techniques présentées celles qui vont servir à définir notre démarche. Nous optons pour un processus de sélection général où on passe par trois étapes principales :

- Définition des critères d'évaluation : Dans cette étape, nous modélisons les besoins des composants, et particulièrement la qualité de service, par le biais d'une ontologie ;
- Recherche et évaluation des composants : Dans cette étape, nous proposons un algorithme d'appariement (QoS matching) de la QoS entre les besoins du développeur et les descriptions des composants. Pour cela, nous proposons un score de pertinence sémantique ;
- Analyse de l'évaluation: Dans cette étape, nous classons les composants (component ranking) du plus pertinent au moins pertinent en utilisant le processus MCDA.

La première étape permet d'amorcer et de garantir les étapes qui la succèdent. Nous devons de ce fait effectuer un choix judicieux quant à la représentation de la QoS. Reposant sur le formalisme ontologique, une technologie du Web sémantique, nous devons explorer dans le prochain chapitre le domaine en question.

Chapitre 3

Technologies du Web sémantique

Le Web sémantique (WS) est une vision qui permet de faciliter et d'améliorer la recherche d'information, faite par l'homme et la machine, grâce à la représentation sémantique du contenu Web. Il étend le Web classique (Berners-Lee et al., 2001) dans le sens où le Web est strictement syntaxique, la structure des ressources accessibles est bien définie mais le contenu du Web reste quasiment inaccessible aux traitements machine.

Les ontologies jouent un rôle très important pour la réalisation du Web sémantique. Elles sont utilisées pour fournir le vocabulaire et la structure des métadonnées associées aux ressources, ou comme représentations pivot pour l'intégration de sources de données hétérogènes, ou pour décrire les Web services ou, en général, partout où il va être nécessaire d'appuyer des modules logiciels sur des représentations sémantiques.

Dans ce contexte, nous pensons que les ontologies peuvent apporter énormément aux systèmes de sélection de composants, particulièrement, ceux qui s'appuient sur les descriptions de composants.

Ce chapitre définit la notion d'ontologie dans le cadre général du WS puis décrit les deux notions qui sont exploitées dans le cadre de notre recherche, à savoir : le raisonnement et l'appariement sémantique.

1. Définition du Web sémantique (WS)

L'expression Web sémantique, attribuée à Tim Berners-Lee (Berners-Lee et al., 2001) au sein du W3C[7], fait d'abord référence à la vision du Web de demain comme un vaste espace

[7] World Wide Web Consortium

d'échange de ressources entre êtres humains et machines qui permet une meilleure exploitation de grands volumes d'informations et de services variés. Le Web sémantique, à la différence du Web actuel, devrait décharger les utilisateurs d'une bonne partie de leurs tâches de recherche, de construction et de combinaison des résultats, grâce aux capacités accrues des machines à accéder aux contenus des ressources et à effectuer des raisonnements sur ceux-ci.

Le Web sémantique, concrètement, est d'abord une infrastructure de langages pour permettre l'exploitation automatique de connaissances formalisées sur le contenu informel du Web, même si aucun consensus n'existe sur le degré de cette formalisation.
Cette infrastructure doit :

- permettre de localiser, d'identifier et de transformer des ressources (des documents ou des services) de manière robuste tout en renforçant l'esprit d'ouverture du Web aux utilisateurs;
- s'appuyer sur des consensus, par exemple, sur les langages de représentation ou sur les ontologies;
- contribuer à assurer, le plus automatiquement possible, l'interopérabilité et les transformations entre les différents formalismes et les différentes ontologies ;
- faciliter la mise en œuvre de calculs et de raisonnements complexes tout en offrant des garanties sur leur validité ;
- offrir des mécanismes de protection (droits d'accès, d'utilisation et de reproduction).

Les recherches actuellement réalisées s'appuient énormément sur des recherches en représentation ou en ingénierie des connaissances (Gandon, 2002). Or, leur utilisation et leur acceptation à l'échelle du (ou d'une partie du) Web posent de nouveaux problèmes et défis : changement d'échelle dû au contexte de déploiement, le Web et ses dérivés (intranet, extranet), nécessité d'un niveau élevé d'interopérabilité, ouverture, standardisation, diversités des usages, distribution bien sûr et aussi impossibilité d'assurer une cohérence globale.

2. Approche ontologique

2.1. Définition(s) d'une ontologie

En Intelligence Artificielle, la définition communément admise d'une ontologie est énoncée par T. Gruber (Gruber, 1993) comme la "spécification explicite d'une conceptualisation".
Cette définition fait suite à un premier essai en 1991 (Gruber, 1991) :

"An ontology defines the basic terms and relations comprising the vocabulary of a topic area, as well as the rules for combining terms and relations to define extensions to the vocabulary".

Cette définition a ensuite été précisée (Studer et al., 1998) pour devenir :

"La spécification formelle et explicite d'une conceptualisation partagée".

Dans cette définition, par "spécification explicite", les auteurs indiquent qu'une ontologie est un ensemble de concepts, de propriétés, d'axiomes, de fonctions et de contraintes explicitement définis.

Le terme "formel" précise que cette conceptualisation doit pouvoir être comprise et interprétée par des agents logiciels. En effet, la formalisation est nécessaire pour que ces agents puissent être munis de capacités de raisonnement permettant de décharger les différents utilisateurs d'une partie de leur tâche d'exploitation et de combinaison des ressources du Web.

Le terme "partagé" précise l'aspect consensuel du vocabulaire employé. Ce terme rappelle que l'on doit assurer une réutilisation de la formalisation choisie pour permettre l'exploitation des ressources du Web par différentes applications ou agents logiciels.

Enfin, le terme conceptualisation implique également l'aspect intentionnel, lié à un objectif de réalisation.

En bref, une ontologie fournit un vocabulaire de termes et de relations pour modéliser les connaissances d'un domaine d'application.

Pour représenter les ontologies, le W3C a standardisé le langage OWL[8]. Le langage OWL s'appuie sur le langage DAML+OIL, produit de la combinaison de l'américain DAML[9] et OIL[10] provenant de projets européens. Il est actuellement construit sur RDFS[11], et apporte ainsi aux langages du Web sémantique, l'équivalent d'une logique de description tout en disposant d'une syntaxe XML.

[8] Ontology Web Language
[9] Darpa Agent Markup Language
[10] Ontology Inference Layer
[11] Resource Description Framework Schema

2.2. Intérêt des ontologies

Plusieurs recherches (Grüninger and Lee, 2002 ; Russ et al., 1999 ; Uschold and Grüninger, 1996 ; Miszoguchi and Bourdeau, 2000; Staab and Maedche, 2001 ; Psyché et al., 2003), se sont intéressées à l'intérêt que pourrait avoir l'utilisation des ontologies dans les systèmes à base de connaissances (SBC) et le Web sémantique :

Les connaissances du domaine d'un SBC : les ontologies servent à représenter explicitement les connaissances d'un SBC. En particulier, elles servent de schéma à la représentation des connaissances de domaine dans la mesure où elles décrivent les objets, leurs propriétés et la façon dont ils peuvent être combinés pour constituer des connaissances explicites et complètes du domaine.

La communication : les ontologies peuvent intervenir dans la communication entre personnes, organisations et logiciels (Uschold and Gruninger, 1996). En effet, les ontologies servent, par exemple, à créer au sein d'un groupe ou d'une organisation un "vocabulaire conceptuel commun". Dans ce cas, on est plutôt dans le cadre d'une ontologie informelle. Dans le cas de la communication entre personnes et systèmes, l'ontologie est formelle et effectue en général une tâche précise dans le SBC ou le système d'information. L'ontologie est un puissant moyen pour lever les ambiguïtés dans les échanges.

L'interopérabilité : "le développement et l'implantation d'une représentation explicite d'une compréhension partagée dans un domaine donné, peut améliorer la communication, qui à son tour permet une plus grande réutilisation, un partage plus large et une interopérabilité plus étendue" (Uschold and Gruninger, 1996). L'interopérabilité est donc une spécialisation de la communication qui permet de répertorier les concepts que des applications peuvent s'échanger même si elles sont distantes et développées sur des plateformes différentes.

L'aide à la spécification de systèmes : Dans (Mizoguchi and Ikeda., 1997), on explique que "la plupart des logiciels conventionnels sont construits avec une conceptualisation implicite et que la nouvelle génération des systèmes utilisant les travaux en intelligence artificielle devrait être basée sur une conceptualisation explicitement représentée". En effet, l'ontologie fournit une description explicite des objets que doit manipuler le système.

L'indexation et la recherche d'information : dans le Web sémantique, les ontologies y sont utilisées pour déterminer les index sémantiques décrivant les ressources sur le Web.

3. Langages du Web sémantique

La proposition du W3C s'appuie au départ sur une pyramide de langages dont seulement les couches basses sont aujourd'hui relativement stabilisées. La figure 3.1 montre l'organisation en couches des langages du WS proposée par le W3C. Deux types de bénéfices peuvent être attendus de cette organisation :

- Elle permet une approche graduelle dans les processus de standardisation et d'acceptation par les utilisateurs ;
- Par ailleurs, si elle est bien conçue, elle doit permettre de disposer du langage au bon niveau de complexité, celle-ci étant fonction de l'application à réaliser.

Un aspect central de l'infrastructure du WS est sa capacité d'identification et de localisation des diverses ressources. Elle repose sur la notion d'URI (Uniform Resource Identifier) qui permet d'attribuer un identifiant unique à un ensemble de ressources, sur le Web bien sûr mais aussi dans d'autres domaines (documents, téléphones portables, personnes, etc.). Cette notion est à la base même des langages du W3C.

Une autre caractéristique de tous ces langages est d'être systématiquement exprimables et échangeables dans une syntaxe XML. Ceci permet de bénéficier de l'ensemble des technologies développées autour de XML : XML Schemas, outils d'exploitation des ressources XML (bibliothèques JAVA, etc.), bases de données gérant des fichiers XML, même si des langages de requêtes spécifiques (par exemple SPARQL[12], RDF[13] Query) sont nécessaires pour les langages construits sur XML comme RDF.

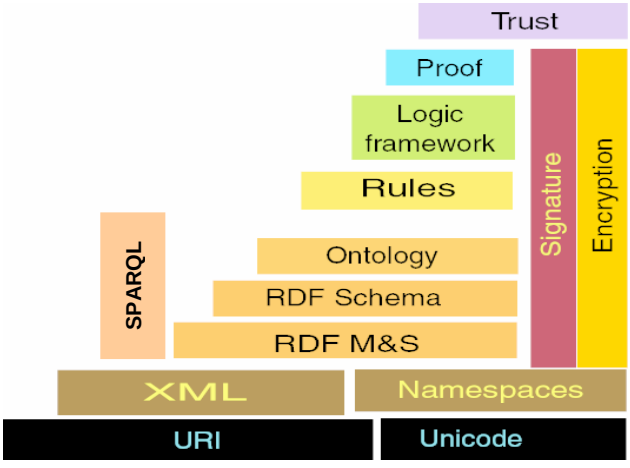


Figure 3.1 : Les couches du Web sémantique (W3C)

[12] Simple Protocol and RDF Query Language
[13] Resource Description Framework

3.1. RDF (Resource Description Framework)

RDF est un modèle de graphe destiné à décrire de façon formelle les ressources du Web et leurs métadonnées, de façon à permettre le traitement automatique de telles descriptions. RDF se base sur des vocabulaires formels et précis, tels que ceux exprimés en RDFS ou en OWL. Il est développé par le W3C et constitue le langage de base du Web sémantique. Une des syntaxes (sérialisation) de ce langage est RDF/XML.

En annotant des documents non structurés et en servant d'interface pour des applications et des documents structurés (par exemple des bases de données), RDF permet une certaine interopérabilité entre des applications échangeant de l'information non formalisée et non structurée sur le Web.

Un document structuré en RDF est un ensemble de triplets de la forme {*sujet, prédicat, objet*} où :

- Le *sujet* représente la ressource à décrire ;
- Le *prédicat* représente un type de propriété applicable à cette ressource ;
- L'*objet* représente une donnée (un littéral) ou une autre ressource : c'est la valeur de la propriété.

Le sujet, et l'objet dans le cas où c'est une ressource, peuvent être identifiés par une URI ou être des nœuds anonymes. Le prédicat est nécessairement identifié par une URI. Les documents RDF peuvent être écrits en différentes syntaxes, y compris en XML. Un document RDF ainsi formé correspond à un multi-graphe orienté et étiqueté. Chaque triplet correspond alors à un arc orienté dont le label est le prédicat, le nœud source est le sujet et le nœud cible est l'objet

Par exemple, dans la figure 3.2, nous modélisons par un graphe RDF le fait qu'un module de systèmes d'information '*MCSI* ' est enseigné par l'enseignante '*Lamia Yessad*' (une ellipse représente une ressource, un rectangle représente un littéral et une flèche représente une propriété).

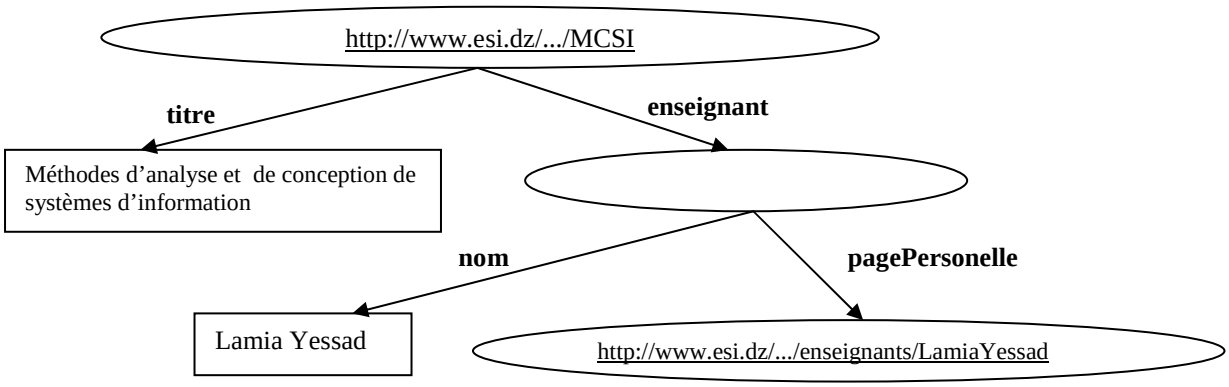


Figure 3.2 : Exemple d'un graphe RDF

Ce graphe peut être sérialisé en RDF/XML comme suit :

```
<rdf:Description rdf:about='http://www.esi.dz/…/MCSI'>
  <titre>Méthodes de conception de systèmes d'information</titre>
  <enseignant>
    <nom> Lamia Yessad </nom>
    <pagePersonelle>
     <rdf:Description rdf:about='http://www.esi.dz/… /enseignants/Lamia Yessad'>
    </pagePersonelle>
  </enseignant>
</rdf:Description>
```

3.2. RDFS (RDF Schema)

RDFS ajoute à RDF la possibilité de définir des hiérarchies de classes et de propriétés dont l'applicabilité et le domaine de valeurs peuvent être contraints à l'aide des attributs rdfs:domain et rdfs:range. A chaque domaine applicatif peut être ainsi associé un schéma identifié par un préfixe particulier et correspondant à une URI. Les ressources instances sont ensuite décrites en utilisant le vocabulaire donné par les classes définies dans ce schéma. Les applications peuvent alors leur donner une interprétation opérationnelle. On peut noter que RDFS n'intègre pas en tant que tel de capacités de raisonnement. Les principales caractéristiques de RDFS sont :

- *rdfs:Class* permet de déclarer une ressource RDF comme une classe pour d'autres ressources. Par exemple nous pouvons définir en RDFS la classe *Cours* qui décrit les cours d'une école :

```
<rdfs:Class rdf:ID='Cours'/>
```

L'expression formelle 'MCSI rdf:type ex:Cours' en RDF traduit l'énoncé en langage naturel : 'MCSI est un cours'

- *rdfs:subClassOf* permet de définir des hiérarchies de classes. Par exemple, 'un cours de DEA est un cours':

```
<rdfs:Class rdf:ID='CoursDEA'>
        <rdfs:subClassOf rdf:resource='#Cours'/>
</rdfs:Class>
```

- RDFS précise la notion de propriété définie par RDF en permettant de donner un type ou une classe au sujet et à l'objet des triplets. Pour cela, RDFS ajoute les notions de 'domain', correspondant au domaine de définition d'une fonction en anglais, et 'range', son ensemble d'arrivée :
- *rdfs:domain* définit la classe des sujets liée à une propriété.

- *rdfs:range* définit la classe ou le type de données des valeurs d'une propriété.

Nous pouvons, par exemple, exprimer en RDFS que l'enseignant d'un cours est une personne et que le nom d'une personne est un littéral (une chaîne de caractères) :

```
<rdf:Property rdf:ID='enseignant '>
        <rdfs:domain rdf:resource='#Cours'/>
        <rdfs:range rdf:resource='#Personne'/>
</rdf:Property>

<rdf:Property rdf:ID='nom'>
        <rdfs:domain rdf:resource='#Personne'/>
        <rdfs:range rdf:resource= '&rdfs;Literal'/>
</rdf:Property>
```

Nous pouvons, par exemple, exprimer qu'une propriété 'enseignant' relie 'MCSI' (de type *Cours*) à 'Lamia Yessad' (de type *Personne*) par les triplets RDF :

```
<Cours rdf:about=' http://www.esi.dz/.../MCSI'>
        <enseignant>
          <Personne rdf:about=' http://www.esi.dz/.../enseignants/LamiaYessad'>
                <nom>Lamia Yessad</nom>
          </Personne>
        </enseignant>
</Cours>
```

Pour résumer, XML peut être vu comme la couche de transport syntaxique, RDF comme un langage relationnel de base et RDFS offre des primitives de représentation de structures ou des primitives ontologiques.

3.3. OWL (Web Ontology Language)

OWL est un langage basé sur RDF. Il enrichit le modèle RDF Schema en définissant un vocabulaire riche pour la description d'ontologies complexes. OWL est basé sur une sémantique formelle définie par une syntaxe rigoureuse. Il existe trois versions du langage : OWL Lite, OWL DL, et OWL Full.

OWL et RDFS sont tous deux des vocabulaires RDF permettant de définir des vocabulaires. RDFS définit le plus petit nombre de notions et de propriétés nécessaires à la définition d'un vocabulaire simple, essentiellement :

- les notions de classe, de ressource et de littéral
- les propriétés de sous-classe, de sous-propriété, de champ de valeur, de domaine d'application

OWL est un langage beaucoup plus riche qui, aux notions définies par RDF Schema, ajoute les propriétés de classe équivalente, de propriété équivalente, d'identité de deux ressources, de différences de deux ressources, de contraire, de symétrie, de transitivité, de cardinalité, etc., permettant de définir des rapports complexes entre des ressources.

3.4. Positionnement de notre approche

OWL est le langage le plus riche et le plus complet dans le domaine de la construction d’ontologies. C’est ce langage que nous utilisons et surtout dans sa version OWL-DL afin de pouvoir exploiter toute la puissance du raisonnement logique.

4. Raisonnement logique

4.1. Logique de description (LD)

Traditionnellement, un système LD comporte deux composants : (1) le premier est la base de connaissances (BC) qui est encore divisée en deux blocs appelés TBox et ABox, et (2) le second est le moteur d'inférence qui implémente les services d'inférence. Un TBox stocke les connaissances conceptuelles (terminologiques) du domaine d'application tandis qu'un ABox présente les connaissances assertionnelles. Ce formalisme est fondé sur trois types d'entités :

- Les *concepts* (ou *classes*) qui représentent des classes d'individus ayant des propriétés communes. Ces individus sont l'*extension* du concept ;

Exemple : le concept Component représente un ensemble de composants logiciels, cet ensemble de composants est l'extension du concept Component.

- Les individus (ou instances des concepts) ;

Exemple : les individus VideoCamera et VideoWindow sont des instances du concept Component ;

- Les rôles (ou propriétés) qui représentent des relations binaires entre les concepts.

Exemple : le rôle hasCharacteristic entre le concept Component et le concept Characteristic désigne la relation entre instances de ces deux concepts ;

Chaque rôle possède un *domaine* et un *co-domaine*.

- Le domaine est le concept où est définit le rôle (concept de départ).
- Le co-domaine est le concept avec lequel le rôle établit une relation (concept d'arrivée).

Exemple : la relation hasCharacteristic qui indique qu'un composant logiciel (instance du concept Component) possède une caractéristique (instance du concept Characteristic) a pour domaine le concept Component et pour co-domaine le concept Characteristic.

Les concepts (et les rôles le cas échéant) sont organisés en une hiérarchie par la relation de subsomption, notée $\sqsubseteq$, où $C \sqsubseteq D$ se lit "C subsume D" ou "D est subsumé par C". Un concept C subsume un concept D si et seulement si C est plus général que D. De plus, chaque concept partage les propriétés de ses subsumants dans la hiérarchie de concepts.

Remarque : On nomme Top ou Thing, noté généralement⊤, le concept le plus général. Ce concept est à la racine de la hiérarchie des concepts et subsume tous les autres concepts.

Les définitions de concepts font intervenir les constructeurs suivants :

- La conjonction de concepts, notée ⊓;
- La cardinalité qui fixe le nombre minimal et maximal de valeurs élémentaires que peut prendre un rôle. Elle est notée ≤, ≥, ou = suivant qu'il s'agit d'une cardinalité maximale, minimale ou exacte.

La syntaxe d'une Logique de description est donnée dans le tableau 3.1.

Tableau 3.1 : Syntaxe d'une LD

La syntaxe d'une logique de description			
$C \sqsubseteq A$	Subsomption de concepts	$\forall r.C$	Restriction universelle
$\bot$	Concept absurde	$\exists r.C$	Restriction existentielle
$\top$	Concept universel	$(\geq n\ r)$	Restriction supérieure de cardinalité
$C \sqcap D$	Conjonction de concepts	$(\leq n\ r)$	Restriction inférieure de cardinalité
$C \sqcup D$	Disjonction de concepts	$r \sqsubseteq a$	Subsomption de rôles
$\neg C$	Négation de concept	$r \wedge s$	Composition de rôles
- C et D sont des expressions de concepts, r et s sont des expressions de rôles - A est un concept primitif et a est un rôle primitif - n est un entier non nul			

Les différents types de raisonnement possibles en utilisant la logique de descriptions sont :

- vérification de la consistance : Un raisonneur est capable de s'assurer de la consistance d'une base de connaissance, autrement dit de vérifier qu'il n'y a pas de faits contradictoires dans la base de connaissance, tels qu'une assertion dans l'ABox qui soit contraire aux définitions présentes dans la TBox. Ainsi, si la TBox définit une propriété 'enseignant' comme devant avoir un individu de la classe 'Cours' pour sujet et un individu de la classe 'Personne' pour objet. Une inconsistance apparaît si l'ABox

contient une assertion telle que : enseignant (MCSI, rouge) où rouge n'est pas une instance de la classe Person.

- Satisfiabilité de concept : Il s'agit de déterminer si une classe peut posséder une instance. Une classe insatisfiable entraîne une inconsistance si une instance d'une telle classe est définie. Un exemple de classe insatisfiable est une classe définie comme l'intersection d'une classe *Woman* et d'une classe *Father* étant donné qu'une personne ne peut être à la fois une femme et un père.

- Classification : Elle consiste à établir les liens hiérarchiques entre toutes les classes d'une ontologie, c'est-à-dire identifier les différentes sous-classes de classes. Ceci permet par exemple de fournir toutes les sous-classes d'une classe donnée.

- Réalisation : Elle consiste à établir la classe la plus spécifique d'un individu, ce qui nécessite d'avoir réalisé une classification auparavant. Par exemple, s'il existe une classe *Person* et une classe *Woman* qui est une sous-classe de la classe *Person*, la classe la plus spécifique d'une femme sera la classe *Woman* et non pas la classe *Person* qui est plus générale.

Parmi les raisonneurs existants, citons RacerPro[14], FaCt++[15] et Pellet[16]. Ce dernier, écrit dans le langage Java, est un raisonneur OWL-DL complet, c'est-à-dire qu'il supporte toute l'expressivité du langage OWL-DL. Il a de plus été étendu afin de supporter certaines propriétés apportées par OWL2, entre autres les assertions de négation de propriétés, les propriétés disjointes et le punning qui est une technique permettant à un individu et une classe de partager la même URI.

4.2. Appariement sémantique de Paolucci

Cette section décrit l'algorithme de (Paolucci et al., 2002). Cet algorithme possède en entrée une requête OWL-S (service requis) et boucle sur chaque annonce OWL-S (service offert, disponible dans un répertoire) afin de déterminer leur appariement (matching). Il retourne un ensemble de services offerts matchées et triés selon leurs degrés d'appariement. Un service offert matche avec le service requis si leur entrées et leurs sorties matchent.

[14] http://www.racer-systems.com/products/download/
[15] http://owl.man.ac.uk/factplusplus/
[16] http://clarkparsia.com/pellet/download/

Etant données $Query_{out}$ et $Advt_{out}$ des listes de concepts de sorties relatives, respectivement, au service requis et au service offert. Le matching entre Query et Advt nécessite le matching entre deux listes de concepts, $Query_{out}$ and $Advt_{out}$, comme suit:

$$\forall c \in Query_{out}, \exists d \in Advt_{out}, \\ \text{s.t. } match(c,d) \neq Fail$$

Idem, $Query_{in}$ and $Advt_{in}$ représentent les listes de concepts d'entrées liées, respectivement, au service requis et au service offert. Le matching des entrées est régi par la règle suivante :

$$\forall c \in Advt_{in}, \exists d \in Query_{in}, \\ \text{s.t. } match(c,d) \neq Fail.$$

Nous constatons que l'ordre du service requis et du service offert est interverti dans la seconde expression. Nous supposons que outQ $\in$ $Query_{out}$ et outA $\in$ $Advt_{out}$ sont deux concepts. Dans ce cas, la fonction de matching des sorties (match (outQ, outA)) prend outQ and outA en entrée et retourne le degré de matching entre eux. Ces degrés sont définis comme suit :

- ***Exact***: si le concept outA est équivalent au concept outQ.
- ***Plugin***: si le concept outA subsume le concept outQ.
- ***Subsume***: si le concept outQ subsume le concept outA.
- ***Fail***: si aucune des conditions précédentes n'est satisfaite.

Ces degrés sont classés comme suit: *Exact* > *Plugin* > *Subsume* > *Fail* (x > y indique que x est meilleur que y en terme de matching).
Cependant, quand il s'agit d'apparier les concepts d'entrées, le matching subsume est meilleur que le matching plugin, c'est-à-dire *Exact* > *Subsume* > *Plugin* > *Fail*.

Le classement entre deux concepts permet de déterminer le classement entre le service requis et le service offert. Pour cela, l'algorithme adopte l'approche gloutonne (Greedy approach) afin de matcher deux listes de concepts. Par exemple, dans le cas du matching des sorties, pour chaque concept, on détermine le concept correspondant ayant un maximum degré de matching. Une fois tous les matchings maximaux sont calculés, la solution optimale correspond au degré global le moins élevé entre le service requis et le service offert.

4.3. Discussion

Les degrés d'appariement proposés par (Paolucci et al., 2002) sont définis entre deux concepts d'entrées (ou de sorties) alors qu'un service (demandé ou offert) constitue une liste de concepts d'entrées et de sorties. Il est donc primordial de déterminer la stratégie de sélection d'un service offert correspondant au service demandé : l'approche gloutonne est adoptée dans le cadre des travaux de (Paolucci et al., 2002).

Par analogie à l'appariement de services, nous proposons un processus d'appariement de composants basé sur des critères de qualité de service. Il s'agit d'apparier un composant candidat avec le composant recherché, chacun d'eux décrit par une liste de concepts de qualité offerte et requise. Ces concepts appartiennent au même formalisme à savoir l'ontologie QoSOntoCS qui sera présentée dans le chapitre 5. L'appariement que nous proposons utilise donc les degrés d'appariement de (Paolucci et al., 2002) pour définir une nouvelle stratégie de sélection : l'approche gloutonne n'étant pas adapté à la recherche multi-critères.

De plus, dans (Paolucci et al., 2002), on ne tient pas compte de la structure extensionnelle des concepts d'entrées/sorties. Notre mesure exploite les propriétés data-types des concepts de qualité à savoir : les métriques de qualité et leur direction (voir chapitre 6).

5. Conclusion

Dans ce chapitre nous avons présenté le Web sémantique en explicitant ses objectifs et ses langages. En effet, Les spécifications RDF, RDFS et OWL combinées forment une partie de la base du Web Sémantique. Celui-ci est basé sur les concepts suivants :

- un schéma de nommage global (les URI ou IRI) ;
- une syntaxe standard pour décrire une information (RDF) ;
- un moyen standard de décrire les propriétés de cette information (RDF Schema) ;
- un moyen standard de décrire les relations entre informations (OWL) ;
- un moyen de créer un réseau de confiance sécurisé à propos de ces informations.

Nous avons également montré l'intérêt que pouvaient avoir les ontologies, une technologie du Web sémantique, dans la recherche et l'évaluation de composants et particulièrement l'appariement de ces derniers. En s'inspirant de la technique présentée dans (Paolucci et al.), nous proposons une nouvelle mesure de similarité (score de pertinence) afin de répondre aux besoins de la sélection.

Partie II

Contribution

Chapitre 4

Proposition de l'architecture QoS-CSA

La sélection de composants est un processus qui inclut plusieurs étapes qui vont de la spécification de besoins (déterminer les critères de sélection) à l'analyse des résultats d'évaluation (déterminer la performance du résultat). Ces étapes nécessitent des connaissances expertes. Ainsi, pour faciliter la mise en œuvre de ces étapes, nous avons élaboré une architecture de sélection de composants logiciels baptisé QoS-CSA (pour **QoS-**based **C**omponent **S**election **A**rchitecture en anglais). L'analyse et la conception d'un tel système de sélection sont présentées dans la suite de ce chapitre.

1. Cadre de développement : Processus AUP

Dans le cadre de ce travail de thèse, nous avons adopté, comme démarche de développement, le processus unifié agile (AUP[17]) qui s'étend de la phase inception (comprendre le périmètre du projet), en passant par les phases d'élaboration et de construction, à la phase de transition (préparer le lancement du produit). Ce chapitre est consacré aux deux premières phases (inception et élaboration).

La phase d'inception se conclut par le jalon LCO (Life Cycle Objective) et comprend les activités (ou tâches) suivantes :

- Comprendre le périmètre du système ;
- Décrire les cas d'utilisation.

Quant à la phase d'élaboration, elle se conclut par le jalon LCA (Life Cycle Architecture) et comprend les activités suivantes :

[17] http://www.ambysoft.com/unifiedprocess/agileUP.html

- Analyser les principaux cas d'utilisation ;
- Construire une architecture globale;
- Réduire les risques techniques.

Dans ce qui suit, nous commençons par la phase d'inception qui énonce le problème et les cas d'utilisation.

2. Inception du système

2.1. Problématique de recherche

L'intérêt principal de notre projet de recherche est de proposer une architecture d'un système qui permet de construire efficacement des applications à base de composants réutilisables. Ces derniers sont recherchés à partir d'une base descriptive de composants et sont jugés comme pertinents et répondant au mieux aux besoins des utilisateurs. Ce système est destiné à être utilisé par des développeurs pendant la phase de conception et leur permet, en particulier, de mettre en œuvre le processus de sélection de composants logiciels en prenant en compte les aspects de qualité de service (QoS) de ces composants. Nous mettons donc l'accent sur la qualité de service, un sous-ensemble de la qualité logicielle, qui décrit les propriétés observables et mesurables des composants. Les caractéristiques de la QoS appartiennent à deux catégories : (1) les caractéristiques statiques qui sont stables pendant le développement d'une application et (2) les caractéristiques dynamiques qui peuvent changer significativement au moment de son exécution. Tandis que la première catégorie regroupe les caractéristiques liées aux composants eux-mêmes ou à leur configuration, la seconde regroupe les caractéristiques liées au fonctionnement des composants.

Nous travaillons sur une base locale où les composants sont décrits par des connaissances fournies lors du cycle « Design for reuse » par leurs développeurs. Ces derniers deviennent les indexeurs du cycle de développement par réutilisation (Design by reuse). Leur tâche consiste à annoter leurs composants en conformité avec les modèles exigés. En premier lieu, le modèle fonctionnel externe est la description des aspects externes d'un composant tels que le nom du composant et les noms des différentes interfaces de ce composant, etc. En second lieu, le modèle de QoS est la description des caractéristiques de QoS des différents artefacts : le composant et ses interfaces (Yessad and Boufaida, 2010). De ce fait, la sélection qui se base sur la description externe doit d'abord être effectuée avant de passer à la sélection sur des

critères de QoS. Pour la sélection préliminaire, nous pouvons choisir n'importe quelle technique de recherche comme par exemple la classification externe par mots clés ou par facettes (voir chapitre 2). Ensuite, notre approche de sélection intervient pour sélectionner des composants pertinents en se basant sur des critères de QoS appartenant aux deux catégories précédentes. Une condition indispensable à la réussite du processus de sélection des composants est que les composants doivent être indexés conformément à l'ontologie QoSOntoCS (pour **QoS Onto**logy for **C**omponent **S**election en anglais) que nous proposons et que nous détaillons dans le chapitre suivant.

L'architecture que nous proposons dans ce chapitre répond au fait que dans le domaine de la sélection de composants logiciels, la majorité des travaux s'intéressent à la sélection en se basant uniquement sur les fonctionnalités spécifiées de manière syntaxique et rares sont les travaux qui présentent une démarche de sélection multi-niveaux[18] (George, 2007) et sémantique (Masmoudi et al., 2008). C'est ce dernier point qui constitue l'intérêt majeur de notre travail.

Dans ce qui suit, nous donnons une vue d'ensemble des besoins auxquels nous nous attaquons :

Besoin 1 : Prise en compte des besoins (ou exigences) de la QoS

D'après les interviews que nous avons menées auprès de développeurs, il est apparu que les critères de QoS sont décisifs dans le processus de sélection de composants. Dans le cadre de cette thèse, notre objectif est de proposer un environnement pour assister les développeurs à construire des applications par assemblage de composants logiciels. Nous nous intéressons uniquement aux caractéristiques de QoS des composants et supposons que les composants satisfont, à priori, les caractéristiques fonctionnelles.

Besoin 2 : Plus de sémantique dans la spécification de la QoS

La sélection automatique basée QoS est une tâche extrêmement difficile à cause de l'hétérogénéité des spécifications de la QoS. Nous proposons une description standard et formelle de la QoS afin d'améliorer l'automatisation de la recherche et l'évaluation de composants logiciels. Nous avons choisi d'utiliser des ontologies pour spécifier sémantiquement la QoS. Les ontologies permettent à un indexeur de composants logiciels

[18] Niveaux syntaxique, comportemental, de synchronisation et de QoS

d'utiliser un modèle commun et formel pour décrire la QoS de ses composants. Le succès du processus de spécification de la QoS dépend d'un autre acteur (ingénieur de connaissances) qui doit intervenir pour assister l'indexeur dans sa tâche. Le résultat de ce processus est un ensemble d'annotations sémantiques de la QoS conformes aux concepts, propriétés et axiomes de l'ontologie que nous proposons (voir chapitre 5).

Besoins 3 : Un processus de sélection efficace et fiable

Le but ultime du développement par réutilisation est de rendre les processus logiciels plus efficaces. Notre système permet d'augmenter l'efficacité de ces processus en évitant de sélectionner le même composant autant de fois qu'on l'utilise. De plus, quand de nouveaux besoins de QoS apparaissent le système offre la possibilité au sélectionneur d'insérer ces besoins sans pour autant refaire le processus de sélection à zéro.

La fiabilité du processus de sélection participe aussi à rendre le processus de développement plus efficace. La sélection est effectuée d'abord dans la base de connaissances locale par l'exploitation des requêtes capitalisées. S'il y a un silence de requête, cette dernière est relaxée. L'idée de la relaxation est que le sélectionneur sacrifie une ou plusieurs besoins de la QoS pour éviter les silences même s'il obtient des résultats moins pertinents.

2.2. Cas d'utilisation

Dans notre système de sélection de composants, différents acteurs peuvent intervenir : (1) Des sélectionneurs pour lancer les processus de sélection basée QoS, (2) des indexeurs pour annoter les composants logiciels disponibles et (3) des ingénieurs de connaissances qui maintiennent la base de connaissances et collaborent avec les indexeurs et les sélectionneurs afin qu'ils mènent à bien leurs tâches respectives.

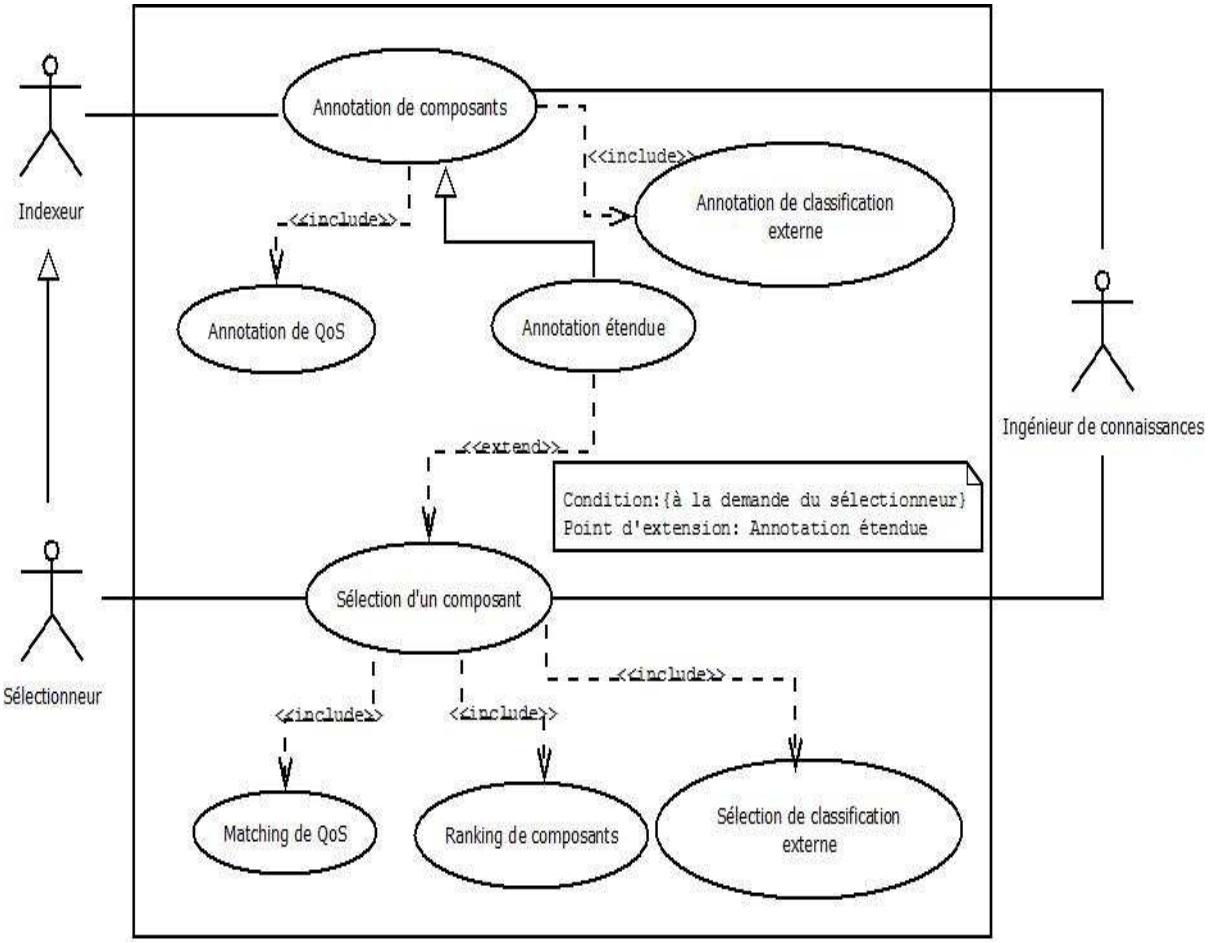


Figure 4.1 : Diagramme de cas d'utilisation du système

Description des principaux cas d'utilisation

A)

Description du cas « Annotation de composants »
Pré-conditions L'indexeur s'authentifie avec succès. ***Enchaînement nominal*** 1. L'indexeur accède au formulaire 2. L'indexeur remplit le formulaire. 3. L'indexeur clique sur le bouton «suivant». 4. L'indexeur reprend au point 2. 5. Appel du cas « Annotation de classification externe » 6. Appel du cas « Annotation de QoS » ***Enchaînements alternatifs*** **A1 : L'indexeur n'a plus de composants.** L'enchaînement démarre après le point 2 de la séquence nominale : 3. L'indexeur clique sur le bouton « terminer ». La séquence nominale reprend au point 5.

Enchaînements d'exception
E1 : L'indexeur ne remplit aucun champ obligatoire.
L'enchaînement démarre après le point 3 de la séquence nominale :
4. Le système affiche un message de notification.

Post-conditions
La base de connaissances est augmentée par les informations contenues dans les annotations externe et de QoS.

Rubriques optionnelles
Contraintes
L'annotation de composants s'effectue à n'importe quel moment.

B)

Description du cas « Sélection d'un composant »

Pré-conditions
Le sélectionneur s'authentifie avec succès.

Enchaînement nominal
1. Le sélectionneur accède à un formulaire.
2. Le sélectionneur spécifie sa requête (composant recherché) y compris ses préférences.
3. Le sélectionneur clique sur le bouton « sélectionner».
4. Appel du cas « Sélection de classification externe».
5. Le système affiche le résultat et demande au sélectionneur s'il veut relaxer sa requête.
6. Le sélectionneur clique sur le bouton « non ».
7. Appel du cas « Matching de QoS ».
8. Le système affiche le résultat et demande au sélectionneur s'il veut relaxer sa requête.
9. Le sélectionneur clique sur le bouton « non ».
10. Appel du cas « Ranking de composants ».
11. Le système affiche l'annotation du meilleur composant.
12. Le système demande au sélectionneur s'il veut mettre à jour l'annotation du composant sélectionné.
13. Le sélectionneur clique sur « oui ».
14. Appel du cas « Annotation étendue ».

Enchaînements alternatifs
A1 : Le sélectionneur veut relaxer sa requête après la « Sélection de classification externe ».
L'enchaînement démarre après le point 5 de la séquence nominale :
6. Le sélectionneur clique sur le bouton « oui ».
La séquence nominale reprend au point 2.
A2 : Le sélectionneur veut relaxer sa requête après le « Matching de QoS ».
L'enchaînement démarre après le point 8 de la séquence nominale :
9. Le sélectionneur clique sur le bouton « oui ».
La séquence nominale reprend au point 2.
A3 : Le sélectionneur n'est pas satisfait du résultat final.
L'enchaînement démarre après le point 12 de la séquence nominale :
13. Le sélectionneur clique sur le bouton « non ».
14. Le système affiche un message de notification.

Enchaînements d'exception

E1 : Le sélectionneur ne spécifie pas le composant recherché.
L'enchaînement démarre après le point 3 de la séquence nominale :
4. Le système affiche un message de notification.

Post-conditions
L'annotation du composant sélectionné peut être étendue.

Rubriques optionnelles
Contraintes
Les processus « Sélection de classification externe», « Matching de QoS » et « Ranking de QoS » sont séquentiels et doivent être exécutés dans cet ordre.

3. Elaboration du système

3.1. Réduction des risques

Le processus AUP est itératif, c'est-à-dire, permet d'intégrer incrémentalement les fonctionnalités du système. En effet, nous pouvons produire d'abord une architecture de référence en élaborant les fonctionnalités de base et ensuite l'augmenter par d'autres composants afin de réaliser les fonctionnalités restantes. Ainsi, le feedback des futurs utilisateurs du logiciel sur les artefacts produits au fur et à mesure du développement est un moyen efficace de réduction des risques.

Un autre moyen de réduire les risques est de déterminer les outils nécessaires à la phase de construction de notre système. Nous avons particulièrement besoin d'une API qui nous permet d'accéder à la base de connaissances à partir d'un programme JAVA. L'API Jena dispose de méthodes pour lire et écrire du RDF comme du XML.

3.2. Analyse des principaux cas d'utilisation

Un diagramme de séquence permet de détailler les différentes interactions entre les différentes parties du système (acteurs, processus, stockages) dans un ordre chronologique.
Dans ce qui suit, nous élaborons un scénario pour chacun des cas d'utilisation décrits dans la phase inception.

3.2.1 Annotation de composants

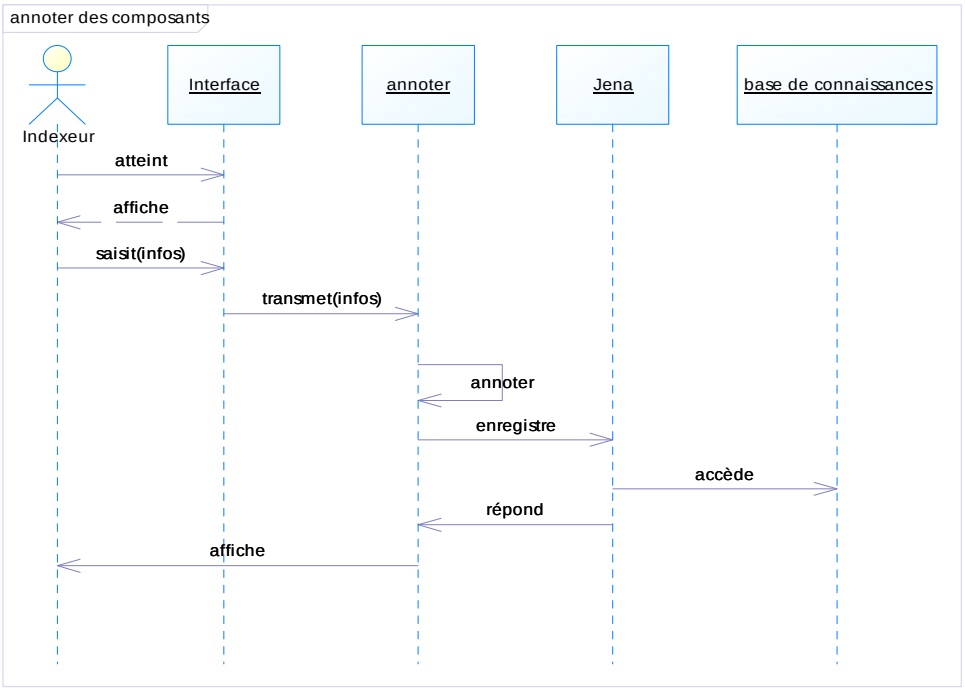


Figure 4.2 : Diagramme de séquence « Annotation de composants »

3.2.2 Sélection d'un composant

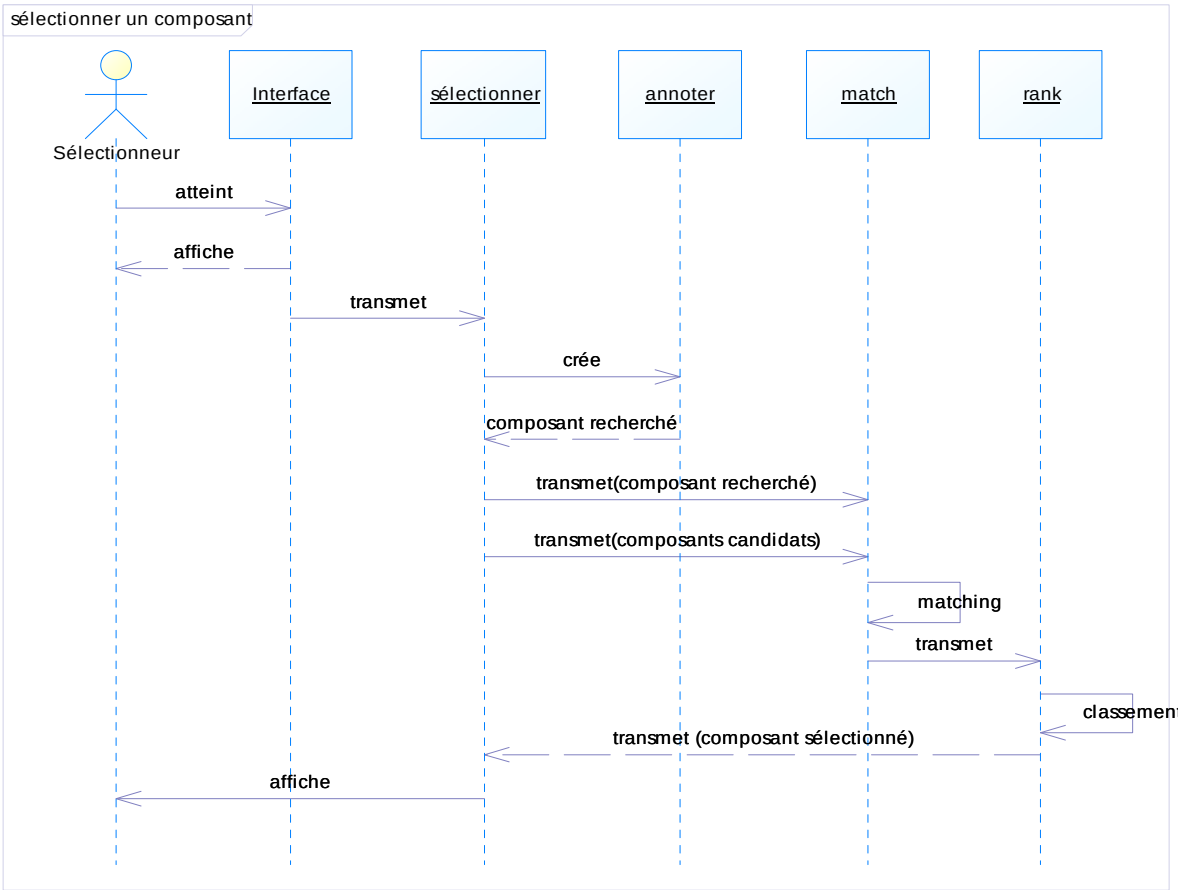


Figure 4.3 : Diagramme de séquence « Sélection d'un composant»

3.3. Description de l'architecture globale

La sélection de composants est un processus qui inclut plusieurs étapes qui vont de l'identification des critères d'évaluation (les besoins de QoS) à l'analyse de l'évaluation de composants logiciels. Ces étapes nécessitent des choix de mise en œuvre et des connaissances expertes. Ainsi, pour structurer ces étapes, nous proposons l'architecture de sélection de composants logiciels QoS-CSA dont les différents modules sont schématisés ci-dessous (voir la figure 4.3).

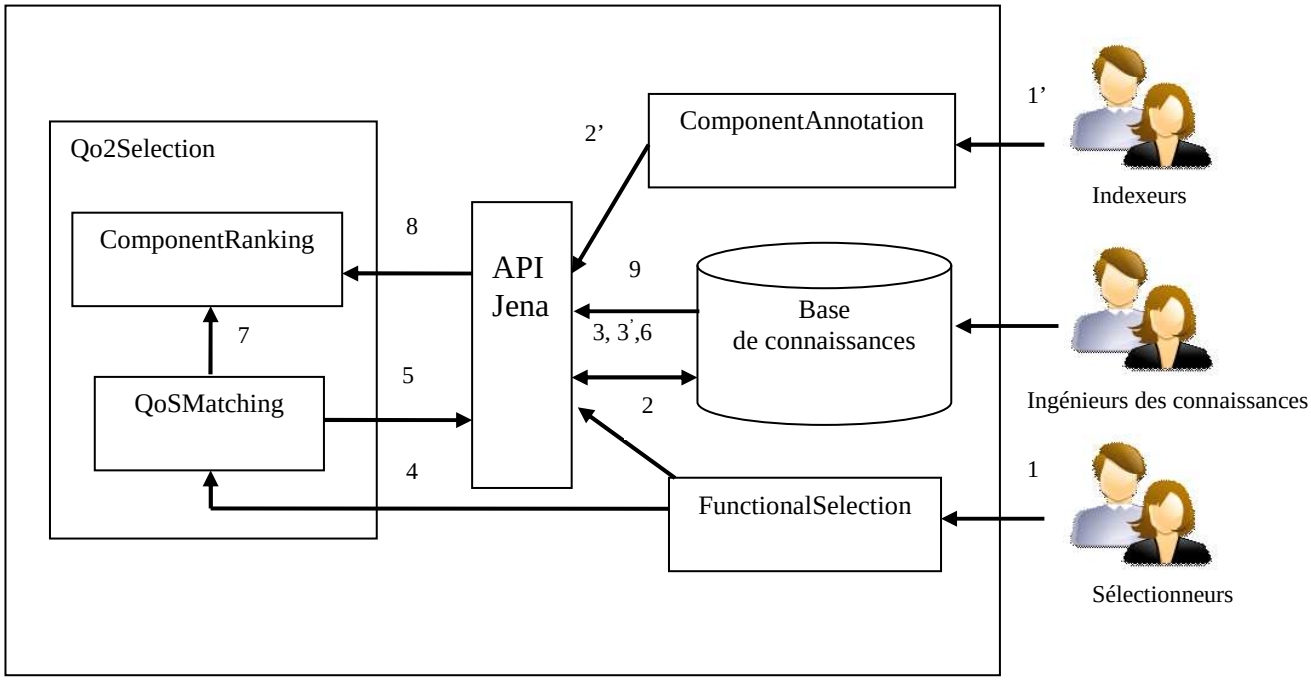


Figure 4.4 : Architecture QoS-CSA

3.3.1 Architecture QoS-CSA

Cette architecture comprend plusieurs modules où chacun a un rôle précis :

- ***ComponentAnnotation***

Ce module permet de peupler l'ontologie QoSOntoCS en annotant les composants disponibles. Il permet l'annotation de plusieurs composants à la fois. Il peut s'exécuter en parallèle du processus de sélection de composants. On parle dans ce cas d'un processus d'annotation en « arrière-plan ».

- ***FunctionalSelection***

Ce module permet la sélection en se basant sur des caractéristiques fonctionnelles. Nous considérons que les composants sont des boites noires. Dans cette optique, un composant est décrit par un ensemble d'interfaces fournies et/ou requises. Cette sélection précède la sélection basée QoS et produit, de ce fait, une liste de composants candidats avec des scores d'appariement fonctionnel.

- ***Qo2Selection***

Ce module effectue la sélection en se basant sur la qualité de service spécifiée par l'ontologie QoSOntoCS. Sa tâche consiste à rechercher puis à évaluer les composants selon un ensemble de critères de QoS spécifiés par le développeur. Pour cela, deux sous-composants sont nécessaires : QoSMatching et ComponentRanking.

- ***QoSMatching***

C'est un sous-composant de Qo2Selection. Son rôle est d'obtenir automatiquement un vecteur de scores pour chaque composant candidat. Chaque score correspond à un critère de qualité et est obtenu en utilisant la formule que nous avons proposée dans le dernier chapitre.

- ***ComponentRanking***

C'est un sous-composant de Qo2Selection. Il utilise les résultats obtenus par le sous-composant QoSMatching afin de classer les différents composants candidats. Il utilise pour cela une des techniques de prise de décision multi-critères en se basant sur les scores obtenus lors du matching et sur les degrés de préférence formulés par le sélectionneur.

3.3.2 Fonctionnement de QoS-CSA

Dans QoS-CSA, les processus de sélection et d'annotation peuvent se dérouler en parallèle. Le premier processus est initié par le sélectionneur (1) alors que le second est initié par l'indexeur (1'). Dans les deux cas, les processus accèdent à la base de connaissances via l'API Jena (3, 3'). Le processus de sélection produit d'abord une liste restreinte de composants candidats à partir des composants disponibles, et en se basant uniquement sur les critères fonctionnels (1→2→3). Il continue ensuite à évaluer les composants en prenant en compte, cette fois-ci, leurs critères de QoS (4→5→6). La sélection est donc effectuée par « Filtrage progressif » (Mohamed et al., 2007). Après l'étape d'évaluation, les composants filtrés sont

analysés par le processus de classification (ranking) de composants (7→8→). Notons que chaque processus nécessite l'accès, en lecture, à la base de connaissances.

Cette architecture stipule que la sélection basée QoS doit être précédée par une sélection fonctionnelle. Cette dernière n'est pas traitée dans le cadre de ce travail de thèse mais son rôle est important : elle permet de réduire l'ensemble de composants logiciels sur lequel nous allons travailler aux seuls composants fonctionnellement pertinents. La performance du résultat dépend donc, à la fois, des critères fonctionnels et des critères de QoS.

4. Aspect statique et Construction de l'application

La partie conceptuelle de la phase élaboration doit inclure la modélisation de la partie statique (données) de notre système. Dans un processus AUP, on utilise le diagramme de classes pour décrire le modèle des données. Certes, nous pouvons décrire notre base de connaissances à l'aide d'un diagramme de classes mais nous omettons cette partie car nous allons le faire dans le chapitre suivant en adoptant une démarche spécifique aux ontologies. Nous définissons donc une ontologie qui permet d'implémenter notre base de connaissance. Une fois la phase élaboration est terminée, nous passons à la phase construction. Dans cette dernière, nous montrons comment implémenter notre système (base de connaissances et algorithmes proposés). La phase construction doit indiquer les outils avec lesquels nous travaillons dans le cadre de ce projet. La phase construction commence au milieu du prochain chapitre et s'étend jusqu'à la fin où des tests sont conduits dans le but de déterminer la performance de notre application.

5. Conclusion

Notre méthode de sélection comporte principalement les étapes suivantes :

- Identification des besoins de QoS ;
- Recherche de composants en se basant sur des critères fonctionnels ;
- Evaluation de composants en prenant en compte des critères de QoS ;
- Analyse des résultats d'évaluation de composants.

Nous avons choisi d'abord de représenter les caractéristiques de QoS par une ontologie. Ensuite, nous utilisons la technique de « filtrage progressif » pour rechercher les composants qui répondent d'abord aux besoins fonctionnels puis à évaluer les composants recherchés en se basant sur des critères de QoS. En effet, la recherche de composants doit précéder

l’évaluation basée sur notre ontologie. Enfin, les résultats d’évaluation sont analysés en appliquant la technique MCDA.

Lors de ce chapitre, nous avons proposé notre architecture QoS-CSA en décrivant tous les modules nécessaires pour la mise en œuvre des fonctionnalités de recherche (Recherche fonctionnelle), d’évaluation (Matching de QoS) et d’analyse de cette évaluation (Ranking de QoS). L’implémentation des deux dernières fonctions est détaillée dans le dernier chapitre alors que le prochain chapitre se focalise sur comment représenter les connaissances utilisées dans QoS-CSA et particulièrement les caractéristiques de QoS.

Chapitre 5

Représentation de la qualité de service dans QoS-CSA

Nous avons élaboré une architecture de référence (QoS-CSA) pour les systèmes de sélection de composants. Notre système prend en charge les aspects de qualité de service des composants logiciels. D'autres itérations peuvent s'avérer nécessaires afin d'affiner ou d'ajouter d'autres fonctionnalités. Par ailleurs, quelque soit la taille de l'application, son noyau comprend toujours une base de connaissances. Cette base représente les données du système qui, en l'occurrence, sont des connaissances sur les composants logiciels et des connaissances sur la sélection de ces composants.

1. Modélisation des connaissances dans QoS-CSA

Comme nous l'avons déjà étudié, l'architecture QoS-CSA répond au besoin de réutilisation de composants logiciels par le moyen d'un processus de sélection. Nous rappelons que ce processus consiste à rechercher puis à évaluer les composants logiciels fonctionnellement pertinents afin de choisir ceux dont la QoS est la plus « proche » à celle du composant recherché (requête du développeur). Dans notre architecture, chaque composant logiciel candidat est annoté en se basant sur les concepts d'une ontologie. Ce choix permet de :

- Lever les ambiguïtés sémantiques entre les modèles descriptifs de composants (le concept *ExternalContract*) ;
- Décrire les caractéristiques de QoS des composants réutilisables (*QoSContract*) ;
- Améliorer la réutilisation des connaissances sur la sélection de composants (*ExtendedContract*).

Pour répondre à ces besoins, nous avons choisi de décrire les connaissances du système avec deux modèles selon deux niveaux d'abstraction différents: Le modèle générique et le modèle spécifique au domaine (MSD). Le premier modèle correspond à une ontologie qui décrit les

concepts et les propriétés génériques liés aux composants, leurs caractéristiques de QoS et éventuellement les résultats de leur sélection. Le second modèle est l'ontologie de QoS qui consiste en la hiérarchisation des caractéristiques de QoS dans le domaine de la réutilisation de composants logiciels. Il est conforme au standard ISO 9126.

Une fois le composant logiciel sélectionné, l'ontologie proposée permet de garder le résultat de la sélection : il s'agit de mémoriser la requête déclenchant cette sélection ainsi que les scores de pertinence du composant sélectionné.

2. Manipulation de la base de connaissances

Dans l'architecture QoS-CSA, différents acteurs interagissent avec la base de connaissances en jouant deux rôles possibles:

Producteur : ce rôle est joué par l'expert qui produit la connaissance. Il spécifie de manière semi-formelle (structurée) les concepts sur les composants, leurs caractéristiques de QoS et/ou leurs résultats de sélection satisfaisant certaines requêtes (ou critères). Plusieurs personnes peuvent jouer le rôle de producteur : L'indexeur, le sélectionneur et l'ingénieur de connaissances.

Consommateur : ce rôle est joué par les sélectionneurs. Ces derniers jouent un double rôle : (1) ils utilisent la base de connaissances pour extraire les informations nécessaires au processus de sélection et (2) ils transmettent les connaissances sur le résultat de la sélection en vue de son stockage dans la base de connaissances.

Dans notre travail, les indexeurs et les sélectionneurs sont contraints d'utiliser exclusivement le vocabulaire spécifié dans les ontologies. Cette contrainte assure l'utilisation d'un vocabulaire commun entre les différents acteurs/modules du système et réduit le risque d'hétérogénéité sémantique. Enfin, le rôle de l'ingénieur de connaissances est donc complémentaire : il doit vérifier la cohérence des données introduites par les indexeurs et les sélectionneurs avec les ontologies.

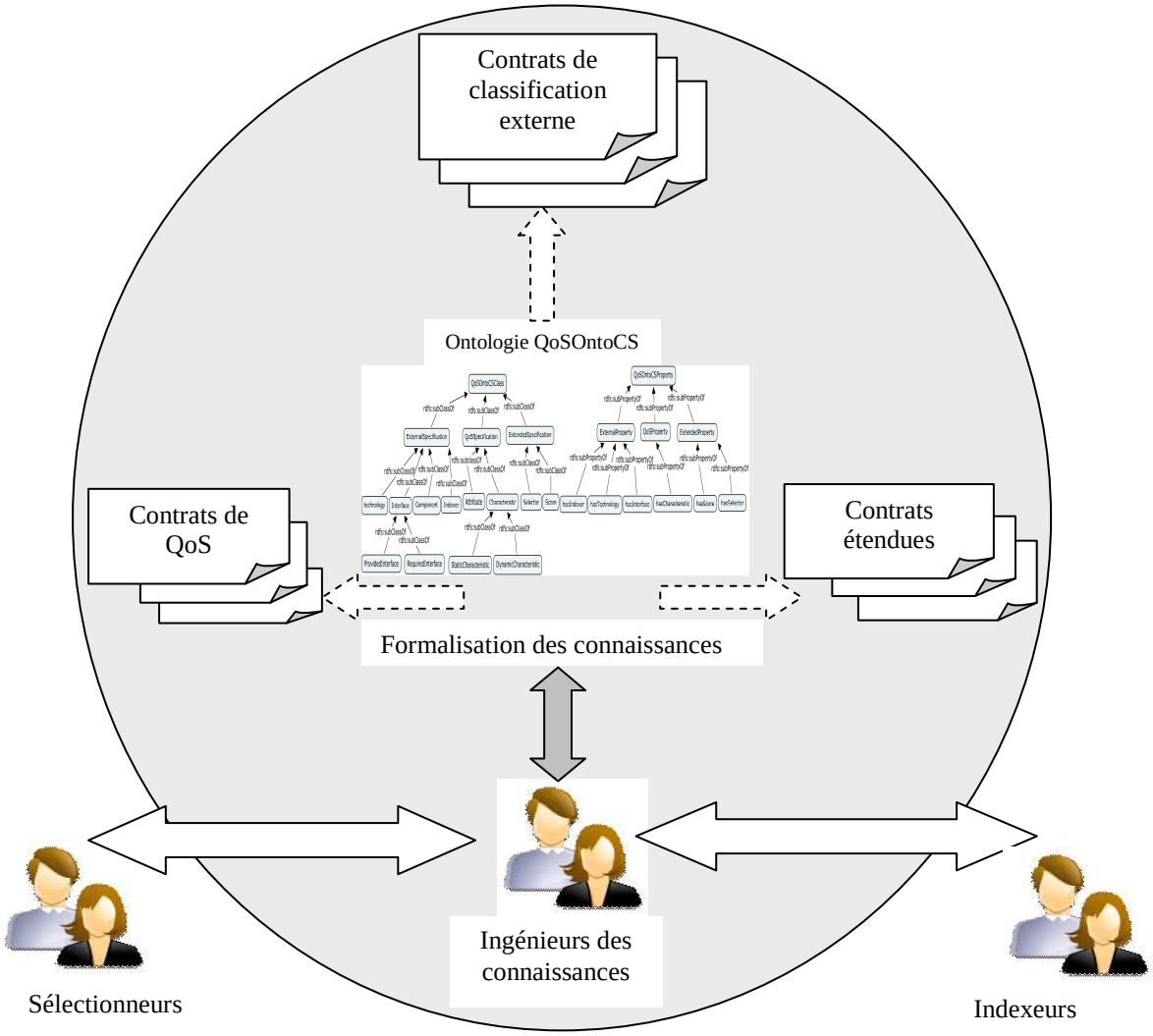


Figure 5.1 : Acteurs et organisation des connaissances dans QoS-CSA

3. Ontologie QoSOntoCS

Une ontologie comprend deux hiérarchies d'entités et d'éventuels axiomes. Ces entités sont les concepts du domaine à formaliser et les propriétés sémantiques qui existent entre les instances de ces concepts. Pour les besoins de notre recherche, nous avons défini une ontologie baptisée QoSOntoCS (pour QoS Ontology for Component Selection, en anglais). Le rôle de cette ontologie est de fournir une description formelle et standardisée pour spécifier la QoS des composants logiciels (Yessad and Boufaida, 2011). De plus, l'ontologie QoSOntoCS permet de formaliser les connaissances sur la sélection de composants logiciels : un sélectionneur peut enregistrer, s'il est satisfait, les résultats d'une sélection antérieure afin de les réutiliser dans de nouvelles sélections.

3.1. Définition informelle

La figure 5.2 représente l'ontologie QoSOntoCS qui a été construite dans le cadre du projet QoS-CSA.

L'ontologie QoSOntoCS permet de formaliser des connaissances sur la sélection de composants logiciels basée sur la QoS. Elle décrit deux types de connaissances: Des concepts organisés hiérarchiquement et des propriétés organisées hiérarchiquement. Pour chaque propriété, on définit un concept de départ (le domaine de la propriété) et un concept d'arrivé (le co-domaine de la propriété).

Les concepts de l'ontologie QoSOntoCS sont organisés hiérarchiquement avec un concept racine et abstrait appelé *QoSOntoCSClass*. Ce dernier possède trois sous-concepts (voir la figure 5.2): *ExternalContract*, *QoSContract* and *ExtendedContract*. Dans cette thèse, nous focalisons sur le concept *QoSContract* généralisant les deux concepts *Characteristic* et *Attribute*. Le concept *Characteristic* est spécialisé en deux sous-concepts : *StaticCharacteristic* et *DynamicCharacteristic*. Tous ces concepts généralisent des caractéristiques de QoS spécifiques au domaine.

De la même manière, les propriétés sont organisées hiérarchiquement avec une propriété racine abstraite appelée *QoSOntoCSProperty*. Cette dernière est aussi spécialisée en trois sous-propriétés : *ExternalProperty*, *QoSProperty*, *ExtendedProperty*.

Les connaissances de QoSOntoCS servent à décrire les aspects externes des composants (par exemple, *Component* ou *Interface*), les aspects de QoS indépendants de tout domaine (par exemple, *Characteristic* ou *Attribute*), les aspects de QoS relatifs aux composants logiciels (conformément au modèle CQM détaillé dans l'annexe A) et les résultats des sélections déjà effectuées (par exemple, *Selector ou Score*).

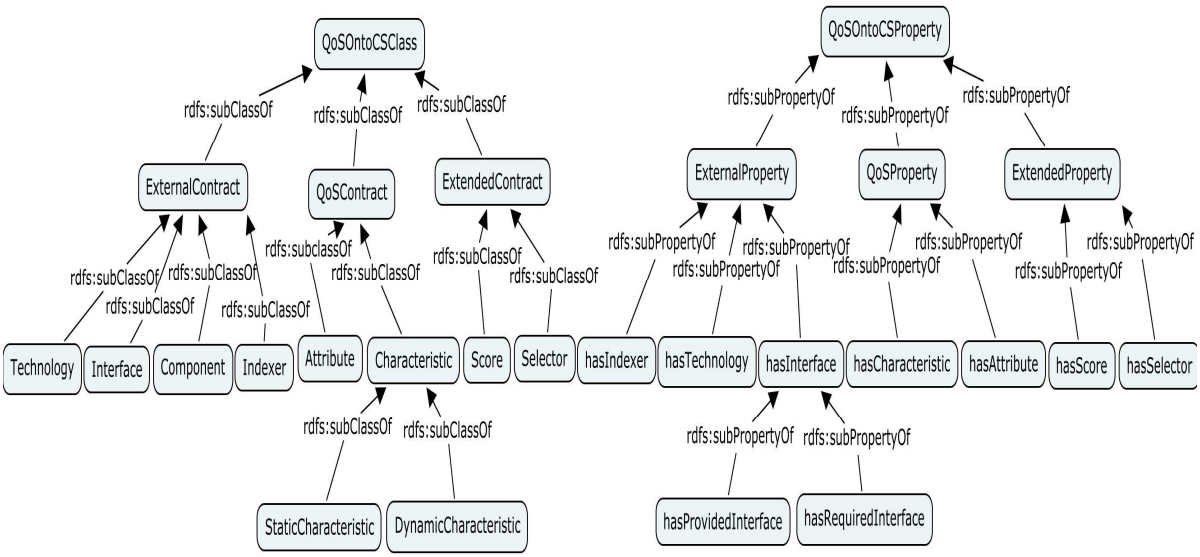


Figure 5.2 : Un extrait de l'ontologie QoSOntoCS (Modèle générique)

Nous notons que la propriété reliant deux concepts est orientée du concept « domaine » vers le concept « co-domaine ». Par exemple la propriété *hasAttribute* relie le concept *Component* (domaine de la propriété) au concept *Attribute* (le co-domaine de la propriété) et l'inverse n'est pas vrai.

3.2. Définition formelle

Formellement, nous définissons l'ontologie QoSOntoCS comme suit :
QoSOntoCS = (C, P, H^c, H^p, Signature, Rules), où:

- C et P sont deux ensembles disjoints. Les éléments de C sont appelés concepts et les éléments de P sont appelés propriétés. Chacun des éléments de ces deux ensembles est identifié par une IRI (International Resource Identifier).
- H^c est la hiérarchie qui relie les différents concepts. Elle représente un graphe acyclique et orienté (un arbre) avec $H^c \subseteq C \times C$. L'écriture $H^c(c_1, c_2)$ signifie que c_1 est un sous-concept de c_2.
- H^p est la hiérarchie qui relie les différentes propriétés. Elle représente un graphe acyclique et orienté (un arbre) avec $H^p \subseteq PxP$. $H^p(p_1,p_2)$ signifie que p_1 est une sous-propriété de p_2.
- Signature: $P \rightarrow C \times C$ est une fonction qui définit une propriété p reliant deux concepts en explicitant le concept de départ (le domaine) et le concept d'arrivée (le co-

domaine). La fonction domain: P→C avec domain(p)=∏1(Signature(p)) renvoie le domaine de p et la fonction range(P)=∏2(Signature(P)) renvoie son co-domaine.

- Rules est un ensemble de règles sur C et P.

L'ontologie QoSOntoCS décrit la QoS liée aux composants logiciels. Les caractéristiques du modèle CQM sont reprises afin de pouvoir compléter cette ontologie. Nous ne pouvons pas faire abstraction de ces caractéristiques dans un domaine où la réutilisation est primordiale: *Reusability* est un sous-concept de *Portability* ou encore *Adaptability* est un sous-concept de *Portability*. Cette ontologie sert aussi à annoter les composants logiciels en construisant des annotations sémantiques à partir des instances qui se trouvent dans la base de connaissances.

Contrairement au modèle générique, le MSD décrit des concepts spécifiques à la qualité de service des composants logiciels. Par exemple, nous retrouvons dans l'ontologie générique le concept *StaticCharacteristic* (une caractéristique statique) et *DynamicCharacteristic* (une caractéristique dynamique) qui représentent des notions génériques alors que l'ontologie du domaine décrit des notions concrètes qui appartiennent au domaine des composants (par exemple *Reusability* et *Scalability*). Le fait que *Reusability* soit un sous-concept de *StaticCharacteristic* exprime explicitement la relation qui existe entre l'ontologie générique et l'ontologie du domaine (voir figure 5.3). Idem pour *Scalability* qui est un sous-concept de *DynamicCharacteristic*.

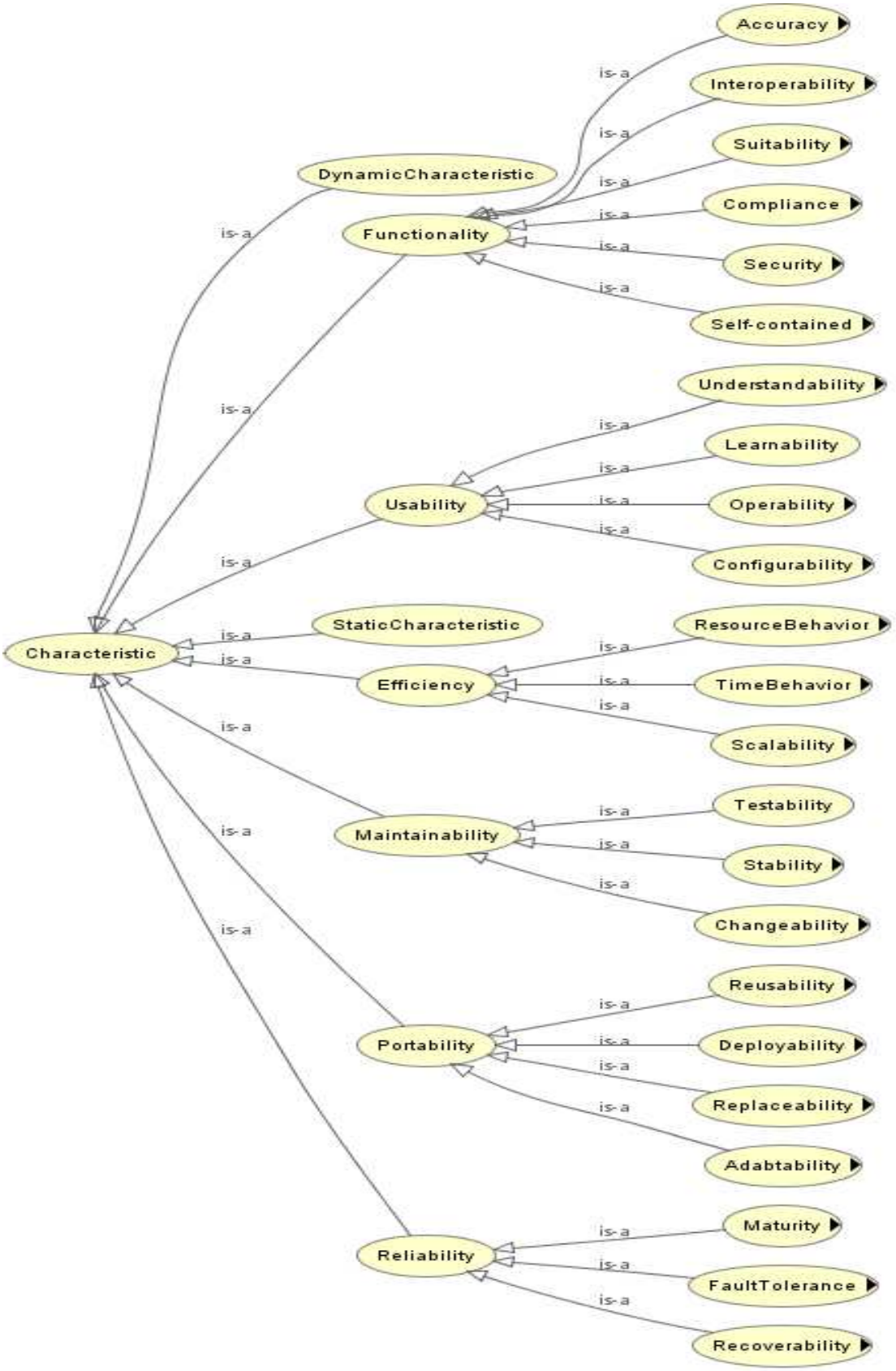


Figure 5.3 : Un extrait du MSD (caractéristiques et sous-caractéristiques)

3.3. Langage de description de l'ontologie QoSOntoCS

Dans l'architecture QoS-CSA, la représentation des connaissances consiste à choisir des formalismes de représentation pour l'ontologie QoSOntoCS et les annotations de composants logiciels.

Les standards actuels de représentation ontologique permettent :

- La formalité (logique du premier ordre pour RDF et logique de description pour OWL) ;
- La multi-instanciation : instances de plusieurs classes ;
- La réutilisation de plusieurs schémas grâce aux namespaces ;
- Des règles d'inférence standards pour raisonner sur les connaissances (OWL).

Dans notre travail, le formalisme choisi pour la modélisation des connaissances doit permettre:

- l'exploitation non ambigüe des différentes connaissances (entre les deux niveaux) ;
- la possibilité de substituer le modèle de QoS du domaine par un nouveau modèle correspondant à un type particulier d'application ;
- la validité du modèle de QoS par rapport au modèle générique (niveau générique) ;
- l'annotation d'un composant par la jointure des deux schémas.

La description de l'ontologie QoSOntoCS est faite dans le langage OWL. Etant donné, d'une part, les caractéristiques de ce langage de description d'ontologies (voir chapitre 3) et d'autre part les besoins cités ci-dessus, OWL est parfaitement adapté pour décrire les concepts de *C*, les propriétés de *P* et leur signature *Signatures*, leurs liens de subsomption H^C et H^P et les règles *Rules* qui s'appliquent sur les concepts et les propriétés.

3.4. Exemples de sérialisation de l'ontologie QoSOntoCS dans OWL

La classe *QoSOntoCSClass* représente le concept racine de tous les concepts de *C*.

```
<owl:Class rdf:ID="QoSOntoCSClass"/>
```

Les concepts *Technology*, *Interface* et *Component* sont des sous-concepts du concept *ExternalContract* qui représente les aspects de classification externe du composant.

```
<owl:Class rdf:ID="ExternalContract">
        <rdfs:subClassOf rdf:resource="#QoSOntoCSClass"/>
</owl:Class>
<owl:Class rdf:ID="Technology">
        <rdfs:subClassOf rdf:resource="#ExternalContract"/>
</owl:Class>
<owl:Class rdf:ID="Interface">
```

```
        <rdfs:subClassOf rdf:resource="#ExternalContract"/>
</owl:Class>
<owl:Class rdf:ID="Component">
        <rdfs:subClassOf rdf:resource="#ExternalContract"/>
</owl:Class>
```

Les concepts *Characteristic* et *Attribute* sont utilisés pour établir le lien entre le modèle générique et le modèle du domaine de la réutilisation. *Characteristic* et *Attribute* sont des sous-concepts de *QoSContract* et permettent pour le premier sous-concept d'instancier des sous-caractéristiques de QoS et pour le second d'instancier des attributs de QoS. Notons que les caractéristiques de QoS définies dans le MSD ne sont pas instanciables.

```
<owl:Class rdf:ID="QoSContract">
         <rdfs:subClassOf rdf:resource="#QoSOntoCSClass"/>
</owl:Class>
<owl:Class rdf:ID="Characteristic">
         <rdfs:subClassOf rdf:resource="#QoSContract"/>
</owl:Class>
<owl:Class rdf:ID="Attribute">
        <rdfs:subClassOf rdf:resource="#QoSContract"/>
</owl:Class>
```

Le concept *ExtendedContract* décrit de manière abstraire les connaissances concernant les résultats de sélection. Il est spécialisé par le concept *Selector et le concept Score* qui servent à conserver la trace des sélections antérieures.

```
<owl:Class rdf:ID="ExtendedContract">
        <rdfs:subClassOf rdf:resource="#QoSOntoCSClass"/>
</owl:Class>
<owl:Class rdf:ID="Selector">
        <rdfs:subClassOf rdf:resource="#ExtendedContract"/>
</owl:Class>
<owl:Class rdf:ID="Score">
        <rdfs:subClassOf rdf:resource="#ExtendedContract"/>
</owl:Class>
```

Par ailleurs, les propriétés *ExternalProperty*, *QoSProperty* et *ExtendedProperty* sont des sous-propriétés de la propriété racine *QoSOntoCSProperty*. Ces trois propriétés permettent de d'organiser des propriétés correspondantes, respectivement, aux aspects externes, de QoS et étendues des composants logiciels.

```
<owl:ObjectProperty rdf:ID="hasInterface">
         <rdfs:subPropertyOf rdf:resource="#ExternalProperty"/>
         <rdfs:domain rdf:resource="#Component"/>
         <rdfs:range rdf:resource="#Interface"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasCharacteristic">
         <rdfs:subPropertyOf rdf:resource="#QoSProperty"/>
         <rdfs:domain rdf:resource="#QoSContract"/>
```

```
          <rdfs:range rdf:resource="#Characteristic"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasScore">
          <rdfs:subPropertyOf rdf:resource="#ExtendedProperty"/>
          <rdfs:domain rdf:resource="#Component"/>
          <rdfs:range rdf:resource="#Score"/>
</owl:ObjectProperty>
```

Les propriétés *hasProvidedInterface* et *hasRequiredInterface* sont, à leur tour, des sous-propriétés de la propriété *hasInterface*.

```
<owl:ObjectProperty rdf:ID="hasProvidedInterface">
          <rdfs:subPropertyOf rdf:resource="#hasInterface"/>
          <rdfs:domain rdf:resource="#Component"/>
          <rdfs:range rdf:resource="#Interface"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasRequiredInterface">
          <rdfs:subPropertyOf rdf:resource="#hasInterface"/>
          <rdfs:domain rdf:resource="#Component"/>
          <rdfs:range rdf:resource="#Interface"/>
</owl:ObjectProperty>
```

Dans OWL, certaines classes peuvent avoir plusieurs super-classes (héritage multiple) mais il est d'usage de représenter la hiérarchie de classe comme étant un arbre simple. La gestion des liens d'héritage multiple est du ressort du raisonneur. Par exemple, le concept TimeBehavior dans le modèle MSD est un sous-concept, à la fois, d'E*fficiency* et de *DynamicCharacteristic*. Dans le même modèle, un attribut de QoS est mesurable et doit donc avoir les propriétés (Data-type) suivantes :

- *hasValue* : mesure la valeur d'un attribut par le biais d'un entier;
- *hasDirection* : caractérise la valeur de métrique pouvant être croissante ou décroissante ;
- *hasUnit* : représente l'unité de la métrique par le biais d'une chaîne de caractères.

```
<owl:DatatypeProperty rdf:ID="hasValue">
    <rdfs:domain rdf:resource="#Attribute"/>
    <rdfs:range rdf:resource="&xsd;#integer"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasDirection">
    <rdfs:domain rdf:resource="#Attribute"/>
    <rdfs:range rdf:resource="&xsd;#boolean"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasUnit">
    <rdfs:domain rdf:resource="#Attribute"/>
    <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>
```

Une fois les différents concepts et propriétés définis dans l'ontologie OWL, ils sont éventuellement instanciés pour construire les contrats d'un composant. C'est ce que nous allons aborder dans la section suivante.

3.5. Peuplement de QoSOntoCS

L'ontologie QoSOntoCS décrite ci-dessus est instanciée pour construire les contrats des composants. Cette étape de création des instances est appelée « Peuplement d'une ontologie ». Cela permet d'annoter les composants disponibles.

3.5.1 Description formelle

Un composant Comp est donc défini comme suit :

Comp= (QoSOntoCS, I, InstC, InstP), où:

- QoSOntoCS=(C, P, H^c, H^p, Signature, Rules) représente l'ontologie QoSOntoCS.
- I est un ensemble d'instances (C, P et I sont des ensembles disjoints).
- InstC: C→ 2^I est une fonction qui permet d'associer des instances aux concepts de QoSOntoCS.
- InstP: P→2^{IxI} est une fonction qui permet d'associer des instances aux propriétés de QoSOntoCS.

L'annotation d'un composant est agrégée de trois annotations : une instance du concept ExternalContract, une instance du concept QoSContract et éventuellement d'une instance du concept ExtendedContract.

Il ressort de cette définition formelle que le lien conceptuel qui relie les différentes annotations à l'ontologie QoSOntoCS est le lien d'instanciation, c'est-à-dire, les concepts et les propriétés de QoSOntoCS sont instanciés pour construire l'annotation d'un composant. Nous traitons dans ce qui suit les trois types d'annotation.

3.5.2 Langage de description

La description des différentes annotations de composant (annotation externe, annotation de QoS et annotation étendue) est faite grâce au langage RDF. Les annotations RDF instancient les concepts et les propriétés d'un modèle OWL pour la description de ressources quelconques (les composants dans notre cas).

La figure 5.4 synthétise le lien qui relie l'ontologie QoSOntoCS aux différentes annotations. Il y apparaît clairement que les annotations RDF sont l'instanciation de connaissances décrites au niveau du modèle OWL. Par exemple le triplet < *VideoCamera, hasProvidedInterface,*

VideoStreamItf> est l'instanciation de la connaissance décrite au niveau de l'ontologie OWL : *hasProvidedInterface (Component, ProvidedInterface)* (en français, un composant logiciel possède une interface fournie). Cette annotation RDF fait partie de la spécification externe qui consiste à décrire les caractéristiques externes d'un composant.

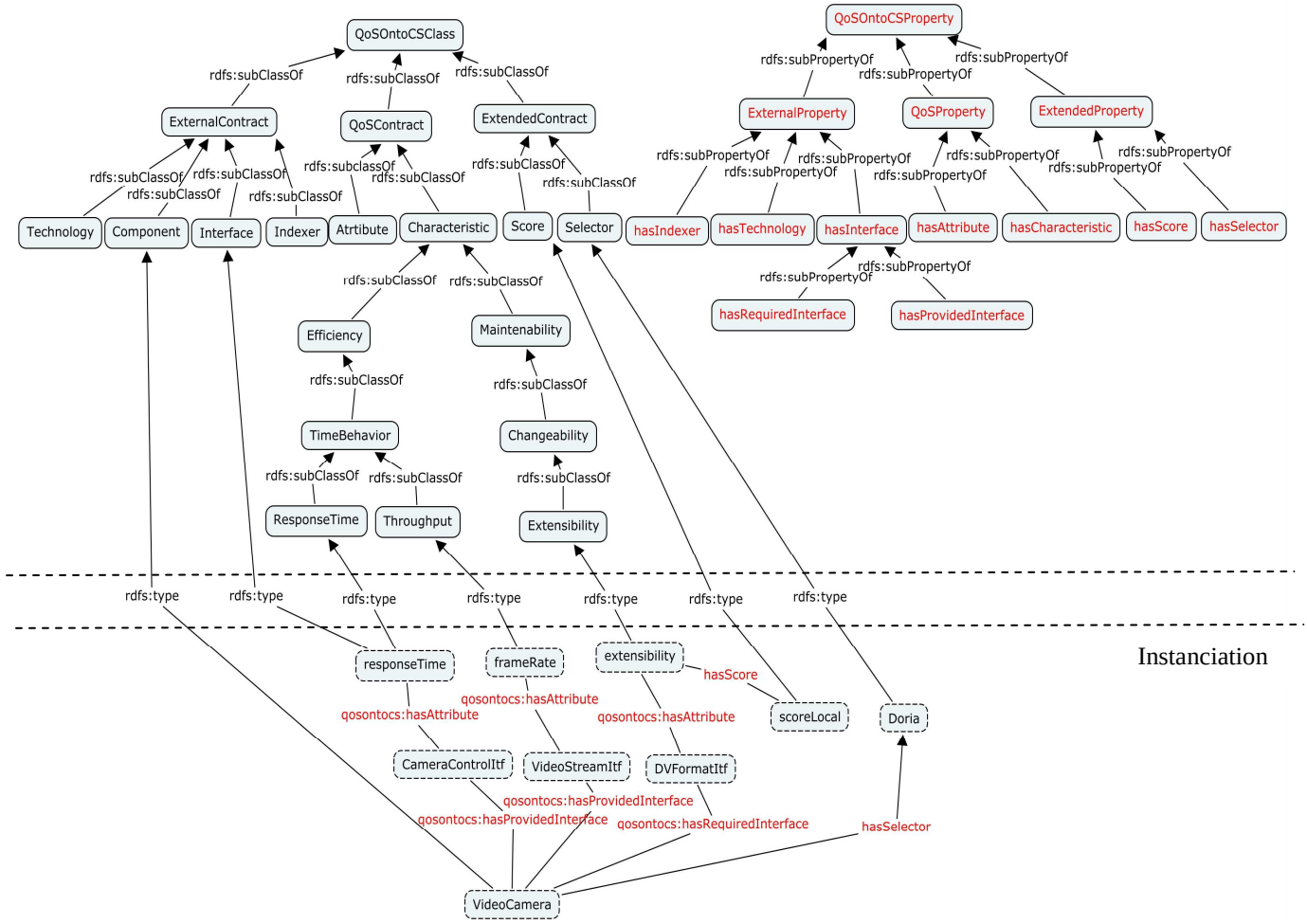


Figure 5.4: Extrait du peuplement ontologique

Dans la figure ci-dessus, nous avons modélisé l'exemple d'un composant VideoCamera adapté à partir des travaux de (Aagedal, 2001). Ce composant possèdent deux interfaces fournies (provided interfaces) de types VideoStream et CameraControl et requiert une interface (required interface) de type DVFormat. La QoS de ce composant est spécifiée comme suit:

- VideoCamera doit envoyer au moins 25 fps (frames per second);

- Il doit démarrer au plus tard 10 millisecondes (milliseconds) après l'appel de la méthode "start";
- Il requiert un composant DV extensible à 99,5% (percent).

Cette spécification de QoS est sérialisée en OWL suivant le schéma montré dans la figure 5.4.

4. Développement des ontologies : Environnement Protégé

Protégé[19] est un système développé à l'Université de Stanford. C'est un outil très populaire dans le domaine du Web Sémantique et très reconnu au niveau de la recherche en informatique. Il est licencié GNU/GPL. Protégé peut lire et sauvegarder des bases de connaissances dans la plupart des formats d'ontologies : RDF, RDFS, OWL, etc. A l'origine Protégé est un outil mono-utilisateur de construction d'ontologie, mais sa version bêta de la version multi-utilisateurs est actuellement accessible pour le travail en groupe. Dans cette version, l'ontologie en cours de développement est stockée dans une base de données partagée, où les utilisateurs multiples peuvent lire la même base de données et faire des modifications incrémentales ou des modifications qui ne sont pas en conflit entre elles. Le mot "multi-utilisateurs" est utilisé pour décrire le fait que la version bêta a des fonctions pour soutenir le travail collaboratif limité entre concepteurs.

Protégé utilise le protocole ouvert de connectivité de base de connaissance OKBC (Open Knowledge Base Connectivity) pour l'interrogation de sa base de connaissances et son interface de construction afin d'atteindre l'interopérabilité avec d'autres systèmes de représentation de la connaissance. De nombreux utilisateurs préfèrent Protégé pour sa convivialité et l'adaptation de ses plugins.

5. Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'aspect représentation des connaissances dans l'architecture QoS-CSA et nous avons montré l'utilisation du formalisme ontologique pour la description de ces connaissances. Notre démarche d'ingénierie combine l'ingénierie de sélection réalisée par des sélectionneurs et l'ingénierie des connaissances réalisée par des ingénieurs de la connaissance. D'une part, les sélectionneurs définissent de manière structurée les caractéristiques de composants primitifs. D'autre part, les ingénieurs des connaissances formalisent ces connaissances par une ontologie QoSOntoCS qui définit deux modèles : (1)

[19] http://protege.stanford.edu/

Modèle générique et (2) modèle spécifique au domaine ou MSD. Ce dernier décrit la QoS relative aux composants logiciels où les aspects de réutilisabilité sont indispensables.

Nous avons choisi le couple (OWL, RDF) pour la représentation ontologique de QoSOntoCS et des annotations de composants. Ce choix est en conformité avec notre choix conceptuel qui décrit l'annotation comme une instanciation de l'ontologie QoSOntoCS pour un composant particulier. La construction de ces représentations est réalisée par le biais de l'environnement Protégé.

Chapitre 6

Algorithmes proposés dans QoS-CSA et évaluation

Dans ce chapitre, nous traitons deux problématiques de sélection. En effet, nous proposons deux solutions destinées à mettre en œuvre l'étape 4 et 5 du processus GCS. La première solution permet d'effectuer l'étape d'évaluation de composants alors que la seconde vise à analyser les résultats d'évaluation. En effet, nous proposons un algorithme appelé « matching de QoS » qui permet de comparer chaque composant candidat avec le composant recherché. Aussi, nous appliquons une technique de prise de décision multi-critères MCDA pour évaluer le résultat du matching et par conséquent classer les composants selon leur ordre de pertinence par rapport au composant recherché.

Avant de présenter notre travail, nous mettons sous hypothèse le fait qu'une sélection fonctionnelle (classification externe, par facette, …) ait été déjà effectuée et que la sélection basée sur la QoS opère sur le résultat de cette étape c'est-à-dire les composants candidats. Nous pouvons commencer par présenter l'environnement de développement intégré que nous utilisons.

1. Outils de développement

En plus de Protégé (utilisé pour le développement de QoSOntoCS), l'implémentation s'effectue avec le langage JAVA sous Eclipse. Cet environnement de développement intégré permet l'utilisation de différents packages, entre autres ceux relatifs au raisonneur Pellet.

1.1. Environnement Eclipse

Eclipse IDE (Eclipse Integrated Developement Environment), développé par IBM, est un environnement de développement intégré libre dont le but est de fournir une plate-forme permettant de créer des projets de développement mettant en œuvre n'importe quel langage de programmation (JAVA, C++, PHP...). Eclipse IDE est principalement écrit en Java.

Eclipse utilise énormément le concept de modules nommés « plug-ins » dans son architecture. D'ailleurs, hormis le noyau de la plate-forme nommé « Runtime », tout le reste de la plate-forme est développé sous la forme de plug-ins. Ce concept permet de fournir un mécanisme pour l'extension de la plate-forme et ainsi fournir la possibilité à des tiers de développer des fonctionnalités qui ne sont pas fournies en standard par Eclipse.

Eclipse possède de nombreux points forts qui sont à l'origine de son énorme succès dont les principaux sont :

- Une plate-forme ouverte pour le développement d'applications et extensible grâce à un mécanisme de plug-ins, plusieurs versions d'un même plug-in peuvent cohabiter sur une même plate-forme, et en plus, Eclipse propose le nécessaire pour développer de nouveaux plug-ins.
- Un support multi langage grâce à des plug-ins dédiés : Cobol, C, PHP, C# , ...
- Support de plusieurs plate-formes d'exécution : Windows, Linux, Mac OS X, ...
- Très rapide à l'exécution.
- Une ergonomie entièrement configurable qui propose selon les activités à réaliser différentes « perspectives ».

1.2 Raisonneur Pellet

Pellet[20] est un des projets du MINDSWAP Group, un groupe de recherche sur le web sémantique de l'université du Maryland.

Il est disponible en OpenSource et offre des évolutions fréquentes. Pellet travaille sur des ontologies décrites en RDF ou OWL et permet les requêtes avec RDQL et SPARQL sur la ABox et la TBox. Pellet possède plusieurs atouts:

- Pellet est open-source et développé en Java.
- Pellet est un raisonneur OWL DL complet.

[20] http://clarkparsia.com/pellet/

- Pellet propose en cas d'incohérence dans l'ontologie des réparations possibles, ainsi qu'une heuristique permettant d'obtenir les informations à ajouter dans l'ontologie pour passer au sous-langage OWL inférieur (OWL Full> OWL DL> OWL Lite).
- Pellet permet l'utilisation des E-Connections et des types de données utilisateurs.
- Pellet peut être intégré à Eclipse.

2. Techniques utilisées

2.1. Score de pertinence proposé

Pour le calcul de ce score, nous proposons une mesure de similarité qui se base à la fois sur les concepts, les instances et les relations de subsomption d'une ontologie. En effet, nous exploitons les relations de subsomption pour définir le matching entre deux concepts R et A. Inspirés de (Paolucci et al., 2002), nous utilisons les niveaux de matching suivants :

- Matching «Exact » si les deux concepts R et A sont conceptuellement équivalents, noté A≡R ;
- Matching «Plugin » si R est un sous-concept de A, noté A ⊑ R ;
- Matching «Subsume » si R est un super-concept de A, noté R ⊑ A ;
- Matching « Partial » si R et A ont le même super-concept, noté $A_j^p \sqcap R_i^p \sqsubseteq \perp$
- Matching «Fail » sinon.

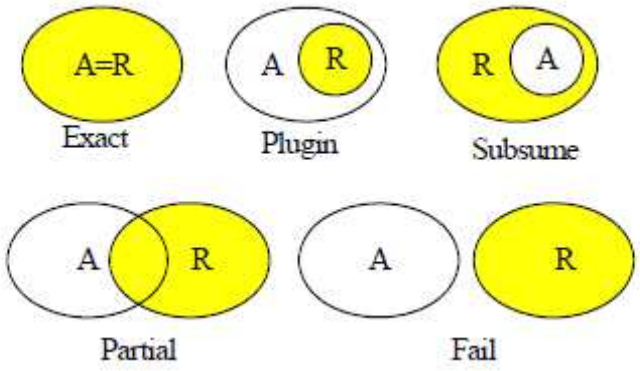


Figure 6.1 : Illustrations des différentes relations du matching

Dans ce qui suit, nous définissons le score de pertinence du composant candidat par rapport au composant recherché exploitant les niveaux définis précédemment. Le niveau le plus élevé est le matching « Exact » pour lequel la valeur δ (δ>6) est attribuée. De même, les valeurs 6,

4 et 2 sont respectivement affectées aux niveaux « Plugin », « Subsume » et « Partial ». Le niveau le plus bas correspond à un échec du matching (« Fail ») et prend la valeur 0. Cependant, cet ordre n'est valable que pour une qualité fournie (qualité liée à une interface fournie). Si la qualité est requise (qualité liée à une interface requise), les niveaux «Plugin » et « Subsume » doivent être intervertis. Evidemment, dans le cas de la qualité requise, moins la qualité requise est satisfaite mieux est le composant candidat.

Quand le développeur spécifie la requête (composant recherché), il introduit un nombre de concepts de qualité de type fourni ou requis. Ces concepts représentent les critères de qualité sur lesquels se base l'évaluation. Pour chaque critère, le composant candidat se voit attribuer un score local obtenu par soit la formule (1) soit la formule (2). Etant donnés un composant recherché $Q(R_1^p, \dots, R_u^p, R_1^r, \dots, R_v^r)$ et un composant candidat $C(A_1^p, \dots, A_m^p, A_1^r, \dots, A_n^r)$:

$$Score_{R_i^p}(C) = \sum_{j=1}^{m} Sim\left(R_i^p, A_j^p\right) / m \qquad \dots(1)$$

$$Score_{R_i^r}(C) = \sum_{j=1}^{n} Sim\left(R_i^r, A_j^r\right) / n \qquad \dots(2)$$

Où $Sim\left(R_i^p, A_j^p\right)$ est le degré de similarité entre deux concepts de qualité fournie (p pour provided, en anglais). Son calcul suit le raisonnement énoncé précédemment, c'est-à-dire :

$$Sim(R_i^p, A_j^p) = \begin{cases} \delta, & si\ A_j^p \equiv R_i^p \\ 6, & si\ A_j^p \sqsubseteq R_i^p \\ 4, & si\ R_i^p \sqsubseteq A_j^p \\ 2, & si\ A_j^p \sqcap R_i^p \sqsubseteq \bot \\ 0, & sinon \end{cases}$$

Idem, le calcul du degré de similarité entre deux concepts de qualité requise (r pour required, en anglais) est défini comme suit :

$$Sim(R_i^r, A_j^r) = \begin{cases} \delta, & si\ A_j^r \equiv R_i^r \\ 6, & si\ R_i^r \sqsubseteq A_j^r \\ 4, & si\ A_j^r \sqsubseteq R_i^r \\ 2, & si\ A_j^r \sqcap R_i^r \sqsubseteq \bot \\ 0, & sinon \end{cases}$$

Le résultat est donc un vecteur de scores locaux, de dimension (u+v), attribué à chaque composant candidat. Un score local évalue le composant par rapport à un critère de qualité (fournie ou requise) décrit dans le composant recherché. L'ensemble des critères est hiérarchisé dans l'ontologie QoSOntoCS et plus exactement au niveau du MSD. Des exemples de critères de qualité sont : *Reliability* et *Efficiency* ou bien des critères plus affinés tels que : *Fault tolerance* et *Time Behavior*. La relation entre ces critères est déterminée par le raisonneur Pellet.

Dans le cas où deux critères (ou concepts) A_j^p et R_i^p (resp. A_j^r et R_i^r) sont équivalents, les instances correspondantes doivent être considérées : nous calculons la conformité entre leurs valeurs de métrique respectives.

2.2. Conformité contractuelle

La conformité contractuelle (ou contract conformance, en anglais) est une relation utilisée pour le matching des aspects non fonctionnels (Sriram et al., 2002). Deux contrats sont conformes si toutes leurs contraintes sont conformes. Afin de définir la conformité d'une contrainte, nous utilisons la relation « est meilleur que» entre les valeurs d'une métrique de QoS. Dans certains cas, la plus petite valeur de métrique peut être conceptuellement meilleure que la plus grande. L'information qui permet de déterminer cette relation est la direction d'une métrique. La direction croissante indique que la qualité est meilleure quand sa métrique croît (par exemple le taux de trames). Inversement, la direction décroissante sous-entend que la qualité est meilleure quand sa métrique décroît (par exemple, le temps de réponse).

Il est donc primordial de considérer, dans l'étape d'évaluation, les instances liées aux critères de QoS quand leurs concepts sont équivalents. En s'appuyant sur la notion de conformité, la valeur de δ est fixée comme suit :

$$\delta = \begin{cases} 10, & si \ \left(A_j^p.V > R_i^p.V \wedge D \nearrow\right) \vee \left(A_j^p.V < R_i^p.V \wedge D \searrow\right) \\ 8, & sinon \end{cases}$$

Dans le cas de la qualité requise, nous définissons la conformité d'une contrainte par la relation « est pire que » entre les valeurs d'une métrique de QoS. Cette relation est l'opposée de la relation « est meilleur que ». De ce fait, la valeur de δ s'inverse :

$$\delta = \begin{cases} 10, & si \ \left(A_j^p.V < R_i^p.V \wedge D \nearrow\right) \vee \left(A_j^p.V < R_i^p.V \wedge D \searrow\right) \\ 8, & sinon \end{cases}$$

2.3. Technique MCDA

Les résultats obtenus lors de l'évaluation doivent être classés afin de pouvoir sélectionner le meilleur composant évalué. Pour cela, nous appliquons une technique de décision multicritères comme étudiée dans le chapitre 2. Nous optons pour la méthode MCDA pour son efficacité et sa facilité d'utilisation. Nous reprenons la formule déjà présentée en la transcrivant avec nos propres notations :

$$Score\ (C) = \sum_{i=1}^{u} w_i \times Score_{R_i^p}(C) + \sum_{j=1}^{v} w_j \times Score_{R_i^r}(C)$$

Où w_i (resp. w_j) est le poids lié au critère R_i^p (resp. R_i^r).

Quand le développeur spécifie le composant recherché, il ne se contente pas de spécifier les critères de QoS désirée mais également ses préférences pour l'un des critères ou l'autre. Ce sont ces préférences qui vont servir à la distribution des poids w_i et w_j.

3. Algorithmes proposés

3.1. Algorithme « QoS matching »

Généralement, on définit le matching de spécifications comme étant "un processus qui détermine si deux spécifications matchent" (Zaremski and Wing, 1995(b)). Dans ce qui suit, nous considérons que le matching des fonctionnalités a été déjà effectué et nous focalisons sur le matching de QoS en utilisant l'ontologie QoSOntoCS. Intuitivement, cet algorithme implémente le score local de pertinence décrit dans la section 2.1. Son rôle est également *décisif* dans le cas où les besoins du développeur changent :

- ***Augmentation :*** le développeur ajoute d'autres critères de QoS ;
- ***Adaptation :*** le développeur maintient les mêmes critères mais modifie leurs valeurs ;
- ***Relaxation :*** le développeur élimine certains critères de QoS.

Ces trois opérations sont exclusives et doivent intervenir avant le processus de ranking de composants décrit dans la section 3.2.

3.1.1 Enoncé et description

Algorithme 1: Matching de QoS

Entrées :

$Q(R_1^p, \cdots, R_u^p, R_1^r, \ldots, R_v^r)$ est le composant recherché par le développeur

$C(A_1^p, \ldots, A_m^p, A_1^r, \ldots, A_n^r)$ est un composant candidat

Sortie :

localScores [] est le résultat du matching entre le composant Q et le composant C

L'algorithme de matching de QoS est comme suit:

```
1.  localScores []= Array of integer; k←0
2.  for each R_i^p (1≤i≤u) do
3.    begin
4.      Sim← 0  // Initialisation
5.      for each A_j^p (1≤j≤m) do
6.        begin
7.          case match (R_i^p , A_j^p ) of
8.            "Exact":  Sim← Sim+conform(R_i^p , A_j^p )
9.            "Plugin": Sim← Sim+1/2
10.           "Subsume": Sim← Sim+1/4
11.         end
12.       localScores[k]←Sim/m
13.       k←k+1
14.   end
15. for each R_i^r (1≤i≤v) do
16.   begin
17.     Sim← 0  // Initialisation
18.     for each A_j^r (1≤j≤n) do
19.       begin
20.         case match (R_i^r , A_j^r ) of
21.           "Exact":  Sim← Sim+conform(R_i^r , A_j^r )
22.           "Subsume": Sim← Sim+1/2
23.           "Plugin": Sim← Sim+1/4
24.         end
25.       localScores[k]←Sim/n
26.       k←k+1
27.   end
28. return localScores[]
```

Figure 6.2 : Enoncé de l'algorithme « Matching de QoS »

Cet algorithme implémente le score de pertinence que nous avons proposé. Il combine le matching de concepts (match(R_i^p ,A_j^p) ou match (R_i^r ,A_j^r)) et d'instances (conform(R_i^p ,A_j^p) ou

conform(R_i^r,A_j^r)). En premier lieu, une fonction de matching (lignes 7&19) est appelée, paramétrée par les deux concepts R_i^p et A_j^p. Un raisonnement logique effectué par Pellet permet de déterminer la relation de subsomption entre ces deux concepts comme suit :

```
static String match (OntClass a, OntClass b){
      if (a.hasEquivalentClass(b)){
          return "Exact";
      }else
          if (a.hasSuperClass(b)){
              return "Plugin";
          }else
              if (a.hasSubClass(b)){
                  return "Subsume";
              }else
                  if (a.getSuperClass()==b.getSuperClass()){
                      return "Partial";
                  }else
                      return "Fail";
}
```

Figure 6.3 : Relation hiérarchique entre deux concepts a et b (Code source)

En second lieu, dans le cas où R_i^p et A_j^p sont équivalents c'est-à-dire match(R_i^p,A_j^p) = "Exact", une fonction de vérification de conformité (lignes 8&20) est appelée, paramétrée par les deux valeurs de métrique $R_i^p.V$, $A_j^p.V$ et leur direction *D*.

Nous utilisons une requête SPARQL afin de récupérer la valeur d'une métrique et sa direction (voir la figure 6.4).

```
          // Construction de la requête
           String queryString =   myOntologyPrefix + NL
                                + rdfPrefix + NL +
                                  "SELECT ?hasValue ?hasDirection WHERE
{?individu rdf:type qosontocs:DataEncryption}" ;

           Query query = QueryFactory.create(queryString) ;

           QueryExecution qexec = QueryExecutionFactory.create(query, m) ;

           // Exécution de la requête
           try {
                   ResultSet rs = qexec.execSelect() ;
                        }
           }
           finally {
             qexec.close() ;
           }
```

Figure 6.4 : Extraction des propriétés data-tyes d'une métrique

3.1.2 Exemple

Nous reprenons l'exemple du composant *VideoCamera* (voir le chapitre 5). Le tableau 6.1 illustre les entrées du système à savoir une requête (Q) et trois composants (C_1, C_2 et C_3).

Tableau 6.1 : Spécification de QoS des components C_1, C_2 et C_3

	Q	C_1	C_2	C_3
R_1^p	frameRate= 25 fps	frameRate = 15 fps	frameRate = 30 fps	frameRate = 25 fps
R_2^p	responseTime = 10 ms	responseTime = 10 ms	responseTime= 10 ms	responseTime = 50 ms
R_1^r	extensibility = 99.5%	changeability = 99.5%	extensibility = 99.5%	extensibility= 99%

Tableau 6.2 montre les sorties du matching de QoS concernant l'exemple du tableau 6.1. On calcule pour, chaque composant, un vecteur de scores locaux. Ces derniers servent pour le calcul du score de pertinence global permettant le classement des différents composants et la sélection du meilleur d'entre eux.

Tableau 6.2 : Résultats du matching de QoS

	C_1	C_2	C_3
$Score_{R_1^p}(C)$	10/3	12/3	12/3
$Score_{R_2^p}(C)$	12/3	12/3	10/3
$Score_{R_1^r}(C)$	4/3	4/3	8/3

A ce niveau, il n'est pas possible de sélectionner le meilleur composant. De ce fait, il est primordial d'appliquer un processus permettant de classer les différents composants : il s'agit du processus de ranking de composants.

3.2. Algorithme « Ranking de composants »

Cet algorithme implémente la formule de la technique MCDA. Il utilise les résultats du matching et les préférences du sélectionneur afin d'obtenir un score global pour chaque composant évalué par rapport au composant recherché (représenté par un ensemble de critères

de QoS). Une distribution de poids est effectuée de manière automatique à partir des préférences du sélectionneur. La somme de ces poids doit être égale à 1 (ou 100%). Nous notons qu'il existe d'autres travaux permettant la distribution des poids sans que l'utilisateur ait contraint à les spécifier. Dans ce cas, on se base sur l'importance de chaque critère dans une application donnée.

3.2.1 Enoncé et description

Algorithme 2: Ranking de composants

Entrées:

W[] est le vecteur d'entiers représentant des poids liés aux critères de qualité

localScores[] est le vecteur d'entiers représentant les scores locaux du composant C

Sortie :

GlobalScore est un entier représentant le score global du composant C

L'algorithme de ranking de composants est comme suit:

1. K←1, GlobalScore←0
2. for each K ($1 \leq k \leq (u+v)$) do
3. GlobalScore←GlobalScore+ localScores[k] x W[k]
4. return GlobalScore

Figure 6.5 : Enoncé de l'algorithme « Ranking de composants »

Cet algorithme implémente la formule MCDA. Le but de cette formule est d'associer un score global à chaque composant candidat. Il utilise pour cela le vecteur de scores locaux obtenu lors du matching. Ce score permet de classer les candidats et par conséquent trouver le meilleur d'entre eux.

3.2.2 Exemple

Reprenons les résultats de l'exemple précédent tout en ajoutant les poids correspondant à chaque critère (un critère par ligne).

Tableau 6.3 : Résultats du « Ranking de composants »

	Poids (préférences du sélectionneur)	$Score_{R_1^p}(C_1)$	$Score_{R_1^p}(C_2)$	$Score_{R_1^p}(C_3)$
R_1^p	0,5	10/3	12/3	12/3
R_2^p	0,25	12/3	12/3	10/3
R_1^r	0,25	4/3	4/3	8/3
Global Score	**1**	**9/3**	**10/3**	**10,5/3**

Nous pouvons désormais dire que le composant C_3 car il possède le score global le plus élevé et est similaire au composant recherché spécifié par le triplet (R_1^p, R_2^p, R_1^r).

4. Evaluation

4.1. Expérimentation

L'évaluation de notre stratégie de sélection est établie sur un exemple concret tiré du référentiel de composants ComponentSource. Pour chaque composant candidat, ComponentSource fournit une courte description, une présentation du constructeur, un descriptif des différentes licences disponibles, une version d'évaluation à télécharger, éventuellement une documentation fournie par le constructeur et une section "compatibilités" qui regroupe un ensemble d'informations utiles, parmi lesquelles nous pouvons trouver :

- Liste des systèmes d'exploitation compatibles : Windows 2000, Windows XP, Mac OSX...
- Liste des conteneurs compatibles : Visual Studio 5, Visual Basic, Borland Delphi 6...
- Type du composant : contrôle ActiveX, classe .NET, JavaBeans, DLL...
- Présence ou non de propriétés de qualité recensées par ComponentSource, telles que : l'inclusion d'une signature numérique, l'annotation "cryptage sécurisé", l'annotation "initialisation sécurisée", la compatibilité avec certains protocoles...

- Ensemble des tests effectués et validés par ComponentSource, tels que : l'installation, la désinstallation, la vérification antivirus, l'installation et la désinstallation de la version d'évaluation, l'examen de la documentation, l'examen d'un échantillon de code, et un test avec .NET RCW pour savoir si le composant est ".NET ready".

D'un point de vue non-fonctionnel, cette section "compatibilités" peut être considérée comme le modèle de qualité informel de ComponentSource. Pour des raisons de simplicité, nous n'allons considérer que cinq attributs : Data Encyption, Memory Usage, Disk Usage, Mobility et Cost. Notons que ces attributs sont couverts par notre modèle MSD.

Dans cette étude de cas, nous opérons donc avec un nombre variable (16, 32 puis 50) de composants candidats (voir tableau 6.4). Chacun de ces composants offre une fonctionnalité de téléchargement FTP avec une certaine qualité mesurée par les attributs suivants :

- Data Encyption: indique si le composant gère le protocole SSL ou pas;
- Memory Usage: indique la taille de mémoire consommée par le composant ;
- Disk Usage : indique l'espace disque nécessaire pour le composant ;
- Mobility : recense les différents conteneurs où le composant peut être déployé ;
- Cost : indique le coût (licence) du composant.

Généralement, le sélectionneur exige une bonne qualité avec une utilisation limitée des ressources et un coût raisonnable. Nous avons testé donc la requête suivante :

- *Critère 1* : Data Encyption avec la valeur « True » ;
- *Critère 2* : Memory Usage avec la valeur « 64 » ;
- *Critère 3* : Disk Usage avec la valeur « 20 » ;
- *Critère 4* : Mobility avec la valeur « 5 » ;
- *Critère 5* : Coût avec la valeur « 1000 ».

Quand le sélectionneur émet sa requête, le système lui renvoie le résultat, c'est-à-dire un ensemble de composants qu'il juge pertinents. Afin de déterminer la pertinence du résultat, nous allons calculer trois paramètres à savoir : la précision, le rappel et la F-mesure.

Tableau 6.4: Exemples de composants candidats

Composant	Description
C_1	dataEncryption =false, memoryUsage =8, diskUsage =3, mobility =13, cost = 294
C_2	dataEncryption =false, mobility =8, cost = 294
C_3	dataEncryption =true, diskUsage =10, mobility =3, cost = 681
C_4	dataEncryption =true, memoryUsage =16, mobility =18, cost = 1394
C_5	dataEncryption =true, memoryUsage =64, diskUsage =2, cost = 949
C_6	dataEncryption =true, memoryUsage =64, diskUsage =2, mobility =5, cost = 747
C_7	dataEncryption =true, memoryUsage =64, diskUsage =2, mobility =1, cost = 746
C_8	dataEncryption =false, diskUsage =2, cost = 569
C_9	dataEncryption =false, memoryUsage =32, diskUsage =10, mobility =15, cost = 289
C_{10}	dataEncryption =false, diskUsage =5, mobility =5, cost = 879

Précision

La précision P est le nombre de composants pertinents sélectionnés rapporté au nombre total de composants retournés par l'algorithme « matching de QoS » suite à une requête donnée.

Le principe est le suivant: quand un sélectionneur interroge la base locale, il souhaite que les composants proposés en réponse à son interrogation correspondent à son attente. Tous les composants retournés superflus ou non pertinents constituent du bruit. La précision s'oppose à ce bruit de sélection. Si elle est élevée, cela signifie que peu de composants inutiles sont retournés par le système et que ce dernier peut être considéré comme "précis". On calcule la précision P avec la formule suivante:

$$P= |\{ \text{composants pertinents}\} \cap \{ \text{composants retournés}\}| / |\{ \text{composants retournés}\}|$$

Rappel

Le rappel R est défini par le nombre de composants pertinents retrouvés au regard du nombre de composants pertinents que possède la base locale. Cela signifie que lorsque le sélectionneur interroge la base il souhaite voir apparaître tous les composants qui pourraient répondre à ses besoins. Si cette adéquation entre la requête du sélectionneur et le nombre de composants présentés est importante alors le taux de rappel est élevé. A l'inverse si le système possède de nombreux composants intéressants mais que ceux-ci n'apparaissent pas on parle de silence. Le silence s'oppose au rappel.

On calcule le rappel R avec la formule suivante

R= |{ composants pertinents} ∩ { composants retournés}|/ |{ composants pertinents}|

F-mesure

La F-mesure est une mesure qui combine la précision et le rappel. On calcule la F-mesure avec la formule suivante :

F= 2* (P*R) / (P+R)

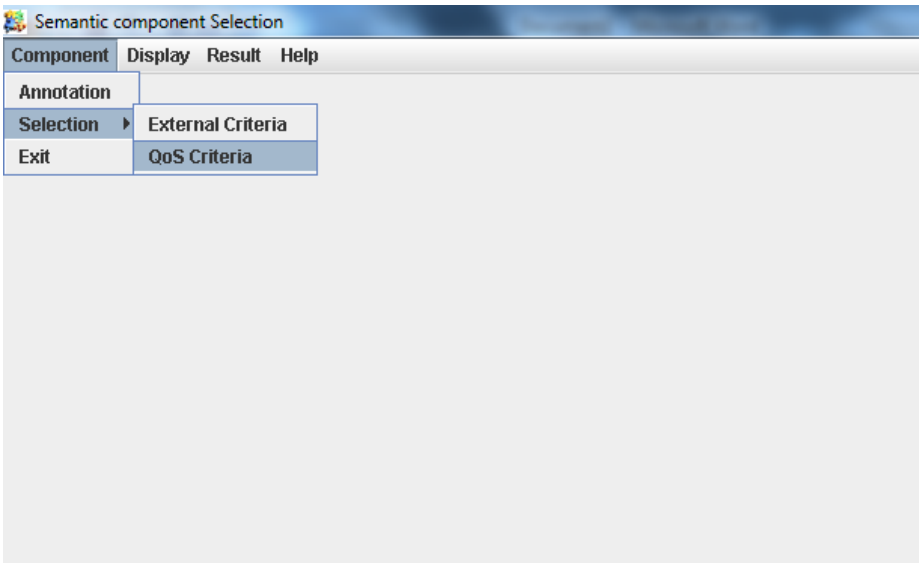
En général, un système de recherche parfait fournira des réponses dont la précision et le rappel sont égaux à 1 (l'algorithme trouve la totalité des composants pertinents - rappel - et ne fait aucune erreur - précision). Dans la réalité, les algorithmes de recherche sont plus ou moins précis, et plus ou moins pertinents. Il sera possible d'obtenir un système très précis (par exemple un score de précision de 0,99), mais peu performant (par exemple avec un rappel de 0.10, qui signifiera qu'il n'a trouvé que 10% des réponses possibles). Dans le même ordre d'idées, un algorithme dont le rappel est fort (par exemple 0.99 soit la quasi totalité des composants pertinents), mais la précision faible (par exemple 0.10) fournira en guise de réponse de nombreux composants erronés en plus de ceux pertinents: il sera donc difficilement exploitable.

4.2. Prototype expérimental

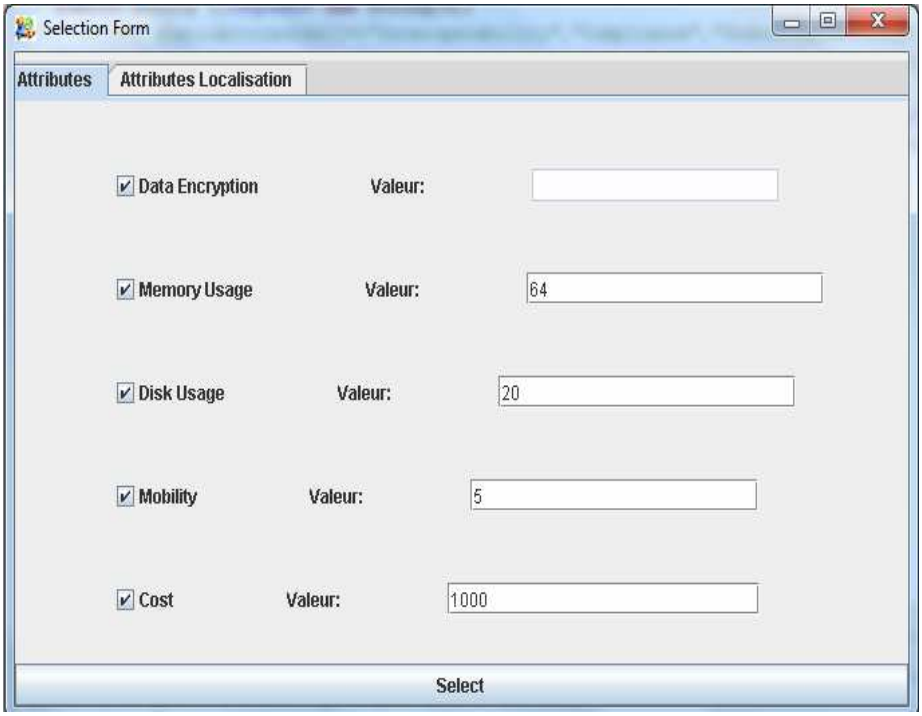
Nous avons expérimenté notre approche de sélection par un programme JAVA. Nous avons implémenté les algorithmes énoncées précédemment en utilisant la requête spécifiée plus haut dans la section 4.1 ainsi que les composants candidats disponibles dans la base de connaissances locale dont un extrait est représenté par le code RDF suivant :

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.semanticweb.org/ontologies/2011/11/Ontology1323712015303.owl#"
   ……..
   <Component rdf:ID="CompInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasAttribute rdf:resource="#CostInstance1"/>
      <hasAttribute rdf:resource="#DiskInstance1"/>
      <hasAttribute rdf:resource="#EncrInstance1"/>
      <hasAttribute rdf:resource="#MemoInstance1"/>
      <hasAttribute rdf:resource="#MobInstance1"/>
      <hasName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">XceedFTPLibrary</hasName>
   </Component>
   <Component rdf:ID="CompInstance2">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasAttribute rdf:resource="#CostInstance1"/>
      <hasAttribute rdf:resource="#EncrInstance1"/>
```

```
      <hasAttribute rdf:resource="#MobInstance2"/>
      <hasName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">XceedFTPDotNet</hasName>
   </Component>
   ……..
   <owl:Class rdf:ID="Cost">
      <rdfs:subClassOf rdf:resource="#Marketability"/>
   </owl:Class>
   <Cost rdf:ID="CostInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasDirection
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</hasDirection>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">294</hasValue>
   </Cost>
   ……..
   <DiskUsage rdf:ID="DiskInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasDirection
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</hasDirection>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">3</hasValue>
   </DiskUsage>
   ……..
   <DataEncryption rdf:ID="EncrInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</hasValue>
   </DataEncryption>
  ……..
   <MemoryUsage rdf:ID="MemoInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasDirection
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</hasDirection>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">8</hasValue>
   </MemoryUsage>
   ……..
   <Mobility rdf:ID="MobInstance1">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasDirection
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</hasDirection>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">13</hasValue>
   </Mobility>
      <Mobility rdf:ID="MobInstance2">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
      <hasDirection
rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</hasDirection>
      <hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">8</hasValue>
   </Mobility>
   ……
   <rdf:Description rdf:about="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</rdf:RDF>
```



a) Page d'accueil



a) Fiche des critères de sélection

Figure 6.6: Quelques captures d'écran du prototype expérimental

4.3. Résultats et discussion

Les attributs de qualité manipulés dans l'étude de cas possèdent des correspondances dans le modèle CQM. Selon une étude effectuée par (Alvaro et al., 2006), la couverture du modèle représente plus de 70% des composants ComponentSource. Nous exécutons le programme de matching à plusieurs reprises en utilisant à chaque fois un ensemble plus grand de composants

candidats. Cela nous permet de calculer la précision, le rappel et la F-mesure après chaque résultat. Les mesures obtenues sont représentés par le graphe de la figure 6.7.

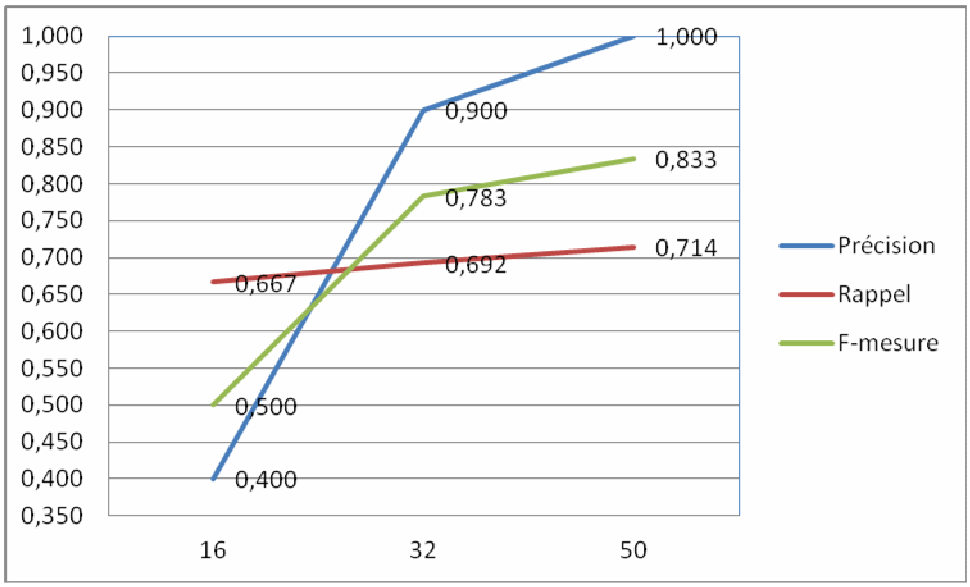


Figure 6.7 : Variation des performances du score global en fonction du nombre de composants

Nous avons ensuite implémenté l'algorithme de Paolucci décrit dans le chapitre 5. Notre stratégie s'est révélée plus performante avec une bonne tendance, c'est-à-dire des résultats croissant avec l'augmentation du nombre de composants. Une comparaison entre les résultats des deux approches est schématisée ci-dessous.

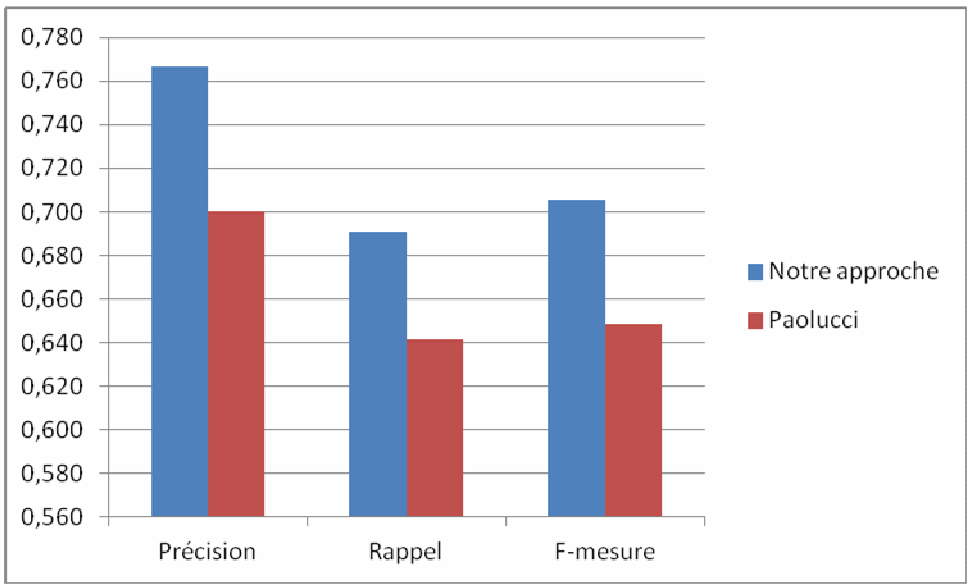


Figure 6.7 : Comparaison entre notre approche et celle de Paolucci

5. Conclusion

Le matching de QoS permet d'obtenir le score d'un composant candidat relativement à un critère de QoS (spécifié par le composant recherché). De ce fait, chaque composant candidat se voit attribuer un vecteur de scores relativement à un composant recherché. Le problème que nous avons recensé, lors de notre étude sur les techniques de prise de décision multi-critères, est désormais résolu : les scores sont affectés automatiquement et non manuellement. De plus, ces scores tiennent en compte de la sémantique des concepts en exploitant les relations de subsomption définies dans l'ontologie QoSOntoCS. Nous pouvons ensuite appliquer MCDA en utilisant les préférences du sélectionneur comme poids d'importance.

Nous avons présenté un exemple à travers lequel nous avons déroulé nos algorithmes. Encore mieux, nous avons testé nos programmes sur une étude de cas traitant ainsi un nombre plus grand de composants. Enfin, nous avons comparé notre approche avec celle de Paolucci : Un meilleur résultat et une meilleure tendance pour notre approche.

Conclusion générale

Conclusion Générale

Bilan

Proposition d'une architecture de sélection

Nous avons élaboré une architecture pour un système de sélection de composants logiciels. Nous nous sommes intéressés uniquement aux critères de qualité de service (QoS) qui sont très peu pris en compte dans les processus de sélection existants. Le processus de sélection que nous avons proposé est sémantique grâce à la définition d'une ontologie qui spécifie les concepts de QoS et leurs relations. Ce processus englobe les étapes suivies par la plupart des méthodes de sélection de COTS (composants sur étagère), à savoir : l'identification des critères d'évaluation, l'évaluation de composants et leur sélection.

Pour atteindre notre objectif, nous avons d'abord construit l'ontologie QoSOntoCS qui formalise les caractéristiques de QoS par des concepts et des propriétés. Les indexeurs construisent ensuite de manière semi-automatique, en collaboration avec les ingénieurs des connaissances, des annotations conceptuelles. Ces annotations décrivent les caractéristiques fonctionnelles et non fonctionnelles des composants disponibles dans les référentiels et sont des triplets RDF obtenus par instanciation des concepts de l'ontologie QoSOntoC. Enfin, ces annotations sont exploitées par le processus de sélection pour déterminer la similarité sémantique entre le composant recherché par le développeur et les composants annotés.

Les expérimentations menées, dans le cadre de cette thèse, montrent que l'architecture QoS-CSA améliore la fiabilité et l'efficacité du processus de sélection des composants et donc permet d'atteindre nos objectifs en terme d'économies de temps, d'effort et de coût du processus de développement.

Proposition d'une ontologie pour la gestion des connaissances

La description et la gestion des connaissances dans QoS-CSA sont réalisées grâce à des modèles et des langages du Web sémantique. Les connaissances sur la QoS sont décrites dans l'ontologie QoSOntoCS de manière formelle et partagée entre les acteurs du système.

L'ontologie QoSOntoCS est organisée selon deux niveaux d'abstraction: d'une part, un modèle générique qui décrit des concepts génériques et indépendants de tout domaine. D'autre part, un modèle spécifique au domaine (MSD) qui permet de décrire des concepts

plus concrets et modélisant la QoS des composants logiciels. Le peuplement de l'ontologie QoSOntoCS permet d'ajouter davantage de composants dans la base de connaissances locale. Un composant annoté regroupe l'annotation fonctionnelle externe, l'annotation de QoS et éventuellement l'annotation étendue, qui décrit respectivement les caractéristiques externes du composant, les caractéristiques de QoS et les scores obtenus lors de la sélection.

Calcul automatique des scores

Nous avons étudié trois techniques de prise de décision multi-critères dont la faiblesse principale était d'évaluer manuellement le score de chaque composant candidat pour chaque critère de QoS. Pour pallier cette difficulté, nous avons proposé un algorithme de matching de QoS. Ce dernier se base sur l'ontologie QoSOntoCS afin de calculer le score de pertinence du composant candidat (annoté). A partir des degrés de préférence spécifiés par le sélectionneur et des résultats du matching, la méthode MCDA est appliquée dans le but de classer les composants candidats et de choisir le meilleur d'entre eux. Ainsi, nous avons proposé un second algorithme de ranking de composants.

Les deux processus précédents permettent de réaliser respectivement deux étapes importantes dans une démarche classique de sélection de composants : l'évaluation des composants candidats et l'analyse de cette évaluation.

Tests expérimentaux

Afin de tester la performance de la mesure de similarité implémentée par le matching de QoS, nous avons évalué les résultats obtenus en utilisant certains paramètres : précision, rappel et F-mesure. La précision est le nombre de composants pertinents par rapport au nombre de composants sélectionnés, le rappel consiste à déterminer le nombre de composants sélectionnés pertinents par rapport à tous les composants candidats pertinents, alors que la F-mesure est calculée à partir de ces deux paramètres.

D'abord, nous avons testé notre approche sur un cas concret : il s'agit de la sélection d'un composant de téléchargement FTP du référentiel ComponentSource.

Nous avons opéré sur un nombre relativement élevé de composants (16, 32 puis 50) avec une requête exigeant une bonne qualité et des contraintes raisonnables.

Sur les dix premiers composants retournés, le résultat a démontré un bon rappel et une bonne précision de notre système. Ensuite, nous avons appliqué l'approche de Paolucci sur la même étude de cas ; nous avons constaté une amélioration des résultats apportée par notre système.

Notons que notre approche réutilise le principe de matching de Paolucci déterminant les relations de subsomption entre concepts. Ce matching a été exploité par plusieurs travaux dans le domaine des Web services.

Perspectives de recherche

L'ontologie que nous proposons peut être étendue par d'autres connaissances : nous pouvons ajouter d'autres attributs de qualité pour couvrir d'autres domaines d'application. Ces derniers peuvent être placés dans la hiérarchie ISO 9126 après une étude experte. Par exemple, dans (Behkamal et al., 2009), on y a inséré les critères traceability, availability, customizability et navigability constituant d'importantes caractéristiques dans le cas des applications B2B.

Il est complémentaire à notre travail de traiter la sélection des composants composites. Un composant composite ou boite grise est une hiérarchie de sous-composants. Nous pouvons nous baser sur le matching proposé dans le cadre de ce travail de thèse afin d'attribuer des scores de pertinence aux composants composites. Ainsi, le problème d'intégration de ces composants constitue une perspective importante pour ce travail. En effet, la qualité de service rendue par un composant ne peut pas être garantie lors de son assemblage.

L'ouverture de notre système sur le Web constitue une perspective intéressante pour augmenter le choix des composants candidats. Pour cela, il faudrait extraire des connaissances sur la QoS à partir des descriptions textuelles disponibles sur le Web. Le challenge est important car ces descriptions textuelles ne sont pas standardisées, ambigües, informelles, et donc difficilement exploitables par la machine.

Références bibliographiques

Aagedal J.Ø. (2001). Quality of Service Support in Development of Distributed Syfrstems. PhD thesis, University of Oslo.

ACCORD (2002). Etat de l'art sur les Langages de Description d'Architectures. Projet ACCORD at http:// www.infres.enst.fr.projet/accord/.

Almeida, E.S, A. Akvaro, V.C. Garcia, J. Cordiro, P. Mascena, V. A. Arruda Buriegio, L. M. Nascimento, D. Lucriedio and S.L. Meira (2007). C.R.U.I.S.E. : Component ReUse In Software Engineering. C.E.S.A.R. (Recife Center for Advanced Studies and Systems).

Alnusair, A. and T. Zhao (2010). Component Search and Reuse: An Ontology-based Approach, The 11th IEEE International Conference on Information Reuse and Integration (IRI-2010), IEEE Computer Society Press, pages 258-261. Las Vegas, NV, USA.

Al Shalabi, L., Z. Shaaban and B. Kasasbeh (2006). Data Mining: A Preprocessing Engine. Journal of Computer Science 2 (9), pages 735-739.

Alvaro, A., E. S. de Almeida and S. Meira (2006). A software component quality model: A preliminary evaluation. In Proceedings of the32 nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICROSEAA),Dubrovnik, Croatia.

Andreasen T., H. Bulskov and R. Knappe (2003). Similarity for Conceptual Querying. Proceedings for the 18th International Symposium on Computer and Information Sciences, pages 268-275.

Aufaure M. A., R. Soussi, and H. Baazaoui (2007). SIRO: On-line semantic information retrieval using ontologies. 2nd International Conference on Digital Information Management, ICDIM'07, pages 321-326.

Badaro, N., Th. Moineau (1991). Rose-ada: A method and a tool to help reuse ada code, In Proceedings, Ada Europe Conference. Springer Verlag (Lecture Notes in Computer Science, vol 499).

Barais, O., L. Duchien and R. Pawlak (2003). Separation of concerns in software modeling : A framework for software architecture transformation. IASTED International Conference on Software Engineering Applications (SEA), Los Angeles, USA, November 2003, pages 663-668, ACTA Press. ISBN : 0-88986-394-6.

Bass, L., P. Clements, and R. Kazman (2003). Software Architecture in Practice. AddisonWesley, second edition.

Baziz, M. (2005). Indexation conceptuelle guidée par ontologie pour la recherche d'information. Thèse, Université Paul Sabatier, Toulouse.

Bidoit, M. (1989). Pluss, un langage pour le développement de spécifications algébriques modulaires, PhD Thesis, Université de Paris-Sud – Centre d'Orsay.

Boehm, B., C. Abts and E. Bailey (1998). COCOTS Software Integration Cost Model: an Overview. In Proceedings of the California Software Symposium.

Chou, S.C., J.Y. Chen and C.G. Chung (1996), A behavior-based classification and retrieval technique for object oriented specification reuse, Software – Practice and Experience, 26(7), pages 815-832.

Bruneton, E., T. Coupaye and J. B. Stefani (2004). The Fractal Component model specification. ObjectWeb Consortium, France Telecom and INRIA.

Chung, L., B. Nixon, E. Yu and J. Mylopoulos (2000), Non-Functional Requirements in Software Engineering. Kluwer Academic Publisher.

Crnkovic I., M. R. V. Chaudron and S. Larsson (2006), Component-Based Development Process and Component Lifecycle, ICSEA, page 44.

Crnkovic I., M. Chaudron, S. Sentilles and A. Vulgarakis (2007), A Classification Framework for Component Models, Proceedings of the 7th Conference on Software Engineering and Practice in Sweden.

Devanbu, P., R. Brachman, P. Selfridge and B. Ballard (1991). Lassie: AKnowledge-based software information system. Communications of the ACM, 34(5):34-39.

Erickson J. and K. Siau (2008). Web Services, Service-Oriented Computing, and Service-Oriented Architecture : Separating Hype from Reality. Database Manag., vol. 19, no 3, pages 42-54.

G. Finnie, G. Wittig, and D. Petkov (1995). Prioritizing software development productivity factors using the analytic hierarchy process. Journal of Systems and Software, 2, pages 129–139,

Frakes W.B. and B.A. Nejmeh (1987), An information system for software reuse, In Proceedings, 10th Minnowbrook Workshop on Software Reuse, Minnowbrook, NY.

Frølund, S. and J. Koistinen (1998). Quality of service in distributed object systems design. In 4th USENIX Conference on Object-Oriented Technologies and Systems (COCOTS), Santa Fe (New Mexico). USENIX.

Gandon, F. (2002). Distributed Artificial Intelligence and Knowledge Management: Ontologies and Multi-Agent Systems for a Corporate Semantic Web. Thèse de INRIA et University of Nice - Sophia Antipolis.

George, B. (2007). Un processus de sélection de composants logiciels multi-niveaux. Thèse de l'université de Bretagne sud.

Gligorov R., W. ven Kate, Z. Aleksovski and F. van Harmelen (2007). Using Google Distance to Weight Approximate Ontology Matches. Proceedings of the 16th international conference on World WideWeb, pages 767 – 776.

Guarino N, C. Masolo and G. Vetere (1999). OntoSeek: Content-Based Access to the Web. IEEE Intelligent Systems, 14 (3), pages 70-80.

Gaudel, M.C. and Th. Moineau (1991). A theory of software reusability, In Lecture Notes in Computer Science, Volume 300, pages 115-130 Springer-Verlag.

Hall, R.J. (1993), Generalized behaviour-based retrieval, In Proceedings, International Conference on Software Engineering, Baltimore, MD.

Hemer, D. and P. Lindsay (2001). Specification-based Retrieval Strategies for Module Reuse. In D. Grant and L. Sterling, editors, Proceedings 2001 Australian Software Engineering Conference, 27-28 August 2001, Canberra, Australia, pages 235-243, IEEE Computer Society.

Helm, R. and Y.S. Maarek (1991). Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries, In proceedings, OOPSLA'91, 26(11), pages 47-61.

Hock-koon, A. and M. Oussalah (2012). Toward a Conceptual Comparison Framework between CBSE and SOSE. CAiSE Forum 2012. Pages 82-89.

Isakowitz, T. and R. Kauffman (1996). Supporting search for reusable software objects. IEEE Transactions On Software Engineering, 22 (6), pages 407-423.

ISO (2001). ISO/IEC 9126-1: 2001. Software Engineering – Product Quality - PartI: Quality model. Geneva, Switzerland.

ISO (2010). ISO/IEC 25010: 2010. Software Product Quality Model. Geneva, Switzerland.

ITU-T (1994). Terms and definitions related to quality of service and network performance including dependability. CCITT/ITU, Report: Recommendation E.800 (08/94).

Karlsson, E.A. (1995). Software Reuse – A Holistic Approach. Willey Edition.

Kontio, J. (1996). A case study in applying a systematic method for cots selection. In Proceedings of International Conference on Software Engineering (ICSE).

Koopmans, T. (1951). Analysis of production as an efficient combination of activities. In Activity Analysis of Production and Allocation, pages 33–97. John Wiley and Sons.

Kuhn, H. and A. Tucker (1951). Nonlinear programming. In Proceedings of the 2nd Berkeley Symposium onMathematical Statistical andProbability, pages 481–491.

Laprie, J. C., B. Randell and C. Landwehr (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. In IEEE Transactions on Dependable & Secure Computing. Vol. 1, No. 1, pages 11–33.

Lau, K. and Z. Wang (2007). Software component models. In Software Engineering, IEEE Transactions on. Vol.33, Issue 10, pages 709-724.

Le Meur, A., P. Lahire, G. Waignier, P. Collet, N. Plouzeau, M. Dao, L. Duchien, A. Ozanne, N. Rivierre (2007). Spécification d'une architecture pour la contractualisation de services : expression du besoin. Research Report RNTL Faros, number F-1.2, pages 1-30.

Lozano-Tello, A. and A. Gomez-Perez (2002). Baremo : How to choose the appropriate software component using the analytic hierarchy process. In Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy.

Maarek, Y.S., D.M. Berry and G.E. Kaiser (1991). An information retrieval approach for automatically constructing software libraries. IEEE Trans. On Soft. Eng, 17(8), pages 800-813.

Macedo, K. (2004). Modélisation d'aspects qualité de service en UML : Application aux composants logiciels. Thèse de l'université de Rennes 1.

MacKenzie, M., K. Laskey, F. McCabe, P. Brown and M. Rebekah (2006). Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS.

Maedche, A. and S. Staab (2002). Measuring similarity between ontologies. In Proceeedings of the European Conference on Knowledge Acquisition and Management - EKAW -LNCS, Springer.

Maiden, N., C. Ncube, and A. Moore (1997). Lessons learned during requirements acquisition for cots systems. Communications of the ACM, 40 (12), pages 21–25.

Maiden, N. and C. Ncube (1998). Acquiring cots software selection requirements. IEEE Transactions on Software Engineering, 24 (3), pages46–56.

McIlroy, M. (1968). Mass produced software components. In NATO Software Engineering Conference Report, pages 79–85, Garmish, Germany.

Masmoudi, A., G. Paquette and R. Champagne (2008). Metadata-driven software components aggregation process with reuse. International Journal Advanced Medi and Communication, pages 35-58.

Matsumoto,Y. (1987). A software factory: An overall approach to software production, in "Software Reusability" ed. by P. Freeman, IEEE Computer Society, pages 155- 178.

Mishne, G. (2004). Source Code Retrieval using Conceptual Similarity, In: Proceedings RIAO 2004, pages 539-554.

Mili, H. et al. (2001). Automating the Indexing and Retrieval of Reusable Software Components. In 6th International Workshop NLDB'01, Madrid, Spain.

Miller, G. A (1995). Wordnet : A lexical database for english. Communication ACM, 38(11), pages 39–41.

Mizoguchi, R. and M. Ikeda (1996). Towards Ontological Engineering (AI-TR-96-1). Osaka: ISIR, Osaka.

Mizoguchi, R., and J. Bourdeau (2000). Using Ontological Engineering to Overcome Common AI-ED Problems. In International Journal of Artificial Intelligence and Education, vol.11 (Special Issue on AIED 2010), pages 107-121.

Mohamed, A., Ruhe, G., Eberlein, A. (2007). COTS Selection: Past, Present, and Future. In Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07).

Moineau, Th. and M.C. Gaudel (1991). Software reusability through formal specifications, In Prieto-Diaz R., Schaefer W., Cramer J., editors, Proceedings, First international Workshop on software reusability, number Memo Nr 57 in Unido, Dortmund.

Moldovan, D. and R. Mihalcea (2000). Improving the search on the Internet by using WordNet and lexical operators. IEEE Inter. Comp. 4(1), pages 34-43.

Ncube, C. (2000). A Requirements Engineering Method for COTS-Based Software Development. PhD thesis, City University, London, UK.

Ncube, C. and J. Dean (2002). The limitations of current decision-making techniques in the procurement of cots software component. InProceedings ofthe1stInternational Conference on COTS-Based Component Systems (ICCBSS), pages 176–187, Orlando, Florida, USA.

Nitto, E. D., C. Ghezzi, A. Metzger, M. P. Papazoglou and K. Pohl (2008). A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng.,vol. 15, no 3-4, pages 313-341.

Noy, N.F. and C.D. Hafner (1997). The State of the Art in Ontology Design, AI Magazine. pages 53-74.

OMG, Object Management Group (2002). CORBA Component Model, v3.0, formal/2002-06-65.

Paolucci, M., T. Kawamura, T. R. Payne and K.Sycara (2002). Semantic Matching of Web Services Capabilities. Proc. 1st Int'l Semantic Web Conf. (ISWC2002), pages 333-347.

Papaioannou, I. V., D. T. Tsesmetzis, I. G. Roussaki, and M. E. Anagnostou (2006). A QoS Ontology Language for Web Services. In Proc. of the 20th International Conference on Advanced Information Networking and Applications (AINA2006), IEEE Computer Society, pages 18-25.

Park, Y. and P. Bais (1997), Generating samples for component retrieval by execution, Technical report, University of Windsor, Windsor, Ontario, Canada.

Paul, S., A. Prakash (1994). Querying Source Code using an Algebraic Query Language, International Conference on Software Maintenance, IEEE Press, pages 127-136.

Peat, H.J. and P. Willett (1991), The limitations of term co-occurrence data for query expansion in document retrieval systems, J. of the ASIS, 42(5), pages 378-83.

Podgursky, A. and L. Pierce (1993), Retrieving reusable software by sampling behavior, acm Trans on Software Engineering and Methodology, 2(3), pages 286-303.

Poulin, J. and K.J. Werkman (1995). Modeling structured abstracts and the world wide web for retrieval of reusable components. In Proceedings, Symposium on software Reuse, Seattle, Washington.

Prieto-Diaz, R. (1991). Implementing faceted classification for software reuse, Communications of the acm, 34(5), pages 88-97.

Psyché, V., O. Mendes and J. Bourdeau (2003). Apport de l'ingénierie ontologique aux environnements de formation à distance. STICEF, Vol. 10, ISSN : 1764-7223.

Ralalason, B. (2010). Représentation multi-facette des documents pour leur accès sémantique. Thèse de l'université Toulouse III, Paul Sabatier.

Rijsbergen C.J. (1979). Information Retrieval. Butterworth and Co., London, 2nd edition.

Ritti M. (1989). Using types as search keys in function libraries. In Proceedings of the fourth international conference on Functional programming languages and computer architecture, Imperial College, London, United Kingdom, pages 174 – 183.

Ritti, M. (1992). Retrieving library identifiers via equational matching of types. Technical Report 65, Programming Methodology Group, Dept of Computer Science, Chalmers University of Technology and University of Goteborg, Goteborg, Sweden.

Rollins, E.J. and J. Wing (1991). Specifications as search keys for software libraries. In Proceedings of the Eighth International Conference on Logic Programming, pages 173--187.

Roy, B. (1996). Multicriteria Methodology for Decision Systems. Kluwer Academy, Dordrecht.

Ruhe, G. (2004), Intelligent Support for Selection of COTS Products, LNCS, Springer, vol. 2593, pages 34-45.

Russ, T., A. Valente, R. MacGregor and W. Swartout (1999). Experiences in Trading Off Ontology Usability and Reusability. The Twelfth Banff Knowledge Acquisition for Knowledge based Systems Workshop, Alberta, Canada.

Saaty, T. (1992). Analytic hierarchy. In Encyclopedia of Science and Technology, pages 559–563. McGraw-Hill.

Salton, G. (1986). On the use of term associations in automatic information retrieval. Proceedings of the 11th Intern. Conf. on Computational Linguistics, pages 380-386.

Simao, R. and A. Belchior (2003). Quality characteristics for software components : Hierarchy and quality guides. In LNCS, editor, Component-Based Software Quality, pages 184–206, Springer-Verlag.

Staab, S., A. Maedche (2001). Axioms are Objects, too – Ontology Engineering beyond the Modeling of Concepts and Relations. In Internal Report 399, Institute AIFB, Karlsruhe University.

Steigerwald, R.A (1992). Reusable component retrieval with formal specifications. In proceedings of the 5th Annual Workshop on Software Reuse, Palo Alto, CA.

Stojanovic, Z. and A. Dahanayake (2005). Service-oriented Software System Engineering Challenges And Practices, IGI Publishing, Hershey, PA, USA.

Sugumaran, V. and V.C. Storey (2003). A semantic-based approach to component retrieval. SIGMIS Database. 34(3), pages 8-24.

Szyperski, C. (2002). Component Software : Beyond Object-Oriented Programming. Second

Edition. Addison-Wesley.

Uschold, M., and M. Gruninger (1996). Ontologies : Principles, Methods and Applications Knowledge, Engineering Review, 11(2).

Vogel, A., B. Kerhervé, G. Bochmann and J. Gecsei (1995). Distributed multimedia and QoS - A survey. In IEEE Multimedia, volume 2, pages 10–19.

Voorhees, E. (1994), Query expansion using lexical-semantic relations, Proceedings of the 17th Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval, pages 61-69, Dublin, Ireland.

Wirsing M. (1991). Algebraic specification. In Handbook of Theoretical Computer Science, Vol. B. North-Holland, Amsterdam, pages 677-788.

Xiaomeng S. and J. G. Atle (2006). An information retrieval approach to ontology mapping. Data& Knowledge Engineering, Vol. 58 Issue 1, pages 47-69.

Yessad, L. (2007). Vers un modèle QoS-Fractal: Gestion de la qualité de service des composants Fractal. In Proceedings of ISPS'07, 7-9 mai, Alger, Algérie, pages 88-95.

Yessad, L. and Z. Boufaida (2010). QoS-based Component Selection using Semantic Web Technologies. In Proceedings of ICWIT'10, 16-19 June, 2010, Marrakech, Morocco, pages 231-240.

Yessad, L. and Z. Boufaida (2011). A QoS Ontology-based Component Selection. In International Journal on Soft Computing (IJSC), Vol.2, No.3, pages 16-30. ISSN: 2229 – 7103.

Zaremski, A.M. and J.M. Wing (1993). Signature matching: A key to reuse, In Proceedings, SIGSOFT'93: ACM SIGSOFT Symposium on the Foundations of Software Engineering,

Redondo Beach, CA.

Zaremski, A.M. and J.M. Wing (1995(a)). Signature matching: A tool for using software libraries, ACM Transactions on Software Engineering and Methodology, 4(2), pages 146-170.

Zaremski A.M. and J.M. Wing (1995(b)). Specification matching of software components, In Proceedings, SIGSOFT'95: third ACM SIGSOFT Symposium on the Foundations of Software engineering, New York, NY: ACM Press, October 1995.

Zhang, Z., L. Svensson, U. Snis, C. Srensen, H. Fgerlind, T. Lindroth, M. Magnusson and C. Stlund (2000). Enhancing Component Reuse Using Search Techniques, Proceedings of IRIS 23. Laboratorium for Interaction Technology, University of Trollhttan Uddevalla.

Annexe A

Annexe A : Le modèle CQM (Alvaro et al., 2006)

{ P :Presence

IV : IValues

R :RatioType

}

Tableau A.1 : Le modèle CQM avec deux types de caractéristiques: runtime et life cycle

Caractéristiques	Sous-caractéristiques (Runtime)	Sous-caractéristiques (Life cycle)
Functionality	Accuracy Security	Suitability Interoperability Compliance **Self-contained**
Reliability	Fault Tolerance Recoverability	Maturity
Usability	**Configurability**	Understandability Learnability Operability
Efficiency	Time Behavior Resource Behavior **Scalability**	Changeability Testability
Maintainability	Analyzability Stability	
Portability	*Deployability*	Replaceability Adaptability **Reusability**
Marketability	**Development time** **Cost** **Time to market** **Targeted market** **Affordability**	

Tableau A.2 : Attributs de qualité mesurables à l'exécution

Sous-caractéristiques (Runtime)	Attributs	Métriques	Type
Accuracy	1. Correctness	Tests results /precision	*R*
Security	2. Data Encryption	Mechanism implemented	*P*
	3. Controllability	Nº. of interfaces/ kind of controllability	*R*
	4. Auditability	Mechanism implemented	*P*
Recoverability	5. Error Handling	Mechanism implemented	*P*
Fault Tolerance	6. Mechanism available	Mechanism identification	*P*
	7. Mechanism efficiency	Amount of errors tolerate / total errors found	*R*
Configurability	8. Effort for configure	Time spend to configure correctly	*IV*
Time Behavior	9. Response time	Time taken between a set of invocations	*IV*
	10. Latency a.Throughput ("out")	Amount of outputs produced with success / period of time	*IV*
	b.Processing Capacity ("in")	Amount of inputs produced with success / period of time	*IV*
Resource Behavior	11. Memory usage	Memory used	*IV*
	12. Disk usage	Disk used	*IV*
Scalability	13. Processing capacity	A large set of calls / response time of each call	*R*
Stability	14. Modifiability	A set of modifications, if possible / % of correct behavior	*R*
Deployability	15. Complexity level	Time taken for deploy	*IV*

Tableau A.3 : Attributs de qualité mesurables lors du développement

Sous-caractéristiques (Life cycle)	Attributs	Métriques	Type
Suitability	1. Coverage	Specified functionality /% implemented	*R*
	2. Completeness	Implemented functionalities / total of specified functionalities	*R*
	3. Pre and Post-conditioned	Verification of the pre and post-conditions	*P*
	4. Proofs of Pre and post-conditions	Proofs verification	*P*
Interoperability	5. Data Compatibility	Analysis of the data standard	*P*
Compliance	6. Standardization	Implementation and Documentation analysis	*P*
	7. Certification	Verify documentation	*P*
Self-contained	8. Dependability	Implementation analysis	*R*
Maturity	9. Volatility	Analysis of the time between commercial versions	*IV*
	10. Failure removal	Nº. of bugs fixed in a version	*IV*
Understandability	11. Doc. available	Documentation Analysis	*P*
	12. Doc. quality	Documentation Analysis	*R*
Learnability	13. Time and effort to (use, config. And expertise) the component.	Component usage through Implementation of some examples and demos	*IV*
Operability	14. Complexity level	All functionalities usage / time to operate	*R*
	15. Provided interfaces	Nº. of provided interfaces	*IV*
	16. Required interfaces	Nº. of required interfaces	*IV*

	17. Effort for operating	Operations in all provided interfaces / total of the provided interfaces	*P*
Changeability	18. Extensibility	% of the functionalities that could be extended	*R*
	19. Customizability	Nº. of parameters to configure the Provided interface / Nº. of interfaces	*R*
Testability	20. Test suit provided	Analysis of the test suites provided	*P*
	21. Extensive component test cases	% of tests cases made / errors found/corrected	*R*
	22. Component tests in a specific environment	Nº. of environments that the component was tested	*IV*
	23. Proofs the component	Proofs Analysis	*P*
Adaptability	24. Mobility	Nº. of containers that the component can be deployed	*IV*
	25. Configuration capacity	Effort needed to transfer a component to other environments	*R*
Replaceability	26. Backward compatibility	Analysis of the compatibility with previous versions	*R*
Reusability	27. Domain abstraction level	Implementation and documentation analysis	*R*
	28. Architecture compatibility	Analysis of the component architecture	*R*
	29. Modularity	Packaging	*R*
	30. Cohesion	Analysis of the inter-related parts	*R*
	31. Coupling	Analysis of the inter-related parts	*R*